

Techniques for Runtime Monitoring and Static Verification of
Concurrent Software

by

Tayfun Elmas

A Thesis Submitted to the
Graduate School of Engineering
in Partial Fulfillment of the Requirements for
the Degree of

Doctor of Philosophy

in

Computer Engineering

Koç University

August, 2010

Koç University
Graduate School of Sciences and Engineering

This is to certify that I have examined this copy of a doctoral dissertation by

Tayfun Elmas

and have found that it is complete and satisfactory in all respects,
and that any and all revisions required by the final
examining committee have been made.

Co-Chairs of Supervisory Committee:

Asst. Prof. Serdar Taşiran

Dr. Shaz Qadeer

Reading Committee:

Prof. Attila Gürsoy

Assoc. Prof. Tolga Etgü

Asst. Prof. Hüsnü Yenigün

Date: _____

To Azize, my wife and best friend

“I was sitting writing at my textbook but the work did not progress; my thoughts were elsewhere. I turned my chair to the fire and dozed. Again the atoms were gambolling before my eyes. This time the smaller groups kept modestly in the background. My mental eye, rendered more acute by the repeated visions of the kind, could now distinguish larger structures of manifold confirmation: long rows, sometimes more closely fitted together all twining and twisting in snake like motion. But look! What was that? One of the snakes had seized hold of its own tail, and the form whirled mockingly before my eyes. As if by a flash of lightning I awoke; and this time also I spent the rest of the night in working out the rest of the hypothesis. Let us learn to dream, gentlemen, then perhaps we shall find the truth... But let us beware of publishing our dreams till they have been tested by waking understanding.”

Friedrich August Kekulé

ABSTRACT

Concurrent implementations of data structures and services form the backbone of many widely-used systems such as databases, Internet servers, and file systems. In order to respond efficiently to a large number of clients accessing concurrently, such software makes use of sophisticated synchronization techniques including fine-grained locking and non-blocking operations. Incorrect use of these techniques makes software prone to concurrency bugs, which can have serious consequences, such as data corruption, operating system crash, or even more catastrophic results, e.g., failure of an aircraft flight control system. Concurrency-related bugs are notoriously difficult to detect, reproduce, and diagnose using code review and testing-based techniques originally developed for sequential programs. This difficulty creates demand for new program analyses that are capable of examining concurrent software precisely with high coverage of its executions. In this thesis, we present two techniques, each following a different direction to respond to this demand.

Data races are often symptomatic of higher-level logical errors and may have unintended consequences. The first part of this thesis presents **GOLDILOCKS**, a Java runtime that monitors program executions and throws a **DataRaceException** when a data race is about to occur. This prevents racy accesses from taking place, and allows race conditions to be handled before they cause errors that may be difficult to diagnose later. Multithreaded executions in our runtime are guaranteed to be race free and thus sequentially consistent as per the Java Memory Model. This strong guarantee provides an easy-to-use, clean semantics to programmers, and helps to rule out many concurrency-related possibilities as the cause of errors. Therefore, the **DataRaceException** is a valuable debugging tool, and, if supported with reasonable computational overhead, can be an important safety mechanism for deployed programs. To support the **DataRaceException**, we developed a novel, precise data-race detection algorithm called **GOLDILOCKS**. The **GOLDILOCKS** algorithm is general, intuitive, and can handle different synchronization patterns and software transactional memory uniformly. We have implemented our algorithm and the **DataRaceException** in the Kaffe Java Virtual

Machine and evaluated our system on a variety of publicly available Java benchmarks. Our experiments indicate that our implementation has reasonable overhead and its performance is competitive with those of other algorithms in the literature.

Runtime monitoring as in GOLDLOCKS cannot fully guarantee absence of bugs. When the concurrent software is a part of, for example, a file system or a standard library, one has to verify that no execution of the software leads to an error. This can be accomplished by doing a static, formal proof on the program text. The interaction between threads over shared memory and at fine levels of concurrency make formal proofs expensive in terms of manual effort, requiring complex annotations and invariants, along with many auxiliary variables. In the second part of the thesis, we present QED, a proof system for static verification of concurrent programs. The key feature of our approach in QED is the use of atomicity as the central proof tool to overcome the challenges in proofs of safety properties. We support two widely-accepted safety specification, assertions and linearizability. While assertions specify local properties of the program, linearizability provides more global and stringent criteria for data structure implementations. A proof is done by rewriting the program with larger atomic blocks in a number of steps. A novel feature of our approach is alternating proof steps that apply reduction and abstraction, each of which improves the outcome of the other in a following step. After reaching a desired level of atomicity, we check assertions sequentially within the atomic blocks of the resulting program and declare the original program correct when we discharge all the assertions. This strategy leads to significantly simplified proofs by allowing the user to express the design intent in a clear way and facilitating a clean separation of concerns about the sequential properties and the concurrency control mechanisms in the program. Our proof system is orthogonal and complementary to existing methods and enables more tractable application of them. Our publicly available software tool, QED-VERIFIER, mechanizes proofs by encoding the low-level logical reasoning required by the proof rules as verification conditions automatically checked by the Z3 SMT solver. We demonstrated the simplicity and practicality of our proof strategy by verifying well-known programs from the literature and showed that atomicity is a powerful tool for reasoning about programs using a wide range of intricate concurrency techniques including fine-grained locking and non-blocking algorithms.

ÖZETÇE

Veri yapılarının ve servislerin koşut-zamanlı gerçekleştirmeleri, veritabanları, Internet sunucuları ve dosya sistemleri gibi geniş kullanım alanına sahip pek çok sistemin omurgasını oluşturmaktadır. Bu yazılımlar, koşut-zamanlı olarak çalışan çok sayıda istemciye verimli şekilde cevap verebilmek için küçük parçalı kilitleme ve bloklanmayan operasyonlar gibi eşleme teknikleri kullanırlar. Bu tekniklerin yanlış kullanımı, veri bozulması ve işletim sisteminin çökmesi, ve hatta bir uçuş kontrol sisteminin arızası gibi ciddi sonuçları olan koşut-zaman hatalarına yol açar. Koşut-zaman hatalarının tespiti, yeniden oluşturulması ve düzeltilmesi aslen ardışık yazılımlar için geliştirilen tekniklerle zor olmaktadır. Bu zorluk, koşut-zamanlı yazılımları kesin olarak, ve koşullarının büyük kısmını kapsayacak şekilde sınavabilecek yeni program analizlerine ihtiyaç oluşturmaktadır. Bu tezde, bu ihtiyaca cevap veren ve farklı yollar izleyen iki teknik sunulmaktadır.

Yarış durumları çoğu kez yüksek seviyede hataların belirtisidir ve istenmeyen koşullara yol açmaktadır. Bu tezin ilk kısmı, programın koşullarını izleyerek, bir yarış durumunun hemen öncesinde bir `DataRaceException` fırlatan çalıştırma ortamı olan `GOLDLOCKS`'ı sunmaktadır. Böylece, yarış durumu oluşturacak veri erişimlerinin oluşması engellenmekte, ve bu yarış durumlarının, teşhisinin zor olacağı hatalara yol açmadan işlenmesine izin verilmektedir. Sunulan çalışma ortamında çok-örgülü koşulların yarışsız, ve Java Bellek Modeli'ne göre ardışık tutarlı olması garantilenmektedir. Bu teminat, programcılara kolay kullanılabilir ve net bir semantik sağlamakta, ve hata ayıklama sırasında koşut-zamanla ilişkili pek çok olasılığın ortadan kaldırılmasına yardımcı olmaktadır. Böylece `DataRaceException`, değerli bir hata ayıklama aracı olmakta, aynı zamanda eğer makul bir hesaplama masrafiyle desteklenirse, yazılım için dağıtımı sonrası önemli bir güvenlik mekanizması olabilecektir. Ayrıca bu tezde, `DataRaceException`'ı desteklemek için `GOLDLOCKS` adlı dinamik ve kesin bir yarış durumu tespit algoritması gerçekleştirilmiştir. `GOLDLOCKS` algoritması genel ve sezgisel olmakla birlikte, farklı eşleme desenlerini ve yazılım işlem belleğini kullanan programları aynı biçimde işleyebilmektedir. Algoritmamız ve `DataRaceException`, Kaffe adlı bir Java

sanal makinesinde gereklenmiř, ve sistemimiz eřitli aık eriřimli Java programları kullanılarak deęerlendirilmiřtir. Deneylerimiz, bu gerekleřtirmenin makul masrafları olduęunu ve bařarımının literatürdeki dięer algoritmalarla rekabeti olduęunu gstermektedir.

GOLDILOCKS gibi alıřma-zamanı izleme teknikleri hataların yokluęunu tamamen garanti edemez. Kořut-zamanlı yazılım, örneęin bir dosya sistemi ya da standart kütüphanenin bir parası ise, hi bir kořusunun hataya yol amayacaęının doęrulanması gerekmektedir. Bu, programın kaynak kodu üzerinde yapılacak duraęan bir biimsel ispat ile bařarılabilir. Örgülerin paylařımlı bellek üzerindeki olan küük paralı etkileřimleri, biimsel ispatların kullanıcının ek girdileri aısından pahalı olmasına yol amaktadır. Tezin ikinci kısmında, kořut-zamanlı programların duraęan doęrulanması için geliřtirdięimiz bir ispat sistemi olan QED sunulmaktadır. Yaklařımımızın kilit noktası, atomiklięin, güvenliik özelliklerinin ispatındaki zorlukların üstesinden gelmek amacıyla merkezi ispat aracı olarak kullanılmasıdır. İddialar ve sıralanabilirlik gibi geniř apta kabul gören iki güvenliik özellięi desteklenmektedir. İddialar, programın yerel özelliklerini belirtmek için kullanılırken, sıralanabilirlik, veri yapıları için daha global ve zorlu ölçütler tanımlamaktadır. İspatlar, programın daha geniř atomik bloklarla adım adım yeniden yazılması ile yapılmaktadır. Buradaki yenilik, indirme ve soyutlama tekniklerini uygulayan ispat adımlarının, birbirinin bir sonraki ıktısını geliřtirecek řekilde deęiřimli olarak kullanılmasıdır. İstenilen atomiklik seviyesine ulařılınca, iddialar, ortaya ıkan programdaki atomik blokların ierisinde ardışık olarak denetlenmekte, iddiaların hepsi ispatlanınca orijinal programın doęruluęu ilan edilmektedir. Bu strateji, kullanıcının tasarım amacını aıka ifade etmesini saęlayarak, ve program ierisindeki ardışık özellikler ve kořut-zaman kontrol mekanizmalarına olan ilginin net bir řekilde ayrımını kolaylařtırarak, ispatları önemli ölçüde basitleřtirmektedir. İspat sistemimiz, var olan yöntemleri tamamlayıcı olup, bunların daha uysal bir řekilde uygulanmasına imkan vermektedir. Aık eriřimli yazılım aracımız QED-VERIFIER, ispat kurallarının gerek duyduęu alt-seviye mantıksal ıkarım iřlemlerini Z3 SMT özücünün otomatik olarak denetleyebileceęi doęrulama řartları olarak ifade ederek, ispatları mekanize etmektedir. QED'deki stratejimizin basit ve pratiklięi, literatürde iyi bilinen programlar doęrulanarak ispatlanmış, atomiklięin, küük paralı kilitleme ve bloklanmayan algoritmalar gibi karışık kořut-zaman teknikleri kullanan programlar hakkında ıkarım yapmak için güçlü bir araç olduęu gsterilmiřtir.

ACKNOWLEDGMENTS

I would like to express my immense gratitude to my supervisors *Serdar Taşiran* and *Shaz Qadeer*. Their experience, understanding, and patience added considerably much to my graduate experience, and their valuable comments and advices helped me in adapting intensive academical research. I would like to thank the other members of my thesis committee, *Attila Gürsoy*, *Tolga Etgü*, and *Hüsnü Yenigün* for critical reading of this thesis and for their strong guidance throughout this study. I am also grateful to the anonymous reviewers of my publications and all others who contributed highly to this work with their invaluable comments and suggestions.

I am grateful to The Scientific and Technical Research Council of Turkey (TÜBİTAK) for providing financial support under the National Doctoral Funding Programme (2211). The research presented in this thesis was also supported by TÜBİTAK under grant 104E058, the Turkish Academy of Sciences Distinguished Young Scientist Award (TÜBA-GEBİP), and research gifts from Microsoft Research.

I would like to thank *Jim Larus*, *Roy Levin*, and *Sriram Rajamani* for making our collaboration with Microsoft Research possible. I would like to express my appreciation to the members of the Research in Software Engineering (RiSE) group for his helpful discussions and contributions to my thesis. I am also indebted to the Spec# team, particularly *Rustan Leino* and *Mike Barnett*, for their support with using the Boogie/Spec# framework.

I am grateful to all my friends from Koç University, for their close friendship during the whole course of my graduate study, especially the members of *ENG-230*, *Soner*, *Erkan*, *Ömer*, and *Ali*. I am thankful to my very special friends *Temel*, *Galip*, *Murat*, *Ali*, and *Okta*y for their continuous support throughout my study.

I must give immense gratitude to my parents *Süleyman* and *Zeynep*, my dear sister *Hatice*, and the new members of my family *Adil*, *Feride*, *Behram*, and *Elnura* for their endless support and patience. Finally, I would like to give my heartily thanks to my wife *Azize* for her extreme morale support and for always believing me in hard days of my life.

TABLE OF CONTENTS

List of Tables	xix
List of Figures	xx
List of Theorems	xxiv
Nomenclature	xxv
 I Goldilocks: A Race-Aware Java Runtime	 1
Chapter 1: Introduction	2
1.1 Contributions and Outline of Part I	6
Chapter 2: The State of The Art in Dynamic Data-Race Detection	9
2.1 Memory Models and Data Races	9
2.2 Dynamic Race Detection Algorithms and Tools	11
Chapter 3: The Goldilocks Algorithm	18
3.1 A Generic Memory Model	18
3.1.1 Variables and Actions	19
3.1.2 Multithreaded Executions	19
3.1.3 Synchronizes-with and Happens-before	20
3.1.4 Sequential Consistency	21
3.1.5 Data Races	21
3.2 The Generalized Goldilocks Algorithm	22
3.3 Correctness of the Goldilocks Algorithm	25
3.4 Distinguishing Read and Write Accesses	32

Chapter 4:	Specializing Goldilocks to The Java Memory Model	37
4.1	The Java Memory Model (JMM)	37
4.1.1	Data Variables and Operations	38
4.1.2	Synchronization Variables and Operations	38
4.2	The Goldilocks Algorithm for Java	39
4.2.1	Handling Other Synchronization Mechanisms	41
4.3	Example: Precise Data-Race Detection for Java	42
4.4	Example: A Task Queue	44
4.5	Discussion: Handling DataRaceException	47
4.5.1	Default Action for DataRaceException	47
4.5.2	Example: Writing an Exception Handler for DataRaceException	49
4.6	Discussion: Dynamic Race Detection and Sequential Consistency	51
Chapter 5:	Dynamic Race Detection and Software Transactional Memory	53
5.1	Motivating Examples	53
5.1.1	Example 1: Handling Software Transactions	53
5.1.2	Example 2: Locks and Software Transactions	55
5.2	Extending the Algorithm for Software Transactions	56
5.3	Revisiting Example 2: Handling Software Transactions	59
5.4	Discussion: Adapting the Algorithm to Different STM Implementations	60
Chapter 6:	Implementing Goldilocks	62
6.1	Implicit Representation and Lazy Evaluation of Goldilocksets	63
6.2	Decentralized checking of races	65
6.3	Performance Optimizations	66
6.3.1	Short Circuit Checks	66
6.3.2	Direct Ownership Transfer	67
6.3.3	Garbage Collection of Synchronization Event List	67
6.3.4	Partially-Eager Evaluation	68
6.3.5	Sound Static Race Analysis	69

6.4	The Pseudo-code of the Implementation	71
6.5	Handling Transactions in the Implementation	75
Chapter 7:	Experiments with Goldilocks	77
7.1	Comparing Goldilocks with Eraser and Vector Clock	77
7.1.1	Benchmarks	78
7.1.2	Discussion about the Experimental Results	81
7.2	Experience with Optimized Goldilocks Runtime	83
7.2.1	Experiments with Transactions	87
7.3	Discussion: Race-Detection Overhead	90
Chapter 8:	Conclusion	93
8.1	Future Work: Specializing GOLDBLOCKS to The C++ Memory Model	94
II	The QED Proof System	96
Chapter 9:	Introduction	97
9.1	Contributions and Outline of Part II	101
Chapter 10:	The State of The Art in Static Verification	108
10.1	Specifying Programs	108
10.1.1	Assertional Specifications	108
10.1.2	Behavioral Specifications	110
10.2	Reasoning about Concurrency	111
10.3	Static Verification Techniques for Concurrent Software	112
10.3.1	The Hoare Logic	113
10.3.2	The Owicki-Gries Method	115
10.3.3	The Rely/Guarantee Approach	116
10.3.4	Concurrent Separation Logic	118
10.3.5	Summary and Discussion	119
10.4	Abstraction and Reduction in the Literature	120

Chapter 11:	Motivation and Overview	124
11.1	Preliminaries	124
11.1.1	Notation	124
11.1.2	Informal Introduction to Reduction and Abstraction	125
11.2	Lock-based Increment	126
11.3	Lock-free Increment	130
11.4	A Device Cache	134
11.4.1	Device Cache Operation	135
11.4.2	Synchronization Mechanisms	136
11.4.3	Specification	137
11.4.4	Verification	137
Chapter 12:	The ActionPL Programming Language	140
12.1	Expressions: Syntax and Semantics	140
12.1.1	Informal Notation for Expressions	141
12.2	Program	142
12.2.1	Types	143
12.2.2	Functions	144
12.2.3	Constants	144
12.2.4	Axioms	145
12.2.5	Global Variables	145
12.2.6	Procedures	146
12.2.7	Program Variables and Scoping	146
12.2.8	State and Transition Predicates	147
12.3	Statements: Syntax	148
12.3.1	Gated Actions	148
12.3.2	Procedure Calls	150
12.3.3	Compound Statements	150
12.4	Operational Semantics	151
12.4.1	Execution Model	151

12.4.2	Program State	152
12.4.3	Program States and State/Transition Predicates	154
12.4.4	Labeled Statements and Exception Handling	155
12.4.5	Semantics of Statements and State Transitions	156
12.4.6	Inlining Procedures	159
12.4.7	Runs of a Program	160
12.4.8	Good and Bad Programs	161
Chapter 13:	The QED Proof System	163
13.1	Proving Programs Correct in QED	163
13.1.1	Proof State	163
13.1.2	Proof Steps	164
13.1.3	Proving Assertions	165
13.2	Invariants	166
13.3	Abstraction	166
13.3.1	Borrowing Assertions	167
13.3.2	Read and Write Abstractions	168
13.3.3	Validating Gated Actions	169
13.4	Variable Addition and Hiding	170
13.4.1	Adding Auxiliary Variables	170
13.4.2	Hiding Existing Variables	171
13.5	Reduction	172
13.5.1	Atomic Composition of Actions	173
13.5.2	Movers	174
13.5.3	The Checks for Movers	176
13.5.4	Deciding Right and Left Movers	178
13.5.5	Role of Assertions in Mover Checks	180
13.5.6	Deciding Movers After Reduction Steps	181
13.5.7	Reducing Sequential Composition	181
13.5.8	Reducing Nondeterministic Choice	182

13.5.9	Reducing Parallel Composition	182
13.5.10	Reducing loops	184
13.6	Reducing Procedures	186
13.6.1	Recursive Procedures	186
13.7	Soundness of the Proof System	191
Chapter 14:	The QEDPL Programming Language	207
14.1	Syntax and Informal Semantics	208
14.1.1	Exception Handling	211
14.2	Denotational Semantics	212
14.2.1	Translating Atomic Statements	213
14.2.2	Translating Compound Statements	215
14.3	Extra Features	216
14.3.1	While Loops	217
14.3.2	Maps	218
14.3.3	Records	220
Chapter 15:	The Proof Tactics for QEDPL	225
15.1	Proof Transformations on QEDPL	225
15.2	Proof Tactics	227
15.2.1	Invariant Introduction	229
15.2.2	Abstraction	230
15.2.3	Variable Introduction and Hiding	233
15.2.4	Reduction	236
15.3	Code Transformations	244
15.3.1	Peeling Out the Last Iteration of While Loops	245
15.3.2	Injecting Prefix to If Statements	249
15.3.3	Splitting If Statements	251
15.3.4	Extracting Assumptions	251
15.4	Proof of Theorem 7	252

Chapter 16:	Proving Linearizability	258
16.1	Motivation and Overview	260
16.1.1	Challenges in a Typical Refinement Proof for Multiset	262
16.1.2	Proof by Reduction and Abstraction	263
16.2	The Verification Approach and Its Soundness	264
16.2.1	Linearizability	265
16.2.2	Behavioral Simulation	266
16.2.3	Connecting Behavioral Simulation to Linearizability	278
16.3	Related Work in Proving Linearizability	279
Chapter 17:	The Implementation and Proof Commands	282
17.1	Our Software Tool: QED-VERIFIER	282
17.2	Proof Strategy	286
17.3	Proof Commands	287
17.3.1	Introducing Invariants	290
17.3.2	Abstraction	290
17.3.3	Variable Addition and Hiding	291
17.3.4	Reduction	292
Chapter 18:	Proof Templates for Synchronization Idioms	296
18.1	Motivation and Overview	298
18.1.1	The example	298
18.1.2	Synchronization Idioms	300
18.1.3	Annotating Procedures	302
18.1.4	Debugging The Synchronization Idiom	305
18.1.5	Naive Attempts	307
18.2	The Annotation Method	309
18.2.1	Describing Synchronization Idioms	309
18.2.2	Relating the Idiom to the Program	310
18.2.3	Annotating Programs - Template for Manual Insertion	312
18.2.4	Annotating Programs - Algorithm	314

18.2.5	Mutual Exclusion	317
18.2.6	Reentrant Locks	318
18.2.7	Event Synchronization	319
18.3	Barrier Synchronization	322
18.4	Experience	323
18.5	Related Work in Annotation Generation for Static Verification	326
Chapter 19:	Experience with QED	329
19.1	Purity Benchmarks	330
19.2	Multiset	333
19.3	Linked-list with Hand-over-hand Locking	336
19.4	Two-lock Queue	340
19.5	Lock-based Synchronous Queue	343
19.6	The Bakery Algorithm	345
19.7	Obstruction-free Deque	347
19.8	Treiber’s Non-Blocking Stack	349
19.9	Michael’s Non-Blocking Stack	351
19.10	Non-Blocking Queue	354
19.10.1	Eliminating the Superficial Conflict	358
19.11	Non-Blocking Mutex-lock	361
19.11.1	Eliminating the Superficial Conflict	363
Chapter 20:	Conclusion	366
20.1	Future Work	367
20.1.1	Applying QED to Other Languages	367
20.1.2	Improving the Mover Checks	368
20.1.3	Using Mover Checks for Precise Commutativity Analysis	369
20.1.4	Concise Proof Commands To Simplify User-Driven Proofs	369
20.1.5	Automated Derivation of Proof Command Sequences	370
20.1.6	Using Runtime Analysis to Help Static Verification	370
20.1.7	Proof System for Parallelizing Programs	371

Bibliography	372
Vita	388

LIST OF TABLES

7.1	Runtime statistics of the benchmark programs	81
7.2	Runtime statistics when fields with races detected on them are disabled . . .	82
7.3	Results of experiments with the race-aware Kaffe runtime	85
7.4	Statistics on experiments with static analyses	87
7.5	Performance of checking races for transactional Multiset	89
17.1	Checks performed in QED-VERIFIER using the Z3 SMT solver	283
17.2	The proof commands provided by QED-VERIFIER	290

LIST OF FIGURES

2.1	Example demonstrating the need for precise data-race detection	12
3.1	Transferring ownership of x , and $GLS(x)$	23
3.2	The core Goldilockset update rules	24
3.3	Interpretation of $\pi(i) \xrightarrow{hb} \pi(j)$ and $t_i \neq t_j$ for Lemma 1.	25
3.4	Interpretation of the contents of GLS_j for Lemma 2.	27
3.5	Happens-before between reads and writes, and read goldilockset of x	32
3.6	Distinguishing read and write accesses	34
4.1	The Goldilockset update rules specialized to Java	40
4.2	Precise data-race detection example	43
4.3	Pseudo-code for the task queue example.	44
4.4	Example 1: Demonstrating use of <code>DataRaceException</code>	50
5.1	Code for Example 1	54
5.2	Code for Example 2	55
5.3	The Goldilockset update rules for software transactions	57
5.4	Handling software transactions example	59
6.1	The synchronization event list	63
6.2	Lazy evaluation of Goldilockset $GLS(x)$	65
6.3	Garbage collection of synchronization events	67
6.4	Partially-eager evaluation of Goldilocksets	69
6.5	Prelude for the pseudo-code in Figure 6.6	71
6.6	Pseudo-code for the <code>GOLDILOCKS</code> implementation	72
7.1	Per access race checking time against the increasing number of threads	78
7.2	The <code>SharedSpot</code> and <code>LocalSpot</code> microbenchmarks.	79

7.3	The <code>Multiset</code> code modified for the experiment.	88
9.1	Presentation of contributions in Part II	102
9.2	The relationship among proof rules, tactics, and commands.	105
11.1	Lock-based atomic increment	127
11.2	Lock-based atomic increment after being transformed	127
11.3	Intermediate transformations on the lock-based increment	128
11.4	The lock-free increment example	132
11.5	Client of <code>inc</code>	132
11.6	Original device cache program	134
11.7	Transformed device cache program	134
12.1	Syntax of expressions	141
12.2	Top-level elements of a program	143
12.3	The syntax of ACTIONPL statements	148
12.4	The rules of the small-step operational semantics of ACTIONPL	157
13.1	Composing gated actions	174
13.2	Rules for deciding movers	175
13.3	Composing a gated action from a compound statement.	189
14.1	The syntax of the QEDPL language.	209
14.2	Syntactic sugars for commands throwing special exception labels.	211
14.3	Compiling atomic statements to gated actions.	214
14.4	Example translations from QEDPL to ACTIONPL.	215
14.5	Compiling compound statements to ACTIONPL statements.	216
15.1	The relationship between QED proof tactics and rules.	226
15.2	The proof tactics for QEDPL	229
15.3	The first part of the loop-reduction example.	248
15.4	The second part of the loop-reduction example.	249

16.1	The concurrent implementation and the sequential specification of multiset . .	261
17.1	The interaction between QED-VERIFIER, the Boogie tool, and Z3.	284
17.2	The proof strategy for verifying a program using proof commands	287
17.3	Commands for adding (a) and hiding (b) a global variable	291
18.1	Threads accessing the (x, v) pair	299
18.2	The procedures used in Figure 18.1	301
18.3	The annotated version of the procedures in Figure 18.2	303
18.4	The annotated procedures with weaker assertions	307
18.5	The annotated procedures with stronger assertions	308
18.6	The idiom template for the r/w lock idiom	310
18.7	The proof script for adding annotations	313
18.8	The inference algorithm	315
18.9	The idiom template for the mutual exclusion idiom	317
18.10	The idiom template for a variation of the mutual exclusion idiom	318
18.11	Example implementation of reentrant locks	319
18.12	The idiom template for the reentrant locks	320
18.13	Synchronizing threads with events	321
18.14	The idiom template for the event synchronization idiom	321
18.15	Rules for hoisting actions out of a parallel statement	322
19.1	Optimistic concurrency control using transaction retry	331
19.2	Translated form of the purity example	331
19.3	Proof script of the purity example	332
19.4	Implementation of the multiset example in QEDPL	333
19.5	The atomic multiset after the QED transformations.	334
19.6	The proof script used for the multiset example in QEDPL	335
19.7	Linked-list based set implementation with hand-over-hand locking.	337
19.8	The proof script used for the linked-list based set implementation.	339
19.9	The two-lock queue implementation in QEDPL	341

19.10	The proof script for the two-lock queue	341
19.11	The final version of the two-lock queue after the QED transformations.	342
19.12	The lock-based synchronous queue implementation in QEDPL	343
19.13	The final version of the lock-based synchronous queue.	344
19.14	The proof script for the lock-based synchronous queue	344
19.15	The implementation of the bakery algorithm in QEDPL	346
19.16	The proof script for the bakery algorithm	346
19.17	Example operation of obstruction-free deque	348
19.18	The implementation of Treiber’s non-blocking stack in QEDPL	349
19.19	The proof script for Treiber’s non-blocking stack.	350
19.20	The final version of Treiber’s non-blocking stack after the QED transformations.	351
19.21	Michael’s algorithm with hazard pointers	352
19.22	The final version of Michael’s stack after the QED transformations.	353
19.23	The proof script for Michael’s algorithm with hazard pointers	354
19.24	The non-blocking queue implementation in QEDPL	355
19.25	The proof script for the non-blocking queue	358
19.26	The reduced implementation of the non-blocking queue and its specification .	358
19.27	The version of the non-blocking queue after hiding <code>Tail</code>	359
19.28	The non-blocking mutex-lock implementation in QEDPL	361
19.29	The proof script for the non-blocking mutex-lock	362
19.30	The reduced implementation and the specification of the lock	363
19.31	The lock implementation after introducing <code>thread</code> , <code>H</code> and <code>owner</code>	364

LIST OF THEOREMS

Lemma 1	Immediate happens-before	26
Lemma 2	Preservation of Goldilockset invariants	27
Theorem 1	Correctness of GOLDDILOCKS update rules	31
Theorem 2	Correctness of extended GOLDDILOCKS update rules	34
Theorem 3	The mover check $\mathcal{MC}$ is modular	179
Lemma 3	Eliminating right movers from a violation	192
Lemma 4	Moving right movers to the end of a run	192
Lemma 5	Gathering atomic statements	193
Theorem 4	Preservation of $\preceq$ by the QED proof rules	195
Theorem 5	Soundness of the QED proof system	202
Corollary 6	Soundness of proving assertions using QED	205
Theorem 7	Preservation of $\preceq$ by the proof tactics	226
Theorem 8	Soundness theorem of QED for QEDPL programs	227
Lemma 6	Behavioral simulation and code maps	267
Lemma 7	$\blacktriangleleft$ is reflexive and transitive	268
Lemma 8	Final state determines the behavior	268
Lemma 9	Behavioral simulation and right movers	268
Lemma 10	Behavioral simulation and left movers	269
Theorem 9	Preservation of $\preceq$ by the QED proof rules	270
Theorem 10	Equivalence of linearizability and behavioral simulation	278
Corollary 11	Soundness of proving linearizability using QED	279

NOMENCLATURE

A/C	Assumption/Commitment based Compositionality
ACE	The Assertion Checking Environment
ADT	Abstract Data Type
C++MM	The C++ Memory Model
CAS	Compare-And-Swap Operation
CEGAR	Counter-example Guided Abstraction Refinement
CPU	Central Processing Unit
CSL	Concurrent Separation Logic
CSP	Communicating Sequential Programs
CTL	Computational Tree Logic
DRF	Data-Race Freedom Property
ESC	Extended Static Checking
FOL	First-Order Logic
GLS	Goldilockset
JML	The Java Modeling Language
JPF	The Java Pathfinder Model Checker
JMM	The Java Memory Model
KLOC	Thousand-lines of Code
LOC	Lines of Code
LS	Lockset
LTL	Linear Time Logic
LTS	Labeled Transition System
OS	Operating System
POR	Partial Order Reduction
R/G	Rely/Guarantee based Compositionality

– *Continued on next page* –

SAT	The Satisfiability Problem
SC	Sequential Consistency
SDV	The Static Driver Verifier
SMT	Satisfiability Modulo Theories
STM	Software Transactional Memory
TLA	Temporal Logic of Actions
VC	Verification Condition

Part I

Goldilocks: A Race-Aware Java Runtime

Chapter 1

INTRODUCTION

In the first part of this thesis, we focus on data races, the most prevalent type of concurrency-related errors. A data race is said to occur when two conflicting accesses to a shared variable by different threads are enabled simultaneously, i.e., at the same program state [1]. An equivalent definition involves an execution in which two threads make conflicting accesses to a variable without proper synchronization actions being executed between the accesses. A common way to enforce race freedom is to associate a lock with every shared variable, and to ensure that threads hold this lock when accessing the shared variable. The lock release by one thread and the lock acquire by the next thread establish the required synchronization between the two threads. In addition to locks, Java provides volatile variables and fork/join operations as other synchronization primitives that can be used to avoid races.

Data races are undesirable for two key reasons. First, a race condition is often a symptom of a higher-level logical error such as an atomicity violation. Thus, race detectors serve as a proxy for more general concurrency-error detection when higher-level specifications such as atomicity annotations do not exist. Second, a race condition makes the outcome of certain shared variable accesses non-deterministic. For this and other reasons, both the Java Memory Model (JMM) [2] and the C++ Memory Model (C++MM) [3] define *well-synchronized programs* to have executions that are all free of race conditions.

For race-free executions, the JMM and C++MM guarantee sequentially-consistent semantics; in particular, every read deterministically returns the value of the “last” write. This semantics is widely considered to be the only simple-enough model with which writing useful concurrent programs is practical. For executions containing race conditions, the semantics is either completely undefined (as is the case for C++MM [3]) or is complicated so that writing a useful and correct program with “benign races” is a challenge. In the latter case, the purpose of defining semantics for racy executions is to provide constraints for compilers

and virtual machines. The recommended way to write programs is to avoid race conditions and to ensure that executions remain sequentially consistent.

Detection and/or elimination of race conditions has been an area of intense research activity. Precise detection and elimination of race conditions at compile time is an undecidable problem, as in the case with `ArrayIndexOutOfBoundsException` and `NullPointerException`. It is possible to perform a more conservative check, i.e., check a sufficient condition for race freedom, for example, adherence to a certain synchronization discipline. It is thus possible to eliminate all races before execution, but this typically requires the programmer to analyze and eliminate a number of false alarms. This is labor intensive and/or error-prone and may not be possible for open programs which rely on external synchronization by the client. As a result, deployed code may potentially contain data races. For this reason, we believe that it is a much more practical compromise to only perform a static check at compile time to determine pairs of accesses guaranteed to be race-free, and, for the rest of the accesses, perform a race condition check at runtime and throw an exception if one occurs. Such a runtime check does not only help in debugging the program during the software development and testing phases, but also provides a viable safety mechanism after the software is deployed in order to react to the races that could not be detected at pre-deployment. Following these ideas, the work presented in this thesis makes two important contributions to this area.

First, for the first time in the literature, we propose that race conditions should be language-level exceptions just like null pointer dereferences and indexing an array out of its bounds. We present a Java runtime called `GOLDILOCKS` that provides a new exception, `DataRaceException`¹, that is thrown when an access that causes an actual race condition is about to be executed. This guarantees that the execution remains sequentially consistent, since a racy execution is never allowed to take place.

`DataRaceException` brings races to the programmer's attention explicitly. When this exception is caught, the recommended course of action is to terminate the program gracefully. This is because, for racy Java programs, a wide range of compiler optimizations are allowed by the JMM, and this makes it complicated to relate program executions to the source code. If the exception is not caught, the `GOLDILOCKS` runtime terminates the thread throwing the

¹We define `DataRaceException` as a subclass of the `RuntimeException` class in the Java standard library.

exception. While not recommended, the programmer could also choose to make optimistic use of a `DataRaceException` by, for instance, retrying the code block that led to the race, hoping that the conflicting thread has performed the necessary synchronization in the meantime. Since it first appeared in our paper on the GOLDDILOCKS runtime [4], the idea that certain concurrency errors, especially ones that result in sequential consistency violations, should result in exceptions has gained significant support and several implementations of the idea have also been investigated by others [5, 6].

Most dynamic race detection algorithms in the literature are primarily intended to be used as stress testing and bug finding tools. For these algorithms, it is a useful feature to point out potential races in executions similar to the one being checked and it is acceptable to declare some false warnings as part of the compromise. However, in the context of runtime exceptions, it is not acceptable to interrupt the execution of a Java program because of a potential or false race. Thus, to support `DataRaceException`, a race detection algorithm needs to be precise, i.e., not produce false alarms or warnings about potential races. The second contribution of our work is the GOLDDILOCKS algorithm, a novel, precise, and general algorithm for detecting data races at runtime.

In this thesis, we present an implementation of the GOLDDILOCKS algorithm that incorporates lazy computation of locksets and “short circuit checks”: constant time sufficient checks for race freedom. We implemented GOLDDILOCKS in the C language and within a Java Virtual Machine (JVM) called Kaffe [7]. We have performed an experimental comparison of the GOLDDILOCKS algorithm with the vector clock algorithm and our implementation of the Eraser algorithm, and demonstrate that our algorithm is much more efficient than the former and nearly as efficient as the latter. Our experiments with GOLDDILOCKS, improved with a number of performance optimizations and implementation features, on benchmarks brought up the new possibility that the overhead of post-deployment precise race detection in a runtime may be tolerable. There has been significant progress in the efficiency of precise race detection since the GOLDDILOCKS runtime was first published (see [8, 9], for example) and this idea appears to be viable today.

The GOLDDILOCKS algorithm is based on an intuitive, general representation for the happens-before relationship (as defined in JMM) as a generalized lockset (Goldilockset) for each variable. In the traditional use of the term, a lockset for a shared variable x at a

point in an execution contains a set of locks. A thread can perform a race-free access to x at that point by first acquiring a lock in this lockset. A Goldilockset generalizes this as follows: At each point in an execution, the Goldilockset for a shared variable x may contain locks, volatile variables, and thread ids. The Goldilockset indicates the threads that have the ownership of x and the synchronization objects that protect access to x at that point. In Java, locks and volatile variables are synchronization objects, and acquiring and releasing a lock, as well as reading from and writing to a volatile variable, are synchronization operations. A thread can perform a race-free access to x if and only if its thread id is in the Goldilockset, or if it first acquires any lock, or reads any volatile variable, that is in the Goldilockset. The Goldilockset is updated during the execution as synchronization operations are performed. As a result, Goldilocksets are a compact, intuitive way to precisely represent the happens-before relationship. Thread-local variables, protecting variables by different locks at different points of the execution, and event-based synchronization with condition variables are all uniformly handled by GOLDILOCKS. Furthermore, Goldilocksets can easily be generalized to handle other synchronization primitives such as software transactions [10] and adapted to handle memory models of languages other than Java, such as C++MM. To facilitate this, in this thesis we first present GOLDILOCKS on a generic memory model and then show how the algorithm can be specialized to JMM and software transactions. In this way, we provide the first formalization of race conditions for executions that use software transactions in addition to other Java synchronization primitives.

To reduce the number of accesses checked at runtime, we use existing static analysis methods at compile time to determine accesses that are statically guaranteed to be race-free. While implementing GOLDILOCKS in Kaffe, we worked with two static analysis tools for this purpose: Chord [11] and RccJava [12]. By processing the outputs of Chord and RccJava we identify a list of fields, classes, and methods that will never be involved in a race condition, and using this information, we enable/disable race checking on the particular accesses. With this pre-elimination, the overheads we obtained on Java benchmarks are competitive with or better than other dynamic race detection algorithms in the literature. These results indicate that the precision of our algorithm does not come at a performance cost. For many benchmark programs, we found the overhead of precise race detection with our approach to be low enough to be possibly tolerable even for deployed code. In other

benchmarks and for programs in which performance is of utmost importance, the safety-performance trade-off provided by `DataRaceException` may not be acceptable. For these programs, our Java runtime can serve as a debugging tool that produces no false alarms.

1.1 Contributions and Outline of Part I

The rest of this part of the thesis is organized as follows:

- Chapter 2 gives the technical background and a survey of existing techniques in dynamic race detection.
- Chapters 3–7 present our contributions in detail.
- Chapter 8 concludes this part of the thesis and provides future research directions.

This thesis presents a novel approach to detecting data-race conditions in Java programs and makes the following technical contributions explained throughout Chapters 3–7.

1. **Runtime exception for data races:** Our Java runtime provides a new runtime exception, `DataRaceException`, that is thrown precisely when a racy access is about to occur. This brings races to the awareness of the programmer and allows him to explicitly handle them.
2. **General, dynamic data-race detection algorithm:** We have developed a novel lockset-based dynamic race detection algorithm called `GOLDILOCKS`. `GOLDILOCKS`, unlike previous variants of lockset algorithms, can uniformly and naturally handle all synchronization idioms such as thread-local variables that later become shared, shared variables that are protected by different locks at different points in time and/or protected indirectly by locks on container objects. In Chapter 3, we first introduce the mathematical model of a generic memory model with the definitions of the happens-before relationship and data races, and then we present the general form of the `GOLDILOCKS` algorithm on this generic model. We show that `GOLDILOCKS` is both sound and precise. Thus, it exactly captures the happens-before relationship defined in our generic memory model (and in the JMM).

3. **Specializing GOLDDILOCKS to the Java Memory Model (JMM):** In Chapter 4, we describe the first precise, purely lockset-based data race detection algorithm for Java. We specialize the general GOLDDILOCKS algorithm to the JMM and show that the generic algorithm can easily be adapted to handle all synchronization primitives in Java.
4. **Specializing GOLDDILOCKS to Software Transactional Memory (STM):** In Chapter 5, we present the first formalization of race conditions for programs that use software transactions along with other Java synchronization primitives. We describe the specialization of the general GOLDDILOCKS algorithm to STM [10] so that GOLDDILOCKS explicitly handles software transactions as a high-level synchronization idiom.
5. **Implementation of GOLDDILOCKS in a real JVM:** We developed implementation features and optimization techniques to make GOLDDILOCKS efficiently implementable, and using them, carried out an implementation of the GOLDDILOCKS algorithm inside the Kaffe JVM. Chapter 6 elaborates on our implementation of the GOLDDILOCKS algorithm in Kaffe. Our implementation incorporates two techniques that significantly enhance its performance: (1) partially-lazy evaluation and implicit representation of Goldilocksets, and (2) a sequence of cheap and sufficient checks for race freedom which allow bypassing of the Goldilockset computation. To reduce the runtime overhead of race detection, we apply existing sound static race analysis tools beforehand to determine and record in Java class files the variables and accesses that are guaranteed to be race free. Our implementation explicitly handles software transactions.
6. **Experience with GOLDDILOCKS:** Using our implementation of GOLDDILOCKS in the Kaffe JVM, we showed that the implementation features and optimizations combined with the capability to skip race-free checks using static pre-analysis in a sound manner result in a precise dynamic race checker whose performance on benchmarks is competitive with or, in some cases, significantly better than existing dynamic race checkers for Java programs. Chapter 7 presents results of two experiments using our implementation. We first give a comparison of an earlier implementation of our algorithm with the prototype implementations of lockset and vector-clock based algorithms, all in the

same framework, and show that GOLDBLOCKS is competitive with other algorithms. Second, we report on our evaluation of the latest version of GOLDBLOCKS with all the optimizations including the static pre-analysis.

Chapter 2

THE STATE OF THE ART IN DYNAMIC DATA-RACE DETECTION

In this chapter, we will discuss the related work in dynamic race detection and related topics such as the effect of races on the semantics of the program.

Outline. Section 2.1 explains the relationship between memory models and race-free and racy programs. Section 2.2 gives information on the state of the art dynamic data-race detection algorithms.

2.1 Memory Models and Data Races

A memory model, in conjunction with the intra-thread sequential semantics of the programming language, specifies the semantics of a multithreaded program, i.e., the set of executions the program may produce at runtime. In particular, the memory model restricts the execution ordering among the actions and the possible outcomes of actions over the variables. Sequential consistency (SC) [13] is the most straightforward memory model that allows programmers to use an interleaving model of execution. In this model, accesses from different threads are interleaved into a total order respecting the intra-thread ordering (also called program order) of variable accesses, and every read sees the value of the most recent write in the total order. Sequential consistency is widely considered to be the only simple-enough model using which writing useful concurrent programs is practical.

The Java Memory Model (JMM) [2] and the C++ Memory Models (C++MM) [3] are *relaxed* memory models, since they do not provide the strong guarantees by SC. However, both JMM and C++MM provide the data-race freedom (DRF) property. A memory model has this property iff the following hold: (1) If an execution is race free, then it is guaranteed to be sequentially consistent, and (2) if all sequentially consistent executions of a program are race free, then the program only exhibits these sequentially consistent executions. The GOLDBLOCKS runtime makes use of (1) and the `DataRaceException` to guarantee for all

programs (whether racy or not) that every concurrent execution is sequentially consistent at the byte-code level¹. This guarantee does not restrict the GOLDBLOCKS runtime’s use as a debugging tool, since the JMM and C++MM ensure that if a program has a racy execution, then it will have at least one execution that is sequentially consistent and racy. Thus, it is sufficient to restrict one’s attention to looking for races in sequentially-consistent executions only.

The Java Memory Model (JMM) [2] precisely defines the semantics for executions including races. We view the primary purpose of this to be specifying constraints for Java compilers and virtual machines in order to provide sanity and security guarantees. For example, the JMM prohibits a program from speculating an “out-of-air” value, a value that will never be generated by a sequentially-consistent execution of the program. The semantics of Java executions with races are very difficult to make use of. The rules in the JMM for defining acceptable executions of a program are rather complicated to interpret and this makes it a challenge to reason about a program’s possible runtime behaviors.

The JMM defines well-synchronized programs to be free of data races. We expect that programmers will strive to ensure that their programs are race free and want to operate under sequential consistency semantics. The Java runtime presented in this thesis raises a `DataRaceException` when an actual (not potential) race condition is about to occur. This provides a mechanism for the program to always operate under sequentially consistent semantics. In this view, an actual race condition is analogous to an out-of-bounds array access or a null pointer dereference. If the programmer has provided exception handling code, the program continues to execute with sequentially consistent semantics.

While the JMM and C++MM give similar guarantees for race-free programs, the latter leaves the semantics of racy programs completely undefined, as defining behavior of badly-synchronized programs results in overly complicated semantics [3]. A C++ compiler and runtime complying with the C++MM may handle a program assuming that the program is race-free, i.e., all the data accesses in the program are separated by synchronization operations. This allows them to perform a set of optimizations, e.g., reordering data variable accesses within a code block without synchronization operations or reordering a data access

¹For a discussion of sequential consistency at the source code vs. the byte-code levels, and the requirements on Java compilers and runtimes in order to provide source-code-level sequential consistency, see Section 4.6.

with a particular kind of synchronization operation, as allowed by the C++MM. For a race-free program, these optimizations are shown to not change the runtime semantics of the program. However, these optimizations modify a racy program so that it may exhibit unexpected behaviors. Moreover, it is argued in [3] that the result of these transformations on badly-synchronized code blocks is rather complex to anticipate. In conclusion, one can never reason about the runtime behavior of a C++ program if it contains a data race.

Since proposed first by this work in [4], the idea that programming platforms should be able to guarantee sequential consistency for *all* programs has gained significant support. The studies by Marino et al. [5] and Lucia et al. [6] present memory models with more conservative and simpler contracts than the JMM and C++MM. It was demonstrated that these contracts can be implemented in hardware, and thus, be checked efficiently. These models provide explicit memory model exceptions for the cases when these contracts are violated by the program. Their approach can be seen as defining and checking a stronger notion of happens-before and races at the hardware level, so that an execution that is declared racy by JMM and C++MM may still be a sequentially consistent execution of the program. In this setting, a data race happens only when two conflicting accesses are concurrent in real time, in which case a hardware exception is thrown and the program is terminated to avoid unanticipated behaviors. The experiments in [5] and [6] using hardware-based monitoring of the execution clearly indicate that hardware support is necessary to reduce overhead of data race checking to levels acceptable for deployed code.

2.2 Dynamic Race Detection Algorithms and Tools

There are two classes of approaches to dynamic data-race detection, one based on locksets and the other based on the happens-before relation. Eraser [14] is a well-known lockset-based algorithm for detecting race conditions dynamically by enforcing the locking discipline that every shared variable is protected by a unique lock. In spite of the numerous papers that refined the basic Eraser algorithm to reduce the number of false alarms [14, 15, 16, 17], there are still cases, such as dynamically changing locksets, that cannot be handled precisely.

To support `DataRaceException`, a precise and efficient dynamic race analysis is needed. Purely vector clock-based algorithms are precise but typically computationally expensive [18]. Lockset-based algorithms are efficient, but not precise. They check adherence to a

```

Class IntBox {  int data; }
IntBox a, b; // Global variables

Thread 1:          Thread 2:          Thread 3:
-----
tmp1 = new IntBox();  acq(ma);          acq(mb);
tmp1.data = 0;        tmp2 = a;          b.data = 2;
                     rel(ma);        tmp3 = b;
                                         rel(mb);

acq(ma)
a = tmp1;            acq(mb);
rel(ma);             b = tmp2;        tmp3.data = 3;
                     rel(mb);

```

Figure 2.1: Example demonstrating the need for precise data-race detection

particular synchronization discipline and declare false races when this discipline is violated. For example, the program given in Figure 2.1 makes use of two idioms typically not handled by earlier lockset algorithms: objects that are local to different threads at different points in the execution, and objects protected by locks of container objects. The Goldilocks race detection algorithm declares no false alarms in this example while other lockset-based algorithms do.

Consider an execution of the program shown in Figure 2.1 in which all actions of T1 happen first, followed by all actions of T2 and then of T3. In the example, certain null pointer checks are not shown for simplicity. The reader may assume that this ordering of actions is the one intended by the programmer and other potential orderings are in fact prevented by proper control statements in the actual code. This example mimics a scenario in which an object is created and initialized and then made globally visible by T1. This `Int` object (referred to as *o* from now on) is a container object for its `data` field (referred to as *x* from now on). The object *o* is referred to by different global variables (`a` and `b`) and local variables (`tmp1`, `tmp2`, and `tmp3`) at different points in this execution. The contained variable *x* is protected by synchronization on the container object *o*, and during the execution, the lock (`La` or `Lb`) protecting *o* and *x* changes depending on which variable (`a` or `b`) points to *o*. Notice that, T2 changes the protecting lock of the container object *o* from `La` to `Lb`, without

accessing x .

The most straightforward lockset algorithm is based on the assumption that each shared variable is protected by a fixed set of locks throughout the execution. To detect a violation of this assumption, the lockset algorithm maintains for each variable x , a lockset $LS(x)$ initialized to the set of all locks in the program. It also maintains for each thread t , a lockset $LH(t)$ representing the set of locks held by thread t at a given point in an execution. This algorithm attempts to infer this set by updating the lockset $LS(x)$ of a data variable x to be the intersection $LH(t) \cap LS(x)$ at each access to x by a thread t . If this intersection becomes empty, this means that x has not been consistently protected by the same lock, thus a race is reported. This approach is too conservative since it reports a false race in many situations, such as during unprotected variable initialization and when the lock protecting a variable changes over time. For instance, this basic algorithm will report a data race on x at the very first access (`tmp1.data = 0`) to x by T1 in Figure 2.1.

Variants of lockset algorithms in the literature use additional mechanisms such as a state machine per shared variable in order to handle special cases such as thread locality and object initialization patterns [14, 15, 16]. However, these variants are neither sound nor precise, and they all report false alarms in scenarios similar to the one in the example above. For instance, in spite of using the state-machine accompanying the Eraser algorithm [14], a data-race will be reported at the last access (`tmp3.data = 3`) to x by T3 in the execution.

There is much work that refines the Eraser algorithm by improving the state machine by adding extra states and transitions to reduce the number of false positives. One such refinement is extending the state-based handling of object initialization and making use of object-oriented concepts [15]. Harrow used *thread segments* to identify the portions of the execution in which objects are accessed concurrently among threads [19]. Another approach is using a basic vector clock algorithm to capture thread-local accesses to objects, and thus, eliminates unnecessary and imprecise applications of the Eraser algorithm [16]. Precise lockset algorithms exist for Cilk programs [20] but they cannot handle concurrency patterns implemented using volatile variables such as barrier synchronization, thus their use for real programs is still under question. The general algorithm in [20] is quite inefficient while the efficient version of this algorithm requires programs to obey the *umbrella locking discipline*, which can be violated by some race-free programs.

The fundamental limitation of existing lockset algorithms is that the lockset of a variable only becomes smaller with time. On the other hand, the GOLDLOCKS algorithm's update rules allow a variable's Goldilocksets to grow during the execution; in fact, the Goldilockset of a variable may be modified even without the variable being accessed. The ability to grow the Goldilockset of x enables GOLDLOCKS to represent threads transferring ownership using different synchronization variables during the execution. These aspects of our algorithm are crucial for detecting changes of ownership during the execution. In Section 4.3 we will give the evaluation of $GLS(x)$, the Goldilockset of x , for every point during the execution of Figure 2.1 described above and show how $GLS(x)$ correctly indicate the absence of a race in this example.

There is a significant body of research on dynamic data-race detection based on computing the happens-before relation [17, 21, 22, 16, 8, 9] using vector clocks [18]. In this approach, every thread and synchronization variable is assigned a vector of logical clocks. A vector clock is a map from thread ids to non-negative integers. The vector clock of a thread t , denoted $VC(t)$, stores its logical clock and that of every other thread. The vector clock of a synchronization object (for example, a mutex lock) o , denoted $VC(o)$, stores the logical clock of every thread last observed by t . These vector clocks are updated when a thread performs a synchronization operation or a data access happens. For example, when a thread t acquires a lock l , $VC(t)$ is updated to $VC(t) \sqcup VC(l)$. The $\sqcup$ operation creates a new vector clock such that each thread u is mapped to the maximum of $VC(t)[u]$ and $VC(l)[u]$. Thus, the lock acquire allows t to observe the logical clock of other threads that used the lock l before. When a thread t release a lock l , $VC(l)$ is updated to $VC(t)$ and $VC(t)$ is updated by incrementing $VC(t)[t]$ by one. Thus, the lock release allows the lock l to observe and carry (to later acquires of l) the logical clock of t . When a thread t accesses a variable x , its vector clock $VC(t)[t]$ must be greater than the vector clock of the last accessor thread, say u , at the point of the last access. This indicates that u and t synchronized on some lock l between the last and the current accesses.

The vector clock algorithm is both sound and precise, thus does not declare any false alarms for the example above and similar scenarios. Tracking the happens-before relationship allows the algorithm to handle various synchronization idioms in which the unique lockset assumption is broken, and thus, cannot be captured by the Eraser-based algorithms. For

example, in the following execution, the variable `x` is not protected by any lock, but the two accesses to `x` by threads `T1` and `T2` are separated by happens before due to the locks `L`. Thus, there is no data race in this example.

```

T1
-----
x := 0;          // VC(T1)=[*T1:0, T2:0] VC(lock)=[T1:0, T2:0]
acquire(lock);  // VC(T1)=[T1:0, T2:0] VC(lock)=[T1:0, T2:0]
...
release(lock);  // VC(T1)=[T1:1, T2:0] VC(lock)=[T1:0, T2:0]

T2
-----
acquire(lock);  // VC(T2)=[T1:0, T2:0] VC(lock)=[T1:0, T2:0]
...
release(lock);  // VC(T2)=[T1:0, T2:1] VC(lock)=[T1:0, T2:0]
x := 1;         // VC(T2)=[T1:0, *T2:1] VC(lock)=[T1:0, T2:0]

```

The code above shows the evolution of the vector clocks of the threads; `*` indicates the vector clocks used in the comparison at the second access point. The updates of `VC(T1)` and `VC(T2)` at the acquire and release points between the two accesses allow the vector clock of `T2` at the second access point to be greater than that of `T1` at the first access point. The acquire and release operations make it explicit that the accesses are separated in time by proper synchronization.

The Trade tool [17] uses a vector clock algorithm implemented at the Java byte-code level and in interpreter mode of JVM as in our implementation of GOLDLOCKS. To reduce the overhead of the vector clocks for programs with a large number of threads, Trade uses object reachability information (which other threads can reach a particular object by following pointers) among the threads in the execution, which makes it more efficient than other similar tools. Schonberg computes for each thread shared variable sets and concurrency lists to capture the set of shared variables between synchronization points of an execution [22]. His algorithm is imprecise for synchronization disciplines that use locks and needs to be extended for asynchronous coordination to achieve precision for these disciplines.

Although the vector clock algorithm is precise, it accomplishes this at significantly in-

creased computational cost compared to our optimized implementation of the Goldilockset update rules. The vector clock algorithm, for an execution with n threads, requires for every thread and synchronization variable a separate vector clock (VC) of size n . At every synchronization operation and data variable access, it performs an $O(n)$ operation to update the vector clocks or compare two vector clocks to check races. In preliminary research, compared to a straightforward implementation of vector clocks, we found GOLDBLOCKS overhead to be significantly lower (see Section 7.1 for details). Another strength of GOLDBLOCKS compared to the vector clock algorithm is the intuitive appeal of a Goldilockset: The Goldilockset of a variable x contains the threads that can access x without causing a race condition, and the synchronization variables on which a thread can perform a synchronization operation in order to gain race-free access to x . The analysis needed to capture this information from vector clocks is quite sophisticated.

Lockset algorithms are sound and efficient, but not precise, and vector clock algorithms are sound and precise, but not efficient. In order to benefit from the advantages of both kind of algorithms, many recent studies have combined the lockset and happens-before analysis into hybrid techniques [16, 8, 9]. For example, RaceTrack [16] uses a basic vector clock algorithm to capture thread-local accesses to objects thereby eliminating unnecessary and imprecise applications of the Eraser algorithm. Similarly, MultiRace [8] presents DJIT⁺, a vector clock algorithm with several optimizations to reduce the number of checks at an access, including keeping distinct vector clocks for reads and writes and using a lockset algorithm as a fast-path check. To the best of our knowledge, FASTTRACK [9], which builds on DJIT⁺, is the best-performing vector clock based algorithm in the literature. By exploiting some access patterns, FASTTRACK reduces the cost of vector clock updates to $O(1)$ on average. Vector clock and GOLDBLOCKS are both precise, but the generalized locksets in GOLDBLOCKS provide an intuitive representation of how shared variables are protected at each point in the execution. In addition, GOLDBLOCKS, for the first time, computes a precise happens-before relation using an implementation that makes use of only locksets. In Section 7.3, we provide a qualitative comparison of the GOLDBLOCKS and FASTTRACK algorithms.

The use of static pre-analysis to eliminate unnecessary runtime race checks for well-protected variables has also been studied. Choi et al. [23] and Praun et al. [15], which focused more on object-level races rather than single field level races, showed that the result

of a static pre-analysis can successfully reduce the runtime overhead of race detection.

In this chapter, we overviewed the relationship between memory models and semantics of data races, and gave pointers to studies that offer similar exceptions to `DataRaceException`. We also summarized the state of the art in dynamic data-race detection algorithms and techniques. We will continue comparing the existing work with our algorithm by examples and experiments in later chapters.

Chapter 3

THE GOLDSLOCKS ALGORITHM

In this chapter, we present a generic memory model and describe the general version of the GOLDSLOCKS algorithm for the data race definition in this model. This generic model allows a uniform treatment of the various synchronization constructs in Java. We also believe that memory models at different levels (e.g., the hardware level) and for different languages (e.g., C++MM) can be expressed as instances of this model. This allows GOLDSLOCKS to be applied in these settings directly.

Outline. Section 3.1 defines the main elements of a generic memory model for data-race checking. Section 3.2 presents the basic GOLDSLOCKS algorithm without distinguishing reads and writes. In Section 3.3, we argue that GOLDSLOCKS is a both precise and sound algorithm for data race checking. Section 3.4 extends the basic algorithm by distinguishing reads and writes.

3.1 A Generic Memory Model

A memory model, in conjunction with the intra-thread sequential semantics of the programming language, specifies the semantics of a multithreaded program, i.e., the set of executions the program may produce at runtime. A multithreaded execution of the program consists of a set of operations over the program variables. In particular, the memory model restricts the execution ordering among the actions and the possible outcomes of actions over the variables. In this section, we describe the kinds of variables and operations, formalize executions and indicate the components of the execution determined by the memory model, and define data races. The next section presents the GOLDSLOCKS algorithm in this generic memory model and describes how the algorithm is specialized to JMM.

3.1.1 Variables and Actions

Program variables are separated into two categories: data variables (*Data*) and synchronization variables (*Sync*). We use x , and o to refer to data and synchronization variables, respectively. Although other kinds of actions (such as arithmetic computation, function calls, etc.) also occur in a real execution of a Java program, these actions are irrelevant for our exposition of race conditions and have consequently been elided. Threads in a program execute actions from the following categories:

- *Data variable accesses*: $\text{read}(t, x, v)$ by thread t reads the current value v of a data variable x , and $\text{write}(t, x, v)$ by thread t writes the value v to x . When referring to an action, we sometimes omit the thread id t and/or value v if they are of no interest, and in some case, we only name its kind when the rest of the information is clear from the context.
- *Synchronization operations*: When threads synchronize using a synchronization mechanism, a thread t_i executes a notification action, which is then observed by other threads t_j . Such a notification-observation pair defines a “synchronizes-with” edge from the former action to the latter. We classify actions that serve as sources and sinks of a synchronizes-with edge as synchronization source and sink actions, respectively.
 - *Synchronization source actions*: $\text{sync-source}(t, o)$ by thread t creates a synchronizes-with source by writing to a synchronization variable o . Lock releases and volatile variable writes in Java are synchronization source actions.
 - *Synchronization sink actions*: $\text{sync-sink}(t, o)$ by thread t creates a synchronizes-with sink by reading from a synchronization variable o . Lock acquires and volatile variable reads in Java are synchronization sink actions.

3.1.2 Multithreaded Executions

Given a concurrent program, the memory model specifies the set of executions that may be generated by program. An execution E is represented by a tuple $E = \langle \text{ThreadId}, \text{Act}, W, \xrightarrow{po}, \xrightarrow{so} \rangle$.

- Tid is the set of identifiers for threads involved in the execution. Each newly-forked thread is given a new unique id from Tid .
- Act is the set of actions that occur in this execution. $Act|_t$ is the set of actions performed by $t \in Tid$, and $Act|_x$ (resp. $Act|_o$) are the sets of actions performed on data variable x (resp. synchronization variable o).
- W is a total function that maps each $read(t, x, v)$ in Act to a $write(u, x, v)$ in Act . W is used to model the write-seen relationship between a read of x and the write to x it sees. In order to make the function W total, we assume an initial write for each variable before any reads.
- $\xrightarrow{po}_t$ is the program order per thread t . For each thread t , $\xrightarrow{po}_t$ is a total order over $Act|_t$ and gives in which order the actions were issued to execute. This order is sometimes referred to as the observed execution order.
- $\xrightarrow{so}_o$ is the synchronization order per synchronization variable $o \in Sync$. For each $o \in Sync$, $\xrightarrow{so}_o$ is a total order over $Act|_o$.

3.1.3 Synchronizes-with and Happens-before

Given an execution with program and synchronization orders, we extract two additional orders called the synchronizes-with ($\xrightarrow{sw}$) and happens-before ($\xrightarrow{hb}$) orders. Data races are defined using these orders, and dynamic data-race detection algorithms compute these orders through an online examination of the execution.

A synchronization operation α_1 by thread t_1 *synchronizes with* α_2 by thread t_2 , denoted $\alpha_1 \xrightarrow{sw} \alpha_2$, if α_1 is a **sync-source** on some synchronization variable o , α_2 is a **sync-sink** on o , and $\alpha_1 \xrightarrow{so}_o \alpha_2$.

The *happens-before* partial order $\xrightarrow{hb}$ on the execution E is defined as the transitive closure of the program orders $\xrightarrow{po}_t$ for all $t \in Tid$ and the synchronizes-with order $\xrightarrow{sw}$.

In this work, we focus only on *well-formed* executions as defined in [2]. By definition, every well-formed execution respects (i) the intra-thread semantics and (ii) the semantics of the synchronization variables and operations. In addition, well-formed executions satisfy two essential requirements for data-race detection:

- *Happens-before consistency*: This property makes use of the happens-before order to restrict the write-seen relationship. For example, for a read action α , $\alpha \xrightarrow{hb} W(\alpha)$ cannot happen, and $W(\alpha)$ cannot be overwritten by another write action β such that $W(\alpha) \xrightarrow{hb} \beta \xrightarrow{hb} \alpha$.
- The union of the program orders for all $t \in Tid$ and the synchronization orders for all variables $o \in Sync$ is a valid partial order. During an execution, our data race-detection algorithm examines a linearization of this partial order and identifies the happens-before edges between data accesses.

3.1.4 Sequential Consistency

Sequential consistency is a property that allows programmers to use an interleaving model of execution. In this model, accesses from different threads are interleaved into a total order, and every read sees the value of the most recent write in this total order. Sequential consistency is widely considered to be the only simple-enough model using which writing useful concurrent programs is practical. Formally, an execution $E = \langle Tid, A, W, \xrightarrow{po}, \xrightarrow{so} \rangle$ is *sequentially consistent* if there exists a total order $\xrightarrow{SC}$ over Act satisfying the following:

- For every thread $t \in Tid$, $\xrightarrow{SC}$ respects the program order $\xrightarrow{po}_t$, i.e., $\xrightarrow{po}_t \subseteq \xrightarrow{SC}$.
- Every read $\alpha = \text{read}(x)$ sees the most recent write to x in $\xrightarrow{SC}$, i.e., there is no other $\beta = \text{write}(x)$ such that $W(\alpha) \xrightarrow{SC} \beta \xrightarrow{SC} \alpha$.

3.1.5 Data Races

Two data variable accesses are called *conflicting* if they refer to the same shared data variable and at least one of them is a write access.

One frequently-used definition of a race condition involves a program state in which two conflicting accesses by two different threads to a shared data variable are simultaneously enabled. To distinguish this definition from others, let us refer to this condition as a *simultaneity race*. The definition of a race condition used in most work on dynamic race detection is what we call a *happens-before race* and involves two conflicting accesses not ordered by the happens before relationship, i.e., not separated by proper synchronization operations.

For C++, these two definitions of a race condition have been shown to be equivalent [3]. In essence, this proof also generalizes to Java executions.¹

Formally, an execution $E = \langle Tid, Act, W, \xrightarrow{po}, \xrightarrow{so} \rangle$ contains a happens-before race if there are two conflicting actions, $\alpha, \beta \in Act|_x$ accessing a data variable x , such that neither $\alpha \xrightarrow{hb} \beta$ nor $\beta \xrightarrow{hb} \alpha$ holds. Conversely, the execution is *race free* if every pair of conflicting accesses to a data variable are ordered by happens-before.

The well-formedness of an execution guarantees that if the execution has no race conditions, then it is sequentially consistent [2]. The GOLDBLOCKS runtime makes use of this and the `DataRaceException` to guarantee for all programs (whether racy or not) that every concurrent execution is sequentially consistent at the byte-code level. In Section 4.6, we give a discussion of sequential consistency at the source code vs. the byte-code levels, and the requirements on Java compilers and runtimes in order to provide source-code-level sequential consistency. Enforcing sequential consistency does not restrict the GOLDBLOCKS runtime's use as a debugging tool for catching races. This is because, for the Java and C++ memory models, it has been proven [2, 3] that if a program has a racy execution, then it is guaranteed to have at least one execution that is both sequentially consistent and racy. Thus, it is sufficient to focus on looking for races in sequentially-consistent executions only.

3.2 The Generalized Goldilocks Algorithm

In this section, we describe our algorithm for detecting data races in an execution $E = \langle Tid, Act, W, \xrightarrow{po}, \xrightarrow{so} \rangle$. For simplicity of exposition, we initially do not distinguish between read and write accesses. In Section 3.4 we extend the algorithm to make this distinction.

The GOLDBLOCKS algorithm processes the actions in Act one at a time, as a sequence. Before a thread t performs an action α in Act , t notifies the GOLDBLOCKS algorithm that α is about to occur. The order in which these notifications from different threads are interleaved and processed by GOLDBLOCKS is represented mathematically by π , where $\pi(i)$ is the i -th action in the sequence. This linear order, by construction, respects the program order for each thread, and the synchronization total order for each synchronization variable. A racy read may appear earlier in π than the write that it sees. If an execution contains a data race

¹The proof relies on the fact that the semantics of `tryLock()` allows spurious failures, i.e., it may fail even though the lock is available.

between a pair of accesses, Goldilocks declares a race at one of these accesses regardless of which linearization π is picked.

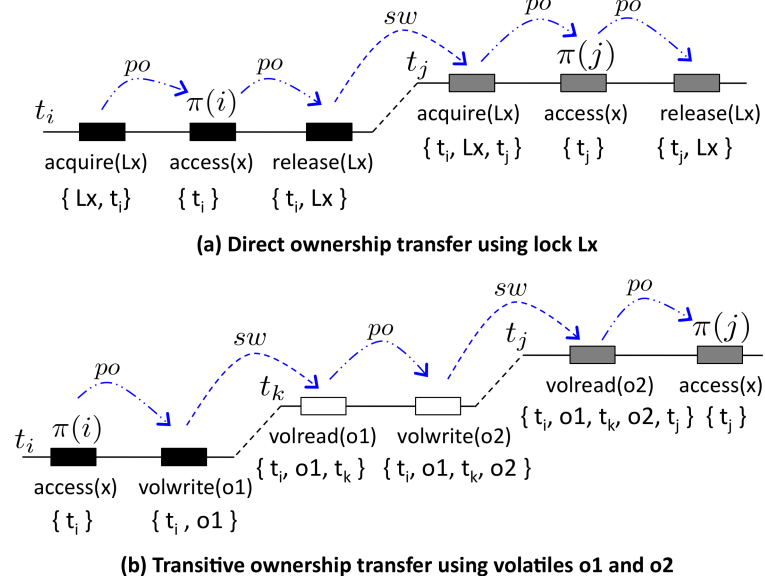


Figure 3.1: Transferring ownership of x , and $GLS(x)$

The GOLDILOCKS algorithm maintains for each data variable a “Goldilockset”: a map GLS such that, for every data variable x , its Goldilockset is a set $GLS(x) \subseteq Tid \cup Sync$. Roughly speaking, $GLS_i(x)$, the Goldilockset of x immediately before processing action $\pi(i)$, consists of (i) the id’s of threads that can access x in a race-free way at that point in the execution, and (ii) the synchronization variables on which a thread can perform a **sync-sink** access in order to gain race-free access to x .

As GOLDILOCKS processes each action $\pi(i)$ from E , it updates the Goldilocksets of variables. Initially, the Goldilockset $GLS(x)$ is empty for all data variables, including static ones. When a new object is created, the Goldilockset for all of its instance fields are initialized to the empty set. After every action, the Goldilockset of every data variable x is potentially updated. For every data variable x , three simple rules specify how $GLS(x)$ is updated after $\pi(i)$ based on whether $\pi(i)$ is (1) a synchronization source, (2) a synchronization sink, or (3) a read or write access to x , as shown in the procedure *ApplyLocksetRules* in Figure 3.2.

If the action $\pi(i)$ is a synchronization operation on a variable o , we update the Goldilock-

```

ApplyLocksetRules( $\pi(i)$ ):

1. if  $\pi(i) = \text{sync-source}(t, o)$ 
    foreach  $x \in \text{Data}$ :
        if  $t \in GLS(x)$ 
             $GLS(x) := GLS(x) \cup \{o\}$ 

2. if  $\pi(i) = \text{sync-sink}(t, o)$ 
    foreach  $x \in \text{Data}$ :
        if  $o \in GLS(x)$ 
             $GLS(x) := GLS(x) \cup \{t\}$ 

3. if  $\pi(i) = \text{write}(t, x)$  or  $\pi(i) = \text{read}(t, x)$ 
    if  $t \in GLS(x)$  or  $GLS(x) = \emptyset$ 
         $GLS(x) := \{t\}$ 
    else
        throw a DataRaceException on  $x$ 

```

Figure 3.2: The core Goldilockset update rules

set $GLS(x)$ for every data variable x in Data . If $\pi(i)$ is a **sync-source** operation, rule 1 adds o to $GLS(x)$ if it contains the id t of the currently accessing thread. Intuitively, this represents that a later **sync-sink** operation by a thread u on synchronization variable o will be sufficient for u to gain race-free access to x . This is formalized by rule 2. If $\pi(i)$ is a **sync-sink**(o) operation, rule 2 checks whether the synchronization variable o is in $GLS(x)$. If this is the case, then t is added to the Goldilockset.

If the action $\pi(i)$ is an access to a data variable x , rule 3 checks the Goldilockset of the variable $GLS(x)$ to decide whether this access is race free or not. If $GLS(x)$ is empty, it indicates that x is a fresh variable which has not been accessed so far and any access to x at this point is race free. If $GLS(x)$ is not empty, only threads whose id's are in $GLS(x)$ can perform race-free accesses to x . If the accessing thread's id t is not in $GLS(x)$ then we throw a **DataRaceException** on x . Otherwise, the access to x is race free and $GLS(x)$

becomes the singleton $\{t\}$, indicating that, without further synchronization operations, only t can access x in a race free manner.

Figure 3.1 shows two cases where the ownership of a data variable x is transferred from a thread t_i to another thread t_j , and indicates how the Goldilocksets evolve in each case. Program order ($\xrightarrow{po}$) and synchronizes-with ($\xrightarrow{sw}$) edges between consecutive actions are indicated in the figure. Fig. 3.1 (a) illustrates direct ownership transfer from t_i to t_j . After accessing x , t_i performs a **sync-source** operation (lock release) on synchronization variable **Lx**. Later, t_j obtains ownership of x by executing a **sync-sink** operation (lock acquire) on synchronization variable **Lx**. Fig. 3.1 (b) illustrates transitive ownership transfer. Threads t_i and t_j do not synchronize on the same synchronization variable. Instead, the synchronization involves a chain of synchronizes-with edges between other threads and synchronization variables. t_i synchronizes with t_k via synchronization variable **o1** and, later t_k synchronizes with t_j via synchronization variable **o2**.

Rules 1 and 2 in Figure 3.2 require updating the Goldilockset of each data variable. A naive implementation of this algorithm would be too expensive for programs that manipulate large heaps. In Chapter 6, we present an efficient way of implementing our algorithm by representing Goldilocksets implicitly and by applying update rules lazily.

3.3 Correctness of the Goldilocks Algorithm

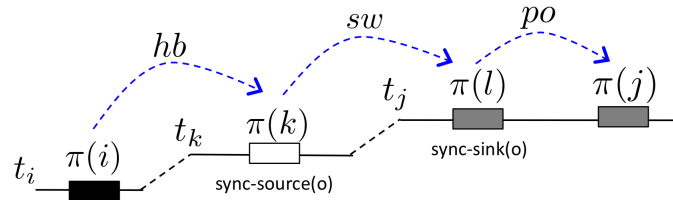


Figure 3.3: Interpretation of $\pi(i) \xrightarrow{hb} \pi(j)$ and $t_i \neq t_j$ for Lemma 1.

Definition 1 (Immediate happens-before edges):

Let $E = \langle Tid, Act, W, \xrightarrow{po}, \xrightarrow{so} \rangle$ be a well-formed execution and π a linear order on Act that respects $\xrightarrow{po}_t$ for all threads $t \in Tid$ and $\xrightarrow{so}_o$ for all synchronization objects $o \in Data$. The immediate happens-before ordering, denoted $\xrightarrow{ihb}$, is a partial order over Act such that

for every pair of actions (α, β) in Act , $\alpha \xrightarrow{ihb} \beta$ holds iff either $\alpha \xrightarrow{po}_t \beta$ for some thread $t \in Tid$ or $\alpha \xrightarrow{so}_o \beta$ for some synchronization object $o \in Data$. ■

Lemma 1 (Immediate happens-before):

Let $E = \langle Tid, Act, W, \xrightarrow{po}, \xrightarrow{so} \rangle$ be a well-formed execution and π a linear order on Act that respects $\xrightarrow{po}_t$ for all threads $t \in Tid$ and $\xrightarrow{so}_o$ for all synchronization objects $o \in Data$.

Let $i < j$, and $\pi(i)$ and $\pi(j)$ be two actions performed by threads t_i and t_j , respectively.

$\pi(i) \xrightarrow{hb} \pi(j)$ iff either $t_i = t_j$ or $t_i \neq t_j$ and there exists k, l such that $i < k < l \leq j$, $\pi(i) \xrightarrow{hb} \pi(k)$, $\pi(k) = \text{sync-source}(t_k, o)$ and $\pi(l) = \text{sync-sink}(t_j, o)$ for some synchronization variable o . ■

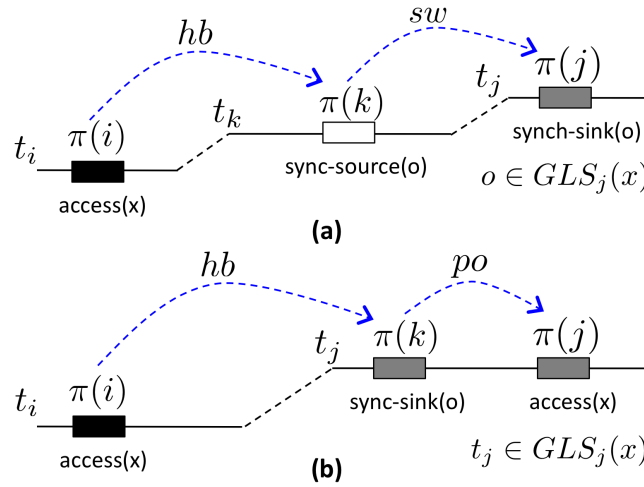
The scenario formalized in Lemma 1 is illustrated in Figure 3.3.

PROOF.

- (1)1. $\implies$: Assume $\pi(i) \xrightarrow{hb} \pi(j)$ and prove that $t_i = t_j$ or $t_i \neq t_j$ and there exists k, l such that $i < k < l \leq j$, $\pi(i) \xrightarrow{hb} \pi(k)$, $\pi(k) = \text{sync-source}(t_k, o)$ and $\pi(l) = \text{sync-sink}(t_j, o)$ for some synchronization variable o .
- (2)1. By the definition of the happens-before relation, there is a chain of edges $\alpha_1 \xrightarrow{ihb} \alpha_2 \xrightarrow{ihb} \dots \xrightarrow{ihb} \alpha_n$ such that $\alpha_1 = \pi(i)$, $\alpha_n = \pi(j)$. For any pair of actions α_k, α_l in this chain such that $1 \leq k \leq l \leq n$, $\alpha_k \xrightarrow{hb} \alpha_l$ holds.
- (2)2. Let $\alpha_1 \xrightarrow{ihb} \alpha_2 \xrightarrow{ihb} \dots \xrightarrow{ihb} \alpha_n$ be the shortest chain between $\pi(i)$ and $\pi(j)$, and assume that $t_i \neq t_j$ (otherwise the shortest chain is $\pi(i) \xrightarrow{po}_{t_j} \pi(j)$). Let α_m be the first action in this chain executed by t_j , and consider the edge $\alpha_{m-1} \xrightarrow{ihb} \alpha_m$. We do a case analysis on the type of this edge:
- (3)1. $\alpha_{m-1} \xrightarrow{po}_{t_j} \alpha_m$: This cannot happen, since we assumed that α_m is the first action in the chain executed by t_j . Thus α_{m-1} and α_m are by different threads.
- (3)2. $\alpha_{m-1} \xrightarrow{sw}_o \alpha_m$ for some synchronization variable o : In this case, α_{m-1} is a **sync-source** operation on o and α_m is a **sync-sink** operation on o . Taking $\pi(k) = \alpha_{m-1}$ and $\pi(l) = \alpha_m$, $\pi(i) \xrightarrow{hb} \pi(k)$ holds because of the chain of immediate happens-before edges $\pi(i) = \alpha_1 \xrightarrow{ihb} \dots \xrightarrow{ihb} \alpha_{m-1} = \pi(k)$; this proves the claim.
- (1)2. $\impliedby$: Assume that either $t_i = t_j$ or $t_i \neq t_j$ and there exists k, l such that $i < k < l \leq j$, $\pi(i) \xrightarrow{hb} \pi(k)$, $\pi(k) = \text{sync-source}(t_k, o)$ and $\pi(l) = \text{sync-sink}(t_j, o)$ for some synchronization variable o , and prove that $\pi(i) \xrightarrow{hb} \pi(j)$.

- (2)1. If $t_i = t_j$, $\pi(i) \xrightarrow{po}_{t_j} \pi(j)$. Thus $\pi(i) \xrightarrow{hb} \pi(j)$ holds, by the definition of the happens-before relation.
 (2)2. Since $\pi(k)$ is a **sync-source** on o and $\pi(l)$ is a **sync-sink** on o , $\pi(k) \xrightarrow{sw}_o \pi(l)$ by the definition of synchronizes-with order.
 (2)3. Since both $\pi(j)$ and $\pi(l)$ is performed by t_j , $\pi(l) \xrightarrow{po}_{t_j} \pi(j)$ holds by the definition of the program order.
 (2)4. By the definition of the happens-before relation, $\pi(i) \xrightarrow{hb} \pi(k) \xrightarrow{sw}_o \pi(l) \xrightarrow{po}_{t_j} \pi(j)$ implies that $\pi(i) \xrightarrow{hb} \pi(j)$ holds.

□

Figure 3.4: Interpretation of the contents of GLS_j for Lemma 2.

In the following, let $GLS_j(x)$ refer to the lockset of x right before $\pi(j)$ is processed using *ApplyLocksetRules*.

Lemma 2 (Preservation of Goldilockset invariants):

Let $E = \langle Tid, Act, W, \xrightarrow{po}_t, \xrightarrow{so}_o \rangle$ be a well-formed execution and π a linear order on Act that respects $\xrightarrow{po}_t$ for all threads $t \in Tid$ and $\xrightarrow{so}_o$ for all synchronization objects $o \in Data$. Let $i < j$, and $\pi(i)$ and $\pi(j)$ be two actions performed by threads t_i and t_j , respectively. Assume that $\pi(i)$ accesses a data variable x , and there is no access $\pi(a)$ to x in between ($i < a < j$). The following hold for $GLS_j(x)$:

I₁. $o \in GLS_j(x)$ iff there exists $\pi(k) = \text{sync-source}(t_k, o)$ such that $i < k < j$ and $\pi(i) \xrightarrow{hb} \pi(k)$.

I₂. $t_j \in GLS_j(x)$ iff either $t_i = t_j$, or there exists $\pi(k) = \text{sync-sink}(t_j, o)$ for some o such that $i < k < j$ and $\pi(i) \xrightarrow{hb} \pi(k)$. ■

In the case of **I₁**, if $\pi(j)$ is a **sync-sink** action on o , then $\pi(i) \xrightarrow{hb} \pi(j)$ holds, and later, t_j can access x without a data race after $\pi(j)$. This scenario is illustrated in Figure 3.4(a). In the case of **I₂**, $\pi(i) \xrightarrow{hb} \pi(j)$ holds, and if $\pi(j)$ is an access to x then $\pi(j)$ is race free. This scenario is illustrated in Figure 3.4(b).

PROOF.

⟨1⟩1. We prove the lemma by induction on n , the number of actions between $\pi(i)$ and $\pi(j)$ ($n = j - i - 1$).

BASE CASE ($n = 0$): In this case $j = i + 1$ and there is no action between $\pi(i)$ and $\pi(j)$. Thus, we consider the Goldilockset of x right after x is accessed: $GLS_j(x) = \{t_i\}$ and **I₁** and **I₂** hold trivially.

INDUCTIVE CASE ($n > 0$): Assume that both **I₁** and **I₂** hold for $i + 1, \dots, j - 1$. We prove that the invariants hold for $\pi(j)$. Since this depends on how the Goldilockset of x changes when $\pi(j - 1)$ is processed, we do a case analysis on the action $\pi(j - 1)$.

⟨2⟩1. CASE $\pi(j - 1) = \text{sync-source}(t_{j-1}, o')$ for some o' .

In this case, rule 1 is applied: if $t_{j-1} \in GLS_{j-1}(x)$ add o' to $GLS_j(x)$.

⟨3⟩1. PROOF OF **I₁**:

⟨4⟩1. $\implies$: Assume $o \in GLS_j(x)$ and prove that $\pi(k) = \text{sync-source}(t_k, o)$ exists such that $\pi(i) \xrightarrow{hb} \pi(k)$. Either $o \neq o'$ or $o = o'$.

⟨5⟩1. If $o \neq o'$, then $o \in GLS_{j-1}(x)$ must hold, since rule 1 does not add o , but o' .

Take the same $\pi(k)$, given by the inductive hypothesis (**I₁**) for $j - 1$, for j .

⟨5⟩2. If $o = o'$, either o was also in $GLS_{j-1}(x)$, or it is added to $GLS(x)$ when processing $\pi(j - 1)$.

⟨6⟩1. In the former case, take the same $\pi(k)$, given by the inductive hypothesis (**I₁**) for $j - 1$, for j as well.

⟨6⟩2. In the latter case, $t_{j-1} \in GLS_{j-1}(x)$ must hold for rule 1 to be applied.

By **I₂** for $j - 1$, there must be a $\pi(m) = \text{sync-sink}(t_{j-1}, o)$ such that $i <$

$m < j - 1$ and $\pi(i) \xrightarrow{hb} \pi(m)$. By **I₁** for m , such $\pi(k)$ exists between $\pi(i)$ and $\pi(m)$.

⟨4⟩2. $\Leftarrow$: Assume $\pi(k) = \text{sync-source}(t_k, o)$ exists such that $\pi(i) \xrightarrow{hb} \pi(k)$ and prove that $o \in GLS_j(x)$. Either $k < (j - 1)$ or $k = (j - 1)$.

⟨5⟩1. If $k < (j - 1)$, then $o \in GLS_{j-1}(x)$ by the inductive hypothesis. Since rule 1 does not remove synchronization objects from $GLS(x)$, $o \in GLS_j(x)$ must hold.

⟨5⟩2. If $k = (j - 1)$, then $\text{sync-source}(t_k, o) = \pi(k) = \pi(j - 1) = \text{sync-source}(t_{j-1}, o')$ and $o = o'$. Since $\pi(i) \xrightarrow{hb} \pi(k) = \pi(j - 1)$, by Lemma 1, either $t_i = t_{j-1}$ or there exists a $\pi(m) = \text{sync-sink}(t_{j-1}, o'')$ ($m < j - 1$) on some variable o'' between $\pi(i)$ and $\pi(j - 1)$.

⟨6⟩1. If $t_i = t_{j-1}$, $GLS_{i+1}(x) = \{t_{j-1}\}$ by rule 3, and $t_{j-1} \in GLS_l(x)$ for all $i + 1 \leq l \leq j - 1$ by the Goldilockset update rules. Thus, while processing $\pi(j - 1)$, rule 1 adds o to $GLS_j(x)$, and the claim holds.

⟨6⟩2. In the other case, by **I₂**, $t_{j-1} \in GLS_{j-1}(x)$ holds. Thus, while processing $\pi(j - 1)$, rule 1 adds o to $GLS_j(x)$ and the claim holds.

⟨3⟩2. PROOF OF **I₂**:

⟨4⟩1. $\Rightarrow$: Assume $t_j \in GLS_j(x)$ and prove that $\pi(k) = \text{sync-sink}(t_j, o)$ exists such that $\pi(i) \xrightarrow{hb} \pi(k)$. Since rule 1 does not add or remove thread ids to/from $GLS(x)$, $t_j \in GLS_{j-1}(x)$ holds. Take the same $\pi(k)$, given by the inductive hypothesis (**I₂**) for $j - 1$, for j as well.

⟨4⟩2. $\Leftarrow$: Assume $\pi(k) = \text{sync-sink}(t_j, o)$ exists such that $\pi(i) \xrightarrow{hb} \pi(k)$ and prove that $t_j \in GLS_j(x)$. Either $k < (j - 1)$ or $k = (j - 1)$.

⟨5⟩1. If $k < (j - 1)$, then $t_j \in GLS_{j-1}(x)$ by the inductive hypothesis for $j - 1$. Since rule 1 does not add or remove thread ids to/from $GLS(x)$, $t_j \in GLS_j(x)$ holds.

⟨5⟩2. The case $k = (j - 1)$ cannot happen, since $\pi(k)$ is a **sync-sink** operation and $\pi(j - 1)$ is a **sync-source** operation.

⟨2⟩2. CASE $\pi(j - 1) = \text{sync-sink}(t_{j-1}, o')$ for some o' .

In this case, rule 2 is applied: if $o' \in GLS_{j-1}(x)$ add t_{j-1} to $GLS_j(x)$.

⟨3⟩1. PROOF OF **I₁**:

⟨4⟩1. $\implies$: Assume $o \in GLS_j(x)$ and prove that $\pi(k) = \text{sync-source}(t_k, o)$ exists such that $\pi(i) \xrightarrow{hb} \pi(k)$. Since rule 2 does not add or remove o to/from $GLS(x)$, $o \in GLS_{j-1}(x)$ holds. Take the same $\pi(k)$, given by the inductive hypothesis (**I₁**) for $j - 1$, for j as well.

⟨4⟩2. $\Leftarrow$: Assume $\pi(k) = \text{sync-source}(t_k, o)$ exists and prove that $o \in GLS_j(x)$. Either $k < (j - 1)$ or $k = (j - 1)$.

⟨5⟩1. If $k < (j - 1)$, then $o \in GLS_{j-1}(x)$ by the inductive hypothesis (**I₁**) for $j - 1$. Since rule 2 does not shrink $GLS(x)$, $o \in GLS_j(x)$ holds.

⟨5⟩2. The case $k = (j - 1)$ cannot happen, since $\pi(k)$ is a **sync-source** operation and $\pi(j - 1)$ is a **sync-sink** operation.

⟨3⟩2. PROOF OF **I₂**:

⟨4⟩1. $\implies$: Assume $t_j \in GLS_j(x)$ and prove that $\pi(k) = \text{sync-sink}(t_j, o)$ exists. Either t_j is in $GLS_{j-1}(x)$ or t_j is added to $GLS_j(x)$ while processing $\pi(j - 1) = \text{sync-sink}(t_{j-1}, o')$.

⟨5⟩1. If $t_j \in GLS_{j-1}(x)$, take the same $\pi(k)$, given by the inductive hypothesis (**I₂**) for $j - 1$, for j as well.

⟨5⟩2. If t_j is added to $GLS_j(x)$ while processing $\pi(j - 1)$, then $o \in GLS_{j-1}(x)$ and $t_j = t_{j-1}$ by rule 2. Either $t_i = t_{j-1}$ or not.

⟨5⟩3. If $t_i = t_{j-1}$, then $\pi(i) \xrightarrow{hb} \pi(j - 1)$ and taking $k = j - 1$ and $\pi(k) = \pi(j - 1) = \text{sync-sink}(t_j, o)$ proves the claim.

⟨5⟩4. Assume that $t_i \neq t_{j-1}$. Since $o \in GLS_{j-1}(x)$, by the inductive hypothesis (**I₁**), there exists $\pi(m) = \text{sync-source}(t_m, o)$ such that $i < m < j - 1$ and $\pi(i) \xrightarrow{hb} \pi(m)$. By Lemma 1, $\pi(i) \xrightarrow{hb} \pi(j - 1)$. Taking $k = j - 1$ and $\pi(k) = \pi(j - 1) = \text{sync-sink}(t_j, o)$ proves the claim.

⟨4⟩2. $\Leftarrow$: Assume $\pi(k) = \text{sync-sink}(t_j, o)$ exists and prove that $t_j \in GLS_j(x)$. Either $k < (j - 1)$ or $k = (j - 1)$.

⟨5⟩1. If $k < (j - 1)$, then $t_j \in GLS_{j-1}(x)$ by the inductive hypothesis (**I₂**). Since rule 2 does not shrink Goldilocksets, $t_j \in GLS_j(x)$ holds.

⟨5⟩2. If $k = (j - 1)$, then $\pi(k) = \pi(j - 1)$, $t_{j-1} = t_j$, and $o = o'$. Since $\pi(i) \xrightarrow{hb} \pi(k) = \pi(j - 1)$, by Lemma 1, either (1) $t_i = t_{j-1}$ or (2) there exists a $\pi(m) = \text{sync-sink}(t_{j-1}, o'')$ ($m < j - 1$) on some variable o'' between $\pi(i)$ and

$\pi(j-1)$.

$\langle 6 \rangle 1$. If (1) holds, then $GLS_{i+1}(x) = \{t_{j-1}\}$ by rule 3, and $t_{j-1} \in GLS_l(x)$ for all $i+1 \leq l \leq j-1$ by the Goldilockset update rules. Since $t_{j-1} = t_j$, $t_j \in GLS_{j-1}(x)$ also holds. $t_j \in GLS_j(x)$ holds since rule 2 does not remove thread ids from $GLS(x)$.

$\langle 6 \rangle 2$. If (2) holds, by **I₂**, $t_j \in GLS_{j-1}(x)$ holds. Since rule 2 does not shrink Goldilocksets, $t_j \in GLS_j(x)$ holds.

$\langle 2 \rangle 3$. CASE $\pi(j-1) = \text{read}(t_{j-1}, y)$ or $\pi(j-1) = \text{write}(t_{j-1}, y)$ for some y .

In this case, rule 3 is applied: if there is no race, $GLS_j(x)$ is reset to $\{t_{j-1}\}$. If $x = y$, then $i = j-1$ and this contradicts with the assumption $i < j$. Thus, in the following, we will assume that $x \neq y$ holds.

$\langle 3 \rangle 1$. PROOF OF **I₁**:

$\langle 4 \rangle 1$. $\implies$: Assume $o \in GLS_j(x)$ and prove that $\pi(k) = \text{sync-source}(t_k, o)$ exists such that $\pi(i) \xrightarrow{hb} \pi(k)$.

$\langle 5 \rangle 1$. Since rule 3 for y does not modify $GLS(x)$, $o \in GLS_{j-1}(x)$ holds. Take the same $\pi(k)$, given by the inductive hypothesis (**I₁**) for $j-1$, for j .

$\langle 4 \rangle 2$. $\Leftarrow$: Assume $\pi(k) = \text{sync-source}(t_k, o)$ exists and prove that $o \in GLS_j(x)$.

$\langle 5 \rangle 1$. By the inductive hypothesis (**I₁**) for $j-1$, $o \in GLS_{j-1}(x)$. Since $x \neq y$ and $\pi(j-1)$ accesses y , rule 3 for y does not modify $GLS(x)$. Thus $o \in GLS_j(x)$.

$\langle 3 \rangle 1$. PROOF OF **I₂**:

$\langle 4 \rangle 1$. $\implies$: Assume $t_j \in GLS_j(x)$ and prove that $\pi(k) = \text{sync-sink}(t_j, o)$ exists.

$\langle 5 \rangle 1$. $t_j \in GLS_{j-1}(x)$ holds since $x \neq y$ and rule 3 for y does not modify $GLS(x)$. Take the same $\pi(k)$, given by the inductive hypothesis (**I₂**) for $j-1$, for j as well.

$\langle 4 \rangle 2$. $\Leftarrow$: Assume $\pi(k) = \text{sync-sink}(t_j, o)$ exists and prove that $t_j \in GLS_j(x)$.

$\langle 5 \rangle 1$. Since $\pi(j-1)$ is a data access, $k < j-1$. Taking $\pi(k)$, given by the inductive hypothesis (**I₂**) for $j-1$, $t_j \in GLS_{j-1}(x)$ holds.

$\langle 5 \rangle 2$. $t_j \in GLS_j(x)$ holds since $x \neq y$ and rule 3 for y does not modify $GLS(x)$.

□

The following theorem expresses the fact that the Goldilocks algorithm is both *sound*, i.e., detects all actual races in a given execution, and *precise*, i.e., never reports false alarms.

Theorem 1 (Correctness of GOLDILOCKS update rules):

Let $E = \langle Tid, Act, W, \xrightarrow{po}, \xrightarrow{so} \rangle$ be a well-formed execution, x a data variable, and π a linear order on Act that respects $\xrightarrow{po}_t$ for all threads $t \in Tid$ and $\xrightarrow{so}_o$ for all synchronization objects $o \in Data$. Let $i < j$, and let $\pi(i)$ and $\pi(j)$ be two accesses to x performed by threads t_i and t_j , with no other action $\pi(k)$ in between ($i < k < j$) accessing x . Then $t_j \in GLS_j(x)$, i.e., the access $\pi(j)$ is declared to be race free by the Goldilocks algorithm iff $\pi(i) \xrightarrow{hb} \pi(j)$. ■

PROOF.

- (1)1. $\implies$: Assume that $t_j \in GLS_j(x)$. By Lemma 2, if $t_j \in GLS_j(x)$, then either (1) $t_i = t_j$ holds, or (2) there exists $\text{sync-sink}(t_j, o)$ for some o such that $i < k < j$ and $\pi(i) \xrightarrow{hb} \pi(k)$. If (1) holds, $\pi(i) \xrightarrow{po}_{t_i} \pi(j)$ and thus $\pi(i) \xrightarrow{hb} \pi(j)$, satisfying the claim. If (2) holds then $\pi(i) \xrightarrow{hb} \pi(j)$ holds because of $\pi(i) \xrightarrow{hb} \pi(k)$ and $\pi(k) \xrightarrow{po}_{t_k} \pi(j)$.
- (1)2. $\impliedby$: Assume $\pi(i) \xrightarrow{hb} \pi(j)$. By Lemma 1, there are two cases: Either (1) $t_i = t_j$ or (2) $t_i \neq t_j$ and there exists an action $\pi(k) = \text{sync-sink}(t_j, o)$ for some variable o such that $\pi(i) \xrightarrow{hb} \pi(k)$. In both cases, **I**₂ of Lemma 2 guarantees that $t_j \in GLS_j(x)$.

□

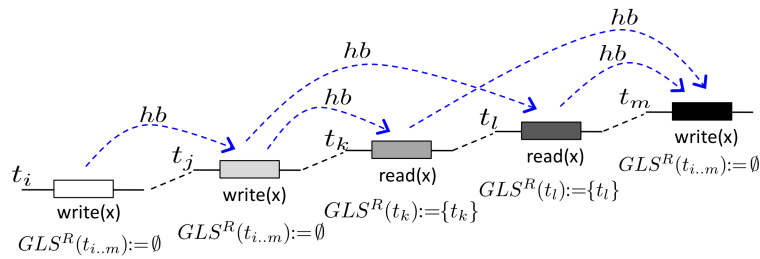
3.4 Distinguishing Read and Write Accesses

Figure 3.5: Happens-before between reads and writes, and read goldilockset of x .

The basic GOLDILOCKS algorithm in Figure 3.2 tracks the happens-before relationship between any two accesses to a variable x . In order to perform race detection, we must check the happens-before relationship only between conflicting actions, i.e., at least one action in the pair must be a write access. This distinction is illustrated in Figure 3.5; only the

indicated happens-before edges have to be checked. In this section, we describe how the basic algorithm is generalized in order to check for happens-before relationships between pairs of conflicting actions.

In the extended algorithm, we check the existence of an happens-before edge between the current access to x and the most recent conflicting accesses (in π) to x . For this, we keep track of the following Goldilocksets per data variable x :

- $GLS^W(x)$, the “write Goldilockset of x ” is used to check the happens-before between an access (read or write) to x and the most recent write to x .
- $GLS^R(x, t)$, the “read Goldilockset of t and x ” is used to check the happens-before between a write access to x and the most recent read of x by thread t .

The update rules in *ApplyLocksetRules* are adapted to maintain these Goldilocksets, but have essentially the same form as the rules in Figure 3.2. In the extended algorithm, it is sufficient to check happens-before between the current access to x and the most recent accesses (in the linear order π) to x . In response to **sync-source** and **sync-sink** operations, for all data variables x , both the write and the read Goldilocksets ($GLS^W(x)$ and $GLS^R(x, t)$ for all $t \in Tid$) are updated as specified in rules 1 and 2 in Figure 3.2. For read and write accesses, the update rules for read and write Goldilocksets are given in Figure 3.6.

Figure 3.5 shows updates to read Goldilocksets. Notice that after the write access by t_i we reset GLS^R to empty sets for all threads, thus when checking race for the access by t_j , only GLS^W is considered. When t_k and t_l read from x , $GLS^R(x, t_k)$ and $GLS^R(x, t_l)$ are reinitialized to contain t_k and t_l , respectively. Thus, when t_m writes to x , we check happens-before between each of these reads and the write by t_m . After this write by t_m , the read Goldilocksets are again reset to empty sets, so that when checking race for a later access, we do not have to consider the reads by t_k and t_l but only the write by t_m .

The following theorem states the correctness of the extended GOLDILOCKS algorithm presented in this section. In words, for every linearization π of actions in a given execution, (1) if there is no happens-before edges between a pair of conflicting actions, then the algorithm throws a **DataRaceException**, and (2) if all the conflicting accesses are separated by the happens-before relationship, then the algorithm throws no exception. Note that, at each synchronization action, the Goldilocksets GLS^W and GLS^R are updated according to

$$\begin{aligned}
\text{Empty}^R(x) &\triangleq \forall t \in \text{Tid}. \text{GLS}^R(x, t) = \emptyset \\
\text{Check}^W(t, x) &\triangleq \text{GLS}^W(x) = \emptyset \text{ or } t \in \text{GLS}^W(x) \\
\text{Check}^R(t, x) &\triangleq \forall u \in \text{Tid} \setminus \{t\}. \text{GLS}^R(x, u) = \emptyset \text{ or } t \in \text{GLS}^R(x, u)
\end{aligned}$$

```

3(a). if  $\pi(i) = \text{write}(t, x)$ 
    if  $\text{Empty}^R(x)$  //check most recent access
        if  $\text{Check}^W(t, x)$  //most recent access is "write"
             $\text{GLS}^W(x) := \{t\}$ 
        else
            throw a DataRaceException on  $x$ 
    elseif  $\text{Check}^R(t, x)$  //most recent access is "read"
         $\text{GLS}^W(x) := \{t\}$ 
        foreach  $u \in \text{Tid}$ :  $\text{GLS}^R(x, u) := \emptyset$ 
    else
        throw a DataRaceException on  $x$ 

3(b). if  $\pi(i) = \text{read}(t, x)$ 
    if  $\text{Check}^W(t, x)$ 
         $\text{GLS}^R(x, t) := \{t\}$ 
    else
        throw a DataRaceException on  $x$ 

```

Figure 3.6: Distinguishing read and write accesses

rules 1 and 2 of Figure 3.2, and at each data-variable access, rules 3(a) and 3(b) re-initialize GLS^W and GLS^R to a singleton containing only the id of the currently accessing thread. Therefore, the happens-before relationship between every pair of conflicting accesses are tracked following the core algorithm given in Figure 3.2. We will make use of this fact and refer to Theorem 1 while proving the theorem below.

Theorem 2 (Correctness of extended GOLDILOCKS update rules):

Let $E = \langle \text{Tid}, \text{Act}, W, \xrightarrow{po}., \xrightarrow{so}.\rangle$ be a well-formed execution, x a data variable, and π a linear order on Act that respects $\xrightarrow{po}_t$ for all threads $t \in \text{Tid}$ and $\xrightarrow{so}_o$ for all synchronization objects $o \in \text{Data}$. Let $\pi(j)$ be an access to x performed by threads t_j .

1. If $\pi(j) = \text{write}(t_j, x)$, then one of the following holds:

- (a) There is no $\pi(i)$ such that $i < j$ and $\pi(i)$ accesses x .
- (b) There exists $\pi(i) = \text{write}(t_i, x)$ such that $i < j$, there is no $\pi(k)$ ($i < k < j$) between $\pi(i)$ and $\pi(j)$ that accesses x , and $t_j \in GLS_j^W(x)$ iff $\pi(i) \xrightarrow{hb} \pi(j)$.
- (c) There exists $\pi(i) = \text{read}(t_i, x)$ such that $i < j$, there is no $\pi(k)$ ($i < k < j$) between $\pi(i)$ and $\pi(j)$ that reads from x and $t_k = t_i$ or writes to x , and $t_j \in GLS_j^R(x, t_i)$ iff $\pi(i) \xrightarrow{hb} \pi(j)$.

2. If $\pi(j) = \text{read}(t_j, x)$, then one of the following holds:

- (a) There is no $\pi(i) = \text{write}(t_i, x)$ such that $i < j$.
- (b) There exists $\pi(i) = \text{write}(t_i, x)$ such that $i < j$, there is no $\pi(k)$ ($i < k < j$) between $\pi(i)$ and $\pi(j)$ that writes to x , and $t_j \in GLS_j^W(x)$ iff $\pi(i) \xrightarrow{hb} \pi(j)$. ■

PROOF.

CASE: $\pi(j) = \text{write}(t_j, x)$

- ⟨2⟩1. If 1(a) holds, then there is no action $\pi(i)$ ($i < j$) that conflicts with $\pi(j)$.
- ⟨2⟩2. Assume that $\pi(i) = \text{write}(t_i, x)$ such that $i < j$, and there is no $\pi(k)$ ($i < k < j$) between $\pi(i)$ and $\pi(j)$ that accesses x .
 - ⟨3⟩1. After the access $\pi(i)$, $GLS^W(x) = \{t_i\}$ according to rule 3(a). Since there is no access to x between $\pi(i)$ and $\pi(j)$, only rules 1 and 2 update $GLS^W(x)$ for actions $\pi(i+1)$ through $\pi(j-1)$. By Theorem 1, $t_j \in GLS_j^W(x)$ iff $\pi(i) \xrightarrow{hb} \pi(j)$.
- ⟨2⟩3. Assume that $\pi(i) = \text{read}(t_i, x)$ such that $i < j$, and there is no $\pi(k)$ ($i < k < j$) between $\pi(i)$ and $\pi(j)$ that writes to x .
 - ⟨3⟩1. After the access $\pi(i)$, $GLS^R(x, t_i) = \{t_i\}$ according to rule 3(b). Since there is no write to x between $\pi(i)$ and $\pi(j)$, only rules 1 and 2 update $GLS^R(x, t_i)$ for actions $\pi(i+1)$ through $\pi(j-1)$. By Theorem 1, $t_j \in GLS_j^R(x, t_i)$ iff $\pi(i) \xrightarrow{hb} \pi(j)$.

CASE: $\pi(j) = \text{read}(t_j, x)$

- ⟨2⟩1. If 2(a) holds, then there is no action $\pi(i)$ ($i < j$) that conflicts with $\pi(j)$.
- ⟨2⟩2. Assume that $\pi(i) = \text{write}(t_i, x)$ such that $i < j$, and there is no $\pi(k)$ ($i < k < j$) between $\pi(i)$ and $\pi(j)$ that writes to x .

$\langle 3 \rangle 1$. After the access $\pi(i)$, $GLS^W(x) = \{t_i\}$ according to rule 3(a). Since there is no write to x between $\pi(i)$ and $\pi(j)$, only rules 1 and 2 update $GLS^W(x)$ for actions $\pi(i+1)$ through $\pi(j-1)$. By Theorem 1, $t_j \in GLS_j^W(x)$ iff $\pi(i) \xrightarrow{hb} \pi(j)$.

□

In this chapter, we presented the basic GOLDILOCKS algorithm and extended it to distinguish read and write accesses. This general formulation of the algorithm over a generic memory model allowed us to separate the fundamental aspects of the algorithm from the JMM-specific details, and in this way, to describe the algorithm clearly. In the next chapter we will express the JMM as a special case of this generic memory model and explain the adaptation of GOLDILOCKS to the JMM.

Chapter 4

SPECIALIZING GOLDBLOCKS TO THE JAVA MEMORY MODEL

The previous chapter presented the generalized GOLDBLOCKS algorithm to detect data races as defined by a generic memory model. Specializing the GOLDBLOCKS algorithm to a particular memory model requires one to specify (1) data and synchronization variables and operations on them, and (2) the synchronizes-with order between synchronization operations. In this chapter, we will establish this for the Java Memory Model (JMM) as defined in [2].

Outline. Section 4.1 defines the JMM as a special case of the generic memory model presented in the previous chapter. In Section 4.2 we give the instantiation of the GOLDBLOCKS algorithm for the JMM. Section 4.3 revisits the example in Figure 2.1 to demonstrate the operation of GOLDBLOCKS. In Section 4.4, we give a more detailed example demonstrating the use of thread locality, dynamically changing locksets, fork and join operations to ensure mutually exclusive access and show how our algorithm is able to uniformly capture all of these idioms. In Section 4.5, we discuss alternative approaches to handling `DataRaceException` by the JVM and by writing an exception handler. Section 4.6 discusses the relationship between detecting data races at runtime and guaranteeing sequential consistency.

4.1 The Java Memory Model (JMM)

The mathematical model presented in Section 3.1 is a relaxation of the JMM [2]. The JMM requires that *all* synchronization operations be ordered by a total order $\xrightarrow{so}$, whereas in our execution model, a separate total order $\xrightarrow{so}_o$ per synchronization variable is sufficient. The relaxation of the JMM's total order requirement does not appear to pose problems in defining the synchronizes-with relation, defining and checking races in executions, and defining sequentially consistent executions.

4.1.1 Data Variables and Operations

Let *Addr* be the set of all possible object references and *Field* be the set of all non-volatile fields. Here, we do not restrict *Addr* to *Field* to a particular Java program, but we consider all possible Java programs while defining these sets. In Java, every data variable is in the form of (o, d) where $o \in \text{Addr}$ is an object and $d \in \text{Field}$ is a non-volatile field. For static data fields, we let o refer to an object of the special class **Class** created internally by the JVM. The action kind $\text{alloc}(o)$ represents allocation of a new object o . Reads and writes to/from data variables are denoted $\text{read}(o, d)$ and $\text{write}(o, d)$ and realized by the byte-code instructions $x\text{load}$ and $x\text{store}$ in a Java virtual machine, respectively. The notations $x\text{load}$ and $x\text{store}$ define a collection of byte-code instructions where x changes depending on the type of the field. For example, $i\text{load}$ is used to read from an integer field of an object from in memory and $f\text{store}$ is used to write to a float field of an object in the memory.

4.1.2 Synchronization Variables and Operations

The JMM specifies three synchronization mechanisms: monitors, volatile fields, and fork/join operations.

Monitors

In Java, a monitor per object (denoted by m_o) provides a reentrant lock for each object o . Let *Monitor* be the set of all monitors. Acquiring the lock of an object o (**acquire**(o)) corresponds to a **sync-sink** operation on m_o , while releasing the lock of o (**release**(o)) corresponds to a **sync-source** operation on m_o ; This is because, in the JMM, all the **acquire** and **release** operations on object o are totally ordered by $\xrightarrow{so}_{m_o}$, and every **release**(o) synchronizes-with the next **acquire**(o) operation in $\xrightarrow{so}_{m_o}$. Since no other thread can acquire a monitor m_o as long as it is held by a thread, every **release**(o) in fact synchronizes-with all the **acquire**(o) operations in $\xrightarrow{so}_{m_o}$. Here, we only consider the first **acquire**(o) and last **release**(o) in a sequence of (reentrant) acquires and releases on the same lock o by the same thread; nested acquires and releases in between are treated no-ops.

Volatile Variables

Let *Volatile* denote the set of all volatile fields. Each volatile variable is denoted (o, v) where $o \in \text{Addr}$ is an object, and $v \in \text{Volatile}$ is a volatile field. For static volatile fields, we let o refer to an object of the special class **Class** created internally by the JVM. Each read **volread** (o, v) from a volatile variable (o, v) , and each write **volwrite** (o, v) to (o, v) is implemented by the *xload* and *xstore* byte-code instructions, respectively. While **volread** (o, v) corresponds to a **sync-sink**, **volwrite** (o, v) corresponds to **sync-source** operation on (o, v) . This is because, in the JMM, all the **volread** and **volwrite** operations on volatile variable (o, v) are totally ordered by $\xrightarrow{so}_{(o,v)}$, and every **volwrite** (o, v) synchronizes-with all the subsequent **volread** (o, v) operations in $\xrightarrow{so}_{(o,v)}$.

Fork/Join Synchronization

In order to handle synchronization using the fork and join operations, we use a fictitious synchronization object $\bar{t}$ for each thread $t \in \text{Tid}$. Let $\bar{\text{Tid}}$ be the set of all such variables. Creating a new thread with id t (**fork** (t)) synchronizes with the first action of thread t , denoted **start** (t) . The last action of thread t , denoted **end** (t) synchronizes with the join operation on t , denoted **join** (t) . Thus, for each thread t , **fork** (t) and **end** (t) correspond to **sync-source** operations on the variable $\bar{t}$, and **start** (t) and **join** (t) correspond to **sync-sink** operations on $\bar{t}$. The JMM guarantees that for each thread t , there exists an order $\xrightarrow{so}_{\bar{t}}$ such that: **fork** $(t) \xrightarrow{so}_{\bar{t}} \text{start}(t) \xrightarrow{so}_{\bar{t}} \text{end}(t) \xrightarrow{so}_{\bar{t}} \text{join}(t)$.

4.2 The Goldilocks Algorithm for Java

In this section, we will describe the specialized version of the GOLDILOCKS algorithm for the JMM. For simplicity, the exposition in this section does not distinguish between read and write accesses. In the specialized algorithm the Goldilockset of a variable (o, d) is defined to be a set $GLS(x) \subseteq \text{Tid} \cup \text{Monitor} \cup (\text{Addr} \times \text{Volatile}) \cup \bar{\text{Tid}}$. For each data variable (o, d) , the lockset $GLS(o, d)$ may contain thread identifiers, monitors, volatile variables, or fictitious variables for threads. Initially, the map *GLS* assigns the empty set to every data variable (o, d) . The algorithm updates *GLS* as each element in the linearization π is processed. The set of rules for these updates is shown in Figure 4.1. The rules in the figure take as input

Let $\pi(i)$ be executed by thread t	if $\pi(i) = \text{fork}(u)$
	foreach $(o', d') \in (\text{Addr} \times \text{Field})$:
if $\pi(i) = \text{volread}(o, v)$	if $t \in \text{GLS}(o', d)$
foreach $(o', d') \in (\text{Addr} \times \text{Field})$:	$\text{GLS}(o', d') := \text{GLS}(o', d') \cup \{\bar{u}\}$
if $(o, v) \in \text{GLS}(o', d)$	
$\text{GLS}(o', d') := \text{GLS}(o', d') \cup \{t\}$	if $\pi(i) = \text{start}(t)$
	foreach $(o', d') \in (\text{Addr} \times \text{Field})$:
if $\pi(i) = \text{volwrite}(o, v)$	if $\bar{t} \in \text{GLS}(o', d)$
foreach $(o', d') \in (\text{Addr} \times \text{Field})$:	$\text{GLS}(o', d') := \text{GLS}(o', d') \cup \{t\}$
if $t \in \text{GLS}(o', d)$	
$\text{GLS}(o', d') := \text{GLS}(o', d') \cup \{(o, v)\}$	if $\pi(i) = \text{end}(t)$
	foreach $(o', d') \in (\text{Addr} \times \text{Field})$:
if $\pi(i) = \text{acquire}(m_o)$	if $t \in \text{GLS}(o', d)$
foreach $(o', d') \in (\text{Addr} \times \text{Field})$:	$\text{GLS}(o', d') := \text{GLS}(o', d') \cup \{\bar{t}\}$
if $m_o \in \text{GLS}(o', d)$	
$\text{GLS}(o', d') := \text{GLS}(o', d') \cup \{t\}$	if $\pi(i) = \text{join}(u)$
	foreach $(o', d') \in (\text{Addr} \times \text{Field})$:
if $\pi(i) = \text{release}(m_o)$	if $\bar{u} \in \text{GLS}(o', d)$
foreach $(o', d') \in (\text{Addr} \times \text{Field})$:	$\text{GLS}(o', d') := \text{GLS}(o', d') \cup \{t\}$
if $t \in \text{GLS}(o', d)$	
$\text{GLS}(o', d') := \text{GLS}(o', d') \cup \{m_o\}$	if $\pi(i) = \text{read}(o, d)$ or $\pi(i) = \text{write}(o, d)$
	if $\text{GLS}(o, d) = \emptyset$ or $t \in \text{GLS}(o, d)$
if $\pi(i) = \text{alloc}(o)$	$\text{GLS}(o, d) := \{t\}$
foreach $d \in \text{Field}$: $\text{GLS}(o, d) := \emptyset$	else
	throw a <code>DataRaceException</code> on (o, d)

Figure 4.1: The Goldilockset update rules specialized to Java

a linearization π of the actions in the execution that respects $\xrightarrow{po}_t$ for all threads $t \in \text{Tid}$ and $\xrightarrow{so}_o$ for all synchronization objects (monitors, volatiles, and threads) $o \in \text{Data}$. The intuitive interpretation of a lockset $\text{GLS}(o, d)$ is as follows:

- If $\text{GLS}(o, d)$ is empty, this indicates that (o, d) has not been accessed since it was allocated. Therefore, an access to (o, d) is necessarily race free when $\text{GLS}(o, d)$ is

empty. Note that, $GLS(o, d)$ never becomes empty after the first access to (o, d) according to the Goldilockset update rules. $GLS(o, d)$ is initialized to the empty set and is reset to the empty set whenever o is an object returned by a memory allocation.

- If a thread identifier t is in $GLS(o, d)$, then t is an owner of (o, d) and an access by t to (o, d) is race-free.
- If a monitor $m_{o'}$ is in $GLS(o, d)$, a thread can become an owner of (o, d) by acquiring the monitor $m_{o'}$. This is because the thread that last accessed (o, d) released $m_{o'}$ subsequently.
- If a volatile variable (o', v) is in $GLS(o, d)$, a thread can become an owner of (o, d) by reading (o', v) . This is because the thread that last accessed (o, d) wrote to (o', v) subsequently.
- If a fictitious variable $\bar{t}$ is in $GLS(o, d)$, there are two cases: Either thread t has been created but has not started yet, or thread t has terminated. In the former case, thread t becomes an owner of (o, d) when it starts. In the latter case, a thread joining t by running `join(t)` can become an owner of (o, d) .

Note that, a naive implementation of this algorithm would be too expensive for programs that manipulate large heaps, and in Chapter 6 we present an efficient scheme to implement our algorithm by applying these updates lazily.

4.2.1 Handling Other Synchronization Mechanisms

Using the lockset update rules above, GOLDSILLOCKS is able to uniformly handle various approaches to synchronization such as dynamically changing locksets, permanent or temporary thread-locality of objects, container-protected objects, ownership transfer of a variable without accessing the variable (as in the example in Section 4.3 below). Furthermore, GOLDSILLOCKS can also handle `wait/notify(All)` (see below), and the synchronization idioms in the `java.util.concurrent` package such as semaphores and barriers, since these primitives are built using only locks and volatile variables.

Wait/Notify Synchronization

Java programs commonly use wait/notify synchronization along with synchronized blocks. The GOLDILOCKS algorithm needs no special rules for wait and notify operations for the following reasons: The Java specification requires each call to `o.wait()` and `o.notify()` on an object `o` be surrounded by the `synchronized(o)` block. Thus `o.wait()` and `o.notify()` are already surrounded by explicit invocations of `acquire(o)` and `release(o)` operations. In addition, the `x.wait()` primitive is implemented by the JVM using the `acquire(o)` and `release(o)` operations internally; these operations are detected and processed by our algorithm. Except for the `acquire` and `release` operations referred to above, the `wait/notify(All)` synchronization in Java does not cause any more happens-before edges to be handled by GOLDILOCKS.

4.3 Example: Precise Data-Race Detection for Java

In this section, we revisit the example in Section 2.2—Figure 2.1 to demonstrate the precision of the Goldilocks algorithm compared to other lockset-based algorithms from the literature. Unlike GOLDILOCKS, which is both sound and precise, other lockset algorithms based on the Eraser algorithm [14] are sound but not precise. Figure 4.2 shows an execution of the program and the evolution of the Goldilockset $GLS(x)$. Note that x represents the variable `o.data` where `o` is the `IntBox` object created at the first line of the execution. The figure also shows the Goldilockset update rules applied at each point; the rules that successfully updates $GLS(x)$ are given in boxes.

The example in Figure 4.2 makes use of two idioms typically not handled by earlier lockset algorithms such as Eraser [14]: objects that are local to different threads at different points in the execution, and objects protected by locks of container objects. The Eraser-based algorithms are based on checking the synchronization discipline that each shared variable is protected by a fixed set of locks throughout the execution. For our example, this approach is too conservative, and the lockset algorithm will report a data-race on x at the very first access of the execution in Figure 4.2.

Observe that the update rules allow a variable's Goldilockset to grow during the execution. This enables them to represent threads transferring ownership using different synchronization variables during the execution. In this example, this ownership transfer takes

Execution	Goldilockset update rule applied	$GLS(x)$
Thread 1 (T1):		
<code>tmp1 = new IntBox();</code>	Initialize lockset	$\emptyset$
<code>tmp1.data = 0;</code>	First access	$\{T1\}$
<code>acquire(La);</code>	$La \in GLS(x) \rightarrow \text{add } T1 \text{ to } GLS(x)$	$\{T1\}$
<code>a = tmp1;</code>	No access to x	$\{T1\}$
<code>release(La);</code>	$T1 \in GLS(x) \rightarrow \text{add } La \text{ to } GLS(x)$	$\{T1, La\}$
Thread 2 (T2):		
<code>acquire(La);</code>	$La \in GLS(x) \rightarrow \text{add } T2 \text{ to } GLS(x)$	$\{T1, La, T2\}$
<code>tmp2 = a;</code>	No access to x	$\{T1, La, T2\}$
<code>release(La);</code>	$T2 \in GLS(x) \rightarrow \text{add } La \text{ to } GLS(x)$	$\{T1, La, T2\}$
<code>acquire(Lb);</code>	$Lb \in GLS(x) \rightarrow \text{add } T2 \text{ to } GLS(x)$	$\{T1, La, T2\}$
<code>b = tmp2;</code>	No access to x	$\{T1, La, T2\}$
<code>release(Lb);</code>	$T2 \in GLS(x) \rightarrow \text{add } Lb \text{ to } GLS(x)$	$\{T1, La, T2, Lb\}$
Thread 3 (T3):		
<code>acquire(Lb);</code>	$Lb \in GLS(x) \rightarrow \text{add } T3 \text{ to } GLS(x)$	$\{T1, La, T2, Lb, T3\}$
<code>b.data = 2;</code>	$T3 \in GLS(x) \rightarrow \text{Race-free access}$	$\{T3\}$
<code>tmp3 = b;</code>	No access to x	$\{T3\}$
<code>release(Lb);</code>	$T3 \in GLS(x) \rightarrow \text{add } Lb \text{ to } GLS(x)$	$\{T3, Lb\}$
<code>tmp3.data = 3;</code>	$T3 \in GLS(x) \rightarrow \text{Race-free access}$	$\{T3\}$

Figure 4.2: Precise data-race detection example

place without the variable itself being accessed. For example, after T2 finishes in Figure 4.2, $GLS(x)$ has the value $\{T1, La, T2, Lb\}$, meaning that a thread can access x without data race by locking either La or Lb . Then T3 makes Lb the only protector lock by acquiring Lb and accesses x .

Note that the vector clock algorithm [18] is precise and does not declare a false race in this example and similar scenarios. However, it accomplishes this at significantly increased memory and computational cost compared to our optimized implementation of the lockset update rules. This is discussed in detail in Section 7.3.

4.4 Example: A Task Queue

The example in this section, for which pseudo-code is provided in Figure 4.3, demonstrates the use of thread locality, dynamically changing locksets, fork and join operations to ensure mutually exclusive access and how our algorithm is able to uniformly capture all of these idioms. This example consists of a program that schedules tasks (represented by class `Task`) into a queue named `tQ`, dequeues and executes them one by one. Each `Task` instance contains an array `subTasks` of subtasks. Each subtask is a `SubTask` instance. The computation required for a single subtask is represented by a function `Perform` that takes a `SubTask` and produces an integer output. The sum of all the outputs are the final result of the task and this value is also stored in its `out` field. The `Task` object is protected by `Tlock` and the task queue is protected by `Qlock`.

```

class Task {
    SubTask subTasks[n];
    int out;
    Task(SubTask sT[]) { subTasks = sT; out = 0;}
}

Queue<Task> tQ;

CreateTask(SubTask sTs[])      ParallelTaskHelper(oneTask)
1  oneTask = new Task(sTs);    1  foreach (i < n)
2  acquire(Qlock)              2    children[i] = fork(PerformSubTask,oneTask,i);
3    tQ.Enqueue(oneTask);      3  foreach (i < n)
4  release(Qlock);             4    join(children[i]);
                               5  print(oneTask.out);

PerformNextTask()              PerformSubTask(oneTask, i)
1  acquire(Qlock)              1  subTaskResult = Perform(oneTask.subTasks[i]);
2    oneTask = tQ.Dequeue();   2  acquire(Tlock);
3  release(Qlock);             3    oneTask.out += subTaskResult;
4  ParallelTaskHelper(oneTask); 4  release(Tlock);

```

Figure 4.3: Pseudo-code for the task queue example.

CreateTask, given an array **sTs** of subtasks, creates a new task object and enqueues it in the task queue. **PerformNextTask** dequeues a task from **tQ** and calls **ParallelTaskHelper**, which actually performs the task. **ParallelTaskHelper** forks for each subtask a new thread that runs **PerformSubTask**. **PerformSubTask** computes the partial result for the given subtask and adds it to the final result of the task.

Consider the following interleaving of actions during a scenario which begins with creation of two threads **T1** and **T2**:

1. A thread **T1**, by running **CreateTask** with an array containing two subtasks **st0** and **st1** as input,
 - (a) creates a new task **oneTask** by calling the **Task** constructor (line 1),
 - (b) acquires **Qlock**, calls **tQ.Enqueue (oneTask)**, releases **Qlock** (lines 2-4).
2. A second thread **T2**, runs **PerformNextTask** which
 - (a) acquires **Qlock** and calls **tQ.Dequeue ()** that returns **oneTask** (lines 1-3),
 - (b) calls **ParallelTaskHelper (oneTask)** (line 4), which creates two threads **T_st0** and **T_st1**, to handle **st0** and **st1** respectively (lines 1-2).
3. **T_st0**, by running **PerformSubTask (oneTask, 0)**,
 - (a) calls **Perform ()** (line 1), acquires **Tlock**, and
 - (b) adds **subTaskResult** to **oneTask.out**, releases **Tlock** (lines 2-4).
4. The second thread **T_st1**, by running **PerformSubTask (oneTask, 1)**,
 - (a) calls **Perform (oneTask.subTasks [1])** (line 1), acquires **Tlock**, and
 - (b) adds **subTaskResult** to **oneTask.out**, releases **Tlock** (lines 2-4).
5. Thread **T2**, continuing running **ParallelTaskHelper**,
 - (a) joins both threads **T_st0** and **T_st1** (lines 3-4),
 - (b) prints **oneTask.out**.

Let us focus on the shared variable `oneTask.out`. In the execution described above, there is no race on `oneTask.out` but the lock protecting it changes dynamically. For example, `oneTask.out` is local to T1 at the beginning and to T2 at the end of the scenario.

In the following, we show how our lockset algorithm handles this execution. Each item below explains how $GLS(\text{oneTask.out})$ changes after each action during the scenario.

$GLS(\text{oneTask.out})$ is initially undefined. Our algorithm handles thread-locality by treating thread identifiers similar to locks, and allowing GLS to contain thread identifiers.

1. (a) In the constructor of `Task`, `oneTask.out` is first accessed by T1. At this point the algorithm sets $GLS(\text{oneTask.out}) = \{T1\}$.
 (b) After T1 releases `Qlock`, the rule for release actions adds `Qlock` to the lockset, which yields $GLS(\text{oneTask.out}) = \{T1, Qlock\}$.
2. (a) After T2 acquires `Qlock`, since $Qlock \in GLS(\text{oneTask.out})$ the rule for acquire actions adds T2 to the lockset so that $GLS(\text{oneTask.out}) = \{T1, Qlock, T2\}$.
 (b) After T2 forks `T_st0`, since $T2 \in GLS(\text{oneTask.out})$, `T_st0` gets added to the lockset to yield $GLS(\text{oneTask.out}) = \{T1, Qlock, T2, T\_st0\}$.
 The same update applies when T2 creates `T_st1` such that $GLS(\text{oneTask.out}) = \{T1, Qlock, T2, T\_st0, T\_st1\}$.
3. (a) After `T_st0` acquires `Tlock`, since $Tlock \in GLS(\text{oneTask.out})$, `T_st0` gets added to the lockset which leaves the lockset unchanged.
 (b) After `oneTask.out` is written we check whether $T\_st0 \in GLS(\text{oneTask.out})$. Since the check succeeds, no race is declared and the lockset becomes $GLS(\text{oneTask.out}) = \{T\_st0\}$. When `T_st0` releases `Tlock`, since $T\_st0 \in GLS(\text{oneTask.out})$ the rule for release actions adds `Tlock` to the lockset to yield $GLS(\text{oneTask.out}) = \{T\_st0, Tlock\}$.
4. (a) After `T_st1` acquires `Tlock`, since $Tlock \in GLS(\text{oneTask.out})$, `T_st1` gets added to the lockset, which yields $GLS(\text{oneTask.out}) = \{T\_st0, Tlock, T\_st1\}$.
 (b) When `oneTask.out` is written, since $T\_st1 \in GLS(\text{oneTask.out})$ before the write action, no race is declared. After the write action we set $GLS(\text{oneTask.out}) =$

$\{T\_st1\}$. When T_st1 releases $Tlock$, since $T\_st1 \in GLS(oneTask.out)$ the rule for release actions adds $Tlock$ to the lockset to yield $GLS(oneTask.out) = \{T\_st1, Tlock\}$.

5. (a) Before $T2$ joins T_st1 , $T\_st1 \in GLS(oneTask.out)$, therefore, after the join action, we add $T2$ to the lockset to obtain $GLS(oneTask.out) = \{T\_st1, Tlock, T2\}$.
- (b) When `oneTask.out` is accessed by `print`, we check whether $T2 \in GLS(oneTask.out)$. This check succeeds and we do not declare a race. After the access, we set $GLS(oneTask.out) = \{T2\}$.

The description above illustrates although $GLS(oneTask.out)$ shrinks at an access to $GLS(oneTask.out)$, it can grow whenever a thread executes a release or a fork operation. It is this ability to grow the lockset that is fundamental to capturing dynamic locking idioms.

4.5 Discussion: Handling *DataRaceException*

When GOLDILOCKS detects that an actual race condition is going to happen, it raises a *DataRaceException*. Then, the programmer can select between (1) allowing the JVM to perform the default action for *DataRaceException*, and (2) catching the exception and reacting to the race. Section 4.5.1 discusses the alternative default actions, and Section 4.5.2 demonstrates writing an exception handler for *DataRaceException* that terminates the racy operations gracefully.

4.5.1 Default Action for *DataRaceException*

For runtime exceptions, the Java specification defines the default action to be terminating the thread causing the exception. If this action is chosen for the *DataRaceException* as well, the thread that attempts to make the second, racy access –and detects that a race is going to happen– must be terminated. However, in contrast to other runtime errors in Java, a data race is caused by more than one thread. This fact leads us to recommend the following default actions as alternatives to the one above.

A conservative default action is to terminate all the threads involved in the race. This approach may become necessary for the following reasons:

- The action of terminating only the last thread assumes that the race checking and the actual access are performed in one atomic step indivisible by other threads, for example using a unique lock per variable (see Section 6.2). In this case, the order of accesses to x and the order of the corresponding race checks coincide, and thus the thread detected the race is the one that last accessed x . If this atomicity requirement is not satisfied, then the relative ordering between the race checks and the accesses may differ due to the context switches between the check and the access. Notice that, in the extreme case, when the `DataRaceException` is thrown, none of the accesses involved in the race may yet have taken place. In this case, terminating all the threads involved in the race becomes necessary, since it may not be possible to determine at the time of the exception the ordering between their accesses.
- A `DataRaceException` is thrown as a result of missing or inconsistent synchronization between a thread, say t , currently accessing to a variable x and one or more different threads that accessed x before. While the thread t seems to be the one violating the synchronization discipline on x , the discipline may have already been violated by the previously accessing threads. Although the `DataRaceException` provides information on all the threads involved in the race, deciding which thread(s) to blame in this situation may depend on a more stringent –and possibly global– analysis on the program, which is usually difficult or costly to do at runtime.

If the default action is chosen to be terminating the threads involved in the race, these threads are determined as follows (where the “current access” refers to the one for which the race checking throws the `DataRaceException`):

- If the current access is a read from x , then the race is between the current thread and the thread that wrote to x most recently¹.
- If the current access is a write to x , then the race is between the current thread and either (1) the thread that wrote to x most recently or (2) the thread(s) that read from x and came after (in the happens-before order) the most recent writer in (1).

¹As long as no race happens on a variable x , all the write accesses to x are totally-ordered (which is a projection of the happens-before ordering). The most recent write to x is one that comes last in this total order before the race happens.

A more conservative option is to terminate the entire program upon a data race. The JMM gives a rather complicated semantics to racy programs, and thus, a racy execution is most likely one that is not intended by the programmer. Therefore, the program may behave in an unexpected manner even after terminating all the threads involved in the race, and terminating the entire program becomes a reasonable action to prevent the unanticipated consequences of the execution. If this option is chosen, then `DataRaceException` should be made a subclass of `Error` instead of `RuntimeException`, since it now indicates an error that the program is not expected to handle or recover from. Studies on memory model exceptions [5, 6] suggest terminating the program as the default action, as well.

4.5.2 Example: Writing an Exception Handler for `DataRaceException`

Figure 4.4 demonstrates the use of `DataRaceException` to terminate gracefully a thread about to perform an access that would have caused a data race before the race leads to an error. This example is adapted from the Apache ftp-server benchmark used in [11]. The `run()` and `close()` methods shown belong to an object representing an ftp connection and are executed by two separate threads. The thread executing the `run()` method, in a `do...while` loop services requests issued by the user on the command line, one command per iteration of the loop. A time-out thread executes the `close()` method of connection objects that have been idle too long. In the original benchmark, which did not contain the `try...catch` block for `DataRaceException`, it is possible for the time-out thread to run the `close()` method and set, for instance, the `m_writer` field of the connection object to `null` right before the other thread accesses it in the `run()` method. This race condition causes a `NullPointerException`. Note that, although in this example a race condition has an immediately noticeable consequence, in other examples the observable manifestation of the race condition may occur far enough in the execution to make it very difficult to trace its origin. Avoiding this race condition by making `run()` a synchronized method would make it impossible for the time-out thread to execute `close()`, thus, in a correct implementation, a sophisticated, finer-grain synchronization needs to be used, as discussed by Keremoglu et al. [24].

In the modified code that uses `DataRaceException`, when the thread executing `run()` is about to access `m_writer` after the unsynchronized access by the thread running `close()`,

```

public void run() {
    ... // Initialize connection conn
    try {

        do {
            String commandLine = m_reader.readLine();
            ...
            m_request.parse(commandLine);
            if(!hasPermission()) {
                m_writer.send(530, "permission", null);
                continue;
            }
            // execute the command
            service(m_request, m_writer);
        } while(!m_isConnectionClosed);

    } catch (DataRaceException e) {

        e.firstThread.exit();
        if(!conn.IsClosed()) {
            conn.close();
        }
        RollbackDBTransactions();
        LogError("Connection closed by " + e.firstThread.Id);
        currentThread.exit();
    }
}

```

```

public void close() {
    synchronized(this) {
        if(m_isConnectionClosed) {
            return;
        }
        m_isConnectionClosed = true;
    }
    ... // Close connection conn
    m_isConnectionClosed = true;
    m_request = null;
    m_writer = null;
    m_reader = null;
}

```

Figure 4.4: Example 1: Demonstrating use of `DataRaceException`

a `DataRaceException` is thrown. Upon catching this exception, `run()` first stops the other thread that involved in the race. Then it tries to close the connection itself if the connection has been not closed. It also rolls back the database transactions started by the `service` routine for the last request. Finally, it writes a message to the log for debugging and termi-

nates the current thread. Here, the exception handling mechanism allows the programmer to roll-back the (possibly corrupted) state of the program to a consistent state. Alternatively, if the information about the remote connection (`conn`) is still available after the race condition, the programmer may choose to restart the thread that performed the racy operation in order to continue servicing the same remote client. Moreover, the programmer may also choose to restart the FTP server to avoid the possible damages of the racy operations on other connections.

4.6 Discussion: Dynamic Race Detection and Sequential Consistency

The Java and C++ memory models provide the data-race freedom (DRF) property [2, 3]. The DRF property guarantees that if all sequentially consistent executions of a *source* program are race free, then the *compiled* program only exhibits these sequentially consistent executions of the source program, after any compiler and hardware optimizations permitted by the memory model. The GOLDILOCKS algorithm checks races by monitoring the executions of the compiled program, and assumes that the compiler and the runtime on which the program is built (hardware or virtual machine) conform to the language and the memory model specifications. Therefore, if the source program is race-free, then any execution of the compiled program corresponds to a sequentially consistent execution of the source program, and no `DataRaceException` is thrown.

If the source program has a race, the GOLDILOCKS runtime still ensures that all executions of the compiled program will run under the sequential consistency semantics, i.e., sequential consistency is guaranteed at the *byte-code level*. This is accomplished by preventing accesses that will cause a data race and throwing a `DataRaceException` right before that access. However, in the case of a racy program, the JMM permits compiler optimizations that result in executions that are not sequentially consistent behaviors of the original source code. In this case, the JMM and the DRF property are not strong enough to allow the Goldilocks runtime to relate byte-code level executions to executions of the source-level program, which makes debugging hard.

To use GOLDILOCKS for debugging purposes, this difficulty can be remedied by disabling compiler optimizations. For post-deployment use, a stronger memory model [5, 6] that is able to relate each (racy and race-free) execution of the compiled program to a sequentially

consistent execution of the source program is needed. Ideally, such strong memory models should be able to relate every race-free execution prefix of the compiled and executed program to a sequentially-consistent, race-free execution prefix of the source code.

In this chapter, we gave the specialized version GOLDILOCKS algorithm to detect data races in Java programs. A comparison between the general, compact version of GOLDILOCKS in Chapter 3 and the algorithm presented in this chapter shows how the former can naturally capture various synchronization mechanisms in Java. In the next chapter, we will extend the algorithm presented above to detect races for programs using software transactions along with the standard synchronization mechanisms of Java.

Chapter 5

DYNAMIC RACE DETECTION AND SOFTWARE TRANSACTIONAL MEMORY

We imagine, at least in the near future, that programmers will use software transactional memory (STM) [10] (or shortly *software transactions*) to manage only some of the accesses to shared data, while using other Java synchronization primitives to manage the rest. This is because most existing transaction implementations do not support I/O and may require libraries to be re-compiled [25]. Thus, it is possible that the same variables can be accessed within transactions and, for example, lock-protected regions at different points in the execution. In this thesis, we provide the first formalization of race conditions for executions that use software transactions in addition to other Java synchronization primitives. In this chapter, we will show how we extend the Goldilockset update rules presented in Chapter 4 in order to handle transactions.

Outline. Section 5.1 gives two examples that motivate data-race checking for programs using software transactions and shows an example data race that happens in a program using both locks and software transactions. In Section 5.2 we extend the GOLDILOCKS algorithm for software transactions. Section 5.3 revisits the example in Section 5.1 and explains the operation of GOLDILOCKS in the case of software transactions. Section 5.4 discusses the adaptation of our algorithm to different software transaction implementations.

5.1 Motivating Examples

5.1.1 Example 1: Handling Software Transactions

In this example (Figure 5.1), software transactions and thread-locality are used at different times in the execution to protect access to the same shared data. A race checking algorithm that does not take into account transactions as a synchronization primitive would declare a false race in this example. Here, `Foo` objects form a linked-list, and while a `Foo` object

```

public class Foo { int data; Foo nxt; }

Thread 1:          Thread 2:          Thread 3:
-----
Foo t1 = new Foo();  Foo iter;          Foo t3;
t1.data = 42;        atomic {          atomic {
atomic {             for (iter = head;    t3 = head;
    t1.nxt = head;    iter != null;          head = t3.nxt;
    head = t1;        iter = iter.nxt)    }
}                    iter.data = 0;      t3.data++;
                    }

```

Figure 5.1: Code for Example 1

is a member of the linked list, access to it is managed by software transactions. A `Foo` object referred to by `t1` is created and initialized by **Thread 1**, during which time it is thread-local. Then, in an atomic transaction, it is added to the head of the linked list by **Thread 1**. While in the linked list, this `Foo` object is modified by the loop in **Thread 2**, which, in an atomic software transaction, modifies the `data` fields of all `Foo` objects in the list. **Thread 3** removes the `Foo` object from the list in an atomic transaction. After this, the object is local to **Thread 3**. Observe that it is possible to make this example more sophisticated, for instance, by having the `Foo` object be shared among threads and making it lock-protected after its removal from the list. This example represents a natural use of software transactions where shared data is protected explicitly by the user through different means before and after it is managed by software transactions. A correct implementation of software transactions would create a happens-before relationship between the transactions in **Thread 1**, **2** and **3**. A race checking algorithm that is not aware of these happens-before edges would falsely declare a race between the accesses to the `data` field of the `Foo` object in **Thread 1** and **Thread 3**. The generalized **GOLDILOCKS** algorithm we present in this thesis correctly infers the absence of a race between these accesses. After describing how we extend **GOLDILOCKS** to support software transactions, we will revisit this example in Section 5.3 and explain how **GOLDILOCKS** handles the scenario given above.

5.1.2 Example 2: Locks and Software Transactions

```
public class Account {
    double bal;
    public synchronized withdraw(int amt) {
        bal = bal - amt;
    }
}

Account savings, checking;

Thread 1:                                Thread 2:
-----                                -----
1 atomic {                                1 checking.withdraw(42);
2   savings.bal = savings.bal - 42;
3   checking.bal = checking.bal + 42;
4 }
```

Figure 5.2: Code for Example 2

In this example (Figure 5.2), shared data is either protected by an object lock or is accessed within a transaction, which might lead one to believe at first glance that there should be no race conditions. `checking` and `savings` are `Account` objects with a synchronized `withdraw` method. `Thread 1` contains a software transaction that transfers money from `savings` to `checking`. `Thread 2` simply performs a withdrawal using the synchronized `withdraw` method. Since the software transaction implementation might be using a mechanism other than the object locks on `checking` and `savings` to implement the atomic transaction, there is a potential race condition between `Thread 1` and `Thread 2`'s accesses to `checking.bal`. This race condition should be signaled regardless of the synchronization mechanism used by the transaction implementation, since this mechanism should not be visible to the programmer. This is necessary for Java programs to be correct independently from which implementation of atomic transactions is chosen. Observe that completely ignoring accesses inside transactions while performing dynamic race checking would overlook the race in this case.

5.2 Extending the Algorithm for Software Transactions

In this section we explain how to extend the GOLDBLOCKS algorithm for software transactional memory. For this, we build on the formalism introduced in Chapter 4 and take the GOLDBLOCKS algorithm presented in Section 4.2 as the basis.

The action $commit(R, W)$ represents the committing of a transaction that reads and writes the sets of shared data variables R and W respectively. We do not allow transaction bodies to include synchronization, therefore $R, W \subseteq (Addr \times Field)$. Each commit action $commit(R, W)$ corresponds to an atomic sequence of reads $read(o_1, d_1), \dots, read(o_m, d_m)$ and writes $write(o_{m+1}, d_{m+1}), \dots, write(o_n, d_n)$ such that $R = \{(o_1, d_1), \dots, (o_m, d_m)\}$ and $W = \{(o_{m+1}, d_{m+1}), \dots, (o_n, d_n)\}$.

There are different ways of defining the happens-before relationship between software transactions in the literature [26, 27, 28, 29, 30]. In this thesis, we follow the interpretation of software transactions in Grossman et al. [27]. All the $commit(R, W)$ actions are totally ordered by the synchronization order $\xrightarrow{so}$ for transactions. The synchronizes-with order between transactions is defined using the sets R and W . For each data variable (o, d) , we define a fictitious synchronization variable denoted by $(\bar{o}, \bar{d})$. We could (in fact our implementation does) use the same variable (o, d) to represent the synchronization variable instead of appealing to a fictitious variable, but we find it useful to separate the actual variables from the synchronization variables for ease of exposition. A transaction synchronizes with (and happens before) another iff they access at least one common variable and the former commits before the latter. In this view, two transactions that access disjoint sets of variables do not synchronize with each other. Formally, for each pair $commit_1(R_1, W_1)$, $commit_2(R_2, W_2)$ of commit actions $commit_1(R_1, W_1) \xrightarrow{sw}_{(\bar{o}, \bar{d})} commit_2(R_2, W_2)$ holds if $commit_1(R_1, W_1) \xrightarrow{so} commit_2(R_2, W_2)$ and $(\bar{o}, \bar{d}) \in ((R_1 \cup W_1) \cup (R_2 \cup W_2))$ for some variable (o, d) . Therefore all commit actions $commit(R, W)$ such that $(o, d) \in (R \cup W)$ are totally ordered by the synchronizes-with order $\xrightarrow{sw}_{(\bar{o}, \bar{d})}$. Given that, each commit action $commit(R, W)$ corresponds to an atomic sequence of actions

$$\text{sync-sink}(\bar{o}_1, \bar{d}_1), \dots, \text{sync-sink}(\bar{o}_n, \bar{d}_n), \text{sync-source}(\bar{o}_1, \bar{d}_1), \dots, \text{sync-source}(\bar{o}_n, \bar{d}_n)$$

such that $(R \cup W) = \{(o_1, d_1), \dots, (o_n, d_n)\}$, in addition to the data read and write actions as indicated above.

Given the above definitions, a linearization of the projection of the happens-before ordering $\xrightarrow{hb}$ onto the *commit* actions corresponds to the atomic order of transactions as defined in [27]. Our treatment of transactions simply formalizes one possible semantics of transactions and does not introduce any extra synchronization or serialization to transaction implementations. In the transaction semantics we model using the happens-before relation, pairs of shared variable accesses where both accesses belong to some transaction are considered to be race-free. This reflects the fact that these accesses are managed by the STM implementation and the programmer and the dynamic race checker should not be concerned with them.

Other ways of specifying the interaction between transactions and the Java memory model can easily be incorporated into our definition of races. For instance, one can choose to define the synchronizes-with order to include *all* synchronization order edges between *commit* actions, or to include a synchronizes-with edge from transaction *commit*(*R*, *W*) to *commit*(*R'*, *W'*) if $R' \cap W \neq \emptyset$. The algorithms and tools presented in this thesis can easily be adapted to such alternative interpretations of software transactions.

```

Let  $\pi(i)$  be executed by thread  $t$ 
if  $\pi(i) = \text{commit}(R, W)$ 
  foreach  $(o, d) \in (\text{Addr} \times \text{Field})$ :
    if  $GLS(o, d) \cap (R \cup W) \neq \emptyset$ 
       $GLS(o, d) := GLS(o, d) \cup \{t\}$ 
    if  $(o, d) \in R \cup W$ :
      if  $GLS(o, d) \neq \emptyset$  and  $\{t, TL\} \cap GLS(o, d) = \emptyset$ 
        throw a DataRaceException on  $(o, d)$ 
       $GLS(o, d) := \{t, TL\}$ 
    if  $t \in GLS(o, d)$ 
      foreach  $(o', d') \in R \cup W$ :  $GLS(o, d) := GLS(o, d) \cup \{(\bar{o}', \bar{d}')\}$ 

```

Figure 5.3: The Goldilockset update rules for software transactions

Having defined the necessary orderings for data-race checking, for programs using software transactions, we extend the Goldilockset of a variable (o, d) as $GLS(o, d) \subseteq \text{Tid} \cup$

$Monitor \cup (Addr \times Volatile) \cup \bar{Tid} \cup (Addr \times \bar{Field})$, where $GLS(o, d)$ may now contain synchronization variables of the form $(\bar{o}, \bar{d})$ and a special value TL (Transaction Lock), in addition to thread identifiers, monitors, volatile variables, and fictitious variables for threads. The value TL models a fictitious global lock that is acquired and released by every transaction when performing its commit action. That is, every transaction produces the following sequence of actions:

acquire(TL) *commit*(R, W) **release**(TL)

The update rules in Figure 5.3 extends the rules in Figure 4.1. The intuitive interpretation of a Goldilockset $GLS(o, d)$ at every point of the execution is extended with the following:

- If TL is in $GLS(o, d)$, then the last access to (o, d) was performed inside a transaction. Hence, there will be no race on (o, d) if the next access is also performed inside a transaction.
- If a synchronization variable of the form $(\bar{o}', \bar{d}')$ is in $GLS(o, d)$, a thread can become an owner of (o, d) by accessing (o', d') inside a transaction. This is because the thread that last accessed (o, d) subsequently executed a transaction in which it accessed (o', d') , and this makes the two transactions accessing (o', d') be related with a synchronizes-with edge.

Given this interpretation of $GLS(o, d)$, we conclude that an access of (o, d) by a thread t outside any transaction is race-free if and only if at that point in the execution either $GLS(o, d)$ is empty or $\bar{t} \in GLS(o, d)$. After this access, $GLS(o, d)$ contains only t , representing the constraint that between this access and any future accesses to (o, d) by other threads, there must be synchronization actions to hand over ownership of (o, d) . Similarly, an access of (o, d) by a thread t inside a transaction is race-free if and only if at that point in the execution either $GLS(o, d)$ is empty or $\bar{t} \in GLS(o, d)$ or $TL \in GLS(o, d)$. After this access, $GLS(o, d)$ contains only $\bar{t}$ and TL , representing the constraint that the next access to (o, d) by a thread t' different from t must either be inside a transaction or there must be synchronization actions to hand over ownership of (o, d) to t' .

Execution	Goldilockset update rule applied	$GLS(x)$
Thread 1 (T1):		
<code>t1 = new Foo();</code>	Initialize the goldilockset	$\emptyset$
<code>t1.data = 42;</code>	First access to x	$\{T1\}$
<code>begin-atomic();</code>	No access to x	$\{T1\}$
<code>t1.nxt = head;</code>	No access to x	$\{T1\}$
<code>head = t1;</code>	No access to x	$\{T1\}$
<code>end-atomic();</code>	$T1 \in GLS(x) \rightarrow \text{add } o.\bar{n}xt \text{ and } \bar{head} \text{ to } GLS(x)$	$\{T1, o.\bar{n}tx, \bar{head}\}$
Thread 2 (T2):		
<code>begin-atomic();</code>	No access to x	$\{T1, o.\bar{n}tx, \bar{head}\}$
<code>iter = head;</code>	No access to x	$\{T1, o.\bar{n}tx, \bar{head}\}$
<code>iter != null</code>	No access to x	$\{T1, o.\bar{n}tx, \bar{head}\}$
<code>iter.data = 0;</code>	No access to x	$\{T1, o.\bar{n}tx, \bar{head}\}$
<code>iter = iter.nxt;</code>	No access to x	$\{T1, o.\bar{n}tx, \bar{head}\}$
<code>iter == null;</code>	No access to x	$\{T1, o.\bar{n}tx, \bar{head}\}$
<code>end-atomic();</code>	$\{\bar{head}, o.\bar{data}, o.\bar{n}tx\} \cap GLS(x) \neq \emptyset \rightarrow \text{add } T2 \text{ to } GLS(x)$	$\{T1, o.\bar{n}tx, \bar{head}, T2\}$
	$\{TL, T2\} \cap GLS(x) \neq \emptyset \rightarrow \text{Race-free access to } x$	$\{TL, T2\}$
	$T2 \in GLS(x) \rightarrow \text{add } \{\bar{head}, o.\bar{data}, o.\bar{n}tx\} \text{ to } GLS(x)$	$\{TL, T2, \bar{head}, o.\bar{data}, o.\bar{n}tx\}$
Thread 3 (T3):		
<code>begin-atomic();</code>	No access to x	$\{TL, T2, \bar{head}, o.\bar{data}, o.\bar{n}tx\}$
<code>t3 = head;</code>	No access to x	$\{TL, T2, \bar{head}, o.\bar{data}, o.\bar{n}tx\}$
<code>head = t3.nxt;</code>	No access to x	$\{TL, T2, \bar{head}, o.\bar{data}, o.\bar{n}tx\}$
<code>end-atomic();</code>	$\{\bar{head}, o.\bar{n}tx\} \cap GLS(x) \neq \emptyset \rightarrow \text{add } T3 \text{ to } GLS(x)$	$\{TL, T2, \bar{head}, o.\bar{data}, o.\bar{n}tx, T3\}$
	$T3 \in GLS(x) \rightarrow \text{add } \{\bar{head}, o.\bar{n}tx\} \text{ to } GLS(x)$	$\{TL, T2, \bar{head}, o.\bar{data}, o.\bar{n}tx, T3\}$
<code>t3.data++;</code>	$T3 \in GLS(x) \rightarrow \text{Race-free access to } x$	$\{T3\}$

Figure 5.4: Handling software transactions example

5.3 Revisiting Example 2: Handling Software Transactions

In this section, we use the program in Example 3 from Section 5.1.1 to demonstrate how Goldilocks handles synchronization due to transactions. Consider an execution of this program shown in Figure 5.4. This execution begins in a state in which `head = null`, and consequently, the linked list of `Foo` objects is empty. Let o denote the address of the object allocated by the first statement. The transaction of **Thread 1** happens-before the transaction of **Thread 2** because both transactions access the variables `head` (address denoted

by `&head`) and `o.data`. The transaction of **Thread 2** happens-before the transaction of **Thread 3** for exactly the same reason. Consequently, the accesses to `o.data` at `t1.data = 42` by **Thread 1**, at `iter.data = 0` by **Thread 2**, and at `t3.data++` by **Thread 3** are ordered by the happens-before relation. Goldilocks is able to detect these happens-before edges and verify that there is no data-race between the three accesses. Figure 5.4 shows the evolution of the Goldilockset for `o.data`. Data variables in the Goldilockset due to transactions are denoted using $\bar{bar}$ (e.g., $\bar{nxt}$). The figure also shows the Goldilockset update rules from Figure 5.3 applied at each point; the rules that successfully updates $GLS(x)$ are given in boxes. All Goldilockset updates at a commit point (`end-atomic()`) are performed atomically at one indivisible step.

5.4 Discussion: Adapting the Algorithm to Different STM Implementations

The work on defining semantics for software transactions is orthogonal to our work; our algorithm can be integrated with all the implementations we are aware of. We require that the semantics of software transactions to specify when a happens-before relationship exists between two transactions without reference to the particular transactions implementation. Our definition of data races in the presence of transactions was particularly influenced in part by [27], a study of the interaction between the synchronization induced by software transactions and weaker memory models. Following this study, we use the specification that there is a happens-before edge between transactions that access at least one common variable. Our primary concerns while detecting races in the presence of transactions were (i) to handle correctly the happens-before relationships guaranteed by the transaction implementation, as they concern the rest of the program execution and (ii) to avoid the runtime overhead and false alarms due to checking variable accesses and synchronization operations performed internally by the transactions implementation.

Our formulation of race conditions facilitates a modular check for races and strong atomicity for programs that use software transactions. The implementer of software transactions verifies that the implementation provides the happens-before edges in the transactions specification. Our race checker assumes that these happens-before edges are implemented properly and checks the race freedom of all accesses, but disregards the internal operation of the transactions implementation. The transaction implementation is only required to provide a

list of the shared variables accessed by each transaction and a commit point. If no `DataRaceException` is thrown, then sequential consistency and strong atomicity are guaranteed. Further, the specification rather than the implementation of transactions is used in order to make this check cheaper. In Section 7.2.1, we will demonstrate this way of handling transactions in our runtime on a hand-coded transactional data structure.

In this chapter, we showed how implementation-independent semantics for transactions, such as the ones in semantics for transactions in [27], can be used to detect races for programs that use software transactions with standard synchronization mechanisms of Java together. In the next chapter, we will describe our implementation features and optimizations that allow us to have `GOLDILOCKS` run at a moderate overhead.

Chapter 6

IMPLEMENTING GOLDBLOCKS

In this chapter, we will elaborate on how to implement the GOLDBLOCKS algorithm to achieve dynamic data-race detection at a tolerable runtime overhead. There are two published implementations of the GOLDBLOCKS algorithm, both of which monitor the execution at the Java bytecode level. At this level, each variable access or synchronization operation corresponds to a single bytecode instruction, and each bytecode instruction can be associated with a source code line and/or variable.

The first Goldilocks implementation, by the author of this thesis, was carried out in Kaffe [7], a clean-room implementation of the Java virtual machine (JVM) in C. In Kaffe, we integrated GOLDBLOCKS into the interpreting mode of Kaffe's runtime engine. While an implementation of our algorithm in the just-in-time (JIT) compilation mode would have been straightforward, it would have required a lot more code, so we chose to use the simpler interpreting mode for our proof-of-concept implementation. Implementing the algorithm in the JVM enables fast access to internal data structures of the JVM that manage the layout of object in the memory and using the efficient mechanisms that exist in the JVM internally, such as fast mutex locks. We also extended our implementation in Kaffe to provide `DataRaceException` support for executions with software transactions. The nature of the Goldilockset update rules and the implementation made it possible to integrate this feature without significant restructuring.

The second implementation of GOLDBLOCKS is by Flanagan and Freund and was carried out using the ROADRUNNER dynamic program analysis tool [31]. In ROADRUNNER, GOLDBLOCKS is implemented as a separate Java library and a program to be monitored is instrumented with this library through byte-code injection at the load-time of the program. This allows the algorithm to benefit from Java compiler optimizations and just-in-time compilation and to be portable to any JVM. Flanagan and Freund showed that this implementation is competitive with ours in Kaffe for most of the common benchmarks [9].

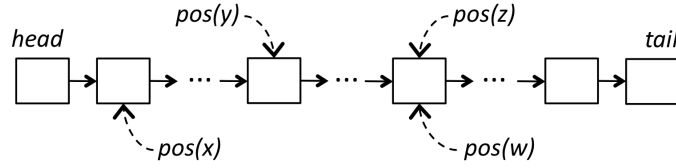


Figure 6.1: The synchronization event list

In this chapter, we will present the most important implementation features and optimizations that both realizations in Kaffe and ROADRUNNER make use of. For simplicity of presentation, we describe our implementation features based on the core algorithm given in Chapter 3. We also give the pseudo-code for the implementation of the algorithm that distinguishes read and write accesses. We defer the explanation of how transactions are handled in the implementation to Section 6.5 to make the initial presentation simpler.

Outline. Section 6.1 describes how we represent Goldilocksets implicitly and compute them lazily during the execution. Section 6.2 explains how the checks are performed in a decentralized manner. In Section 6.3, we describe the optimization in the implementation to reduce the memory and time cost. Section 6.4 gives the pseudo-code for the implementation and shows how to distinguish reads and writes in the implementation level; in Section 6.5 we describe how to extend this code to handle software transactions.

6.1 Implicit Representation and Lazy Evaluation of Goldilocksets

For programs with a large number of data variables, representing Goldilocksets explicitly for each data variable and implementing the GOLDDLOCKS algorithm as described in Figure 3.2 may have high memory and computational cost. First, maintaining an explicit individual Goldilockset data structure for each variable has a memory cost. Second, the update rules in Figure 3.2 require the Goldilockset $GLS(x)$ for *every variable* x to be updated (as opposed to only the accessed variables) for each synchronization event, which may have high computational cost. We avoid the memory cost by representing the Goldilocksets *implicitly* and the computational cost by evaluating Goldilocksets *lazily* as described below.

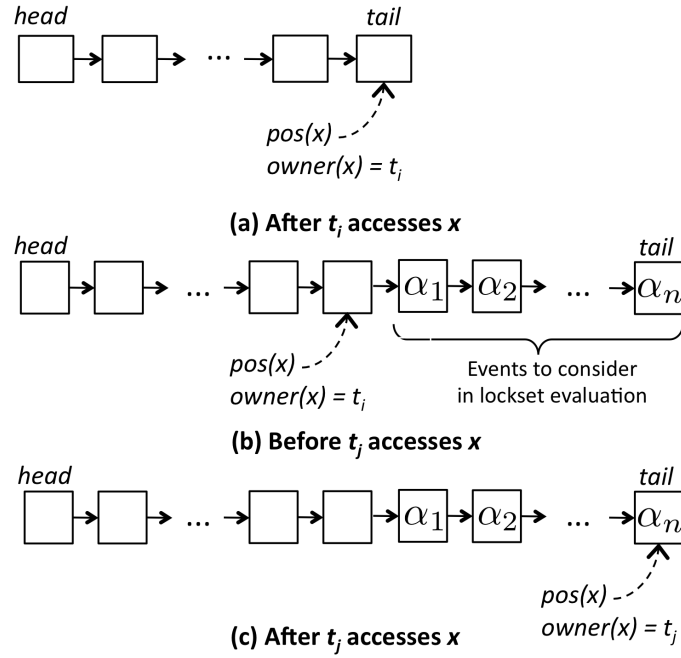
Instead of keeping a separate Goldilockset $GLS(x)$ for each variable x , we represent

$GLS(x)$ implicitly as long as no access to x happens. $GLS(x)$ is computed temporarily when an access happens and only enough to decide if the id of the currently accessing thread is in $GLS(x)$. After the access, the Goldilockset again becomes a singleton (containing only the currently accessing thread id), and we continue to represent it implicitly until the next access. For this, we keep the synchronization events in a single, global linked list called the *synchronization-event list* and represent this list by its *head* and *tail* pointers in Figure 6.1. The ordering of these events in the list is consistent with the program order $\xrightarrow{po}_t$ for each thread t and the synchronization orders $\xrightarrow{so}_o$ for each synchronization variable o . For Java, there is a total order on all synchronization operations, and the entries in the list are in this order. When a thread performs a synchronization action α , it must append a corresponding event to the synchronization-event list atomically with the event. In Kaffe, we make sure this is the case by modifying the implementations of the Java synchronization actions.

In order to represent $GLS(x)$, each variable x in the program is associated with two bits of information regarding the most recent access to x : $owner(x)$ stores the id of the thread that most recently accessed x , and $pos(x)$ points to the last synchronization event in the list that was taken into account when $GLS(x)$ was last computed.

Figure 6.1 shows four variables pointing to entries in the synchronization-event list. Figure 6.2 shows how the lockset $GLS(x)$ is computed using $owner(x)$ and $pos(x)$ when x is accessed. Here we assume the core algorithm presented in Figure 3.2 and compute only one Goldilockset at each access. As demonstrated in Section 6.4, when we distinguish read and write accesses to check write-after-read races, we may need to compute multiple Goldilocksets using a different $(owner(x), pos(x))$ pair per read.

- **6.2(a).** After each access to x by a thread t_i , $owner(x)$ is set to t_i , and $pos(x)$ is set to point to the *tail* of the synchronization event list.
- **6.2(b).** Right before an access to x by thread t_j , temporarily, we represent $GLS(x)$ explicitly. $GLS(x)$ is initialized to $\{t_i\}$ (since $owner(x) = t_i$) and is updated by processing the synchronization events between $pos(x)$ and *tail* (denoted by $\alpha_1, \dots, \alpha_n$) according to the rules 1 and 2 of Figure 3.2. This process stops either when t_j is added to $GLS(x)$ or the last event (α_n) is processed. In the former case, no race is reported according to the rule 3 of Figure 3.2. In the latter case, a race is reported since

Figure 6.2: Lazy evaluation of Goldilockset $GLS(x)$

$t_j \notin GLS(x)$ after the evaluation.

- **6.2(c).** After the race check, $owner(x)$ is set to t_j and $pos(x)$ is set to the *tail* of the synchronization event list.

6.2 Decentralized checking of races

The implementation does not use any extra threads for race detection. JVM's implement threads by associating each program thread with a thread of the underlying platform. The algorithm is performed in a decentralized manner by instrumented threads of the program being checked.

Each thread carries out itself the steps of the algorithm pertaining to the data accesses and synchronization operations it performs. For each data variable x , we use a unique lock $KL(x)$ in order to make atomic the Goldilockset update and the race-freedom check for each access to x as sketched below:

$acquire(KL(x)); RaceCheck(x); Access(x); release(klock(x))$

Before an access to x , the thread acquires $KL(x)$, computes $GLS(x)$ lazily as described above, and performs the race-freedom check. After checking the race, the thread performs the access to x and then releases $KL(x)$. Thus, the race checking and the actual access are atomic, and all the race-freedom checks for x are serialized.

An alternative approach to the above one is to release the lock $KL(x)$ before performing the access to x as sketched below:

$$acquire(KL(x)); RaceCheck(x); release(klock(x)); Access(x)$$

As a result, for two accesses to x , the ordering of race checks and actual accesses may be different if the accesses are not ordered by the happens-before relation; however, the soundness and the precision of the algorithm are not affected. Since accesses to x are not protected by $KL(x)$, threads can access x concurrently, and this allows race checking to have less effect on the concurrency characteristics of the program.

6.3 Performance Optimizations

6.3.1 Short Circuit Checks

The computational cost of race checking for an access to x depends on the number of operations processed during the lazy evaluation of $GLS(x)$. A cheap-to-evaluate sufficient condition for a happens-before edge between two accesses to a variable can reduce race-detection overhead by eliminating the need to compute $GLS(x)$. We make use of two such conditions, called *short-circuit checks*, and bypass the traversal of the synchronization event list when these checks succeed. Both short-circuit checks are performed at constant time.

Recall from Section 6.2 that we serialize all the race checks and accesses to each variable x using $KL(x)$. First, when last two accesses (in this serialization) to a shared variable are performed by the same thread t , the happens-before relationship is guaranteed by the program order of t . This is detected by checking whether $owner(t)$, the last accessing thread, is the same as the thread performing the current access.

In the second short-circuit check, we determine whether the variable x is protected by the same lock during the last two accesses to x . For this, we associate with each variable x a lock $alock(x)$. After each access to x , $alock(x)$ is randomly selected among the locks held

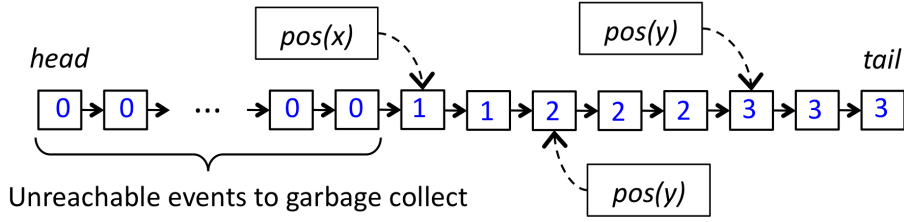


Figure 6.3: Garbage collection of synchronization events

by the currently accessing thread. When a thread t accesses x and if $alock(x)$ is held by t , then that access is race free.

6.3.2 Direct Ownership Transfer

A sound but imprecise optimization is to consider only the subset of synchronization events executed by the current and last accessing thread when examining the portion of the synchronization event list between $pos(x)$ and $tail$.

Recall the Goldilockset computation illustrated in Figure 6.2. At step (b), $GLS(x)$ is evaluated in two phases. In the first phase, after $GLS(x)$ is initialized to $\{t_i\}$, only the actions from $\alpha_1, \dots, \alpha_2$ that were inserted by either t_i ($owner(x)$) or t_j , the currently accessing thread. This stops either when t_j is added $GLS(x)$ or the last event (α_n) is processed. In the former case, there is a direct ownership between t_i and t_j , and thus, no race is reported. In the latter case, $GLS(x)$ is reinitialized to $\{t_i\}$ and is evaluated again, now considering all the actions $\alpha_1, \dots, \alpha_2$. This second phase checks whether there is any happens-before relationship between the accesses of t_i and t_j established through actions of other threads.

This strategy does not give a constant time check, but we found that it saves on the runtime of Goldilockset computation when the happens-before edge between $pos(x)$ and $tail$ is immediate, i.e., is accomplished by a direct transfer of ownership from one thread to another through a single synchronizes-with edge.

6.3.3 Garbage Collection of Synchronization Event List

The synchronization events list is periodically garbage-collected when there are entries in the beginning of the list that are not relevant for the Goldilockset computation of any variable.

This is the case when an entry in the list is not reachable from $pos(x)$ for any data variable x , and is tracked by maintaining incremental reference counts for each list entry.

We keep track of this information by maintaining for every event in the list a reference count. The reference count indicates the number of data variables x that can reach that event by starting from $pos(x)$ and going through the tail of the synchronization event list. For example, in Figure 6.3 the numbers in the boxes representing synchronization events indicate the reference counts of the events. In the figure, the prefix of the event list up to the event pointed by $pos(x)$ have reference counts of zero, thus are not reachable by any data variable. We discard such prefixes and reset *head* to point to the first event with nonzero reference count. Our implementation either keeps these discarded events to re-use them later or explicitly calls the JVM’s garbage collector to release the memory space allocated by the events.

In our GOLDDILOCKS implementation, we use a “delta” encoding to implicitly represent the reference count of each entry in the synchronization list. Instead of the number of data variables that can reach the entry, we keep track of the number of data variables whose *pos* pointers directly refer to the entry. This encoding makes it cheaper to maintain the reference counts, since it only requires two updates when $pos(x)$ for a variable x changes: decrementing the counter of old $pos(x)$ and incrementing the counter of new $pos(x)$.

6.3.4 Partially-Eager Evaluation

While running GOLDDILOCKS on long executions, sometimes the synchronization event list gets too long and it is not possible to garbage-collect the event list as described below. Figure 6.4(a) illustrates such a case where the variable x is accessed early in an execution but is not used afterwards. In this case, the reference count for a long prefix of the list is nonzero and cannot be garbage collected. This phenomenon is a side effect of the initial GOLDDILOCKS implementation performing “completely lazy” evaluation. We address this problem by “partially-eager” Goldilockset evaluation. We move $pos(x)$ forward towards the *tail* to a new position $pos'(x)$, and partially evaluate a Goldilockset $GLS(x)$ of x by processing events (i.e., running *ApplyLocksetRules* on them) between $pos(x)$ and $pos'(x)$. These are the events with reference count 1 in Figure 6.4(a). Figure 6.4(b) shows the result of partially-eager evaluation: The events between $pos(x)$ and $pos'(x)$ got zero reference count, and we

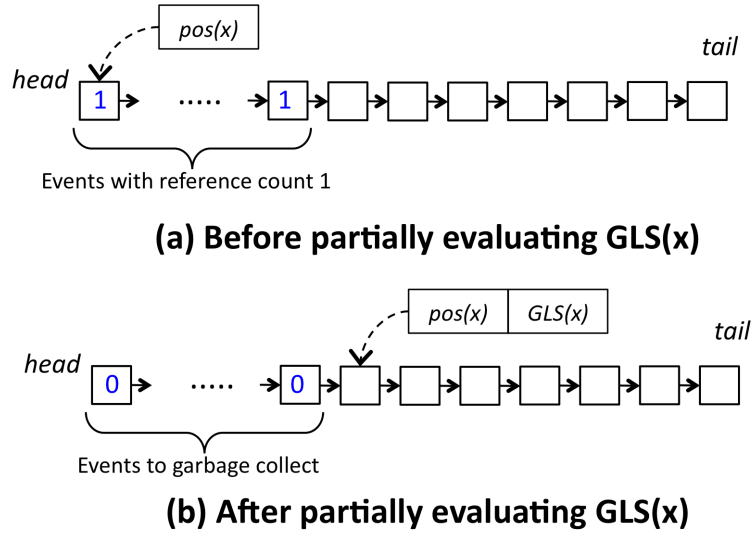


Figure 6.4: Partially-eager evaluation of Goldilocks

can now garbage-collect these events. The resulting Goldilockset is associated with x and kept alive until a subsequent access to x . During the next access the evaluation of $GLS(x)$ starts from the stored Goldilockset, not from $\{owner(x)\}$.

This partially-eager Goldilockset evaluation scheme provides a running time-memory trade off and is necessary for continuously-running programs which may have very long synchronization event lists otherwise. In our implementation in Kaffe, we trigger the partially-eager Goldilockset computation scheme to trim the first 10% of the entries in the synchronization event list when garbage collection is triggered in the Kaffe JVM. We explicitly trigger a garbage collection when the event list grows longer than one million entries. Since we do not maintain reverse pointers from events in the list to data variables, it is necessary to perform the partially-eager computation of Goldilocks when we garbage-collect events with non-zero reference counts.

6.3.5 Sound Static Race Analysis

The runtime overhead of race detection is directly related to the number of data variable accesses checked and the synchronization events that occur. In the worst-case, this overhead is proportional to the product of the following two quantities: (1) the sum of the number of

synchronization events and shared variable accesses, (2) the number of data variables in the execution. This is because in the worst-case, the entire synchronization event list may have to be traversed for each shared variable (in a single step or multiple steps depending on the number of accesses to that variable). This overhead is amortized over the total number of accesses. In practice, we do not encounter this worst-case behavior and observe a constant-time overhead per access. To reduce the number of accesses checked at runtime, we use existing static analysis methods at compile time to determine accesses or access pairs that are guaranteed to be race-free.

We worked with two static analysis tools for this purpose: a newer version of the Chord tool [11] and the RccJava static race detection tool [12]. The output of RccJava is a list of fields that may be involved in a race condition. The output of Chord is a list $\mathcal{R}$ of pairs of accesses (line numbers in the source code) that *may* be involved in a race, i.e., if any execution of the program ever produces a race, the pair of accesses (α_1, α_2) is guaranteed to be in $\mathcal{R}$. Our runtime makes use of $\mathcal{R}$ by parsing Chord's output and inferring from it the sets of object fields ($\mathcal{F}$) and methods ($\mathcal{M}$) that are guaranteed to never be involved in a race. It then annotates the Java class files using the reserved bits of the access flags of classes, fields and methods to enable/disable race checking on the particular class, field or method.

It is also possible to make further use of the output of a tool such as Chord in the following ways:

- One can compute using $\mathcal{R}$ the set of all accesses that are guaranteed to never take part in a race. One can then turn off race detection for those accesses.
- For every pair of consecutive accesses (α_1, α_2) to a variable that occur during an execution, one can define another short-circuit check: determine whether $(\alpha_1, \alpha_2) \in \mathcal{R}$.

These approaches can be thought of as additional short-circuit checks. It is straightforward to integrate the first check into our implementation by associating a flag with each bytecode instruction that indicates whether it represents an access to be checked or not. The second check would involve that the $\mathcal{R}$ be loaded along with the byte-code for classes, stored in a hashtable, and that, for each pair of accesses, a lookup be performed in this hashtable.

We have not implemented these short-circuit checks since the reduction in overhead accomplished by turning off race checking for fields was satisfactory for a demonstration-of-concept implementation.

6.4 The Pseudo-code of the Implementation

The above sections described the main implementation features of our implementation and performance optimizations. In this section, we illustrate how these features can be implemented by giving the pseudo-code of our implementation. The pseudo-code also shows how to use the implementation features discussed above for the extended algorithm that distinguishes reads and writes as described in Section 3.4.

```

record Cell {
  thread: Tid;
  action : Action;
  next: ref(Cell);
}

record Info {
  pos: ref(Cell);
  owner: Tid;
  alock: Addr;
  gls:  $\emptyset(Addr \times Volatile \cup Data) \cup Tid \cup \{TL\}$ ;
  xact: Boolean;
}

head, tail: ref(Cell);
ReadInfo:  $(Addr \times Data \times Tid) \longrightarrow Info$ ;
WriteInfo:  $(Addr \times Data) \longrightarrow Info$ ;

Initially head := new Cell; tail := head;
ReadInfo := EmptyMap; WriteInfo := EmptyMap;

```

Figure 6.5: Prelude for the pseudo-code in Figure 6.6

The pseudo-code for the Goldilocks implementation is given in Figure 6.5 and Figure 6.6. We defer the explanation of how transactions are handled in the implementation to the next section to make the initial presentation simpler.

In Figure 6.5–6.6 the synchronization event list is represented by its *head* and *tail* pointers. Synchronization events are represented by the *Cell* data structure, which stores the synchronization action kind and the thread performing the action. When a thread performs a synchronization action, it atomically appends (see *Enqueue-Synch-Event*) the correspond-

```

    Enqueue-Synch-Event ( $t, \alpha$ ):
1    $tail \rightarrow thread := t$ ;
2    $tail \rightarrow action := \alpha$ ;
3    $tail \rightarrow next := new Cell()$ ;
4    $tail := tail \rightarrow next$ ;

    Check-Happens-Before ( $info_1, info_2$ ):
1   if ( $info_1.xact \wedge info_2.xact$ )
2       return; // no race for transactional variables
3   if ( $(info_2.owner \neq info_1.owner)$ 
4        $\wedge (info_1.alock$  is not held by  $info_2.owner)$ ) {
5       Apply-Lockset-Rules ( $info_1.gls, info_1.pos, info_2.pos, info_2.owner$ ) ;
6        $info_2.alock :=$  (choose randomly a lock held by  $info_1.owner$ );
7   }

    Handle-Action ( $t, \alpha$ ):
1   if ( $\alpha \in \{\text{acquire}(o), \text{release}(o), \text{fork}(u), \text{join}(u), \text{volread}(o, v), \text{volwrite}(o, v), \text{start}(t), \text{end}(t), \text{alloc}(o)\}$ ) {
2       Enqueue-Synch-Event ( $t, \alpha$ );

3   } else if ( $\alpha = \text{read}(o, d)$ ) {
4        $info := new Info ()$ ;
5        $info.owner := t$ ;
6        $info.pos := tail$ ;
7        $info.xact := Is-In-Transaction (t)$ ;
8        $info.gls := \{t\}$ ;
9        $ReadInfo(o, d, t) := info$ 
10      Check-Happens-Before ( $WriteInfo(o, d), info$ );

11  } else if ( $\alpha = \text{write}(o, d)$ ) {
12       $info := new Info ()$ ;
13       $info.owner := t$ ;
14       $info.pos := tail$ ;
15       $info.xact := Is-In-Transaction (t)$ ;
16       $info.gls := \{t\}$ ;
17      for each ( $t \in Tid$ )
18          if ( $ReadInfo(o, d, t) \neq null$ )
19              Check-Happens-Before ( $ReadInfo(o, d, t), info$ );
20      if ( $WriteInfo(o, d) \neq null$ )
21          Check-Happens-Before ( $WriteInfo(o, d), info$ );
22       $WriteInfo(o, d) := info$ 
23      for each ( $t \in Tid$ )  $ReadInfo(o, d, t) := null$ 

24  } else if ( $\alpha = \text{commit}(R, W)$ ) {
25      Enqueue-Synch-Event ( $t, \alpha$ );
26      for each  $\tilde{\alpha}$  in ( $W \cup R$ )
27          Handle-Action ( $t, \tilde{\alpha}$ ); // Check race freedom as regular access
28      }
29  }

```

Figure 6.6: Pseudo-code for the GOLDILOCKS implementation

ing cell to the synchronization event list. This is accomplished by modifying the code in Kaffe implementing synchronization actions so that events are inserted into the update list in the synchronization order as implemented in Kaffe. This required the synchronization actions and *Enqueue-Synch-Event* to be performed atomically in Kaffe. For this purpose, we protected *Enqueue-Synch-Event* with the same internal synchronization mechanism that Kaffe employs to implement Java synchronization primitives. For volatile variables, Kaffe simply relied on the synchronization of accesses to the memory by the underlying hardware, therefore, we had to create one extra explicit compare-and-swap based spinlock to make atomic volatile variable accesses and *Enqueue-Synch-Event*.

The race-freedom check is implemented in a decentralized fashion. For each action α that a thread t performs, it calls *Handle-Action*(t, α). For each shared data variable (o, d) , to serialize the Goldilockset update and race-freedom checks for each access to (o, d) , our implementation uses a unique lock $KL(o, d)$. Before a thread accesses (o, d) , it acquires $KL(o, d)$ and performs the Goldilockset update (computes the Goldilockset associated with the access) and race-freedom check explained in Theorem 1. In this way, each thread carries out the race-freedom check for the accesses it performs, and these checks are linearly ordered by $KL(o, d)$. The implementation of the variable access in Kaffe is not protected by $KL(o, d)$.

The record *Info* corresponds to an access to a data variable (o, d) during the execution. The *owner* field of *Info* stores the id of the thread performed the access, and the *pos* field stores a pointer into the synchronization event list, to the *cell* representing the last synchronization event that the access comes after. The *gls* field contains the Goldilockset of the variable just after the access. The fields *alock* and *xact* will be explained below while discussing the short-circuit checks and the transactions.

After an access to a variable, a new *Info* instance *info* that represents the current access is created. *info.pos* is simply set to the current tail of the list (see lines 6,14 of *Handle-Action*). At this point, according to the variable access rule in Figure 3.2, the Goldilockset of the variable should contain only the id of the currently accessing thread. *info.gls* is set after the access in order to reflect this (see lines 8 and 16 of *Handle-Action*). This *info* instance is referred to as *info₂* by *Check-Happens-Before*.

The *Check-Happens-Before* procedure determines whether there is a happens before relationship between two accesses to (o, d) represented by two *info* data structures: *info₁*

represents the last access to (o, d) and $info_2$ —referred to as *info* in the body of *Handle-Action*—represents the current access to (o, d) . Before applying the Goldilockset update rules of Figure 3.2, *Check-Happens-Before* first applies the short-circuit checks described in Section 6.3.1: The first constant-time short-circuit check is determines whether the *thread* field of the *info* data structure is the same as the thread performing the current access (see line 3 of *Check-Happens-Before*). The second constant-time short-circuit check determines whether a random element of $GLS(o, d)$ at the last access, kept in the *alock* field of *Info*, is also held by the current thread (see line 4 of *Check-Happens-Before*). Our experimental results indicate that the short-circuit checks succeed most of the time, and the Goldilockset update rules are only applied in the case of more elaborate ownership transfer scenarios. If all the short-circuit checks fail to prove the happens-before edge, the Goldilockset of the variable is computed lazily in *Apply-Lockset-Rules*.

Apply-Lockset-Rules, which is called at line 5 of *Check-Happens-Before*, implements the lazy evaluation mechanism for Goldilocksets described in Section 6.1. *Apply-Lockset-Rules* computes the Goldilockset $GLS(o, d)$ of (o, d) by applying to the Goldilockset of the last access, $info_1.gls$, the Goldilockset update rules in Figure 3.2 for the events in the synchronization list between $info_1.pos$ and $info_2.pos$, the current tail of the synchronization list representing the position of the current access. *Apply-Lockset-Rules* traverses the events between $info_1.pos$ and $info_2.pos$ in two phases. During the first phase, it only traverses the events that are created by the last and currently accessing threads in order to check a direct ownership transfer between the two threads as described in Section 6.3.2. If this cannot show the race-freedom of the current access, in the second phase *Apply-Lockset-Rules* traverses all the events between $info_1.pos$ and $info_2.pos$.

In the usual case, $info_1.gls$ starts from a singleton set containing the id of the last accessing thread, and $info_1.pos$ indicates the point where the last access happened. However, as described in Section 6.3.4, the partially-eager computation of Goldilocksets in between the accesses may overwrite $info_1.gls$ and $info_1.pos$ with a partially computed Goldilocksets and a later point than the actual access point, respectively. *Apply-Lockset-Rules* raises a **DataRaceException** if the computed Goldilockset does not contain the id of the current accessing thread. After the data-race checking the *info* structure is used to store the information about the current access. Note that *Handle-Action* sets $info.gls$ to $\{t\}$, the current

accessing thread, so, if no partially-eager evaluation happens, the Goldilocks computation at a later access starts from $\{t\}$.

To distinguish between read and write accesses, the implementation maintains the *Info* records for the last write access to each shared data variable (o, d) ($WriteInfo(o, d)$) and the last read access to (o, d) by thread t ($ReadInfo(o, d, t)$) if this access came after the last write access. These maps are updated after each access to (o, d) . Note that, for each data variable, there are potentially as many pointers into the synchronization event list from *ReadInfo* as the number of active threads, although, in practice, we rarely encounter this situation. In this case, instead of checking the happens-before edge between any two accesses to (o, d) , our algorithm checks 1) for each read access whether it happened before $WriteInfo(o, d)$, and 2) for each write access whether it happened before $ReadInfo(o, d, t)$ for each thread t .

6.5 Handling Transactions in the Implementation

To detect races at runtime, our approach requires a transaction manager to provide or make possible for the runtime to collect for each transaction $commit(R, W)$ the sets R and W and the place of commit point of the transaction in the global synchronization order. Note that transaction implementations need to ensure such an order to implement the required semantics correctly, thus to provide the latter information, transaction implementations do not need to perform any additional synchronization. For the transaction implementations in LibSTM [28] and LibCMT [26], this information is readily available to the runtime, and can be collected easily in SXM [30] at runtime.

In our implementation (Figure 6.6), when a transaction commit action is encountered, we first insert the commit action α (that contains the list of read and write accesses R and W) as a synchronization action into the synchronization event list. Then, all shared variable reads and writes within the transaction (i.e., in R and W) are checked for race-freedom (lines 24-28 of *Handle-Action*) in the same way as accesses outside transactions. Note that the collection of R and W is done in a distributed fashion by each transaction and does not introduce extra synchronization between transactions. We use the *xact* field of *Info* in order to indicate whether an access happened inside a transaction or not (see lines 7,15 of *Handle-Action*). *xact* is used in another short-circuit check (line 1 of *Check-Happens-Before*); as long as the accesses to (o, d) are inside transactions, the Goldilockset computation is skipped.

Notice that, we do not use *TL*, the global lock used in Section 5.2. Instead, setting the *xact* field of *Info* to **true** (line 15 of *Handle-Action*) corresponds to adding *TL* to the Goldilockset of the variable, and checking whether *xact* field is **true** (line 1 of *Check-Happens-Before*) corresponds to checking whether *TL* is in the Goldilockset of the variable.

Note that all actions that are internal to the operation of the transaction manager and invisible to the programmer are ignored by our implementation. In practice, we accomplish this by using two marker actions that indicate the beginning and the end of a transaction. Our race checking implementation ignores all actions between these marker actions except the commit action. This includes actions performed by threads that are launched and joined by the transaction implementation. This way of handling the happens-before relationships induced by transactions avoids processing the synchronization operations performed by a transactions manager that has already been proven correct. It infers exactly the desired set of happens-before edges, and accomplishes this at less of a computational cost than treating transactions implementations as part of the application program. This approach can be viewed as a modular way of checking race-freedom and thus sequential consistency. The implementor of software transactions guarantees the happens-before edges between transactions as described in Section 5.2 and race-freedom for accesses within transactions. Our Java runtime performs the race-freedom check for the rest of the accesses using the happens-before edges provided by the transactions implementation.

In this chapter, we elaborated on how to implement the GOLDILOCKS algorithm to achieve data-race detection at low runtime overhead. The techniques described in this chapter were used in both our implementation in the Kaffe JVM and the ROADRUNNER framework. We explained how we implicitly represent Goldilocksets and compute them lazily at the access point. We introduced optimizations on these features including constant-time short-circuit checks and explained why and how we garbage collect event list and partially-eager evaluate Goldilocksets. Finally we presented detailed pseudo-code that illustrates the main data structures and the algorithms we use while implementing GOLDILOCKS. We also pointed out how our implementation interacts with the transaction implementation and represents information about the transactional accesses to a variable. In the next chapter, we will report our experience with our implementation in the Kaffe JVM.

Chapter 7

EXPERIMENTS WITH GOLDSILLOCKS

In this chapter, we will present the results of two experimental studies using our GOLDSILLOCKS runtime. In the first study, we used an earlier version of the runtime to compare the GOLDSILLOCKS algorithm with two other well-known algorithms, Eraser [14] and vector clocks [18]. In the second study, we used the latest version of our runtime to show the effect of the implementation optimizations on reducing the In practice, though, we do not encounter worst-case behavior and observe a constant-time overhead per access.

Outline. Section 7.1 gives the result of the former study in which we compare GOLDSILLOCKS, Eraser, and vector clocks. Section 7.2 reports on our latest study using the GOLDSILLOCKS runtime. In Section 7.3 we discuss the overheads of the GOLDSILLOCKS and FASTTRACK implementations in the ROADRUNNER framework.

7.1 Comparing Goldilocks with Eraser and Vector Clock

In order to evaluate the performance of our algorithm, we ran the instrumented version of the Kaffe JVM on a set of benchmarks. The version of the GOLDSILLOCKS implementation used the lazy evaluation technique described in Section 6.1 and the short-circuit checks described in Section 6.3.1. However we had not implemented the partially-eager evaluation, the garbage collection of events, and the static pre-analysis of race-free accesses at the time of this experiment.

In order to concentrate on the races in the applications, we disabled checks for fields of the standard library classes. Arrays were checked by treating each array element as a separate variable. We first present our experiments and then discuss their results in Section 7.1.2.

In order to compare our algorithm with traditional lockset and vector clock algorithms, we implemented a basic version of the Eraser algorithm [14] that we call Basic-Eraser and a vector clock based algorithm similar to the one used by Trade [17]. Where possible, we

used the same data structure implementations while implementing the three algorithms. For Basic-Eraser, we used the same code for keeping and manipulating locksets that we developed for GOLDILOCKS.

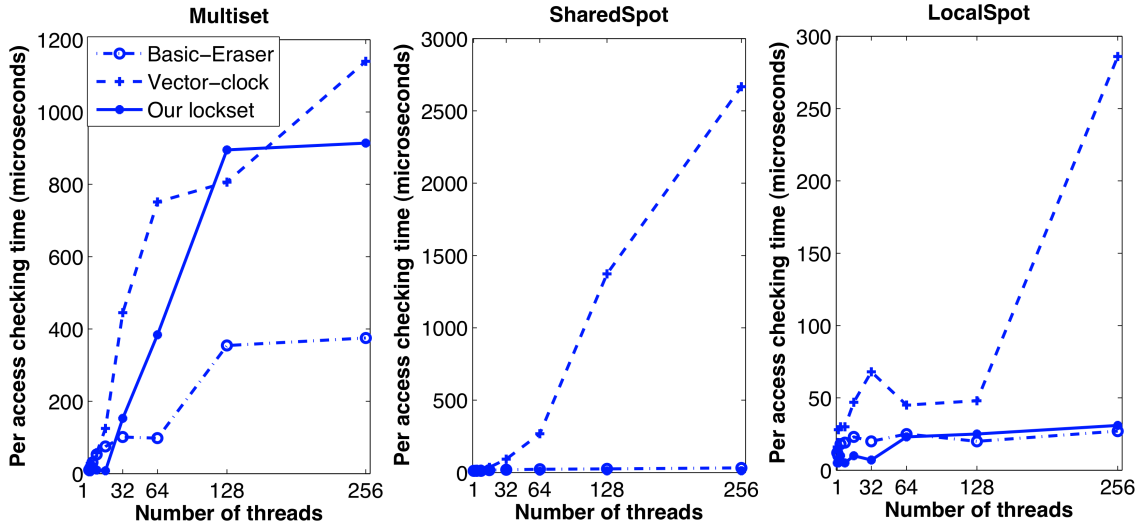


Figure 7.1: Per access race checking time against the increasing number of threads

7.1.1 Benchmarks

Microbenchmarks

The **Multiset** microbenchmark consists of a number of threads accessing a multiset of integers concurrently by inserting, deleting and querying elements to/from it. The **SharedSpot** benchmark illustrates the case in which a number of integers, each of which is protected by a separate unique lock, are accessed concurrently by a number of threads for applying arithmetic operations on them. The **LocalSpot** benchmark is similar to **SharedSpot** but each variable is thread-local. The code for these microbenchmarks is given in Figure 7.2. We ran experiments parameterizing the microbenchmarks with the number of threads starting from 1 and doubling until 256. Figure 7.1 plots for three algorithms the average time spent for checking each variable access against increasing number of threads.


```

public class SharedSpot implements Runnable{

    SharedVar[] vars;
    int anum;

    public SharedSpot(int vnum, int anum) {
        synchronized(this) {
            this.vars = new SharedVar[vnum];
            for(int i = 0; i < vnum; ++i) {
                this.vars[i] = new SharedVar();
            }
            this.anum = anum;
        }
    }

    public void run() {
        int num = 0;
        SharedVar var = null;
        synchronized(this) {
            num = this.anum;
        }
        for(int i = 0; i < num; ++i) {
            synchronized(this) {
                var = this.vars[i];
            }
            var.access();
        }
    }

    public static void main(String args[]) {
        int tnum = Integer.parseInt(args[0]);
        int anum = Integer.parseInt(args[1]);
        int vnum = Integer.parseInt(args[2]);
        SharedSpot spot = new SharedSpot(vnum, anum);

        Thread[] threads = new Thread[tnum];
        for(int i = 0; i < tnum; ++i) {
            threads[i] = new Thread(spot);
        }
        for(int i = 0; i < tnum; ++i) {
            threads[i].start();
        }
        for(int i = 0; i < tnum; ++i) {
            try {
                threads[i].join();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        System.out.println("Finished.");
    }
}

public class LocalSpot implements Runnable{

    LocalVar[] vars;
    int anum;
    int vnum;

    public LocalSpot(int vnum, int anum) {
        this.anum = anum;
        this.vnum = vnum;
    }

    public void run() {
        this.vars = new LocalVar[this.vnum];
        for(int i = 0; i < this.vnum; ++i) {
            this.vars[i] = new LocalVar();
        }
        int i = 0;
        int j = 0;
        while(i < anum) {
            vars[j].access();
            ++i;
            ++j;
            if(j == vars.length) j = 0;
        }
    }

    public static void main(String args[]) {
        int tnum = Integer.parseInt(args[0]);
        int anum = Integer.parseInt(args[1]);
        int vnum = Integer.parseInt(args[2]);
        Thread[] threads = new Thread[tnum];
        for(int i = 0; i < tnum; ++i) {
            threads[i] = new Thread(new LocalSpot(vnum, anum));
        }
        for(int i = 0; i < tnum; ++i) {
            threads[i].start();
        }
        for(int i = 0; i < tnum; ++i) {
            try {
                threads[i].join();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        System.out.println("Finished.");
    }
}

```

Figure 7.2: The SharedSpot and LocalSpot microbenchmarks.

Large Benchmarks

We used six benchmark programs commonly used in the literature to compare the performance of the three algorithms on large programs: **Raja**¹ is a ray tracer ($\approx 6K$ lines). **SciMark**² is a composite Java benchmark consisting of five computational kernels (≈ 2300 lines). Four of our benchmarks are from the Java Grande Forum Benchmark Suite³. They are **moldyn**, a molecular dynamics simulation (≈ 650 lines), **raytracer**, a 3D ray tracer (≈ 1200 lines), **montecarlo**, a Monte Carlo simulation ($\approx 3K$ lines) and **sor**, a successive over-relaxation program (≈ 220 lines).

Table 7.1 presents the performance statistics of the three algorithms on the benchmark programs. The purpose of this batch of experiments is to contrast the overhead that each of the three approaches incur while checking for races. In this batch of experiments, race checking for a variable was *not* turned off after detecting a race on it, as would be the case in normal usage of a race detection tool. The purpose of this was to enable a fair comparison between algorithms. On this set of benchmarks, Basic-Eraser conservatively declared false races on many variables early in the execution. If the race checking on these variables had been turned off after Basic-Eraser detects a race on them, Basic-Eraser would have ended up doing a lot less work and checking a lot fewer accesses than the other two approaches, especially since these variables are typically very likely to have races on them later in the execution as well. This would have made the overhead numbers difficult to compare.

Checks for all the indices of an array were disabled when a race was detected on any index of the array. This is also reasonable because it is highly probable in most cases that a race in an index is an early warning for future races on other indices of the array.

In Table 7.1, we give the number of threads created in each program under the name of the benchmark. The column titled “Uninstrumented” reports the total runtime of the program in the uninstrumented JVM, and the total number of variable accesses (fields+array indices) performed at runtime. Note that, we ignored accesses to the variables in the standard library classes. Each column for an algorithm presents, for each benchmark, the total execution time

¹Raja can be obtained at <http://raja.sourceforge.net/>.

²Scimark can be obtained at <http://math.nist.gov/scimark2/>.

³Java Grande Forum Benchmark Suite can be obtained at http://www.epcc.ed.ac.uk/computing/research_activities/java_grande/threads.html.

and the slowdown ratio of the program with instrumentation. The time values are given in seconds. The slowdown ratio is the ratio of the difference between the instrumented runtime and the uninstrumented runtime to the uninstrumented runtime. The number of variable accesses checked for races is important for assessing the amount of work carried out by the algorithm during execution and average checking time for each variable access.

	Uninstrumented	Vector Clock	Basic-Eraser	GOLDILOCKS
Benchmark	Runtime (sec.)	Runtime (sec.)	Runtime (sec.)	Runtime (sec.)
# threads	# accesses	Slowdown	Slowdown	Slowdown
Raja	8.6	145.1	105.9	70.2
3	5979629	15.7	11.1	7
SciMark	28.2	51.3	46.1	33.1
7	3647012	0.8	0.6	0.2
moldyn	11.2	195	138.9	92.8
7	8610585	16.3	11.3	7.2
raytracer	1.9	122.8	79.8	50
7	5299350	63.1	40.6	25.1
montecarlo	5.7	243.8	160	117.5
7	10491747	41.4	26.8	19.4
sor	27.2	145.9	157.5	107
7	7696597	4.3	4.7	2.9

Table 7.1: Runtime statistics of the benchmark programs

Table 7.2 lists the results of our experiments with GOLDILOCKS where checks for fields on which a race is detected are disabled. This is a more realistic setting to judge the overhead of our algorithm in absolute terms. The measurements reported in the first three rows are the same as the ones in Table 7.1, taken without disabling any checks. The next three rows give the runtime statistics when we followed the approach described above.

7.1.2 Discussion about the Experimental Results

The plots in Figure 7.1 show per access checking times of the three algorithms. The very low acceleration in the per access runtime overhead of our algorithm and Eraser in the

Algorithm	Raja	SciMark	moldyn	raytracer	montecarlo	sor
Runtime	70.2	33.1	92.8	50	117.6	107
Slowdown	7	0.2	7.2	25.1	19.4	2.9
# checks	5979629	3647012	8610585	5299350	10491747	7696597
Runtime*	65.8	35.5	57.0	17.6	111.2	63.8
Slowdown*	6.5	0.2	4	8.2	18.3	1.3
# checks*	5979629	4104754	5268021	1884836	10484544	3416928

* Results after disabling checks to the fields.

Table 7.2: Runtime statistics when fields with races detected on them are disabled

SharedSpot and **LocalSpot** examples is noteworthy. Short circuit checks in our algorithm allow constant time overhead for thread-local variables and variables protected by a unique lock. This makes our algorithm asymptotically better than the vector clock algorithm.

The runtime statistics in Table 7.1 indicate that **GOLDILOCKS** performs better than the vector clock algorithm for large-scale programs. As the number of checks done for variable accesses are the same, we can conclude that per variable access checking time of our lockset algorithm on average is less than the vector clock algorithm.

SciMark, **moldyn** and **sor** are well-synchronized programs with few races and a simple locking discipline. Thus the short circuit checks mostly succeed and the overhead of the lockset algorithm is low. However, more elaborate synchronization policies in **Raja**, **raytracer** and **montecarlo** caused long runs of the lockset algorithm, thus the slowdown ratio increases. These programs have a relatively high number of races.

The results indicate that our algorithm works as efficiently as Basic-Eraser while Basic-Eraser can not handle all the synchronization policies used in the benchmarks. The main reason for our algorithm performing slightly better in our experiments is the fact that Basic-Eraser does lockset intersections while checking the accesses. Intersection is fundamentally an expensive operation. Our algorithm, on the other hand, requires insertions and lookups, which can be implemented in constant amortized time. Clearly, a more optimized implementation of Eraser would have performed better. The goal of this comparison is to demonstrate that our algorithm does not have significantly more cost than other lockset algorithms.

Disabling the checks for accesses to fields on which races were detected dramatically decreases the number of accesses to be checked against races, thus the total runtime of the instrumented program. This can be seen from Table 7.1. For the benchmarks `moldyn`, `raytracer` and `sor`, the differences in the number of accesses point to this effect.

The experiments reported in this section are performed using a preliminary version of the implementation and only to evaluate the performance of GOLDILOCKS over other algorithms. The implementation of GOLDILOCKS we used did not contain the performance enhancing features direct ownership transfer, garbage collection of synchronization event list, partially eager evaluation, and static pre-analysis. Even in this setting, the experimental results given above show that our lockset algorithm is much cheaper than the vector clock algorithm and at very similar cost to Basic-Eraser. Table 7.2 emphasizes the fact that the overhead of race checking can be made more tolerable when fields with previously detected races are not checked. This also gives a rough idea about how a static analysis, which is used to eliminate checks to well-protected fields, can be used to further reduce the runtime cost of our algorithm. In the next section, we will provide our experiments after adding all the optimizations in Chapter 6 including the static pre-analysis of races.

7.2 Experience with Optimized Goldilocks Runtime

In this experiment we ran on the latest version of our race-aware JVM a set of benchmarks well known in the literature. A Linux machine with Intel Pentium-4 2.80GHz CPU was used. The initial and the maximum Java heap space were set to 16 MB and 512 MB, respectively. These limits triggered a moderate number of garbage collection runs which were necessary for proper trimming of the synchronization event list by partially-eager lockset propagation. The experiments were run on the interpreter mode of the Kaffe JVM version 1.1.6 [7]. We collected the runtime statistics using the PAPI interface to the hardware performance counters [32]. Arrays were checked by treating each array element as a separate variable.

To provide a reasonable idea of race checking overhead in our experiments, when a race was detected on a variable, race checking for that variable was turned off during the rest of the execution. Checks for all the indices of an array were disabled when a race is detected on any index of the array. The Java benchmarks we worked on do not have handlers for `DataRaceExceptions`. It is common in these benchmarks that a race on a variable is

followed by many more races on the same variable. Turning off race checking after the first detected race provides a more reasonable idea of overhead in these cases because, in normal operation, we do not expect a program to continuously detect and recover from races on the same variable.

We used eleven benchmark programs to evaluate the performance of our implementation in Kaffe. Six of our benchmarks are from the Java Grande Forum Benchmark Suite.⁴ The benchmark `moldyn` is a molecular dynamics simulation, `raytracer` is a 3D ray tracer, `montecarlo` is a Monte Carlo simulation, `sor` is a successive over-relaxation program, `lufact` is a LU factorization program, and `series` is a program that computes fourier coefficients. We reduced the size of inputs to the Grande benchmarks because of the long running time of the applications. The remaining four benchmarks are from [15].

We were able to apply the Chord static race analysis tool to all of the benchmarks and used the resulting race-prone access pairs as described in Section 6.3.5. We only had access to RccJava output for the following benchmarks: `moldyn`, `montecarlo`, `raytracer`, `sor2`, and `tsp`. In most of these cases, the checks that RccJava was able to eliminate subsumed those eliminated by Chord.

Table 7.3 presents the performance statistics of the GOLDILOCKS algorithm in different settings of the benchmark programs. The column titled “Uninstrumented” reports the total runtime of the programs in the JIT-compilation and the interpreter modes of JVM with the race detection feature disabled. The other columns present the running times and the slowdown ratios of the programs on JVM with race checking enabled. The runtime measurements are given in seconds. The slowdown ratio is the ratio of the runtime with the race checking enabled to the uninstrumented runtime of the program (both in interpreted mode). For the results in the columns titled “with Chord outputs”, “with RccJava outputs”, and “short-circuit checks”, the standard libraries were instrumented as well. For the results in the columns titled “without static information” the libraries were not instrumented. For these experiments, instrumenting libraries at most doubles overhead.

⁴Java Grande Forum Benchmark Suite can be obtained at:
http://www2.epcc.ed.ac.uk/computing/research_activities/java_grande/threads.html

Benchmarks	# Lines	# Threads	Uninstrumented Runtime		Without static information		With Chord outputs		With RccJava outputs		Short-circuit checks (%)	
			Just-in-time	Interpreted	Runtime	Slowdown	Runtime	Slowdown	Runtime	Slowdown	Chord	RccJava
<i>colt</i>	-	10	6.3	6.8	13.2	1.9	7.8	1.2	-	-	66.67	-
<i>hedc</i>	2.5K	10	3.1	7.5	14.5	1.9	11.0	1.5	-	-	31.94	-
<i>lufact</i>	1K	10	0.2	0.5	2.0	4.1	0.9	1.8	-	-	12.73	-
<i>moldyn</i>	650	5	21.6	135.2	730.2	5.4	712.9	5.3	217.8	1.6	99.53	99.99
<i>montecarlo</i>	3K	5	43.8	44.1	97.2	2.2	47.0	1.1	44.8	1.0	99.93	99.98
<i>philo</i>	86	8	2.9	2.6	2.7	1.0	2.7	1.0	-	-	6.62	-
<i>raytracer</i>	1.2K	5	0.4	3.4	61.7	17.9	39.0	11.4	7.4	2.1	51.01	51.01
<i>series</i>	380	10	87.9	88.4	94.1	1.0	93.3	1.0	-	-	10.17	-
<i>sor</i>	220	5	31.5	32.8	44.1	1.3	31.5	1.0	-	-	16.97	-
<i>sor2</i>	252	10	0.6	11.2	70.8	6.3	25.8	2.3	12.4	1.1	0.00	99.00
<i>tsp</i>	700	10	1.6	2.5	5.5	2.2	3.4	1.3	3.1	1.2	64.54	37.28

Table 7.3: Results of experiments with the race-aware Kaffe runtime

Table 7.3 also reports the percentages of succeeding short-circuit checks when the outputs of Chord or RccJava are used. The rest of the accesses require full lockset computations by traversal of the synchronization event list. These results clearly indicate that for some programs, most of the happens-before edges can be captured by using cheaper short-circuit checks, which significantly reduces overhead. The high percentage of lockset computations for `sor2` explains the high overhead incurred for this benchmark. In `lufact`, `philo`, `series` and `sor` the percentage of successful short-circuit checks is high, which led to low runtime overhead. The short-circuit checks do not help to reduce the overhead with Chord outputs for `moldyn` and `raytracer`. This is because `moldyn` and `raytracer` use barrier synchronization which is not captured by Chord, and thus, each volatile variable read and write used to implement barriers is processed separately in the lockset computations. The overhead for these examples is much lower when RccJava outputs are used, because RccJava eliminates checks for most of the variables whose accesses are synchronized using barriers. Note that Eraser-based algorithms do not handle barrier synchronization and would have declared false alarms for `moldyn` and `raytracer`.

One of the purposes of Table 7.3 is to contrast the performance of race checking without making use of any a priori information about race-free fields/accesses and race checking only the fields/accesses not proved to be race-free by the static pre-analysis. Although the runtime overhead without the static analysis may be acceptable for a race debugging tool, this overhead is unacceptable for the end users if the `DataRaceException` feature is intended to be enabled for continuously monitoring for races even after deployment. From this point of view, the results with Chord and RccJava highlight the importance of incorporating a priori analysis results in keeping the runtime overhead of race checking reasonable.

Table 7.4 reports the percentage values for the variables checked among all the variables created, and the variable accesses checked among all the accesses that take place during the execution. Using the output of Chord reduces the slowdown to a small value between 1 and 2 for most of the benchmarks. Using RccJava outputs decreases the slowdown for `moldyn` and `raytracer` to similar levels, whereas overhead remained high with the outputs of Chord for these examples. These experiments demonstrate that, with proper prior static analysis, the overhead of precise race checking at runtime can be reduced significantly. These results also indicate that, to be practical, precise race detection at runtime must be preceded by

	Variables checked (%)		Accessed checked (%)	
	Chord	RccJava	Chord	RccJava
<i>colt</i>	0.1	-	0.0	-
<i>hedc</i>	0.0	-	0.0	-
<i>lufact</i>	1.1	-	2.1	-
<i>moldyn</i>	84.1	83.5	56.6	49.1
<i>montecarlo</i>	15.4	13.7	13.0	39.9
<i>philo</i>	0.2	-	0.1	-
<i>raytracer</i>	80.5	35.3	5.2	1.3
<i>series</i>	0.3	-	2.0	-
<i>sor</i>	0.3	-	0.9	-
<i>sor2</i>	1.3	0.0	36.9	0.0
<i>tsp</i>	60.1	80.0	42.0	23.7

Table 7.4: Statistics on experiments with static analyses

sound static techniques for determining race-free variables.

7.2.1 Experiments with Transactions

To measure the performance of our implementation of a transaction-aware race checker, we mimicked the transactions implementation by source-to-source translation of Grossman et. al. [33] for a concurrently-accessed data structure. We did this because, although we had a prototype implementation of a transaction-aware race checker based on [33], we did not have access to the actual transactions implementation code as of this experiment.

The implementation of transactions in [33] uses Java’s object locks for synchronization. All shared variable reads and writes in an atomic transaction are protected by the object locks for the objects accessed. All shared variable reads and writes that are part of a transaction take place between the first lock acquire and the first lock release associated with the transaction. The first lock release also constitutes the commit point of the transaction.

```

public class Multiset {

    private Element[] A;
    private int capacity = 10;

    public Multiset(int c) {
        this.capacity = c;
        this.A = new Element[capacity];
        for (int i = 0; i < capacity; ++i) {
            this.A[i] = new Element();
        }
    }

    public boolean Insert(int[] x) {
        for(int i = 0; i < x.length; ++i) {
            int j = FindSlot(x[i]);
            if(j == -1) {
                for(int k = 0; k < i; ++k) {
                    A[x[k]].elt = -1;
                }
                return false;
            }
            x[i] = j;
        }
        validate(x, x.length-1);
        return true;
    }

    private void validate(int[] x, int i) {
        synchronized(A[x[i]]) {
            if(i == 0) {
                for(int k = 0; k < x.length; ++k) {
                    A[x[k]].valid = true;
                }
            } else {
                validate(x, i-1);
            }
        }
    }

    public boolean Delete(int e) {
        for (int i = 0; i < capacity; ++i) {
            synchronized (A[i]) {
                if (A[i].elt == e && A[i].valid) {
                    A[i].elt = -1;
                    A[i].valid = false;
                    return true;
                }
            }
        }
        return false;
    }

    public boolean LookUp(int e) {
        for (int i = 0; i < capacity; ++i) {
            synchronized (A[i]) {
                if (A[i].elt == e && A[i].valid) {
                    return true;
                }
            }
        }
        return false;
    }
}

package multiset;

public class MultisetTester extends Thread {

    private Multiset impl;
    int rnd;
    int anum;
    RandomNumberGenerator gen;

    public MultisetTester(Multiset multiset,
                          int anum,
                          RandomNumberGenerator g) {
        this.impl = multiset;
        this.anum = anum;
        this.rnd = 0;
        this.gen = g;
    }

    public void run() {
        for(int i = 0; i < anum; ++i) {
            impl.Insert(new int[] {
                gen.generate(), gen.generate(), gen.generate(),
                gen.generate(), gen.generate(), gen.generate()});
            impl.Insert(new int[] {
                gen.generate(), gen.generate(), gen.generate(),
                gen.generate(), gen.generate(), gen.generate()});
            impl.Delete(gen.generate());
            impl.LookUp(gen.generate());
        }
    }

    public static void main(String args[]) {
        int tnum = Integer.parseInt(args[0]);
        int anum = Integer.parseInt(args[1]);
        int vnum = Integer.parseInt(args[2]);
        RandomNumberGenerator g = new RandomNumberGenerator(0);
        Multiset multiset = new Multiset(vnum);

        Thread[] threads = new Thread[tnum];
        for(int i = 0; i < tnum; ++i) {
            threads[i] = new MultisetTester(multiset, anum, g);
        }
        for(int i = 0; i < tnum; ++i) {
            threads[i].start();
        }
        for(int i = 0; i < tnum; ++i) {
            try {
                threads[i].join();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}

```

Figure 7.3: The Multiset code modified for the experiment.

The data structure we worked on is a **Multiset**, based on the benchmark with the same name in [34]. Figure 7.3 shows the code. The test program for **Multiset** consists of a number of threads accessing a multiset of integers concurrently by performing insertions, deletions and queries. The representation for a multiset is an array **elements** of objects where each object potentially stores an element of the multiset. The **Insert(int[] x)** operation attempts to first allocate space for the **x.length** entries in **elements** using a transaction for each allocation. If this allocation is successful for all **x.length** entries, then all of the new multiset elements are made visible to other threads in an atomic transaction. If the allocation is unsuccessful because of space contention with concurrent threads, the space allocated in **elements** is freed in a single atomic transaction. This mimics transaction rollback. To mimic the use of transactions mixed with other synchronization primitives, the arrays used as arguments to **Insert** were generated by a factory object shared among threads. This object and the array objects were manipulated outside transactions.

To imitate what our JVM expects of a transactions implementation, shared variable reads and writes between the first lock acquire and the first lock release in each transaction were recorded by instrumentation code in the JVM to form the sets R and W associated with a $commit(R, W)$ action. This commit action was inserted in the synchronization event list where the first lock release in a transaction would have been inserted if we were not explicitly considering transactions.

#Threads	Uninstrumented	GOLDILOCKS with transactions		#Accesses	#Transactions
	Runtime	Runtime	Slowdown	x1000	x1000
5	0.35	0.51	1.46	215	21
10	0.66	0.80	1.21	381	45
20	1.11	1.60	1.44	660	87
50	2.89	3.55	1.23	1460	206
100	5.57	8.19	1.47	2819	407
200	12.15	15.30	1.26	5493	802
500	31.78	40.84	1.29	13648	2006

Table 7.5: Performance of checking races for transactional Multiset

We measured, for different numbers of threads sharing a multiset of size 10, the runtime with race checking enabled for the transactions as described in Chapter 5 and Section 6.5 (Table 7.5). The table also reports the number of shared variable accesses and the number of transactions in the executions. The results indicate that the runtime overhead of our approach (which includes the overhead of keeping track of the read and write sets of transactions) when explicitly handling transactions is moderate. When we analyze **Multiset** executions without taking transactions into account (i.e., using the translated source code as is) we incur slowdown factors of more than ten. This indicates that treating software transactions as high-level synchronization primitives may reduce the runtime overhead of race checking.

The overhead discussed above is not representative as **Multiset** executions consist entirely of shared variable accesses each involving a separate lock acquire and release. If we had applied our approach to the **Multiset** disregarding transactions, the static analyses would have detected that the elements of the **Multiset** were each protected by a unique lock throughout the transaction. However, to be able to perform race checking for the variables accessed outside of transactions, we would have still had to include all the lock acquires and releases in the synchronization event list.

7.3 Discussion: Race-Detection Overhead

At the time of the original **GOLDILOCKS** work, the vector clock algorithm [18] was the only precise dynamic-race-detection algorithm in the literature. The vector clock algorithm, for an execution with n threads, requires for every thread and synchronization variable a separate vector clock (VC) of size n and performs $O(n)$ operations (merging or comparing two VCs) whenever a synchronization operation or data access happens. See Section 2.2 for a basic description of the vector clock algorithm. In preliminary research, compared to a straightforward implementation of vector clocks, we found **GOLDILOCKS** overhead to be significantly less [4].

In [4], we measured the overhead of the **GOLDILOCKS** implementation inside Kaffe on a set of widely-used Java benchmarks. This implementation required us to run all programs in interpreted (not just-in-time compiled) mode. We found that, with powerful static analysis tools eliminating much of the monitoring, we were able to obtain a slowdown of within

approximately 2 for all benchmarks. Without static elimination of some checks, overheads remained high; some benchmarks experienced slowdowns of over 15. The overhead results with static pre-elimination were encouraging in that they showed precise race detection to be a practical debugging tool, and they indicated that, with further optimizations, post-deployment runtime race detection to support `DataRaceException` could be viable.

Later work on FASTTRACK [9], a dynamic race detector based on vector clocks, is able to avoid worst-case performance of vector clocks much of the time using optimizations for common cases. This is accomplished by using optimizations that improve the complexity of VC updates in common synchronization cases, and by optimizations that reduce the storage requirement; the average cost reduces to $O(1)$ when a series of consecutive reads are ordered by the happens-before relationship. Experiments in [9] compare a number of race-detection algorithms, including just-in-time compiled implementations of FASTTRACK and GOLDDILOCKS in ROADRUNNER [31]. FASTTRACK achieves significantly better overheads than both implementations of GOLDDILOCKS. The low overheads achieved by FASTTRACK provide further support that a practical race-aware runtime for deployed programs supporting a `DataRaceException` can be built. It is reported in [9] that additional short-circuit checks similar to ones we discussed above dramatically reduce the runtime of FASTTRACK. Most of these checks can be incorporated into GOLDDILOCKS implementations as well. There are more optimizations possible to add in both FASTTRACK and GOLDDILOCKS, and whether a further-optimized version of GOLDDILOCKS would be competitive with FASTTRACK or not is an open question.

In this chapter, we reported on our experience with the GOLDDILOCKS runtime. We showed that the performance of GOLDDILOCKS is competitive with those of the Eraser and vector clock algorithms. In addition, our results on the current version of the runtime showed that the optimizations including the static pre-analysis dramatically reduce the runtime cost of our algorithm. While considering the absolute impact on the runtime, it is important to consider the safety-performance trade-off offered by other runtime exceptions in Java. Since the consequences of not catching these other exceptions can be serious, for them the trade-off is considered acceptable. Observing that the effects of undetected race conditions are serious (unintended semantics, data corruption, etc.), we believe that the overheads offered by our

approach may be tolerable and further reduced by more optimized implementations.

Chapter 8

CONCLUSION

In the first part of this thesis, we presented **GOLDILOCKS**, a Java runtime that continuously monitors the program for data-races. We have built **GOLDILOCKS** motivated by two facts: First, data-races, which are the most prevalent symptoms of concurrency bugs, are difficult to detect and reproduce. Second, racy program semantics, which are precisely defined by the Java Memory Model (JMM), are difficult to understand and write correct programs with. On the other hand, the JMM guarantees that when a program is free of races, it will operate under the easy-to-use sequentially-consistent semantics.

When a data-race is about to occur, our runtime throws a Java runtime exception called **DataRaceException**. This prevents unintended consequences of the data-race and brings the situation to the attention of the programmer. In this way, an actual race condition becomes analogous to other checked errors in Java, for example, null-pointer dereferences or out-of-bound array accesses.

This work is the first to propose **DataRaceException** and show that, in addition to being a debugging tool during software development, **DataRaceException** is a viable mechanism for the enforcement of program safety after deployment.

The **GOLDILOCKS** runtime provides three key benefits. First, it helps to identify concurrency-related bugs that manifest themselves as data-races. By interrupting the execution when a data-race is detected, our runtime allows the programmer to analyze the bug before it causes an error that may be impossible to diagnose later. Second, if no **DataRaceException** is thrown during an execution, an error in that execution can be analyzed by assuming sequential-consistency at the byte-code level and ruling out many possibilities of missing synchronization. Third, by catching the **DataRaceException**, the programmer can terminate gracefully or roll-back the effects of the operation that triggered the race. Alternatively, he can use **DataRaceException** as a conflict-detection mechanism in optimistic uses of concurrency, e.g., software transactional memory (STM).

In order to detect data-races at runtime, we developed GOLDILOCKS, a lockset-based algorithm for computing the happens-before relation as defined in the JMM. GOLDILOCKS is the first algorithm to have the three essential features necessary for the implementation of `DataRaceException`: First, it is *sound*, i.e., it detects all actual data-races. Second, it is *precise*, i.e., it never gives false decisions. Third, it is *efficient*, i.e., it has tolerable runtime overhead, which is competitive with that of the vector-clock and Eraser algorithms. Furthermore, GOLDILOCKS provides the first formalization of race conditions for programs using software transactional memory along with other synchronization primitives in Java.

We have implemented GOLDILOCKS inside the Kaffe JVM, a full-fledged Java virtual machine written in C. Through a combination of static analysis and an efficient implementation of our data-race detection algorithm, we have demonstrated that the runtime overhead of precise data-race detection required for supporting a `DataRaceException` can be made reasonable. With improvement of static analysis techniques and further optimizations in the implementation, we believe that the runtime overhead can be reduced enough to be acceptable for continuous monitoring of program executions during debugging and during deployment for critical programs. Our algorithm was also implemented, using the same set of implementation strategies, in the Chess model checker of Microsoft Research and the ROADRUNNER infrastructure and taught in concurrency-related courses at several universities.

8.1 Future Work: Specializing GOLDILOCKS to The C++ Memory Model

There has been an ongoing effort to define a memory model to C++. The current version of the C++ Memory Model (C++MM) [3] provides a similar formalism to the JMM. In C++MM, variables are separated as data and “atomic variables”, where the latter correspond to synchronization variables. Data variables are either non-atomic global variables defined outside of classes or in the form of (o, d) where o is an object and d is a non-atomic variable.

The main synchronization mechanisms in C++ are locks and atomic variables. There is an ongoing effort on defining a standard semantics for locks and atomic variables provided by different C++ threading libraries. In this semantics, the synchronizes-with ordering for lock operations is defined in the same way it is defined for monitor operations in the JMM. However, while atomic variables in the C++MM and volatile variables in the JMM seem to

coincide in the usage patterns, the synchronizes-with ordering for the atomic variables are defined differently from the synchronizes-with ordering for volatile variables in the JMM. In particular, in the C++MM, an atomic write in C++ synchronizes with *only* the atomic reads that see that write; in the JMM, a volatile write synchronizes with *all* the reads that come later in the synchronization order. This difference makes unsound not only GOLDBLOCKS but also the vector clock algorithm. As a result, either C++MM should be modified to make atomic reads/writes in the C++MM reflect the same, or the data-race checking algorithms for C++ should be fixed. If the latter is chosen, we foresee that it will be much easier to adapt GOLDBLOCKS to the formalism in the C++MM than the vector clock algorithm. Specializing the GOLDBLOCKS algorithm to the C++MM is an interesting problem to address since it will provide the first sound and precise race detection system for the C++MM.

Part II

The QED Proof System

Chapter 9

INTRODUCTION

Because of emerging technologies such as multicore processors and grid computing, concurrency is becoming an important issue in today's software systems and is likely to become even more so in the future. A wide-range of systems, e.g., web servers, databases, and operating systems, make use of highly concurrent data structures and services in order to respond efficiently to a large number of clients accessing simultaneously. Such software makes use of sophisticated synchronization techniques, including fine-grained locking and non-blocking operations, and creates extra threads for internal operations. These techniques require intensive care to use, and bugs due to their incorrect usage can have serious consequences, such as data corruption, operating system crash, or even more catastrophic results, e.g., failure of an aircraft flight control system. Therefore, the functional correctness of software becomes as fundamental as its performance.

The second part of this thesis is concerned with the problem of statically verifying the partial correctness of shared-memory multithreaded programs. In particular we are interested in proving two prevalently used safety criteria: *assertions* and *linearizability*. Assertions state conditions expected to hold when the execution of the program reaches a certain location in the code, thus are widely used to specify thread-local and program-wide global invariants, and procedure pre- and post-conditions [35]. Assertion verification has been studied quite extensively, since many interesting properties of programs such as data integrity, absence of null-pointer dereferences and out-of-bounds array accesses can be expressed as assertions. Linearizability formalizes the requirement that a concurrently-accessed data structure implementation conform to a specification containing atomic versions of the data structure's operations [36]. In this thesis, we provide a static proof system and supporting software tool to show that (1) no execution of a given program leads to an assertion violation and (2) every execution of a concurrent data structure implementation is equivalent to an execution of its atomic specification.

The static verification problem is undecidable [37] and, in theory, no harder than the problem of verifying single-threaded programs. In practice, however, it is significantly more difficult to verify multithreaded programs. For single-threaded programs, the undecidability of program verification is circumvented by the use of annotations—pre-conditions, post-conditions, and loop invariants—to decompose the problem into manageable pieces. The effectiveness of the proof is highly dependent on the selection of these annotations. For a sequential program, these annotations need to refer only to the locally visible part of the program state, and can usually be stated with little difficulty. Given this, for a considerable class of sequential programs, several key ideas have been instrumental in automating assertion checking, which have been implemented in well-known verification tool ESC/Java [38], WHY [39, 40], and Boogie [41].

Reasoning about multithreaded programs, on the other hand, requires significantly more intellectual effort due to the interaction between threads over the shared memory. For example, invariant-based reasoning [35, 42] requires for each line¹ of the program an annotation that is guaranteed to be stable under interference from other threads. Writing such a specification is challenging because at each line, the user needs to consider possible interleavings of a large number of conflicting operations. The resulting annotations in each thread are often non-local and complicated because they refer to the private state of other threads and the state of the synchronization objects, e.g., which thread holds a particular lock. The rely-guarantee approach [43] simplifies specifying and checking the interference between threads by allowing one to compose proof outlines of statements modularly, but the complexity of the required annotations in these proof outlines is still significant due to fine-grained concurrency. Concurrent separation logic [44] has the ability to maintain separation between shared and local memory, enabling sequential reasoning for multithreaded programs; however, this is not particularly useful for programs with high level of interference.

The fundamental difficulty in reasoning about multithreaded programs is the need to reason about concurrent execution threads composed of fine-grained atomic actions. In this thesis, we introduce an approach to proving assertions and linearizability in multithreaded programs aimed at circumventing this difficulty. In our approach, a proof is performed in

¹For brevity in this chapter, we use the word “line” to refer to the granularity of statements executed indivisibly by other threads. We will define such statements more precisely later.

multiple steps. In each step, we rewrite the program by locally transforming its atomic actions to obtain a simpler program. For this purpose, we have developed a static proof system, called QED. QED provides a novel proof strategy in which *atomicity is used as a proof tool*: a program with fine-grained concurrency is transformed iteratively to make it consist of larger atomic actions. This gradually reduces the influence of thread interleavings on the complexity of the reasoning, and therefore permits significantly more tractable proofs than those provided by existing methods. The correctness conditions are checked after the program reaches an atomicity level appropriate for local analyses. For example, we prove assertions by performing *sequential* checks within atomic blocks of the final program. The transformations preserve or expand the set of behaviors of the program, so that assertions proved at the end of a sequence of transformations are valid in the original program. Our proof strategy facilitates a clean separation of concerns where at each phase of the proof, the user worries only about either the sequential properties or the concurrency control mechanisms in the program. This allows the user to clearly express, during the proof steps, his/her design intent about the program. In addition, one can simplify a program with larger atomic blocks using QED and can continue the proof with another method, e.g., concurrent separation logic [45, 46].

Each rewrite of the program performs one of two different kinds of transformations—*abstraction* and *reduction*. Abstraction replaces an atomic action with a more relaxed atomic action allowing extra behaviors. Abstraction transformations include making shared variable reads or writes non-deterministic, and adding extra assertions to atomic actions. Reduction [47] replaces a compound statement consisting of several atomic actions with a single atomic action if certain non-interference conditions hold. This transformation has the effect of increasing the granularity of the atomic actions in the program.

A distinguishing and essential aspect of our method is the iterative and alternating application of abstraction and reduction. While abstraction and reduction have been studied in isolation, in our method, they are *symbiotic*. Reduction creates coarse-grained actions from fine-grained actions and allows a subsequent abstraction step to summarize the entire calculation much as pre-conditions and post-conditions summarize the behavior of a procedure in a single-threaded program. Conversely, suitably abstracting an atomic action allows us to reason that it does not interfere with other atomic actions, and later applications

of reduction are able to merge the action with other actions. Our experience with highly-concurrent benchmark programs (reported in Chapter 19) shows that the combined iterative application of these techniques is surprisingly powerful so that even intricate examples can be proved correct by a short sequence of transformations and a small number of annotations and auxiliary variables. In most cases, we were able to carry the proof forward so that the assertions in the final program can be validated by purely sequential reasoning within atomic actions. In other cases, the atomic actions in the program become large enough for a straightforward application of existing techniques such as rely-guarantee reasoning. Our method is orthogonal and complementary to existing methods that do not make direct use of reduction and abstraction, and subsumes others that do [48, 49, 50, 51].

Another distinguishing feature of our approach is the possibility of introducing assertions at any point during the sequence of transformations while deferring their proof to a later step, until the atomic actions become large enough to make the discharging of these assertions easy. In our framework, annotating an atomic statement with an assertion is a valid program abstraction. The interpretation is that the action is relaxed so that if the assertion is violated, the execution “goes wrong”. In other proof methods, when an additional assertion is introduced, one is forced to prove that it is valid and preserved under interference by other threads. In our approach, we use the introduced assertion without first proving it and take further proof steps that simplify the program. Yet, the soundness of our method is not compromised as long as all assertions are proved eventually. We find the capability to “borrow assertions” from the abstracted program in this sound manner to be quite useful. For example, we borrow assertions to express locally our beliefs about the program state when an action is reached. During a following reduction step, the assertion indicates the absence of an apparent interference between the containing action and another action by a different thread, for example, that the actions are not simultaneously enabled, and enables further application of reduction. When a previously inserted assertion is discharged, this indicates that our beliefs about the program state were correct, and thus, the previous reduction steps relying on that assertion were sound.

We have implemented our verification method in an interactive proof tool called QED-VERIFIER. QED-VERIFIER is open source and can be downloaded from <http://qed.codeplex.com>. Our tool accepts as input a multithreaded program written in QEDPL,

an intermediate programming language based on the Boogie 2 language [52], and a proof script containing a sequence of proof commands. A proof command is used for one of two purposes. First, it may ask QED-VERIFIER to rewrite the input program using abstraction, reduction or a combination of the two. For example, the **abstract read** command is used to abstract the value of a variable in a particular action, and the **reduce all** command triggers an iterative application of various reduction rules in QED on the program. Second, it may provide a concise specification of the behavior of the current version of the program to be used by later proof commands; common specifications include locking protocols and data invariants. For example, the **invariant** command is used to introduce a new invariant for the program, and **annot proc** is used to introduce a pre- and/or post-condition for a procedure. After executing each step in the proof script, QED-VERIFIER allows the user to examine the resulting program, intercept the proof, and give new commands. The tool mechanizes proof steps specified in the proof script by encoding the low-level logical reasoning required by the QED proof rules as verification conditions (VC) automatically checked by Z3 [53], a state-of-the-art solver for satisfiability-modulo-theories (SMT).

We have evaluated QED-VERIFIER by verifying a number of multithreaded programs with varying degree of synchronization complexity. These examples include programs using fine-grained locking protocols and non-blocking data structure implementations. We have found that the iterative approach embodied in QED-VERIFIER provides a simple and convenient way of communicating to the verifier the programmer’s understanding of the computation and synchronization in the program. Interaction with the verifier allows the user to progressively develop a better understanding of the concurrency in the program, and makes it easy to debug errors in the implementation and specification. The proofs in our method are usually cleaner than the proofs based on existing approaches. The proof script of a program serves as a reproducible (by our tool) documentation of its correctness. Therefore, our distribution of QED-VERIFIER also provides a repository of QED-based proofs for a collection of benchmark programs using a rich variety of synchronization techniques.

9.1 Contributions and Outline of Part II

The rest of this part of the thesis is organized as follows:

- Chapter 10 provides the technical background and a survey of existing techniques in

static verification.

- Chapter 11 gives a high-level overview of our verification method by demonstrating the application of the proof system to several examples.
- Chapters 12–19 present our contributions in detail.
- Chapter 20 concludes this thesis and discusses future research directions.

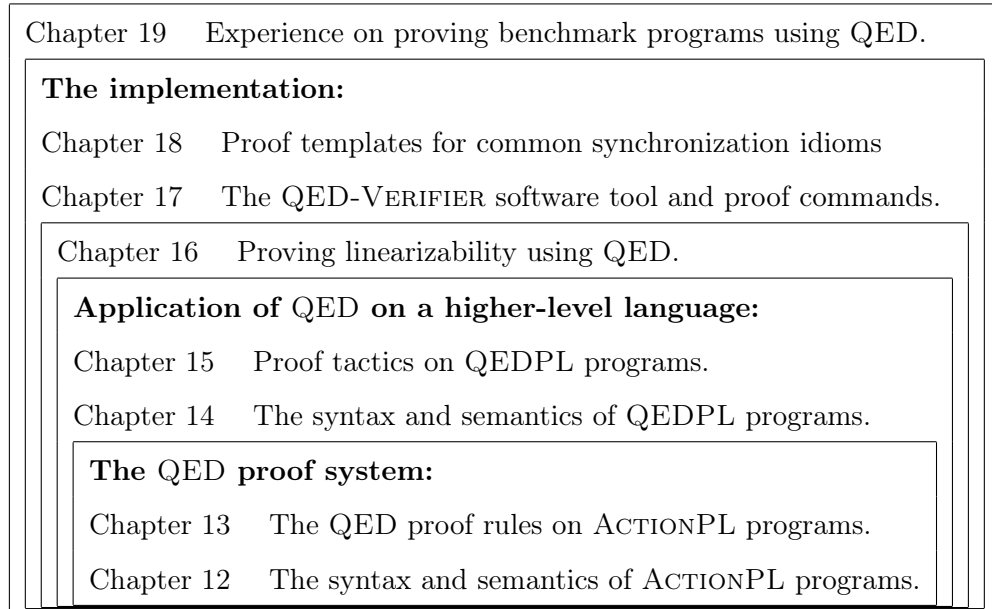


Figure 9.1: Presentation of contributions in Part II

The rest of this section overviews the contributions of this part of the thesis explained in Chapters 12 through 19. Figure 9.1 shows the hierarchical relationship between the contents of these chapters. In this relationship, each box represents part of our contributions that is built upon the ones in the inner boxes.

1. **The QED proof system:** Our main contribution in this part of the thesis is a proof system, called QED, and a novel technique based on QED for statically verifying multithreaded programs for assertions and linearizability. In this technique the input program is rewritten iteratively using abstraction and reduction, producing a program

that can be verified by sequential reasoning methods. Our technique also complements other verification methods by simplifying the program with coarser-grained actions. The core proof system is described in two chapters:

- (a) In order to present our conceptual approach, we developed ACTIONPL, a formal programming language. Chapter 12 describes the syntax and operational semantics of ACTIONPL programs. We have developed ACTIONPL to simplify the technical presentation of the QED proof system. ACTIONPL is a low-level language that is not practical to write programs in, but it provides a convenient way to represent computations for formal treatment, especially to describe the operation of the QED proof rules. It is also expressive enough that the operation of any program in a higher-level language such as Dafny, Java, and our input language QEDPL, can be expressed as an ACTIONPL program. This allows us to lift our QED-based conceptual approach to a more practical setting, by defining sound transformations on a higher-level language.
- (b) In Chapter 13, we present our notion of a *proof* and the core QED*proof rules*. Each proof rule specifies a transformation on a given ACTIONPL program under certain conditions to be satisfied by the program. While doing this, we describe a precise check for an abstraction relation between two actions, and show how to encode commutativity properties of actions, required by reduction, in terms of this relation. These checks can be expressed as verification conditions in the first-order logic that can automatically be verified by using SMT solvers. Therefore, this thesis presents a novel and practical application of Lipton’s reduction theory [47] for static verification. In Chapter 13 we prove that the QED proof rules are sound for assertion checking: Once a program P_1 is transformed to program P_n and all the assertions in P_n are verified, then our soundness theorem guarantees that P_1 satisfies all assertions in it.

2. Application of QED on a higher-level language: In this thesis, we show how the conceptual approach introduced in item 1 to a higher-level language. This consists of two phases:

- (a) Chapter 14 introduces the QEDPL programming language we developed to demon-

strate the application of the proof system to a higher-level language. Our tool implementation takes as input a QEDPL program and carries out the entire proof in this language. Compared to languages like Java and C#, QEDPL is not a full-fledged language, however, it carries the essential properties of an intermediate language: expressiveness and a syntax that is easy to understand and write useful programs with. While developing QEDPL, we take the syntax of Boogie 2 [52] as the basis. Boogie 2 is being developed by Microsoft Research for verifying sequential programs. QEDPL extends the core syntax of Boogie 2 with multithreaded programming constructs such as atomic statements (`atomic{·}`) and structured thread creation (`dopar{·}with{·}`), as well as exception handling and structured data types (records). We give a denotational semantics of QEDPL by translating every QEDPL program to an ACTIONPL program.

- (b) In Chapter 15, we present *proof tactics*, higher-level proof transformations on QEDPL programs. Each proof tactic describes an application of a proof rule referred to in item 1(b) on a QEDPL program. Thus, Chapter 15 explains how the conceptual transformations in QED proof system can be performed in practice. As each QEDPL program is specified via its translation to an ACTIONPL program, the soundness of a proof using these tactics is justified by associating the operation of a proof tactic on a QEDPL program with the operation of the corresponding proof rule on an ACTIONPL program.

3. **Proving linearizability using QED:** To simplify the presentation, Chapters 12–15 focus on verifying assertions. In Chapter 16, we describe a new strategy for proving linearizability by transforming an implementation into its specification within the QED proof system and argue that QED significantly simplifies the reasoning about linearizability for programs using fine-grained concurrency and complex synchronization. In the resulting approach, the abstraction map between the implementation and the specification, an essential part of linearizability proofs, is constructed incrementally, and the need to reason about the location of commit points in the implementation code is eliminated. Chapter 16 gives two soundness results: First, transforming a concurrent data structure implementation $Impl_1$ to $Impl_2$ using QED and proceeding with $Impl_2$

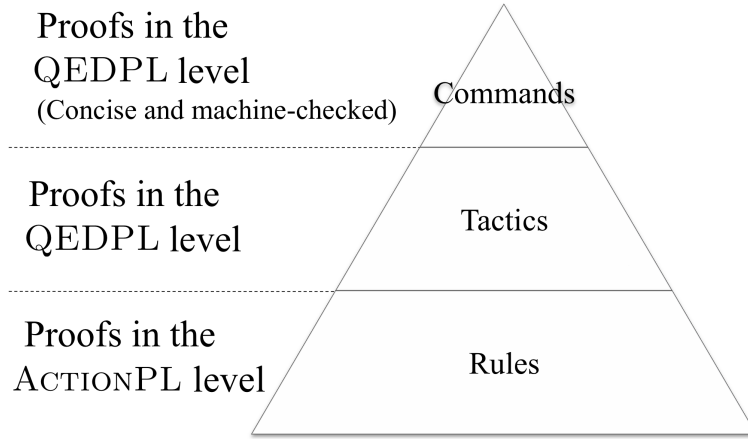


Figure 9.2: The relationship among proof rules, tactics, and commands.

and using another method is a sound approach to verifying the linearizability of $Impl_1$. Second, transforming $Impl_1$ to its sequential specification using QED transformations guarantees the linearizability of $Impl_1$.

4. **The implementation:** We have implemented our verification technique in an interactive software tool called QED-VERIFIER. QED-VERIFIER mechanizes the application of proof tactics given by the user and automatically generates proof templates for common scenarios.

- (a) Chapter 17 describes the architecture of QED-VERIFIER. QED-VERIFIER runs by applying user-provided *proof commands* to QEDPL programs. In our implementation, a proof of the program is performed by applying a sequence of proof commands. A proof command is an intuitive and concise way of instructing the tool to apply QED-based transformations. Figure 9.2 shows this relationship among proof rules, tactics, and commands: Proof rules, tactics, and command are all used to construct a QED-based proof of a program. Proof rules are used to do this in a conceptual level and on programs written in a formal language (ACTIONPL). We use proof tactics to prove a program written in a higher-level, more conventional language. Proof commands provide a concise language to construct proofs in our tool QED-VERIFIER. A proof performed at a level in Figure 9.2

can be expressed as a proof at a lower level. However, the proof becomes more intuitive and concise while going up in Figure 9.2. QED-VERIFIER transforms low-level logical reasoning about the correctness of a concurrent program, which is often carried out using complicated-to-use interactive theorem provers, to intuitive reasoning at the level of program transformations. This is enabled by encoding most of the sophisticated reasoning in the proof rules and automatically performing the required checks using an SMT solver.

- (b) **Proof templates for common synchronization idioms:** Having supported by QED-VERIFIER, in Chapter 18, we explore an approach to improving the practical usability of our method. Synchronization idioms such as mutual exclusion and readers/writer locks are widely-used to ensure atomicity of critical regions. We have developed an annotation assistant within QED-VERIFIER that automatically generates program annotations that express non-interference between program statements, ensured by the synchronization idioms, and are used to identify atomic code regions. We provide proof commands that help the user to specify atomic critical regions concisely by identifying the synchronization idiom that ensures their atomicity. This also allows the programmer to debug the use of the idioms in the program. In Chapter 18:

- We provide an abstract semantics for several common synchronization idioms and, using programmer-provided predicates, a way to link this semantics to its realization in a program.
- For each synchronization idiom, we provide a proof script template that is used to identify atomic code blocks ensured by that idiom. For each idiom, we have developed a proof command that takes programmer-provided predicates, and runs an iterative algorithm that automatically generates the program annotations required for the script template to succeed. The algorithm takes over the manual derivation of the annotations, and thus reduces the user's annotation burden to simply providing the synchronization idiom and a few predicates.

5. **Experience with QED:** We have evaluated the simplicity and practicality of our technique and tool on a variety of small to medium-sized multithreaded programs. In Chapter 19 we present our experience elaborating on the key steps of the proofs we explored. Our experience has produced strong evidence for two claims. First, QED is *effective*: the notions of reduction and abstraction in QED are flexible enough to construct proofs of a wide spectrum of programs using fine-grained locking, non-blocking operations, and optimistic concurrency. Second, QED is *usable*: our iterative proof strategy helps in capturing the design intent of the programmer by *separating concerns about concurrency techniques and data properties* during the proof, and thus helps the user to express and check each correctness argument on a program in a clear way. As the proof script for a program serves as a *reproducible* documentation of its correctness, an important outcome of this work is a proof repository for the benchmarks we proved using QED-VERIFIER.

Chapter 10

THE STATE OF THE ART IN STATIC VERIFICATION

In this chapter we set up a preliminary background on the fundamental concepts in verification of concurrent software. We also extend the discussion in Chapter 9 about strengths and drawbacks of the existing approaches with more specific examples from the literature.

Outline. Section 10.1 describes two techniques to specify programs. Section 10.2 points out the difficulties concurrency brings to program analyses. Section 10.3 briefly introduces existing static verification techniques for concurrent systems. Section 10.4 concludes by discussing abstraction- and reduction-based techniques in the literature.

10.1 Specifying Programs

The specification of a program formally describes the expected properties of the program. For example, accessing to only allocated and initialized data is an expected behavior. In this case, the specification describes valid memory operations by stating how the program is supposed to do dynamic memory allocation and which regions of the memory are allowed to be read from and written to. A formal specification can be as simple as a logical expression such as a propositional formula, or as complex as a program written in a formal programming language, such as Zing [54]. Specification methods are classified into assertional or behavioral. In this thesis we use both assertional and behavioral specifications.

10.1.1 Assertional Specifications

Assertional specifications use a set of constraints that should be satisfied by the states the program will produce. In this thesis, assertional specification of a program is given by inserting *assert* statements throughout the source code. An assert statement, or simply *assertion*, assures that a particular condition holds whenever the assertion is evaluated. For programmers, this is a convenient way of specifying the correctness of a code block.

Assertions specify safety properties [55, 56], which assure that the program will never enter an unintended, “bad” state. The asserted condition is a logical formula over the program variables within the same syntactical scope of the assertion. Then, the entire specification of the program is taken as the conjunction of these assertions, and describes the set of states that an execution of the program can produce. While the first-order propositional logic is the natural language for writing assertion conditions, with the advances in mechanized theorem proving, higher order logics have also been utilized [57] to capture more intricate properties of programs such as polymorphism and recursion.

The following pseudo-code demonstrates a use of assertions. The procedure **Append** appends an item to a queue; `queue_size` and `queue_tail` are global variables storing the size and pointing to the tail of the queue, respectively.

```

procedure Append (Queue queue, Item item) {
    ... get the current tail of the queue
    assert (queue_size > 0) ==> (queue_tail != NULL);
    ... make item the new tail of the queue
}

```

The assertion in the code above must be satisfied whenever the assertion in the middle is reached when executing **Append**. The constraint enforces that, at any time if the size of the queue is a positive integer, the variable `queue_tail` points to a valid location in the memory. Violation of the assertion in an execution indicates that there is a bug in the program causing an inconsistent queue.

Correct operation of procedures can be expressed in this way by inserting assertions at the beginning and the end of the procedures as pre- and post-conditions of the procedures. For example, in the code below, the first assertion indicates that the procedure must be called with the argument `item` pointing to a valid location in the memory. Both assertions indicate that the implementation must be written so that the queue must not contain more than `MAX_QUEUE_SIZE` items at a time, thus any execution of **Append** must leave $(0 \leq \text{queue\_size}) \ \&\& \ (\text{queue\_size} \leq \text{MAX\_QUEUE\_SIZE})$ invariant. The first assertion enforces a condition that must be satisfied right before a call to **Append** is made, and the second assertion enforces a condition that must be satisfied by the code in the body of **Append** assuming that the first assertion is satisfied at the call points. Thus, in modular verification [41] the

first assertion is treated an assumption and it is only checked at the call points (in other procedures).

```

procedure Append (Queue queue, Item item) {
    assert (0 <= queue_size) && (queue_size <= MAX_QUEUE_SIZE) && (item != NULL);
    ... code for the operations
    assert (0 <= queue_size) && (queue_size <= MAX_QUEUE_SIZE);
}

```

Soundness and Completeness for Assertion Checking

A program analysis technique for verifying assertions in a given program answers the question: *Is there any executions of the given program that violates an assertions?* When a *sound* finishes without reporting any violations, the program must not produce any executions in which some assertion is violated, i.e., the analysis should never declare a buggy program as correct.

A *complete* analysis can correctly answer the question above for all programs. However, assertion checking for arbitrary programs is an undecidable problem [58, 59]. Thus, a method can only achieve completeness by either restricting the class of programs it can analyze, which is usually done by abstracting programs to a finite domain (of program states and computations), or requiring manual inputs (such as loop invariants) from the user in order to divide the overall analysis to decidable analyses. For example, Gulwari et al. showed that checking assertions can become decidable by abstracting programs over the theories of linear arithmetic and uninterpreted functions. This new problem was proved to be coNP-hard [60], although the check can be done efficiently in practice. There are algorithms for special cases of the problem that can run in polynomial time [61]. In this thesis, we follow the latter direction, requiring manual guidance and inputs from the user.

10.1.2 Behavioral Specifications

Behavioral specifications describe correct operations of a program using another abstract program (written in the same language or a formal one). This abstract program generates the expected behaviors of a correct program. Then the correctness of a program with respect to the abstract program is usually formalized as a conformance check, also known

as refinement check, between the programs [62, 63, 64, 65]. In this regard, Unity [66] and Zing [54] are well-known languages that support concurrency.

In this thesis, we use *linearizability*, a well-known correctness criterion for specifying the expected behaviors of a concurrent data-structure implementation through a sequential version of the data structure [36]. A concurrent implementation, denoted *Impl*, is said to be *linearizable* with respect to a sequential specification, denoted *Spec*, if every concurrent operation *op* of *Impl* takes effect as if it is performed indivisibly by other threads between its call and return points, where the correct effect is described by a sequential operation *op'* in *Spec*. In our approach, both *Impl* and *Spec* are written in our input language QEDPL. Therefore, in our QED-based technique, QEDPL is a language not only for writing programs, but also for writing the behavioral specifications of programs.

10.2 Reasoning about Concurrency

While concurrency is beneficial in increasing the efficiency of the overall computation, it is also costly and nontrivial to use and reason about. The difficulties indicated below hold for both dynamic and static program analyses. Concurrency brings nondeterminism due to thread schedules along with the nondeterminism due to program input such as start-up arguments of the program and bytes read from a file on the disk. In a computer with a single-core processor, at every point of execution, except for the synchronization points, the scheduler of the operating system (OS) can choose between executing the current thread or switching context to another thread. In a computer with a multicore processor, nondeterminism due to real concurrency (executing threads simultaneously not in an interleaving manner) takes effect in addition to the nondeterminism due to the OS thread scheduler. Moreover, the memory model [2] may allow the source-code level and runtime transformations that may add new executions to the program, especially if the program is racy. The result is a significant increase in the execution space of the program. These executions are hard to predict through manual inspection of the software and inefficient to examine by automated techniques. In fact, concurrency is the primary cause of the state space explosion problem, one of the most fundamental problems of model checking for decades [67]. Thus, the static approaches to sequential program verification have evolved to more sophisticated versions while focusing on concurrent programs. For example, model checking tools need to

use more complex partial-order reduction techniques for concurrent software than sequential software [54, 68].

Moreover, concurrency brings new types of errors. Concurrency-related bugs usually take place because of missing or incorrect synchronization between threads and they manifest themselves as discrepancies in program behavior, e.g., null pointer dereferences and data corruption. Resulting errors induce high computational and memory costs to detect and reproduce through monitoring the execution at runtime. In fact, they may even happen without generating observable results (e.g., system crash or warning) for many executions of the program, by having symptoms only realizable by careful inspection of the program state. Below we give three concurrency-related errors, that appear as symptoms of synchronization deficiencies in the code.

- **Data Races:** A data race happens when two concurrent threads try to access the same memory region simultaneously so that the result of the accesses may be arbitrary. Race conditions may have serious consequences such as data corruption and even cause runtime errors ending with system crash. There is an extensive research on identifying races [69, 4, 70, 14, 16, 8, 9].
- **Atomicity Violations:** Atomicity [71] is an important requirement for systems, e.g. databases, doing highly concurrent data updates. The aim in these systems is to achieve a synchronization discipline so that a sequence of data accesses (read or write) is executed as if the accesses do not interfere with the conflicting operations of other threads. With the introduction of transactional memory, ensuring atomicity has become a big challenge in software engineering [29, 27].
- **Deadlocks:** A deadlock happens when the execution reaches a state in which all the threads in the program are in some nonterminal, blocking state. Deadlock is especially an unintended situation for service oriented applications like web services.

10.3 Static Verification Techniques for Concurrent Software

Because of the expansion in the state space of the program due to the thread interleavings, most successful applications of concurrent system verification originated from the composi-

tional approach to static program analysis. In its simplest meaning, compositionality divides the verification problem by decomposing the program into parts from which sufficient properties are proved that make a property of the entire program hold. From static verification point of view, *compositionality* asserts that correctness of a program should be provable by composing the individual proof of its syntactic components so that the complexity of verifying the program in overall should be on the order of the sum of the complexities of verifying its components [72]. The definition implies that compositionality can be exploited for concurrent programs, whose each thread indeed runs some corresponding syntactic component. This requires deriving and proving an appropriate specification for each thread so that the combination of these specifications do not only comply with each other but also imply the properties of the entire program. Although this is easy for concurrency semantics with well-specified communication semantics such as CSP [73], arbitrary interactions on the shared memory make it hard to precisely infer these specifications [74].

In the rest of this section we first briefly present the Hoare’s methodology for sequential programs [75]. While Floyd developed a similar methodology [76] earlier, his technique is not compositional [77]. We will then explain Owicki-Gries’ (non-compositional) approach [42] for concurrent programs and finish with the basic notion of handling concurrency with rely-guarantee [43] and concurrent separation logic [45, 46].

10.3.1 The Hoare Logic

The basic idea in Hoare’s method [75] is to annotate *every* point of the program code (simply each line in the code) with an assertion over the program variables and possibly some auxiliary variables. Each assertion is intended to hold when the program reaches the point of the code annotated with that assertion. A program statement annotated with assertions at the beginning and the end of the statement forms a *proof outline* for that statement. The common notation for a proof outline is the following:

$$\{P\} S \{Q\}$$

where P and Q are assertions and S denotes a statement. This notation is also called a *Hoare triple*. The semantics of this triple states that if S is executed from a program state satisfying P (1) the execution does not violate any assertions in the statement S , and (2) if

S terminates, it ends at a state satisfying Q . P is often called the pre-condition and Q is called the post-condition of the statement S . Blocking executions of S is ignored, so these techniques compute partial correctness proof. In a partial correctness proof, the program is analyzed under the assumption that it always terminates (thus all of its statements). There are attempts to include rules that assure termination [78] for soundness in the case of checking deadlock.

While Floyd included control point information to these assertions, Hoare skipped this information from the assertions P and Q in order to be independent of which control point the statements is contained in the program. This leads to a compositional proof strategy in which proof outlines of statements are composed to obtain the proof outline of the entire program, which is correct by construction. For this, a Hoare-logic based proof approach defines a set of proof rules for deducing proof outlines from the static form of the program using axioms. For primitive statements, e.g., variable assignments, the semantics of triples are defined simple logical rules such as using Dijkstra's pre-conditions [79]. Here, we will not talk about rules for primitive statements but composition of them. The rules for control statements and compositional constructs allow proof rules to be derived in a recursive way. For example, the basic rule for sequential composition is as follows:

$$\frac{P \Rightarrow P_1, \{P_1\} S_1 \{Q_1\}, Q_1 \Rightarrow P_2, \{P_2\} S_2 \{Q_2\}, Q_2 \Rightarrow Q}{\{P\} S_1; S_2 \{Q\}}$$

where “;” is the sequential composition operator for statements. This allows derivation of a larger statement $S_1; S_2$ to be derived using the proofs of its constituent elements S_1 and S_2 .

The derivation rule for the **if** statement is as follows:

$$\frac{\{P \wedge c\} S_1 \{Q_1\}, \{P \wedge \neg c\} S_2 \{Q_2\}, (Q_1 \vee Q_2) \Rightarrow Q}{\{P\} \text{if } (c) \text{ then } S_1 \text{ else } S_2 \{Q\}}$$

Then the pre- ($preCond_M$) and post-condition ($postCond_M$) of a given method M whose body executes the statement S can be proved correct using the rule below:

$$\frac{preCond_M \Rightarrow P, \{P\} S \{Q\}, Q \Rightarrow postCond_M}{\{preCond_M\} \text{procedure } M() \{ S \} \{postCond_M\}}$$

A local assertion c in the statement $S; \text{assert } c$ in a program can be proved correct by writing the proof outline of S as $\{P\} S \{Q\}$ such that $Q \Rightarrow c$. In the weakest pre-condition [79] based approach, the assertion in $S; \text{assert } c$ is proved by writing the triple

for S as $\{wlp(S, c)\} S \{c\}$, where $wlp(S, c)$ denotes the weakest pre-condition of S with respect to the post-condition c . That is, any execution of S from a state satisfying $wlp(S, c)$, if terminates, results in a state satisfying c . If the program is proved correct using this triple, it indicates that there is no execution of the program visiting a state, after S , that violates the assertion c . If S does not contain any loops and procedure calls, $wlp(S, c)$ can be computed automatically and efficiently. For loops and procedure calls, loop invariants and procedure specifications have to be provided manually. The techniques used in ESC/Java [38], WHY [39, 40], and Boogie [41] are improved versions of the Hoare logic described above.

10.3.2 The Owicki-Gries Method

Now we will see how concurrency complicates the analysis presented above. The Owicki-Gries method [42] is an extension of Hoare's logic to concurrent programs. In this method, a concurrent program is proved by defining a rule for parallel composition of statements, each of which are executed by a separate thread. Similar to the sequential case, every statement of the program is annotated with pre- and post-condition, constructing a proof outline. The rule for parallel composition specifies valid triples: $\{P\} S_1 \parallel S_2 \{Q\}$ where $S_1 \parallel S_2$ indicates that S_1 and S_2 are executed concurrently. The triple $\{P\} S_1 \parallel S_2 \{Q\}$ is obtained using the following rule (where $P = P_1 \wedge P_2$ and $Q = Q_1 \wedge Q_2$):

$$\frac{\{P_1\} S_1 \{Q_1\}, \{P_2\} S_2 \{Q_2\}, S_1 \text{ and } S_2 \text{ are interference free}}{\{P_1 \wedge P_2\} S_1 \parallel S_2 \{Q_1 \wedge Q_2\}}$$

The application of a parallel composition rule consists of two steps. First, the proof outlines for each thread are computed separately using the Hoare logic rules for sequential statements. This step is called the *sequential reasoning*, as the proof is done with respect to the sequential semantics. Second, it is proved that the correctness of the Hoare triple for each individual statement is not affected by concurrent executions of other threads. This is called the *interference-freedom check*. Non-interference between two statements $\{P_1\} S_1 \{Q_1\}$ and $\{P_2\} S_2 \{Q_2\}$ can be proved by the following rules:

$$\frac{\{P_1\} S_1 \{Q_1\}, \{P_2\} S_2 \{Q_2\}, \{P_1 \wedge P_2\} S_1 \{P_2\}, \{P_1 \wedge Q_2\} S_1 \{Q_2\}}{S_1 \text{ does not violate } \{P_2\} S_2 \{Q_2\}}$$

$$\frac{\{P_1\} S_1 \{Q_1\}, \{P_2\} S_2 \{Q_2\}, \{P_2 \wedge P_1\} S_2 \{P_1\}, \{P_2 \wedge Q_1\} S_2 \{Q_1\}}{S_2 \text{ does not violate } \{P_1\} S_1 \{Q_1\}}$$

The first rule simply states that whenever S_2 is executed from a state satisfying pre-condition (or post-condition) of S_1 , this execution ends at a state still satisfying the pre-condition (or post-condition) of S_1 . This obviously implies that the sequential reasoning about S_1 is sound in spite of the concurrent environment. The second rule is symmetric. If some interference is detected during the this proof, i.e. execution of a thread violates the assertion on another thread, the appropriate assertions are updated accordingly.

Recall that, for sequential programs Hoare triple required to prove assertions can be computed automatically (except that loop invariants and procedure specifications are given manually) using weakest pre-conditions. However, automatically computing the Owicki-Gries proof outline for a concurrent program is both inefficient and impractical. This is because the weakest precondition generation should consider all possible interleavings of the program, and when an interference check fails, the pre- and post-condition of the failing proof outline must be strengthened to rule out the interference. Therefore, the Owicki-Gries proofs are performed manually. Since the triples for sequential code can be produced automatically, an Owicki-Gries proof requires as manual input only the assertions for interleaving points.

10.3.3 The Rely/Guarantee Approach

The Owicki-Gries method non-compositional, because the second step of the analysis uses all the statements in the program to prove the interference-freedom condition. When the program is modified the interference freedom checks must be repeated. Notice also that the internal implementation details of the modules visible at both steps of the analysis. Following the above observation, compositional extensions of Owicki-Gries combine the two steps, the sequential reasoning and interference-freedom check, into one by extending the proof outline so that the outline specifies assumptions about the interference between a statement S and other statements concurrently executing:

$$\{P, R\} S \{Q, G\}$$

The pre-condition P and the post-condition Q are extended with the rely condition R and the guarantee G , respectively. While P formulates valid starting states for the statement, R makes an assumption about the concurrent operations of the environment in which the proof outline will be valid. Similarly, while Q formulates possible end states for the statement, G

restricts the operations S can perform so that it does not interfere with other threads in an unintended way. While P and Q are predicates specifying program states, R and G specify pairs of states thus transitions between states. The compositionality is achieved by proving the validity of the proof outline of each statement separately with respect to the operational semantics of the statement. The proof rule for parallel composition is as follows [78]:

$$\frac{\{P, R_1\} S_1 \{Q_1, G_1\}, \{P, R_2\} S_2 \{Q_2, G_2\}, ((R \vee G_1) \Rightarrow R_2), ((R \vee G_2) \Rightarrow R_1), ((G_1 \vee G_2) \Rightarrow G)}{\{P, R\} S_1 \| S_2 \{Q_1 \wedge Q_2\}, G}$$

In the rule above, R and G are the assumption and guarantee specifying the concurrent environment in which $S_1 \| S_2$ will execute. Each of S_1 and S_2 are verified under weaker assumptions (R_1 and R_2) and stronger guarantees (G_1 and G_2), and this ensures the soundness of the isolated proofs of S_1 and S_2 . A rely/guarantee-based proof subsumes the interference-freedom check of Owicki-Gries, i.e., the interference-freedom condition of the program is implied.

The noteworthy examples of these are rely/guarantee (R/G) techniques [78, 80] for shared variable systems, starting from Jones' work [43], and the assumption-commitment (A/C) techniques [81] for the CSP model, and Abadi and Lamport's work [82] on assume-guarantee for open/closed components. While the rely/guarantee methodology distinguishes the assumption and the guarantee conditions from the pre-condition and post-condition of the statement as illustrated above, others combine pre-condition-assumption and post-condition-guarantee pairs into single assertions.

The assume-guarantee based compositional methods have two main promising advantages for real programs. First, *open software*¹ can be verified by defining a safe environment through an assume-guarantee specification. In this case the specification restricts the interaction between the environment and the program so that the latter can be proved correct [82]. Notice that this proof is only valid for the executions of the program in an environment satisfying the assumptions declared in the specification. Second, large programs can be verified in a modular way, by abstracting the cohesive components to derive rely/guarantee specifications only referring to globally visible operations of the modules. Then they are

¹ An open system is one running on a partially-specified environment. Hence, software libraries can be considered open software since they are developed independent of the environment they are used.

used to prove properties about interactions among the modules without dealing with their internal implementation details.

10.3.4 Concurrent Separation Logic

Concurrent Separation Logic (CSL) [45, 46] is an extension of Separation Logic [83, 44] to reason about concurrent programs. The proof outlines used in the approaches above describe the operation of the statement on the whole memory. Thus, P and Q in every triple $\{P\} S \{Q\}$ must specify the state of the entire memory; the unspecified portions of the memory are assumed to change in an arbitrary way. The idea in CSL is to identify independent regions of memory at every point of the program and specify the effect of each statement on a separate region of the memory on which the statement operates on; the other regions of the memory are left intact. This avoids to reason about the effect of each operation on the global memory and simplifies the verification task if the sharing of memory regions among the threads is low. The separated memory regions often correspond to thread-local memory, and the approach reduces to sequential reasoning for each separate region of the memory.

In CSL, the proof outlines are written in the separation logic. The logic allows explicitly specifying different portions of the memory, using the separating conjunct $*$. The assertion $P * Q$ states that the memory can be divided into two distinct parts, one satisfying P and the other satisfying Q . Then, the triple $\{P * R\} S \{Q * R\}$ states that the statement S transforms a memory region specified by P so that it satisfies Q at the end of the statement; the memory region specified by R is not affected by the execution of S .

The frame rule below allows one to extend the effect on a statement on a larger portion of the memory.

$$\frac{\{P\} S \{Q\}}{\{P * R\} S \{Q * R\}}$$

The parallel composition rule below states that if two threads operate on distinct regions of the memory, the parallel composition of the threads can run safely on the combination of these regions without affecting the result of each other on their own regions.

$$\frac{\{P_1\} S_1 \{Q_1\} \quad \{P_2\} S_2 \{Q_2\}}{\{P_1 * Q_1\} S_1 || S_2 \{P_2 * Q_2\}}$$

While CSL has the ability to maintain separation between shared and local memory, however, this is not particularly useful for programs with large regions of shared memory and high level of interference. For this reason, there has been a significant body of research on combining CSL with the rely/guarantee-based reasoning [84, 85, 86], starting with the RGSep framework of Vafeiadis et al. [84]. RGSep provides a proof system that separates the memory into two distinct spaces, local and shared. The effect of program statement is specified using the triple:

$$\{P, A\} S \{Q, G\}$$

where P and Q are separation logic formulas that specify the state of the local and shared memory. While the effect of each operation on the local memory is managed by the separation logic formula, the effect of the operation on the shared memory is managed using the rely/guarantee conditions. The combined approach has been used successfully in proving highly-concurrent data structures with fine-grained locking and non-blocking algorithms [87, 88].

10.3.5 Summary and Discussion

In the Owicki-Gries approach [42], each potential interleaving point in a program must be annotated with an invariant that is valid under interference from other concurrently-executing actions. Rely/guarantee methods [78, 80, 89] make this approach more modular by obviating the need to consider each pair of concurrent statements separately. Instead, the guarantee and rely conditions of a thread provide a summary for transitions taken by this thread and the transitions taken by environment threads, respectively. However, the rely/guarantee approach does not eliminate the need to annotate every possible interleaving point of the program; it only eliminates the need for explicitly performing the interference-freedom check for every pair of statement. Concurrent separation logic [44, 46] has the ability to maintain separation between shared and local memory dynamically. Similarly to our method, it enables sequential reasoning for multithreaded programs. All of the techniques above require at each possible interleaving point annotations. If the program is written using fine-grained actions, there are a lot of these points. If the threads in the program interact with each other using many shared variables and intricate concurrency protocols such as non-blocking variables updates, then the annotations at the interleaving points become complex

since the proof outline should specify all the intermediate states between fine-grained actions.

By converting the original program into a simpler program that uses coarse-grained atomic actions, our method enhances the applicability of all of these approaches. For example, the need for auxiliary variables to encode intermediate states at the interleaving points in a code block may be eliminated by transforming that code block to an atomic statement containing no interleaving points. In addition, the user does not have to write annotations within an atomic statement, as these annotations can be derived automatically using weakest pre-conditions. At the same time, our method can benefit from these approaches as well. For example, the ability of concurrent separation logic to reason about dynamic ownership transfer of heap objects could be useful for establishing that heap accesses are non-conflicting, thereby enabling the key step of reduction in our method. The same symbiotic relationship applies to the approach in [90] where method contracts and object invariants are used to specify sharing and ownership constraints on `Spec#` objects.

10.4 Abstraction and Reduction in the Literature

There has been extensive work on computing precise abstractions for verification purposes [91]. Most of these techniques are utilized a lot in software model checking [92]. Abstract interpretation is a technique to describe the concrete state space and computations of the program in terms of those over an abstract domain of finite size [93]. A special case is predicate abstraction [94], a predominant technique with a lot of variants from the literature. In predicate abstraction, each program state represents a valuation to a set of first order predicates. These predicates are over the program variables and represent knowledge about program states, which is relevant for proving properties of the program. The partition of the states with respect to the values of predicates on these states yields a state space abstraction. Then operations of the program are abstracted as transitions between these partitions, so that each operation changes the values of some predicates in the abstract sense. As each predicate is seen as a conceptual boolean variable, the resulting abstract program is called a boolean program. In cases where the abstraction is too coarse, i.e. the level of the detail is not sufficient, the abstraction can be iteratively refined by adding more predicates to the abstraction that capture more detailed information about the actual program, as done in the Counter-Example Guided Abstraction Refinement (CEGAR) method [95].

In contrast with predicate abstraction, the notion of abstraction in QED abstracts the program to a new program in the same domain, i.e., types of variables do not change but operations are allowed to read or write extra values to/from variables. In addition, the main purpose of using abstraction in our approach is not to model the program over a finite domain and make the analysis decidable, but to eliminate the interference among the actions of the program and to help reduction to compute atomic blocks.

Several verification approaches in the literature use Lipton-based reduction [47] as a key ingredient [48, 49, 50, 51]. These approaches are different from ours in the following aspects:

1. Their commutativity analyses are based on type and effect systems applying simple, syntactic conditions for mover types. For example, the most-used scheme declares lock acquires as right movers, lock releases as left movers, and accesses to local variables and lock protected variables as both movers. This sort of analysis can only reason about commutativity of accesses that are not simultaneously enabled.
2. They are limited to simple synchronization disciplines, mostly to locks and reader-s/write locks. The synchronization information, e.g., which locks are held by a particular thread at each point, is used to reason about show that two accesses execute at different synchronization states.
3. They apply reduction at one level. That is, the reduction rules are not applied after the atomic blocks get large enough that they start interfering with each other. The reduction is not enforced as long as there is interference between actions.

Regarding the first issue, our notion of reduction is general and not restricted to syntactic rules and actions' being not simultaneously enabled. We perform the commutativity check by generating verification conditions (VC) that capture the operation of the actions precisely and verifying these VCs automatically using SMT solvers. This enables us to deduce that certain simultaneously enabled accesses commute.

[96] addresses the second issue above by using auxiliary variables and access predicates to enable wider application of reduction. In addition to regular locking primitives, [51] applies mover-analysis to non-blocking synchronization primitives LL, SC and CAS under certain execution patterns of these primitives. As demonstrated in this thesis, our method supports

intricate synchronization mechanisms including non-blocking synchronization naturally, with moderate annotation burden.

In order to address the third issue, abstractions have been used as a mechanism to prove atomicity in the work on purity [50, 51]. The idea in purity is to specially treat *pure* statements while computing atomicity using reduction. Normal terminations of a pure statement cause apparent interference thus make the containing statements irreducible. However, a pure code block modifies the program state only when it terminates exceptionally. Normal terminations of pure statement are abstracted and are omitted from the atomicity computation. For example, consider a pure loop given below.

```
while(true) { // pure loop
    t := x;    // apparent interference on x
    fx := f(t);
    acquire(lock);
    if(x == t) {
        z := fx;
        release(lock);
        break;
    }
    release(lock);
}
```

The loop body read from a shared variable `x` to the local variable `t` and computes a long running function on `t`. Then in a lock-protected block it checks if `x` has not been overwritten by any other thread. The normal terminations of the loop is represented by its non-final iterations, and the exceptional termination of the loop is represented by its final iteration. The loop writes to shared memory only in its final iteration. In the purity-based atomicity analysis, all the non-final iterations of the loop are abstracted to no-ops, and only its final iteration, which is ensured to be atomic by the use of the lock, is considered. The abstraction step in our method subsumes purity and allows us to eliminate pure code blocks from the analysis through simple variable abstractions (see Section 19.1 for the proof of a similar example to the one above). In addition our method has the ability to combine abstraction with reduction in multiple levels; we can eliminate interference through abstractions whenever reduction gets stuck.

As a matter of fact, experiments with restricted definitions of atomicity can extensively

reduce the state space in model checking [97, 54]. The basic idea in the use of atomicity in model check is that in well-synchronized programs, any computation of a thread is modeled as a sequence of transactions. Each transaction appears to execute atomically to other threads. During state exploration, the model checker schedules the threads only at the boundaries of these transactions. If each shared variable in the program is protected with a lock, then these transactions are determined by using the reduction-based atomicity inference technique in [48]. The formulation of reduction in our method can also be used in similar model checking systems to identify the transactions on-the-fly or offline.

This chapter gave an overview of the existing approaches to the verification of concurrent software. The approach we present in the following chapters will combine the strengths of the current techniques referred to above for a more elaborate treatment to concurrent software.

Chapter 11

MOTIVATION AND OVERVIEW

In this chapter, we overview our approach to verifying assertions in a multithreaded program with reduction and abstraction using several examples. In particular, we demonstrate the key role of reduction in simplifying the assertion checking and the interaction between reduction and abstraction to enable further applications of each other.

Outline. Section 11.1 describes the notation we use in the examples and gives an informal introduction to reduction and abstraction. Section 11.2 illustrates the application of reduction using a lock-based program, and Section 11.3 explains the proof of the lock-free version of this program to demonstrate a simple, alternating use of reduction and abstraction. Section 11.4 presents a nontrivial interaction between reduction and abstraction.

11.1 Preliminaries

11.1.1 Notation

The examples are written in the syntax of our input language QEDPL. We use `if(*)` and `while(*)` to denote that whether the body of `if` and `while` are executed or not is chosen nondeterministically. Assignments are denoted `x := E` where `x` is a variable and `E` is an expression that evaluates to a value whose type is the same as that of `x`. The statement `havoc x` assigns to `x` a nondeterministic value allowed by the type of `x`.

The statements `assume B` and `assert B` take an expression `B` that is evaluated to a boolean value when the statement is executed. If `B` evaluates to `true`, then both `assume B` and `assert B` are equivalent to no-ops, i.e., they do not have any side effects on the execution of the program. If `B` evaluates to `false`, `assume B` makes the current execution to be ruled out (not considered as an execution of the program) by our analysis. If `B` evaluates to `false`, `assert B` we say that an assertion is violated, so the current execution fails. Therefore, we use `assume B` to specify a fact that is assumed to be always true when executing the

statement, and we use `assert B` to specify a fact that must be verified to be true when executing the statement.

In this chapter, when talking about program executions, we will let *action* denote the program statements that are executed in one step, indivisibly by another thread. By an execution of the program we refer to a sequence of actions by different threads. In the example programs, each line of code is assumed to be a separate action. In addition, we use square brackets (`[...]`) to simply express that a code block spanning multiple lines is a single action. In the actual syntax of QEDPL (see Figure 14.1), `[...]` is replaced by the wrapper `atomic{...}`.

In the example programs, we enforce that each line makes at most one access to only one global variable; otherwise, we divide the line into multiple lines each performing at most one global variable access. For example, we split the increment of `x` (`x := x + 1`) in Figure 11.1 and the assignment of `newsize` to `currsz` (`(currsz := newsize)`) in Figure 11.6 (lines 15–16) into multiple lines.

11.1.2 Informal Introduction to Reduction and Abstraction

Reduction produces a single atomic action from the sequential composition of two atomic actions if either the first action is a right mover or the second action is a left mover. An action R commutes to the right of X if the effect of thread t executing R followed by a different thread u executing action X can be “simulated” by first thread u executing X followed by thread t executing R . If R commutes to the right of all actions in the program, it is said to be a right mover. Two actions are said to *conflict* (or make conflicting accesses) if they access to the same variable and at least one of these accesses is a write (the other can be either a read or a write). Common cases where R commutes to the right of X are (1) if R and X do not conflict on any common variables, (2) if R is nondeterministic, i.e., independent of the program store, or (3) if R disables X . A left mover is defined similarly: If R commutes to the left of all actions in the program, it is said to be a left mover. Common cases where an action L commutes to the left of X are (1) if L and X do not conflict on any common variables, (2) if L is nondeterministic, i.e., independent of the program store, or (3) if L fails from every state X can execute without blocking. The typical set of rules for deciding movers in a lock-based program is as follows [48]: Lock acquires are right movers

and lock releases are left movers. Accesses to local variables and race-free accesses to shared variables are both movers, i.e., they are both right and left movers.

Reduction iteratively determines the mover types of actions and combine them to obtain larger actions until no more action can be combined with another. At this point, we use abstraction to replace actions with new actions that will be movers and be combined in a further application of reduction. In QED, an action is abstracted in two ways: (1) by adding an assertion into the statement, and (2) replacing it with another action that will allow more non-failing transitions from the same state. An example to (1) is replacing a line `t := x` with `[assert owner == tid; t := x]`. The new action violates the assertion if `owner` stores a different value than `tid` when the action is executed. An example to (2) is replacing a line `t := x` with `havoc t`. While the old action only allows `t` to store the value of `x` when it terminates, the new action allows `t` to store any value (including that of `x`) when it terminates.

11.2 Lock-based Increment

In Figure 11.1, the procedure `inc` on the left is a lock-based implementation for atomically incrementing a shared, global variable `x`. We assume that only the actions in `inc` access `x` within the entire program. The procedure `main`, run by thread `T1` first sets `x` to 0, and creates two threads `T2` and `T3` using the statement `dopar`. Each branch of the `dopar` statement executes a call to `inc` by a different thread. The variable `t` is a local variable, so each of `T2` and `T3` access a distinct copy of `t` while executing `inc`. While `dopar` is executing, `T1` waits for both `T2` and `T3` to terminate. When both `T2` and `T3` terminate (when both calls to `inc` return), `T1` joins its child threads and asserts that `x == 2`. The assertion checking problem for this program is to show that for every execution of the program, the assertion `x == 2` is satisfied.

Through an iterative application of reduction and abstraction, we can transform bodies of these procedures each into single atomic action as shown in Figure 11.2 and indicated by square brackets. The transformation from Figure 11.1 to Figure 11.2 is performed by a combination of reduction and abstraction steps described below.

The initial sequence of abstraction and reduction steps combines the body of `inc` into a single atomic action (Figure 11.3(a)-(d)). We first show that the procedure `inc` atomically


```

procedure inc() {
  var t:int;
  acquire(lock); // Right mover
  t := x;        // Both mover
  t := t+1;       // Both mover
  x := t;        // Both mover
  release(lock); // Left mover
}

procedure main() {
  x := 0;        // run by T1
  dpar {
    call inc(); // run by T2
  } with {
    call inc(); // run by T3
  }
  assert x == 2; // run by T1
}

```

Figure 11.1: Lock-based atomic increment

```

procedure inc() {
  var t:int;
  [havoc t; x := x+1];
}

procedure main() {
  var t1,t2:int;
  [x := 0;
   havoc t1; x := x+1;
   havoc t2; x := x+1;
   assert x == 2;]
}

```

Figure 11.2: Lock-based atomic increment after being transformed

increments x by obtaining the body of `inc` as shown in Figure 11.3(d). Notice that this actually is the case, since the lock prevents actions of threads running `inc` from being interleaved with each other. Thus, every execution of the program is equivalent to an execution in which `inc` appears to run sequentially, and `inc` is said to be atomic.

An action α being a mover indicates that for any action β in the program (including α), either β does not follow α , or α does not conflict with β , i.e., either α and β access different variables or all their accesses to common variables are reads. The lock ensures the former condition when α and β are both actions of `inc`. Expressing this feature of the lock in the program is essential in showing that the actions guarded by the lock are movers. We accomplish this by annotating the program with assertions. In Chapter 18, we present an annotation assistant implemented in our tool QED-VERIFIER that automatically infers these assertions. The annotated version of `inc` is seen in Figure 11.3(a).

```

procedure inc() {
  var t:int;
  [acquire(lock); m := tid;]
  [assert m==tid; t := x;]
  [assert m==tid; t := t + 1;]
  [assert m==tid; x := t;]
  [assert m==tid;
   release(lock); m := 0;]
}
(a)

procedure inc() {
  var t:int;
  [
    acquire(lock); m := tid;
    assert m==tid; t := x;
    assert m==tid; t := t + 1;
    assert m==tid; x := t;
    assert m==tid;
    release(lock); m := 0;
  ]
}
(b)

procedure inc() {
  var t:int;
  [
    acquire(lock); m := tid;
    t := x;
    t := t + 1;
    x := t;
    release(lock); m := 0;
  ]
}
(c)

procedure main() {
  var t1,t2:int;
  x := 0;
  dopar {
    [havoc t1; x := x+1;] // T1
  } with {
    [havoc t2; x := x+1;] // T2
  }
  [havoc t2; x := x+1;] // T3
}
assert x == 2;
// T1
}
(f)

```

```

procedure inc() {
  var t:int;
  [acquire(lock);
   t := x;
   t := t+1;
   x := t;
   release(lock);]
}
(d)

procedure inc() {
  var t:int;
  [havoc t;
   x := x+1;]
}
(e)

```

Figure 11.3: Intermediate transformations on the lock-based increment

The annotations are inserted to atomic actions as assertions and assignments, which refer to an auxiliary variable, `m`. The domain of `m` is $Tid \cup \{0\}$ where Tid is the set of all thread ids ($0 \notin Tid$). The annotations add auxiliary information to the state of the mutex lock: If `m==0` the lock is free, and otherwise `m` holds the id of the thread that has acquired the lock. Variable `tid` is a special global variable: It stores the id of the currently executing thread. The assertion `m==tid` is used to explicitly mark the lines that a thread executes while holding the lock. Note that, adding an assertion to an action is a legal way of abstracting the action. Therefore, it is made explicit that two actions with these assertions may not immediately follow each other. Then, it is proved that actions of `inc` have proper mover types to combine then into a single action. Figure 11.3(b) shows the new body of `inc`. Once we obtain the body of `inc` as a single action, we can discharge the assertions about `m` by a sequential check. Figure 11.3(c) shows the body of `inc` after discharging the assertions. Since we proved all the assertions about `m` and the variable `m` is no longer needed for the rest of the proof, we transform Figure 11.3(c) to Figure 11.3(d) by hiding the variable `m`.

The assertions are added to the program as extra proof obligations. If the synchronization idiom is properly used in the program, these assertions are eventually proved as valid sequentially after all the actions are merged into one action. Note that, our annotation assistant explained in Chapter 18 automates the annotation of actions with assertions to identify code blocks whose atomicity is ensured by synchronization idioms. One of the following two outcomes of our annotation generation technique indicates a possible bug in the program: (1) An action that is expected to be protected by the synchronization idiom is not annotated with an assertion. (2) Assertions that are added to actions do not help in making that code block atomic by reduction. An example to the former case is an action that is expected to be protected by a lock `lock` but not surrounded by `acquire(lock)` and `release(lock)` operations. As an example to the latter, assume that there exists a `dec` procedure (for decrementing `x`) that accesses `x` but does not use the lock used by `inc`. In this case, our automatic annotation method provides no annotation for the actions of `dec` because of the missing synchronization. Failing to annotate `dec` even though `inc` is annotated makes actions of `inc` non-movers, which prevents `inc` from being reduced to a single action. As a result, the assertions in `inc` will not be validated since this requires having `inc` as a single action. The failing mover analysis between actions of `inc` and `dec` can easily be

used to diagnose the missing synchronization in `dec`.

When we reach the version of the program shown in Figure 11.3(d), we apply abstraction to replace its body with the one in Figure 11.3(e). The new body of `inc` (1) summarizes the multiple-step computation using `t` to a simple increment of `t`, and (2) writes a nondeterministic value to `x`, and (3) does not call the operations on `lock`. (2) and (3) are sound since `t` and `lock` are not needed any more for reasoning about the assertion and `lock` is not modified by `inc`. In this way, the abstraction step decouples `x` from the “implementation-level” variables `t` and `lock`.

Figure 11.3(f) shows the body of `main` after inlining the body of `inc` at the call sites. At this point, there are only two different thread interleavings (either `T1-T2-T3-T1` or `T1-T3-T2-T1`) of the program, while the original version of the program in Figure 11.1 has 252 potential thread interleavings. The actions run by `T2` and `T3` in Figure 11.3(f) are both movers since two increments of `x` commute both to the right and left of each other. This indicates that the executions of the program following the interleavings `T1-T2-T3-T1` and `T1-T3-T2-T1` terminate at the same final state. Therefore, we can replace the parallel composition of actions in `dopar` with a sequential composition these actions and combine them with the actions of `T1`, finally producing a single action given on the right in Figure 11.2. Intuitively, replacing `dopar` with sequential composition means to restrict the new program to only one of the equivalent (with respect to the final state) interleavings `T1-T2-T3-T1` and `T1-T3-T2-T1`.

The assertion is discharged within the sequential body of `main` in Figure 11.2. In Section 13.7 we prove a soundness theorem that states that verifying the assertion in the sequential body of `main` guarantees that every run of the concurrent body of `main` in Figure 11.2 satisfies its assertion. The example clearly shows that our transformations in QED reduce a complex analysis over many executions of a concurrent program to a simpler analysis over sequential code.

11.3 Lock-free Increment

In Figure 11.4, the procedure `inc` on the left uses the `CAS` (Compare-And-Swap) primitive for a lock-free implementation the increment of `x`. `CAS(x, t, t+1)` atomically performs the following operation: It compares the value of `x` with `t`. If `x` with `t` have the same value,

it writes $t+1$ into x and returns **true**; otherwise, it leaves x unchanged and returns **false**. Note that, this program has conflicting accesses to x that are simultaneously enabled, and is consequently more difficult to reason about than the previous version.

In the lock-based increment example, we used a combination of reduction and assertion-based abstraction to show that the body of **inc** in Figure 11.1 is atomic. On the contrary, the body of **inc** in Figure 11.4 is in fact not atomic. For this example, we combine reduction with a new kind of abstraction to obtain a new body of **inc** that can be shown to be atomic and still increments x correctly. The sequence of transformations for this are shown in Figure 11.4.

First, we perform a simple transformation (as in [50]) that peels out the last iteration of the loop, thereby arriving at the version of **inc** in Figure 11.4(b). Next, we argue that it is a valid abstraction to replace the read of x in $t := x$ with **havoc** t since this only increases the set of possible behaviors. We do not expect this abstraction to lead to additional assertion violations since the result of the read action is later verified by **CAS**($x, t, t+1$); reading a value other than the actual value of x would simply lead to a failed **CAS** operation.

```

procedure inc() {
  var t:int, b:bool;
  while (true) {
    t := x;
    call b := CAS(x,t,t+1);
    if (b)
      break;
  }
}

procedure inc() {
  var t:int;
  while (*) {
    t := x;
    assume x!=t;
  }
  t := x;
  [assume x==t;
  x := t+1];
}

procedure inc() {
  var t:int;
  while (*) {
    havoc t;
    skip;
  }
  havoc t;
  [assume x==t;
  x := t+1];
}

[havoc t;
x := x+1];
}

```

(a) (b) (c) (d) (e)

Figure 11.4: The lock-free increment example

```

procedure add(n:int) {
  while (0<n) {
    call inc();
    n := n-1;
  }
}

procedure add(n:int) {
  while (0<n) {
    [x := x+1];
    n := n-1;
  }
}

procedure add(n:int) {
  [if(0<n) {
    x := x+n;
    n := 0;
  }];
}

```

(a) (b) (c)

Figure 11.5: Client of inc

We also abstract each unsuccessful execution of `CAS(x, t, t+1)` inside the loop with the statement `skip` to arrive at the version of `inc` in Figure 11.4(c). This is safe for the assertion, since both the body of the loop (`t := x; assume x!=t`) and the abstracted body do not modify `x`. Replacing `t := x` with `havoc t` makes the new version of `inc` to guess the value of `x` instead of explicitly reading `x`. As a result, although there is not interference on `x`, i.e., `x` is not updated by any other thread between `havoc t` and `CAS(x, t, t+1)`, `inc` may guess a wrong value for `x` and fail to update `x`. However, adding nondeterministic failures to `inc` in this way does not affect the correctness of the program with respect to the assertion in `main`, as the successful termination of `inc` correctly increments `x`.

In the version of `inc` in Figure 11.4(c), every operation except for the last one accesses only local state and is consequently both a right and a left mover. Recall that an action that accesses only local variables is a both mover, since it does not conflict with other actions by different threads. Therefore, we can use reduction to summarize the entire loop with a single `havoc t` statement to arrive at the version of `inc` in Figure 11.4(d). Finally, we reduce the resulting sequential composition to arrive at a single atomic action in the version of `inc` in Figure 11.4(e).

Figure 11.5(a) shows the procedure `add`, a client of `inc`. Procedure `add` has a parameter `n`. If `n` initially has a positive value, then `add` increments `x` by `n` by calling `inc` repeatedly in a loop `n` times. Once the procedure `inc` has been transformed into an atomic increment of `x`, we can reason about `add` by replacing the call to `inc` with an atomic increment of `x` to obtain the version in Figure 11.5(b). In this version, every action other than the atomic increment accesses only local variables. Moreover, atomic increments of `x` commute with each other, and there is no other concurrently running action in the program that accesses `x`. Therefore, every action in the second version of `inc` is both mover. Consequently, the entire loop is reducible to a single atomic action. Now, using purely sequential reasoning techniques and writing an appropriate for the loop in `add`, we perform abstraction on this atomic action. The resulting atomic action is semantically equivalent to the body of the version of `add` in Figure 11.5(c). In Section 15.3.1, we give the full-detailed proof steps for reducing a similar loop in Figure 11.5(b) to the action in Figure 11.5(c) (see Figures 15.3 and 15.4).

11.4 A Device Cache

```

procedure Read (start : int, size : int)
  returns (buffer : [int]int, bytesread : int)
{
  var i, j, tmp : int;

  1  acquire(lock);
  2  i := currsize;
  3  if (start + size <= i) {
  4    release(lock); goto COPY_TO_BUFFER;
  5  } else if (newsize > i) {
  6    size := (start <= i) ? i - start : 0;
  7    release(lock); goto COPY_TO_BUFFER;
  8  } else {
  9    newsize := start + size;
  10   release(lock); goto READ_DEVICE;
  } // end if

  READ_DEVICE:
  11 while (i < start + size) {
  12   cache[i] := device[i];
  13   i := i + 1;
  }
  14 acquire(lock);
  15 tmp := newsize;
  16 currsize := tmp;
  17 release(lock);

  COPY_TO_BUFFER:
  18 j := 0;
  19 while(j < size) {
  20   buffer[j] := cache[start + j];
  21   j := j + 1;
  }
  22 bytesread := size; return;
}

```

Figure 11.6: Original device cache program

```

procedure Read (start : int, size : int)
  returns (buffer : [int]int, bytesread : int)
{
  var i, j, tmp : int;

  A [if (*) {
    havoc i, size;
    assume (size == 0 || start + size <= currsize);
    goto COPY_TO_BUFFER;
  } else {
    i := currsize;
    assume newsize <= i && i < start + size;
    newsize := start + size;
    goto READ_DEVICE;
  }]

  READ_DEVICE:
  B while (i < start + size) {
  C   [assert currsize <= i; cache[i] := device[i]];
  D   i := i + 1;
  }
  E [tmp := newsize;
    currsize := tmp];

  COPY_TO_BUFFER:
  F j := 0;
  G while (j < size) {
  H   [assert start + j < currsize; buffer[j] := cache[start + j]];
  I   j := j + 1;
  }
  J bytesread := size; return;
}

```

Figure 11.7: Transformed device cache program

We have illustrated abstraction and reduction using two small examples. We now demonstrate the symbiotic nature of these two techniques using a larger and nontrivial example. In the following, in order to make it easier to follow the proof steps, sentences that constitute a proof step are indicated by **(S1)**, **(S2)**, etc.

11.4.1 Device Cache Operation

Procedure **Read** in Figure 11.6 reads a number of bytes from a device's physical storage, modeled by the variable **device**. Client threads call **Read** to request to read (up to) **size** bytes from **device** into the output parameter **buffer**, starting from index **start** in the device. **Read** returns the number of bytes it was able to read (possibly less than **size**) from the device in the output parameter **bytesread**. In order to make subsequent requests to **Read** faster, the implementation of **Read** caches the bytes read into an (unbounded) memory buffer **cache**. For this example, we assume the invariant that all the variables of type **int** store non-negative integers. The reader can informally assume that all integers are initialized to 0. The variables **device**, **cache** and **buffer** are all maps indexed by integers starting from 0.

The variable **currsz** stores the number of bytes from the device that are already available in the cache. When there are enough bytes in the cache (lines 2-4), **Read** jumps to **COPY_TO_BUFFER** to copy the contents of **cache** to **buffer** (lines 18-22). In this case, the number of bytes read is the same as the number of bytes requested by the client.

If the cache does not contain all the bytes requested, there are two possibilities:

- If in line 5 a thread t observes **newsize** > **currsz**, this means another thread is in the process of copying bytes from the device to the cache as will be explained below. In this case, t can read the portion of **cache** between 0 and **currsz** (lines 18-21). If **bytesread** < **size** at return, the client thread t may retry later for the rest of the bytes.
- Otherwise, thread t reads the missing bytes from the device into the cache. The driver allows only one thread to read from the device and write to the cache. This is implemented by the following synchronization policy. The first thread that attempts to read from the device sets the variable **newsize** to **start** + **size** (line 9), and then

performs the actual read by jumping to `READ_DEVICE` (line 11). While `newsize` is greater than `currrsize`, no other thread attempts to access the section of `cache` and `device` between `currrsize` and `newsize`; the code at lines 5-7 detects that `newsize > currrsize` and resets `size`, the number of bytes to read from `cache` accordingly. As explained above, during this period, another thread can read the portion of `cache` between 0 and `currrsize`. The thread that jumped to `READ_DEVICE` sets `currrsize` to `newsize` (lines 14-17), which makes the recently read bytes from the device available for further calls to `Read` by all threads.

11.4.2 Synchronization Mechanisms

This example uses two different mechanisms to synchronize the client threads.

- The shared lock `lock` protects all accesses to `currrsize` and `newsize`. The `acquire(lock)` and `release(lock)` operations in Figure 11.6 are used to acquire and release this lock. Using `lock` allows us to reduce the code blocks from lines 1-10 and lines 14-17 into atomic actions (**S1**). This increased granularity of atomicity also allows us to introduce and prove the invariant `currrsize <= newsize`.
- Only one thread at a time is allowed to update `newsize`, make the jump to `READ_DEVICE` from line 10, and update `cache`. We capture this synchronization by introducing a conceptual, auxiliary lock variable `al` that is associated with the locking predicate `currrsize < newsize`.
 - **Acquiring `al`:** The lock `al` is available to a thread in a state where `currrsize == newsize`. A thread increases `newsize` (line 9), making the locking predicate `true`, to acquire `al`. Acquiring `al` fails as long as `currrsize < newsize` holds, which indicates that there is a thread currently holding the lock and accessing device.
 - **Releasing `al`:** A thread sets `currrsize` to `newsize` (line 15–16), making the locking predicate `false` to release `al`.

Thus, the lock `al` is acquired just before the jump to `READ_DEVICE` and held during the execution of lines 11–17.

11.4.3 Specification

We would like to verify that if the input parameters of `Read` satisfy `0 <= start && 0 <= size` at entry, then the output parameters of `Read` satisfy `(forall x:int. start <= x && x < start + bytesread ==> buffer[x] == device[x])` at exit. This task is straightforward if `Read` executes without any interference. Our goal in the following is to convert the entire body of `Read` into an abstract atomic action that is strong enough to verify the aforementioned property.

11.4.4 Verification

We will argue that lines and actions that access shared variables are of the desired mover types in order to allow us to convert the body of `Read` into an atomic action. We will argue that

- the update of `cache` at line 12 is a right mover (**S2**), and
- the read of `cache` at line 20 is a left mover (**S3**)

Note that `buffer` is a local variable and `device` is assumed to be written once before any calls to `Read`, so the only possible conflict between these lines could be due to the variable `cache`. The execution of line 12 is not simultaneously enabled in two different threads because of the lock `a1`. The argument that line 12 does not conflict with line 20 requires the introduction of assertions just prior to the execution of these actions. These assertions are shown on lines C and H in Figure 11.7; they are checked atomically with the execution of the action (shown by surrounding square brackets). These assertions capture why the update and read of `cache` do not conflict: from a state in which both hold, the update and read access different indices of `cache`!

The transformation of line 12 and line 20 in Figure 11.6 into line C and line H in Figure 11.7 introduces a dependency on the variable `currsz` because of the assertion annotations. Therefore, we must also argue that lines C and H commute to the right and left respectively of the action updating `currsz` on line E. Due to the conceptual lock `a1`, line C is not enabled simultaneously with line E. Also, line H commutes to the left of line E when executed from a state satisfying the already proved invariant `currsz <= newsz`.

We now show how to transform the atomic action in lines 1-10 of Figure 11.6 into a right mover. In its original form, this action does not commute to the right of the atomic action in lines 14-17. To see this, consider the scenario where lines 1-10 are executed by a thread t from a state in which `currrsize` < (`start` + `size`) < `newsize`; consequently `size` is reduced to either 0 or `currrsize` - `start`. Suppose that this were followed by another thread u executing the action in lines 14-17 and increasing `currrsize` to `newsize`. Clearly, this scenario is not equivalent to one in which u executes first followed by t , since in this case t would not modify `size`.

To transform lines 1-10 into a right mover, we perform two abstractions to get the action shown on line A in Figure 11.7. We first abstract the check at line 5 of Figure 11.6 to a non-deterministic decision and enable lines 6-7 to run even when `currrsize` == `newsize` (S4). This causes a thread to return fewer bytes than `size` even though it could jump to `READ_DEVICE` and read all bytes it needs from `device`. The second abstraction is to update the variable `size` nondeterministically to a value less than or equal to that assigned by the original program in the case when control jumps to `COPY_TO_BUFFER` (S5). This allows the code to read fewer than the number of bytes available to it in the cache. This abstraction is justified since the additional behavior in the transformed `Read` (Figure 11.7) might have occurred if the call had taken place while an update of `cache` was being executed by another thread. This abstraction is also safe from the point of view of proving that the `buffer` contains a valid snapshot of `device` for the first `bytesread` bytes. The transformed `Read` in Figure 11.7 can now be reduced into an atomic action, because lines A-D are right movers and lines F-J are left movers (S6). Finally, we can prove the relevant condition about output parameters at the return point of `Read` using purely sequential reasoning techniques.

We introduce and verify the invariant (`forall x:int. 0 <= x < currrsize ==> cache[x] == device[x]`) once the code in Figure 11.7 has been converted into a single atomic action (S7). A proof of `Read` using the Owicki-Gries proof method at the initial line-level granularity would require all introduced invariants to be included in the annotations at several lines. In addition, it would require facts about global variables established at assignments and conditional checks to be propagated through the annotations in subsequent lines. Our method can handle this example much more simply by introducing invariants such as those in (S7) when atomic blocks are large enough.

In this chapter, we overviewed our verification technique and demonstrated our notions of abstraction and reduction. We showed that the alternating, symbiotic use of abstraction and reduction enables us to transform a program with an intricate concurrency protocol to a program in which assertions can be verified by only using sequential analysis. Starting with the next chapter, we will formally define a proof in QED and explain QED transformations in detail.

Chapter 12

THE ACTIONPL PROGRAMMING LANGUAGE

In this thesis, we present the QED proof system using two programming languages, ACTIONPL and QEDPL. The two languages share the same top level structure of the program: Each program consists of a collection of definitions; the syntax of these definitions are common to ACTIONPL and QEDPL programs. ACTIONPL and QEDPL differ in the syntax of the statements in the procedure bodies: While ACTIONPL provides a simple, formal language for describing the conceptual aspects of QED, QEDPL provides a richer, high-level language for describing the application of QED to conventional languages in practice. In this chapter we give the syntax and small-step operational semantics of ACTIONPL; QEDPL is presented in Chapter 14.

Outline. Section 12.2 formally defines an ACTIONPL program and presents its elements. Section 12.1 introduces the language for expressions. Section 12.3 and Section 12.4 present the syntax and the operational semantics of ACTIONPL statements, respectively.

12.1 Expressions: Syntax and Semantics

In this thesis we set up our logical framework over the first-order predicate logic (FOL) with existential and universal quantifiers. The logic we use in the expressions is a combination of propositional logic, equality with uninterpreted functions (EUF), and linear arithmetic. This logic can be extended by defining new types and constants, functions over these types, and axioms over constants and functions (see Sections 12.2.1 through 12.2.4 for details).

Figure 12.1 gives the syntax of expressions. The over-arrow notation (e.g., $\vec{E}$) indicates a (possibly empty) sequence of items, separated by commas if multiple. In an expression, every *id* is either a (global or local) variable (x or ℓ), a constant (c), or a bound variable (bv). Every expression is either *id* or a function application over expressions, denoted $f(E^*)$, where E^* denotes a (possibly empty) sequence of expressions.

$$\begin{aligned}
& x \in \text{Global} \quad \ell \in \text{Local} \quad bv \in \text{Bound} \\
\\
\text{Var} : \quad v &::= x \mid \ell \\
\text{Constant} \quad c &::= \text{true} \mid \text{false} \mid \text{Integer literals } (\dots, -1, 0, 1, \dots) \mid \text{User defined constants} \\
\text{Function} \quad f &::= \text{select} \mid \text{store} \mid \text{Boolean operators } (f_{\text{bool}} \in \{\neq, \wedge, \vee, \Rightarrow, \Leftrightarrow, <, >, \leq, \geq\}) \\
& \quad \mid \text{Integer operators } (f_{\text{int}} \in \{+, -, *, /, \%\}) \mid \text{User defined functions} \\
\text{Identifier} : \quad id &::= v \mid c \mid bv \\
\text{BExpr} : \quad B &::= id \mid v' \mid f_{\text{bool}}(\vec{E}) \mid \exists bv. B \mid \forall bv. B \\
\text{Expr} : \quad E &::= id \mid B \mid f(\vec{E})
\end{aligned}$$

Figure 12.1: Syntax of expressions

For linear arithmetic, integer literals $(\dots, -2, -1, 0, 1, 2, \dots)$ are treated as special constants of type **int**. The binary operators $+, -, *, /, \%$ are treated as special functions that are shown in the infix notation. Similarly in propositional logic, **true** and **false** are treated as special constants, and the unary operator $\neg$ and the binary operators $=, \neq, \wedge, \vee, \Rightarrow, \Leftrightarrow, <, >, \leq, \geq$ are treated special functions and are shown in the infix notation.

For propositional logic and linear arithmetic, the semantics of expressions is as usual [98]. In addition, the language of expressions are extended by defining new types, functions, constants. The semantics of custom-defined types, functions, and constants is described by a collection of axioms combined with the theory of equality with uninterpreted functions (EUF).

Expr represents the general set of expressions, while $BExpr \subseteq Expr$ represents boolean expressions, also called *predicates*. In this thesis, E ranges over expressions (of any type) and B ranges over predicates.

Let $\text{fv}(E)$ denote the set of free variables in the expression E . We say that an expression E is over V if $\text{fv}(E) \subseteq V$. The expression $E[X/Y]$ refers to the expression obtained by substituting every “free” occurrence of every variable $x_i \in X$ for $y_i \in Y$ in E .

12.1.1 Informal Notation for Expressions

Figure 12.1 gives the formal syntax of expressions using the standard mathematical alphabet $(\{\wedge, \vee, \Rightarrow, \dots\})$. This notation is useful for formal descriptions. However, we sometimes

find it more convenient to use an textual alphabet ($\{\&\&, ||, ==>, \dots\}$) while showing program text. This is an informal but unambiguous notation for formal syntax. The textual equivalents of the mathematical symbols are as follows:

$$\begin{array}{llllll} \neg ! & \wedge \&\& & \vee || & \Rightarrow ==> & \forall x. B \text{ forall } x:: B \\ = == & \neq != & \leq <= & \geq >= & \exists x. B \text{ exists } x:: B \end{array}$$

12.2 Program

Roughly speaking, our notion of a program of global variables and a collection of procedures that operate on these variables. In addition, a program may contain user-defined types, constants, functions, and axioms; these are to extend the logic used in the expressions, and thus, to simplify the representation of the program state and the computation in procedure bodies. In this thesis, by “program text” we refer to the source code of the program.

Formally, a program is represented by a tuple:

$$P = \langle Type_P, Function_P, Constant_P, Axiom_P, Global_P, Proc_P \rangle \quad (12.1)$$

where each item in the tuple is a set of *top-level program elements* of the same kind. Top-level program elements are types ($Type_P$), functions ($Function_P$), constants ($Constant_P$), axioms ($Axiom_P$), global variables ($Global_P$), and procedures ($Proc_P$). The program text consists of definitions, each for a top-level element of the program. Here, “top-level” is used to mean standalone elements of the program. Thus, the definition of each top-level element in the source code corresponds to a separate parse-tree. Therefore, for example, a constant cannot be defined within a procedure, or a type cannot be defined within a function.

Figure 12.2 gives the syntax for defining top-level elements in the program text. We do not allow overloading for any kind of top-level element, so each element is defined using a program-wide *unique* name. For simplicity of presentation, we refer to elements by their names, so, for example, **p** refers to not only the name of a procedure, but also the overall procedure it belongs to.

In this thesis, we only focus on programs that are loaded and type-checked successfully. Therefore, for example, we assume that every function in an expression (or every procedure call in a statement) is given as argument expressions of correct types. In the rest of this section, we explain the top-level elements of a program.

$$\begin{array}{ll}
v \in \text{Var} & f \in \text{Function} \quad c \in \text{Constant} \quad E \in \text{Expr} \quad B \in \text{BExpr} \quad S \in \text{Stmt} \\
\\
\text{Type} : & \mathbf{t} ::= \mathbf{int} \mid \mathbf{bool} \mid \text{Custom type name} \\
\text{TypeDef} : & \text{type} ::= \text{type Custom type name;} \\
\text{TypedIdentDef} : & \text{tident} ::= v : \mathbf{t} \\
\text{VarDef} : & \text{variable} ::= \text{var tident;} \\
\text{ConstDef} : & \text{constant} ::= \text{const [unique] } c : \mathbf{t}; \\
\text{FunctDef} : & \text{function} ::= \text{function } f(\overrightarrow{\mathbf{t}}) \text{ returns}(\mathbf{t}); \\
\text{AxiomDef} : & \text{axiom} ::= \text{axiom } B; \\
\text{ProcDef} : & \text{procedure} ::= \text{procedure } \mathbf{p}(\overrightarrow{\text{tident}}) \text{ returns}(\overrightarrow{\text{tident}}) \{ \overrightarrow{\text{variable}} S \} \\
\text{ProgDef} : & \text{prog} ::= \overrightarrow{\text{type variable constant function axiom procedure}}
\end{array}$$

Figure 12.2: Top-level elements of a program

12.2.1 Types

Type_P contains the types in the program P . All variables and expressions in ACTIONPL and QEDPL are typed. Let Val be the set of all values. Each type $\mathbf{t}$ specifies to a distinct subset of Val ¹, denoted $\text{Val}_{\mathbf{t}}$, and an expression of type $\mathbf{t}$ can only evaluate to a value from $\text{Val}_{\mathbf{t}}$. For a variable v , we will use the notation $\mathbf{t}(v)$ to refer to its type, and $\text{Val}_{\mathbf{t}(v)}$ to refer to the set of values that v can assume from.

In addition to standard types **int** for signed integers and **bool** for booleans, an ACTIONPL program may define a custom type $\mathbf{t}$ using the syntax:

type $\mathbf{t}$;

Once it is defined, the new type $\mathbf{t}$ can be used in the definition of constants, functions, and variables. An identifier of type $\mathbf{t}$ is denoted by $id : \mathbf{t}$ where id is a constant or a variable.

We give semantics to a custom type $\mathbf{t}$ by (1) naming some values in $\text{Val}_{\mathbf{t}}$ via constants of type $\mathbf{t}$, and (2) relating values in $\text{Val}_{\mathbf{t}}$ to each other using functions and axioms.

¹Without loss of generality, we omit subtyping relationships between types during the formal treatment of types. Our implementation supports polymorphic types but this does not pose any technical problems to the soundness of our approach.

12.2.2 Functions

$Function_P$ contains the functions in the program P . A function is defined using the syntax:

$$\text{function } f(\mathbf{t}_1, \dots, \mathbf{t}_n) \text{ returns}(\mathbf{t});$$

where f is the (unique) function name, $\mathbf{t}_1, \dots, \mathbf{t}_n$ are the types of its parameters and $\mathbf{t}$ is the return type. We assume that every function is *total*, i.e. for every list of values $a_1, \dots, a_n$ such that $a_1 \in Val_{\mathbf{t}_1}, \dots, a_n \in Val_{\mathbf{t}_n}$, there exists a value $b \in Val_{\mathbf{t}}$ such that $f(a_1, \dots, a_n) = b$. Thus, evaluation of an expression does not cause a program to block or fail.

12.2.3 Constants

$Constant_P$ contains the constants in the program P . A constant is defined using the syntax:

$$\text{const } [\text{unique}] \ c : \mathbf{t};$$

where c is the (unique) name of the constant and $\mathbf{t}$ is the type of the constant. Intuitively, c is used to refer to a value from the set $Val_{\mathbf{t}}$, and is equivalent to a nullary function that returns that value. The keyword `unique` expresses the uniqueness of the constant's value from all other constants defined with `unique`. A constant can appear in expressions, including axioms.

An example use of constants is to define enumerations of a custom type. For example, the following code defines a custom type `DaysOfWeek` and defines seven constants of this type. The constants represent possible values of a variable of type `DaysOfWeek`. The keyword `unique` ensures that each constant represents a distinct day of the week. We also define a function `NextDay` that takes a value of type `DayOfWeek` and returns a value of the same type. In Section 12.2.4, we will use axioms to state that (1) the constants below form an enumeration of all the values in $Val_{\text{DaysOfWeek}}$ and (2) `NextDay` returns the next day of a given day.

```
type DaysOfWeek;
const unique Monday: DaysOfWeek;
const unique Tuesday: DaysOfWeek;
const unique Wednesday: DaysOfWeek;
const unique Thursday: DaysOfWeek;
```

```

const unique Friday: DaysOfWeek;
const unique Saturday: DaysOfWeek;
const unique Sunday: DaysOfWeek;
function NextDay(DayOfWeek) returns(DayOfWeek);

```

12.2.4 Axioms

$Axiom_P$ contains the axioms in the program P . An axiom is defined using the syntax:

```
axiom B;
```

The boolean expression B is used by the theorem prover as a background axiom while proving the truth of any predicate about the program. Thus, $fv(B)$ can contain constants and functions but cannot contain program variables.

Axioms are used to specify the meaning of types, functions, and constants defined as above. As an example, recall the custom type `DayOfWeek` defined above and consider the axioms below. The first axiom states that the constants are used to define an enumeration of the type `DayOfWeek`, i.e., every value in the set $Val_{DayOfWeek}$ must be represent by one of the constants. The second axiom states that the function `NextDay` returns the next day of a given day, in other words, defines a total order among the constants.

```

axiom (forall day:DayOfWeek :: day==Monday || day==Tuesday || day==Wednesday
      || day==Thursday || day==Friday || day==Saturday || day==Sunday);

axiom NextDay(Monday)==Tuesday && NextDay(Tuesday)==Wednesday
    && NextDay(Wednesday)==Thursday && NextDay(Thursday)==Friday
    && NextDay(Friday)==Saturday && NextDay(Saturday)==Sunday
    && NextDay(Sunday)=Monday;

```

12.2.5 Global Variables

$Global_P$ contains the global variables of the program P . Let x range over global variables. Global variables are defined outside of procedures. A global variable is defined using the following syntax:

```
var x : t;
```

where $gvar$ is the (unique) variable name and t is the type of the variable.

12.2.6 Procedures

$Proc_P$ contains the procedures of the program. A procedure $\mathbf{p}$ is defined using the following syntax:

$$\text{procedure } \mathbf{p}(\ell_1^{in} : \mathbf{t}_1, \dots, \ell_n^{in} : \mathbf{t}_n) \text{ returns}(\ell_1^{out} : \mathbf{t}_1, \dots, \ell_m^{out} : \mathbf{t}_m) \{ \ell_1, \dots, \ell_k, S \}$$

where $\mathbf{p}$ is the (unique) name, $\{\ell_1^{in}, \dots, \ell_n^{in}\}$ are the *input variables*, $\{\ell_1^{out} : \mathbf{t}_1, \dots, \ell_m^{out}\}$ are the *output variables*, and $\{\ell_1, \dots, \ell_k\}$ are the *private variables*, and S is the *body* of the procedure. Let $\vec{In}_{\mathbf{p}}$ refer to the input variables, $\vec{Out}_{\mathbf{p}}$ to the output variable, $Private_{\mathbf{p}}$ to the private variables, and $Body_{\mathbf{p}}$ the body (the statement S) of the procedure. We call the set $\vec{In}_{\mathbf{p}} \cup \vec{Out}_{\mathbf{p}} \cup Private_{\mathbf{p}}$ the *local variables* of the procedure and denote it with $Local_{\mathbf{p}}$. Here the over-arrow notation in $\vec{In}_{\mathbf{p}}$ and $\vec{Out}_{\mathbf{p}}$ denotes that there is an implicit (total) ordering among the variables in these sets, defined while defining the procedure $\mathbf{p}$ and required for the procedure calls. There is no ordering among the variables in $Private_{\mathbf{p}}$. We omit the subscripts when the program or procedure is clear from the context. We enforce the convention that (1) every variable in $\vec{In}_{\mathbf{p}}$ is read-only throughout the procedure, and (2) every variable in $\vec{Out}_{\mathbf{p}}$ is write-only until its first write, after which it can be both read and written. The variables in $Private_{\mathbf{p}}$ can be both read and written without any restriction. Let ℓ range over local variables.

For every ACTIONPL program P , we assume the existence of a special procedure $Main \in Proc_P$. $Main$ has neither input nor output variables, i.e., $\vec{In}_{Main} = \vec{Out}_{Main} = \emptyset$. The program starts executing by calling the procedure $Main$ and ends when the evaluation of $Main$ ends.

The tuple $\langle \mathbf{p}, \vec{In}_{\mathbf{p}}, \vec{Out}_{\mathbf{p}} \rangle$ is called the *signature* of the procedure $\mathbf{p}$, and denoted $Sig_{\mathbf{p}}$. The signatures of the procedures in $Proc_P$ form the signature of the program, denoted Sig_P . The minimal signature for a program is $Sig_P = \{ \langle Main, \emptyset, \emptyset \rangle \}$.

12.2.7 Program Variables and Scoping

We call the global variables and local variables of procedures *program variables*. In addition, we assume two special program variables per thread exc and tid , which is defined in Section 12.4.4. For a program P , let Var_P be the set of all program variables in P . For

simplicity of presentation, we assume that every program variable has a unique name; otherwise, the local variables can be renamed using the name of the procedure as a prefix to obtain the unique names.

Each ACTIONPL program defines three kinds of scope for program variables: program and procedure, and thread scope:

- Every global variable has the program scope; it is visible throughout the entire program.
- Every local variable of a procedure has procedure scope; it is visible throughout the entire body of the procedure it belongs to.
- The special variables *exc* and *tid* have the thread scope; they are visible throughout the entire thread it belongs to.

We use the notations E_ℓ and B_ℓ to refer to an expression and a predicate, respectively, whose free variables are all local.

12.2.8 State and Transition Predicates

For each program variable v , we also define a primed version of the variable denoted v' . For a set V of variables, let the set V' consist of the primed version of each variable in V , i.e., $V' = \{v' \mid v \in V\}$. For example, $Global'_P$ contains the primed global variables of the program.

We distinguish two kinds of predicates: *state predicates*, which are over unprimed program variables, and *transition predicates*, which are over both unprimed and primed program variables. Let ϕ range over state predicates and τ range over transition predicates. A state predicate ϕ is used to represent sets of states that satisfy ϕ , while a transition predicate τ is used to represent pairs of states, where an unprimed variable v represents the value of v at the first state, and the primed variable v' represents the value of v at the second state. State and transition predicates are used to formally describe an atomic computation on program variables as elaborated below.

$$\phi, \tau \in BExpr$$

$$Lbl : l ::= \mathbf{skip} \mid \mathbf{return} \mid \mathbf{break} \mid \mathbf{exit} \mid \dots$$

$$Atomic : \alpha ::= \phi \triangleright \tau$$

$$Stmt : S ::= \alpha \mid \vec{\ell} := \mathbf{p}(\vec{E}_{\ell}) \mid \mathbf{done} \mid \mathbf{error}$$

$$\mid S^{\circ} \mid S ; S \mid S \square S \mid S \parallel S \mid l:S \mid *:S$$

Syntactic Sugars:

$\mathbf{skip}$	$\mathbf{true} \triangleright \langle \mathbf{true} \rangle_{\emptyset}$
$\mathbf{return}$	$\mathbf{true} \triangleright \langle exc' = \mathbf{return} \rangle_{exc}$
$\mathbf{break}$	$\mathbf{true} \triangleright \langle exc' = \mathbf{break} \rangle_{exc}$
$\mathbf{exit}$	$\mathbf{true} \triangleright \langle exc' = \mathbf{exit} \rangle_{exc}$
S^*	$(S \square \mathbf{break})^{\circ} ; \mathbf{break} : \mathbf{skip}$

Figure 12.3: The syntax of ACTIONPL statements

12.3 Statements: Syntax

Figure 12.3 shows the syntax of statements in ACTIONPL. Primitive building blocks of statements (*Stmt*) are gated actions, procedure calls, **done**, and **error**. The statements **done** and **error** are not allowed in the static code, but are assigned to threads at runtime; we will use these statement while describing operational semantics of ACTIONPL. Figure 12.3 also shows the syntactic sugars for special statements. A *syntactic sugar* is a program construct (usually a statement) that is not part of the core syntax and is defined over the core syntax of the program. We use syntactic sugars to express some special constructs more clearly and concisely.

Other statements are formed by composing of gated actions using sequential, choice, parallel, looping connectors.

12.3.1 Gated Actions

In this language, we express each atomic statement (*Atomic*) with a *gated action* (shortly *action*) α , which is in the form: $\phi \triangleright \tau$. The state predicate ϕ is the *gate* of the action and specifies *assertion* that must be satisfied right before the action is executed. The transition

predicate τ describes the set of atomic computations over program variables allowed by this gated action. For any action α , let ϕ_α and τ_α denote its gate and transition predicate.

Actions and Scoping

Every action appears within the body of some procedure in the program. Let $\mathbf{p}$ be the procedure whose body contains α . We define Var_α as the set of variables visible to α . Var_α contains the global variables of the program, the local variables of the containing procedure $\mathbf{p}$, and the special variables *exc* and *tid* (defined below), i.e., $Var_\alpha = Global \cup Local_{\mathbf{p}} \cup \{exc, tid\}$. We syntactically enforce that $fv(\phi_\alpha) \subseteq Var_\alpha$, and $fv(\tau_\alpha) \subseteq (Var_\alpha \cup Var'_\alpha)$. We also define $Local_\alpha$, the local variables visible to α , as the set $Var_\alpha \setminus Global_P$.

We extend this definition of visible variables to state and transition predicates as follows: For a state predicate ϕ that is the gate of an action α , Var_ϕ refers to Var_α . Similarly, for a transition predicate τ that belongs to an action α , Var_τ refers to Var_α .

The transition predicate of a gated action α does not refer to variables that are not in Var_α , particularly, local variables of other threads. These variables are implicitly assumed to remain unchanged by the action.

It is often the case that an action modifies a small set of variables and leaves all other variables in Var_α unchanged. For every unchanged variable v , the transition predicate must explicitly contain the conjunct $v' = v$; this causes large transition predicates. In the following, we introduce a notation to express transition predicates in a compact way.

Compact Notation for Transition Predicates

The transition predicate τ_α expresses how the action α modifies the program variables in Var_α . We sometimes express the transition predicate of an action α in the compact notation:

$$\langle \tau \rangle_M \triangleq \tau \wedge \bigwedge_{v \in Var_\alpha \setminus M} (v' = v) \quad (12.2)$$

where M intuitively represents the list of variables modified by α . We require that, in the compact notation of a gated action $\phi \triangleright \langle \tau \rangle_M$, M contain *all* the variables updated by the action. For example, we use **skip** as a syntactic sugar for the action $\text{true} \triangleright \langle \text{true} \rangle_\emptyset$, which has not assertion and does not modify any program variables. M can never contain *tid*, since

actions are not allowed to modify *tid*; *tid* is updated by the operational semantics rules as described below.

Definition 2 (Independent actions):

Two actions α and β are said to be independent from each other, if either $\text{Var}_\alpha \cap \text{Var}_\beta = \emptyset$, or the actions can be written as $\alpha = \phi_1 \triangleright \langle \tau_1 \rangle_{M_1}$ and $\beta = \phi_2 \triangleright \langle \tau_2 \rangle_{M_2}$ such that the following holds:

$$((\text{fv}(\phi_1) \cup \text{fv}(\tau_1)) \cap M_2) = ((\text{fv}(\phi_2) \cup \text{fv}(\tau_2)) \cap M_1) = (M_1 \cap M_2) = \emptyset \quad (12.3)$$

In our formalization of ACTIONPL, Var_α for every action α in the program contains global variables *Global* of the program. Thus, the condition $\text{Var}_\alpha \cap \text{Var}_\beta = \emptyset$ is not met as long as *Global* is nonempty. Therefore, we usually determine whether two actions conflict with each other or not by examining their gates and transition predicates and checking the condition (12.3). Two actions α and β are said to *conflict with each other* if they are not independent.

12.3.2 Procedure Calls

A procedure call is denoted $\vec{\ell}_2 := \mathbf{p}(\vec{\ell}_1)$, where $\vec{\ell}_1$ and $\vec{\ell}_2$ are the input and output argument lists of the call. We require that $\vec{\ell}_1$ and $\vec{\ell}_2$ be sets of local variables (of the caller procedure) so that starting a procedure call has no side effects on the global variables.

Definition 3 (Set of procedures closed under call):

A set $\mathcal{M}$ of procedures is closed under call if for every procedure $\mathbf{p}_i \in \mathcal{M}$, if $\text{Body}_{\mathbf{p}_i}$ contains a call to a procedure $\mathbf{p}_j$, then $\mathbf{p}_j \in \mathcal{M}$ holds. ■

12.3.3 Compound Statements

We use the sequential composition ($;$), non-deterministic choice ($\square$), looping ($^\circ$) and parallel composition ($\parallel$) operators to form *compound statements* (*Stmt*). Formally, each atomic action and a procedure call is a compound statement, and for compound statements S_1 and S_2 , $S_1; S_2$, $S_1 \square S_2$, $S_1 \parallel S_2$ and S_1° are also compound statements.

In ACTIONPL, we can label statements and direct the control flow to a labeled statements. This use of labeled statements corresponds to structured exception handling in higher-level languages using the *try-catch* statement. In fact, having this feature enables us

to implement exception handling in the QEDPL language by translating a *try-catch* statement to labeled statements (see Section 14.1.1). We refer to exceptions via labels. Let Lbl be the set of all exception labels. We distinguish the special labels **skip**, **break**, **return**, and **exit**. We use **skip** to indicate “no exception” and explain the usage of other special labels while describing the operational semantics. A labeled statement is denoted either by $l:S$ or $*:S$, where $*$ expresses “any label except **skip**”. We assume that every statement can get at most one label, so situations like $l_1:l_2:S$ are ruled out. Regular statements are not labeled explicitly and are assumed to be labeled with the special label **skip**.

While proving programs, we often need to refer to the set of all gated actions contained in a statement S . This set is denoted by $Atoms(S)$ and is computed recursively as follows:

$$\begin{aligned}
Atoms(\phi \triangleright \tau) &= \{\phi \triangleright \tau\} \\
Atoms(l:S) &= Atoms(S) \\
Atoms(S^\complement) &= Atoms(S) \\
Atoms(S_1; S_2) &= Atoms(S_1) \cup Atoms(S_2) \\
Atoms(S_1 \square S_2) &= Atoms(S_1) \cup Atoms(S_2) \\
Atoms(S_1 \parallel S_2) &= Atoms(S_1) \cup Atoms(S_2) \\
Atoms(\vec{\ell}_2 := \mathbf{p}(\vec{\ell}_1)) &= \emptyset
\end{aligned}$$

We extend the definition of $Atoms$ to programs as follows:

$$Atoms(P) = \bigcup_{\mathbf{p} \in Proc_P} Atoms(Body_{\mathbf{p}}) \quad (12.4)$$

12.4 Operational Semantics

The operational semantics of ACTIONPL specifies the set of executions that a given ACTIONPL program produces. In this section we give the execution model of a program, formalize the program state and present small-step operational semantics of statements.

12.4.1 Execution Model

A program P starts executing by calling procedure *Main* and ends when the evaluation of *Main* ends. If *Main* is not given explicitly by the programmer, we assume a general form of *Main* that creates an arbitrary number of threads and each thread calls at least one

procedure from $Proc_P$ with arbitrary arguments. This allows us to prove an “open” program, e.g., a library or a data structure implementation, for all possible clients.

The program starts executing *Main* on a single thread. Additional threads are created via the parallel construct $\parallel$ while executing *Main*. Let Tid be the set of all thread identifiers. Each thread is assigned a unique identifier from a set Tid . Let t, u range over Tid . The gate and the transition predicate of a gated action may refer to the current thread id through the special (thread-local) variable tid , whose domain is Tid . The variable tid is written once the thread is created by the operational semantics rule *fork*, and actions cannot overwrite tid . When executing the gated action $\phi \triangleright \tau$ by a thread t , we substitute t for tid in both ϕ and τ .

We refer by the distinguished thread id t_{main} to the thread that runs the procedure *Main*. To represent the creation of new threads (through parallel composition $(\parallel)$), we make use of two functions $left, right : Tid \rightarrow Tid$. We assume that $left$ and $right$ together define a binary tree over Tid with t_{main} being the root of the tree. When a thread t executes the statement $S_1 \parallel S_2$, the ids of the new threads executing S_1 and S_2 are selected as $left(t)$ and $right(t)$, respectively. Therefore, for example, the statements S_1 and S_3 (resp. S_2 and S_4) in a statement $(S_1 \parallel S_2); (S_3 \parallel S_4)$ are executed by the same thread.

The operational semantics of our programming language is given in Figure 12.4. We define an interleaving semantics for ACTIONPL programs over the sequentially-consistent memory model [13], where context switches are allowed at only atomic action boundaries. We assume that data races between threads do not affect the semantics of the program, i.e., the outcome of the concurrent data accesses. The program state, defined below, evolves over time as threads execute gated actions. When a context switch happens, a nondeterministic scheduler selects one of the threads in the program and executes the next atomic action of that thread, where the next action is specified by the operational semantics given in Figure 12.4 and explained below.

12.4.2 Program State

A program state s is a pair (σ_s, θ_s) :

- $\sigma_s : Tid \times Var \rightarrow Val$ is a variable valuation, called *store*, that maps a thread id and a variable to a value. For a global variable x , $\sigma_s(t, x) = \sigma_s(u, x)$ for all states s . Thus,

we often refer to global variables by omitting the thread id argument, e.g., $\sigma_s(x)$.

- $\theta_s : Tid \rightarrow Stmt$ is a *code map* that keeps track of a dynamic statement for each thread, such that $\theta_s(t) = S$ means that at program state s , the remaining part of the program to be executed by thread t is given by S . Differently from static code, a statement during the execution can contain the special statements **done** and **error**; these statements indicate the end of a computation by the thread.

Notations for Stores

Given a store σ , we refer to the valuation of variables visible to a thread t using unary stores. A *unary store* $\varsigma : Var \rightarrow Val$ is a map from variables to values. We use the following notation for unary stores. While being used mostly for unary stores, this notation applies to other mappings in this thesis, such as the code map θ . A unary store from variables $x_1, \dots, x_n$ to values $y_1, \dots, y_n$, respectively, is denoted $[x_1 \mapsto y_1, \dots, x_n \mapsto y_n]$. Let us fix the store $\varsigma = [x_1 \mapsto y_1, \dots, x_n \mapsto y_n]$. $\varsigma(x_i)$ denotes the value y_i of variable x_i . $\varsigma[x_i \mapsto z_i]$ agrees with ς on all variables except $\varsigma[x_i \mapsto z_i](x_i) = z_i$. $\varsigma[x_i \mapsto \perp]$ agrees with ς on all variables except that x_i is now undefined. The domain of ς , $\{x_1, \dots, x_n\}$, is denoted $\text{dom}(\varsigma)$. The union of two unary stores ς_1 and ς_2 such that $\text{dom}(\varsigma_1) \cap \text{dom}(\varsigma_2) = \emptyset$ is denoted $\varsigma_1 \cup \varsigma_2$. Two unary stores are equivalent, denoted $\varsigma_1 \equiv \varsigma_2$ if $\text{dom}(\varsigma_1) = \text{dom}(\varsigma_2)$ and for each $x \in \text{dom}(\varsigma_1)$, $\varsigma_1(x) = \varsigma_2(x)$ holds. $\varsigma|_V$ for $V \subseteq \text{dom}(\varsigma)$ represents the projection of ς to a set of variables V , i.e., the store $[v \mapsto \varsigma(v) \mid v \in V]$.

Given a store σ and a thread t , the *t -visible store at σ* , denoted $\sigma|_t$, is the projection of σ to t , i.e., the unary store $[v \mapsto a \mid \sigma(t, v) = a]$. The *t -local store at σ* , denoted $\sigma|_t^\ell$, is the projection of the t -visible store to local variables (including *tid* and *exc*), i.e., the unary store $[v \mapsto a \mid \sigma(t, v) = a, v \notin \text{Global}]$. Given an unary store ς such that $\text{dom}(\varsigma) \cap \text{Global} = \emptyset$ (ς does not refer to any global variables), we use the notation $\sigma[t \mapsto \varsigma]$ to denote the initialize or update of the local store of t ; this does not affect the global variables (for all global variables x , $\sigma(t, x)$ remains as is).

In addition, we define $\sigma|_{\text{Global}}$ as the projection of the store to the global variables. i.e., the unary store $[x \mapsto a \mid \sigma(t, x) = a, t \in \text{Tid}]$.

Initial, Final, and Error States

The program starts executing from an initial state and ends at either a final state or an error state. The threads that exists at a state s is determined by querying $\text{dom}(\theta_s)$, the domain of the code map at s . A program state s is called *initial* if s contains only the main thread, i.e., $\text{dom}(\theta_s) = \{t_{\text{main}}\}$ and $\theta_s(t_{\text{main}}) = \text{Main}()$. A state s is called *final* if $\text{dom}(\theta_s) = \{t_{\text{main}}\}$ and $\theta_s(t_{\text{main}}) = \text{done}$. The statement **done** indicates that $\text{Body}_{\text{Main}}$ has been fully evaluated. State s is called an *error* state if $\theta_s(t) = \text{error}$ for some thread $t \in \text{dom}(\theta_s)$. The statement **error** indicates that the evaluation of a statement has ended by violating an assertion (gate of an action). We use the predicates $\text{Initial}(s)$, $\text{Final}(s)$, and $\text{Error}(s)$ to denote that s is an initial, final, and error state, respectively.

12.4.3 Program States and State/Transition Predicates

Let us fix a state predicate ϕ and a transition predicate τ . We say that a thread t *satisfies* ϕ at store σ , denoted as $\sigma \models_t \phi$, if ϕ evaluates to **true** given the valuations of unprimed variables in the t -visible store $\sigma|_t$. Similarly, the thread t *satisfies* τ for a pair of stores (σ_1, σ_2) , denoted as $(\sigma_1, \sigma_2) \models_t \tau$, if τ evaluates to **true** given the valuations of unprimed variables from the t -visible store $\sigma_1|_t$ and primed variables from the t -visible store $\sigma_2|_t$.

We lift the above definitions to states and pairs of states as follows:

$$\begin{aligned} s \models_t \phi &\quad \Leftrightarrow \quad \sigma_s \models_t \phi \\ (s_1, s_2) \models_t \tau &\quad \Leftrightarrow \quad (\sigma_{s_1}, \sigma_{s_2}) \models_t \tau \end{aligned}$$

The thread is said to *violate* a state predicate (resp. transition predicate) at a state (resp. for a pair of states) if it does not satisfy the predicate at that state (resp. for the state pair).

For a predicate B , the notation $\models B$ expresses that B is *valid*, i.e., B evaluates to **true** for all valuations of $\text{fv}(B)$, the free variables of B . When talking about the validity of a predicate B we sometimes omit the quantification of free variables in B ; the reader should assume that in $\models B$ all the free variables of B are universally quantified implicitly. In this thesis, the notation $\models B$ often indicates a proof obligation that is checked using a theorem prover.

12.4.4 Labeled Statements and Exception Handling

We use exception labels and labeled statements to redirect the control flow to a particular statement. For this, we assign every thread a private (thread-local) variable exc that ranges over labels in Lbl . Every thread starts with $exc = \mathbf{skip}$. A gated action can overwrite this by assigning another label l to exc , in which case the action is said to *throw* l . For example, the action $\mathbf{true} \triangleright \langle exc' = \mathbf{return} \rangle_{exc}$ throws the exception **return** (and leaves variables other than exc as is). The exception thrown by a thread only affects the control flow at that thread; it has no effect on the other threads.

While describing the operational semantics in Section 12.4.5, we assume that every statement in the program is labeled. Regular statements, ones that are not labeled explicitly, are assumed to be labeled with the special label **skip**. A labeled statement, $l:S$, is evaluated only when exc stores the value l ; otherwise, the execution skips over $l:S$. Thus, $l:S$ behaves as an exception handler for l , and when evaluated, it is said to *catch* l . The statement $*:S$ behaves as an exception handler for any label except for **skip**. We say that a statement $l:S$ *suppresses* an exception l when the statement catches l and terminates with $exc = \mathbf{skip}$.

The exception handling mechanism via labeled statements allows us to implement useful features to the language. The following is a list of features that we assume to be used as stated. QEDPL uses the exception-handling mechanism to realize similar features.

- We use the label **break** to implement the non-deterministic looping denoted S^* , which executes the body S zero or more times non-deterministically. S^* is a syntactic sugar for the following statement:

$$(S \sqcap \mathbf{break})^\circ ; (\mathbf{break}:\mathbf{skip}) \quad (12.5)$$

where **break** is also a syntactic sugar for the action $\mathbf{true} \triangleright \langle exc' = \mathbf{break} \rangle_{exc}$. At any iteration, S^* chooses non-deterministically to execute the body S or sets exc to **break**. In the latter case the execution of the loop terminates. The exception **break** is caught by the statement **break:skip**.

- The exception label **return** is used to terminate a procedure call. The action **return** is a syntactic sugar for the action $\mathbf{true} \triangleright \langle exc' = \mathbf{return} \rangle_{exc}$, which throws **return**.

We assume that **return** is not caught by any action. The semantics rule PROC-CALL described below ensures that **return** is caught and suppressed when returning from the procedure call.

- The exception label **exit** is used to terminate a thread. The action **exit** is a syntactic sugar for the action $true \triangleright \langle exc' = \mathbf{exit} \rangle_{exc}$, which throws **exit**. We assume that **exit** is never caught by any action, thus the action **exit** causes all the statement of the current thread to be skipped until the thread's statements become **done**.

12.4.5 Semantics of Statements and State Transitions

In Figure 12.4, we give the small step operational semantics of ACTIONPL programs. In this semantics, a run of the program (defined in the next section) consists of a sequence of state transitions. A state transition is expressed as $(\sigma_1, \theta_1) \xrightarrow{(t, \cdot)} (\sigma_2, \theta_2)$ where $(t, \cdot)$ is a transition label. A *transition label* is either (t, α) for an action $\alpha \in Atomic$ executed by thread t , or (t, λ) where λ represents an *invisible transition*. In an invisible transition, the statement executed by a t is reorganized; no variables except exc are modified. We denote a sequence of labels by $\vec{\mathbf{I}}$ and the i^{th} label in that sequence by $\vec{\mathbf{I}}_i$. We say that a transition $s_1 \xrightarrow{(t, \cdot)} s_2$ is *legal* if the transition is derived using the rules in Figure 12.4. A thread is said to be *blocked* at a state s if no legal transition $s \xrightarrow{(t, \cdot)} s'$ can be derived from the semantics rules. Our formulation of the operational semantics rules out stuttering transition; every transition must evaluate an action or a compound statement.

While specifying the semantics, we assume that when the program starts, every statement, except **done** and **error**, is labeled; if not explicitly labeled, a statement is assumed to be labeled with **skip**. Then every (labeled) statement is evaluated first by the rules CATCH, CATCH-ANY, and CATCH-FAIL. The other semantics rules operate on unlabeled statements. Given a statement $l:S$, these rules check if the exc of the current thread is equal to the label l of the statement. If exc and the label match, then the rule CATCH (1) removes the label of the statement, (2) assigns **skip** to exc and (3) starts evaluating S . If exc and the label do not match, then the rule CATCH-FAIL skips evaluating the labeled statement, by replacing it with **done**. The rule CATCH-ANY evaluates the body of a statement labeled with $*$ if exc is not **skip**.

$(\sigma_1, \theta_1) \xrightarrow{(t, \cdot)} (\sigma_2, \theta_2)$		
CATCH $\frac{\sigma(t, exc) = l \quad \theta(t) = l : S}{(\sigma, \theta) \xrightarrow{(t, \lambda)} (\sigma[exc \mapsto \mathbf{skip}], \theta[t \mapsto S])}$	CATCH-ANY $\frac{\sigma(t, exc) \neq \mathbf{skip} \quad \theta(t) = * : S}{(\sigma, \theta) \xrightarrow{(t, \lambda)} (\sigma[exc \mapsto \mathbf{skip}], \theta[t \mapsto S])}$	
CATCH-FAIL $\frac{\sigma(t, exc) = l_1 \quad \theta(t) = l_2 : S \quad l_1 \neq l_2}{(\sigma, \theta) \xrightarrow{(t, \lambda)} (\sigma, \theta[t \mapsto \mathbf{done}])}$	ATOMIC $\frac{\theta(t) = \phi \triangleright \tau \quad \sigma_1 \models_t \phi \quad (\sigma_1, \sigma_2) \models_t \tau}{(\sigma_1, \theta) \xrightarrow{(t, \phi \triangleright \tau)} (\sigma_2, \theta[t \mapsto \mathbf{done}])}$	
ERROR $\frac{\theta(t) = \phi \triangleright \tau \quad \sigma \models_t \neg \phi}{(\sigma, \theta) \xrightarrow{(t, \phi \triangleright \tau)} (\sigma, \theta[t \mapsto \mathbf{error}])}$	EVALUATE $\frac{\theta_1(t) = S_1; S_2 \quad (\sigma_1, \theta_1[t \mapsto S_1]) \xrightarrow{1} (\sigma_2, \theta_2)}{(\sigma_1, \theta_1) \xrightarrow{1} (\sigma_2, \theta_2[t \mapsto \theta_2(t); S_2])}$	SEQUENTIAL-DONE $\frac{\theta(t) = \mathbf{done}; S}{(\sigma, \theta) \xrightarrow{(t, \lambda)} (\sigma, \theta[t \mapsto S])}$
SEQUENTIAL-ERROR $\frac{\theta(t) = \mathbf{error}; S}{(\sigma, \theta) \xrightarrow{(t, \lambda)} (\sigma, \theta[t \mapsto \mathbf{error}])}$	CHOOSE $\frac{\theta(t) = S_1 \sqcap S_2 \quad i \in \{1, 2\}}{(\sigma, \theta) \xrightarrow{(t, \lambda)} (\sigma, \theta[t \mapsto S_i])}$	LOOP $\frac{\theta(t) = S^\odot}{(\sigma, \theta) \xrightarrow{(t, \lambda)} (\sigma, \theta[t \mapsto S; S^\odot])}$
FORK $\frac{\theta(t) = S_1 \parallel S_2 \quad u = \mathit{left}(t) \quad w = \mathit{right}(t) \quad \varsigma_1 = \mathit{Fresh}(\sigma _t^\ell)[tid \mapsto u, exc \mapsto \mathbf{skip}] \quad \varsigma_2 = \mathit{Fresh}(\sigma _t^\ell)[tid \mapsto w, exc \mapsto \mathbf{skip}]}{(\sigma, \theta) \xrightarrow{(t, \lambda)} (\varsigma[u \mapsto \varsigma_1, w \mapsto \varsigma_2], \theta[t \mapsto \perp; u \mapsto S_1; w \mapsto S_2])}$		
JOIN-DONE $\frac{u = \mathit{left}(t) \quad w = \mathit{right}(t) \quad \theta(u) = \mathbf{done} \quad \theta(w) = \mathbf{done}}{(\sigma, \theta) \xrightarrow{(t, \lambda)} (\sigma[u \mapsto \perp, w \mapsto \perp], \theta[t \mapsto \mathbf{done}; u \mapsto \perp, w \mapsto \perp])}$		
JOIN-ERROR $\frac{u = \mathit{left}(t) \quad w = \mathit{right}(t) \quad \theta(u) = \mathbf{error} \quad \text{or} \quad \theta(w) = \mathbf{error}}{(\sigma, \theta_1) \xrightarrow{(t, \lambda)} (\sigma[u \mapsto \perp, w \mapsto \perp], \theta[t \mapsto \mathbf{error}; u \mapsto \perp, w \mapsto \perp])}$	PROC-CALL $\frac{\theta(t) = \overrightarrow{out} := \mathbf{p}(\overrightarrow{in}) \quad \langle S, V \rangle = \mathit{Inline}(\overrightarrow{out} := \mathbf{p}(\overrightarrow{in})) \quad \varsigma = \sigma _t^\ell \cup [\ell \mapsto a \mid \ell \in V, a \in \mathit{Val}_{\mathbf{t}(\ell)}]}{(\sigma, \theta) \xrightarrow{(t, \lambda)} (\sigma[t \mapsto \varsigma], \theta[t \mapsto S])}$	

Figure 12.4: The rules of the small-step operational semantics of ACTIONPL

Atomic Transitions of Gated Actions

The semantics of a gated action $t : \phi \triangleright \tau$ is given in the rules `ATOMIC` and `ERROR`. We call transitions obtained from `ATOMIC` and `ERROR` *atomic transitions* of the program. Atomic transitions of an action α are labeled with (t, α) , while all other transitions are labeled (t, λ) . These transitions are triggered only when *exc* of thread t stores the label **skip** at the current state. Rules `ATOMIC` and `ERROR` first check if the current state satisfies the gate of the action, $\phi[t]$. `ATOMIC` specifies that if the current state satisfies ϕ , the store is modified atomically consistent with τ , and the thread is assigned the special statement **done**. The statement **done** indicates that the evaluation of a particular statement has ended successfully. Thus, atomic transitions of an action governed by the rule `ATOMIC` are called *successful*, or *normal*, transitions of the action. If the action violates ϕ , `ERROR` assigns the thread the special statement **error**. In this case the action is said to *go wrong*. Such transitions of an action is called *failing* transitions of the action. A thread goes wrong when an action it executes goes wrong, and a run of the program goes wrong when a thread in it goes wrong.

Sequential Composition

Rule `EVALUATE` evaluates the first element of a sequential composition and leaves the second element as is. When the first element of a sequential composition ends with **done**, rule `SEQUENTIAL-DONE` replaces the entire statement with the second element to start evaluating it. When the first element of a sequential composition ends with **error**, the entire compound statement goes wrong and replaced with **error** by rule `SEQUENTIAL-ERROR`.

Nondeterministic Choice and Loops

The rule `CHOOSE` selects one of the elements in a non-deterministic choice composition for evaluation. The rule `LOOP` starts evaluating the next iteration S of an infinite loop S° . It also puts the entire loop S° to the front of the iteration S , so that the evaluation of the loop completes only when its body S throws an exception other than **skip** (e.g., **break** or **return**), in which case α° in the front is skipped.

Forking and Joining Threads

The rule FORK represents the creation of two new threads by a parent thread t . The new threads are assigned the thread ids $left(t)$ and $right(t)$, and separate copies of the local stores of the parent thread. FORK uses the operator *Fresh* to create fresh local stores for the new threads. *Fresh* is defined as follows:

$$Fresh(\varsigma) \triangleq [\ell \mapsto a \mid \ell \in \text{dom}(\varsigma), a \in Val_{t(\ell)}] \quad (12.6)$$

Thus, each new thread starts operating on separate and arbitrary local store. In addition, the parent thread t is removed from the code map θ so that it is not allowed to execute while its children have not terminated. Therefore, FORK specifies a structured form of parallelism.

When both threads in a parallel composition terminate, the rules JOIN-DONE and JOIN-ERROR removes the elements from the state and re-enables the parent thread. If one of the threads goes wrong, JOIN-ERROR specifies that the parent thread also goes wrong at the join.

12.4.6 Inlining Procedures

The rule PROC-CALL inlines a procedure at a call site as explained below. Let $\overrightarrow{out} := \mathbf{p}(\overrightarrow{in})$ be a call to the procedure $\mathbf{p}$ at the body of procedure $\mathbf{p}'$, where $\overrightarrow{In}_{\mathbf{p}} = \{\ell_1^{in}, \dots, \ell_n^{in}\}$ and $\overrightarrow{Out} = \{\ell_1^{out}, \dots, \ell_m^{out}\}$ are the input and output arguments of the call. $Inline(\overrightarrow{out} := \mathbf{p}(\overrightarrow{in}))$ returns a tuple $\langle S, V \rangle$ where S is the inlined body $Body_{\mathbf{p}}$ of $\mathbf{p}$ at this call site and V is a set of fresh local variables that are created on behalf of the inlining. *Inline* is defined as follows:

$$Inline(\overrightarrow{out} := \mathbf{p}(\overrightarrow{in})) \triangleq \langle (Body_{\mathbf{p}}[\overrightarrow{in}/\overrightarrow{In}_{\mathbf{p}}, \overrightarrow{out}/\overrightarrow{Out}_{\mathbf{p}}, V/Private_{\mathbf{p}}]; \mathbf{return} : \text{skip}) , V \rangle \quad (12.7)$$

(where V is fresh for $Private_{\mathbf{p}}$)

Inlining the callee at a call $\overrightarrow{out} := \mathbf{p}(\overrightarrow{in})$ takes place in the following steps.

1. For each local variable in $Private_{\mathbf{p}}$, we create a fresh variable to avoid naming clashes, especially when there are recursive calls to a procedure. We say that V is *fresh for* $Private_{\mathbf{p}}$ to denote that the variables in V are freshly (unique) created to replace the variables $Private_{\mathbf{p}}$, thus there is a bijection F from $Private_{\mathbf{p}}$ to V .

2. Every occurrence of each local variable in $Local_{\mathbf{p}}$ is substituted by another expression. The input and output arguments of the call $(\vec{in}$ and $\vec{out})$ are substituted for the input and output variables of $\mathbf{p}$ ($\vec{In}_{\mathbf{p}}$ and $\vec{Out}_{\mathbf{p}}$), respectively. Every occurrences of each local variable $\ell \in Private_{\mathbf{p}}$ is substituted by a fresh variable $v \in V$ such that $v = F(\ell)$. The result of this is represented by $Body_{\mathbf{p}}[\vec{in}/\vec{In}_{\mathbf{p}}, \vec{out}/\vec{Out}_{\mathbf{p}}, V/Private_{\mathbf{p}}]$ in Equation 12.7.
3. The body of the procedure may execute **return** to terminate the execution of the body. Since **return** sets *exc* to **return**, we compose the body of $\mathbf{p}$ with the statement **return**:**skip** in order to catch and suppress the exception **return** before the control flow passes to the caller.

The rule PROC-CALL replaces the call $\vec{out} := \mathbf{p}(\vec{in})$ with $Inline(\vec{out} := \mathbf{p}(\vec{in}))$ and merges the existing store with the store that maps every fresh variable in the new body to a nondeterministic value of proper type.

12.4.7 Runs of a Program

A run ρ of the program is a sequence of state transitions:

$$\rho = \rho_1 \xrightarrow{\rho(1)} \rho_2 \xrightarrow{\rho(2)} \dots \xrightarrow{\rho(n)} \rho_{n+1}$$

where ρ_i denotes the i^{th} program state, $\rho(i)$ denotes i^{th} transition label (of the form either (t, α) or (t, λ)), and for all $1 \leq i \leq n$ $\rho_i \xrightarrow{\rho(i)} \rho_{i+1}$ is a legal transition according to the operational semantics given in Figure 12.4.

For the definitions that follow, we fix the run ρ above. Let Tid_{ρ} denote the set of threads occurring in ρ . The length of the run ρ , the number of transitions, is denoted $|\rho|$. Let ρ_{last} refer to the last state (ρ_{n+1}) of the run (we still use ρ_1 to refer to the first state).

For a state predicate ϕ , we say that ρ is a run of P from ϕ if $Initial(\rho_1)$ and $\rho_1 \models_{t_{main}} \phi$. The trace of the run is the sequence of transition labels, $\vec{T} = \rho(1) \dots \rho(n)$. We say that the trace $\vec{T}$ moves a state s_1 to s_2 , written $s_1 \xrightarrow{\vec{T}} s_2$ if there is a run r of P over $\vec{T}$ and with $|\rho| = k$, such that $\rho_1 = s_1$, $\rho_{k+1} = s_2$, and for all $1 \leq j \leq k+1$ and $\rho_i \xrightarrow{\vec{T}_i} \rho_{i+1}$. We often use sub-traces of a run to indicate particular states in that run. For example, $s_1 \xrightarrow{tlabels_1} s_m \xrightarrow{(t, \alpha)(u, beta)} s_{m+2} \xrightarrow{\vec{T}_2} s_n$ denotes a run in which $\vec{T}_1$ moves s_1 to s_m , then

action α by thread followed by action β by thread u moves s_m to s_{m+2} , and finally $\vec{T}_2$ moves s_{m+2} to s_n .

We denote by $\vec{T}|_t$ the projection of the trace $\vec{T}$ to the transition labels whose first element is t .

A run ρ is *successful* if it ends at a final state ($Final(\rho_{last})$) and a *violation* if it ends at an error state ($Error(\rho_{last})$). In a violation, we call the first transition $s_i \xrightarrow{1} s_{i+1}$ such that $i \in [1, |\rho|]$, $\neq Error(s_i)$, and $Error(s_{i+1})$ the *first failing transition* of ρ . We often use the notation $s \xrightarrow{\vec{T}} \mathbf{wrong}$ to indicate that there exists a violation from a state s ; here we omit the error states of the violation and replace it with the placeholder **wrong**. A run is *blocking* if ρ_{last} is not a final or an error state and every thread in $t \in \theta_{\rho_{last}}$ is blocked at ρ_{last} . In this thesis, we focus on *finite* and *non-blocking* runs.

12.4.8 Good and Bad Programs

Let ϕ be a state predicate over Var_P . In the following, we define $Good(P, \phi)$ as the set of initial and final global store pairs associated with successful runs of program P from ϕ . Note that, the initial and final states of a successful run contains only the thread t_{main} . $Bad(P, \phi)$ is the set of initial global stores associated with violations of program P from ϕ .

$$\begin{aligned} Good(P, \phi) &= \{ (\sigma_{s_1}|_{Global}, \sigma_{s_2}|_{Global}) \mid Initial(s_1), s_1[t_{main}] \models \phi, \exists \vec{T}. s_1 \xrightarrow{\vec{T}} s_2, Final(s_2) \} \\ Bad(P, \phi) &= \{ \sigma_{s_1}|_{Global} \mid Initial(s_1), s_1[t_{main}] \models \phi, \exists \vec{T}. s_1 \xrightarrow{\vec{T}} \mathbf{wrong} \} \end{aligned}$$

Without loss of generality, we opt for considering only the valuations of the global variables in the definition of *Good* and *Bad*; any local variables can be included in our treatment by using special global variables tracking the local variables.

Definition 4 (Good and bad programs):

P is said to be good from ϕ if *P* cannot go wrong from ϕ , i.e., $Bad(P, \phi) = \emptyset$; it is called bad from ϕ , otherwise. ■

The projections of *Good* and *Bad* to a set of (global) variables V are defined as follows (note that $\sigma|_V$ is the projection of σ to V):

$$Good|_V(P, \phi) = \{ (\sigma_1|_V, \sigma_2|_V) \mid (\sigma_1, \sigma_2) \in Good(P, \phi) \} \quad (12.8)$$

$$Bad|_V(P, \phi) = \{ \sigma_1|_V \mid \sigma_1 \in Bad(P, \phi) \} \quad (12.9)$$

In this chapter, we introduced the programming language, ACTIONPL. The next chapter presents the QED proof system and the verification approach using the formalism of ACTIONPL. In addition, the expressiveness of ACTIONPL allows us to describe the semantics of higher-level languages in terms of ACTIONPL statements with gated actions and reason about the application of the QED proof system on these languages. For example, while we are presenting the high-level input language QEDPL in Chapter 14, we give semantics to QEDPL programs by compiling a QEDPL program to an ACTIONPL program, and in Chapter 15 we reason about correctness of proof transformation on QEDPL using proof rules on ACTIONPL programs.

Chapter 13

THE QED PROOF SYSTEM

This thesis presents a verification approach in which a program is proved correct by transforming it to another program with fewer actions, thus fewer interleaving points. This is performed in a sequence of steps each of which is governed by a proof rule in the QED proof system. In this chapter, we formalize our notion of a proof and introduce the QED proof rules, which specify our notions of abstraction and reduction. In order to simplify the presentation, we concentrate on verifying assertions in this chapter. Chapter 16 explains how we reason about the linearizability of a data structure implementation using QED.

Outline. Section 13.1 introduces our notion of proving a program, by defining the proof state and connecting the rules of the proof system to transitions of proof state. The following sections introduce proof rules for invariants (Section 13.2), abstraction (Section 13.3), variable introduction and hiding (Section 13.4), reduction (Section 13.5), and procedures (Section 13.6). Finally, we state our the soundness theorem in Section 13.7.

13.1 Proving Programs Correct in QED

13.1.1 Proof State

A proof consists of a sequence of QED proof rule applications on proof states. A *proof state*, denoted $(P, \mathcal{I})$, consists of an ACTIONPL program P and a state predicate $\mathcal{I}$ over global variables ($Global_P$) and constants ($Constant_P$). The proof state $(P, \mathcal{I})$ implicitly refers to a set of runs of P that start from states satisfying $\mathcal{I}$.

We sometimes extend the proof state with an annotation function $\mathcal{A}$ for a special case described in Section 13.5. All the inference rules for $(P, \mathcal{I})$ that do not refer to $\mathcal{A}$ hold for the extended proof state $(P, \mathcal{I}, \mathcal{A})$.

The $(P, \mathcal{I})$ notation implicitly states that $\mathcal{I}$ be an *inductive invariant* of program P . Thus, $\mathcal{I}$ not only constrains initial states from which P is run, but also is an invariant

guaranteed to be preserved by every atomic transition of P . A gated action $\phi \triangleright \tau$ *preserves* $\mathcal{I}$, denoted $\phi \triangleright \tau \sqsubseteq \mathcal{I}$, if from program states satisfying $\mathcal{I}$, $\phi \triangleright \tau$ either goes wrong or satisfies $\mathcal{I}$ at the post-store. More formally, $\phi \triangleright \tau \sqsubseteq \mathcal{I}$ iff the following holds:

$$\models (\mathcal{I} \wedge \phi \wedge \tau) \Rightarrow \mathcal{I}' \quad (13.1)$$

Note that, in the condition above states that $(\mathcal{I} \wedge \phi \wedge \tau) \Rightarrow \mathcal{I}'$ evaluates to true for every valuation of $\text{Var}_{\phi \triangleright \tau}$, $\text{Var}'_{\phi \triangleright \tau}$, the union of variables $\mathcal{I}$, ϕ , and τ are defined over.

Definition 5 (Inductive invariant):

$\mathcal{I}$ is an inductive invariant of a program P , denoted $P \sqsubseteq \mathcal{I}$, if $\forall \alpha \in \text{Atoms}(P). \alpha \sqsubseteq \mathcal{I}$. ■

Section 13.2 gives proof rules to update the invariant $\mathcal{I}$ in a proof state $(P, \mathcal{I})$.

Invariants in Program Text

In the notation $(P, \mathcal{I})$, the program P is separated from the invariant $\mathcal{I}$. Adding the invariant in the program text as a top-level element allows us to represent a proof state within the same program text. An invariant of the program is declared using the following syntax:

invariant B ;

where B is a state predicate over global variables and constants. The program text can contain multiple invariants $B_1, \dots, B_n$. The invariant $\mathcal{I}$ in the proof state is equivalent to the conjunction $\mathcal{I} = (B_1 \wedge \dots \wedge B_n)$.

Note that both axioms and invariants are used to state truths about the program. However, there are two distinctions between them: First, axioms are assumed to be true at any time during the execution, while invariants must be true only at atomic action boundaries. Second, axioms cannot refer to global variables, but invariants can.

13.1.2 Proof Steps

Each proof step is expressed as $(P_i, \mathcal{I}_i) \dashrightarrow (P_{i+1}, \mathcal{I}_{i+1})$ and is governed by a proof rule. Each proof rule is written in the following format:

$$\begin{array}{c}
\text{RULE NAME (CORE RULE NAME)} \\
\hline
\frac{\textit{Condition}_1 \quad \textit{Condition}_2 \quad \dots \quad \textit{Condition}_n}{(P_i, \mathcal{I}_i) \dashrightarrow (P_{i+1}, \mathcal{I}_{i+1})}
\end{array}$$

If the rule is derived from a core proof rule, we also give the name of the latter in parenthesis. The consequent of the rule indicates how the current proof state $(P_i, \mathcal{I}_i)$ is updated to obtain the new proof state $(P_{i+1}, \mathcal{I}_{i+1})$ when all the antecedents $\textit{Condition}_1, \dots, \textit{Condition}_n$ of the rule hold at the current proof state $(P_i, \mathcal{I}_i)$. A condition is usually encoded as a state predicate that must hold in the current proof state. We say that a proof step $(P_i, \mathcal{I}_i) \dashrightarrow (P_{i+1}, \mathcal{I}_{i+1})$ is *legal* if the transition of the proof state is derived using one of the rules in our proof system. The rule updates the proof state by changing the program invariant $\mathcal{I}_i$ to $\mathcal{I}_{i+1}$, rewriting the program P_i to P_{i+1} , or updating both the program and invariant. When the invariant is updated, it must hold that $\mathcal{I}_{i+1} \Rightarrow \mathcal{I}_i$ in order to ensure the soundness of the entire transformation sequence; this connection between $\mathcal{I}_i$ and $\mathcal{I}_{i+1}$ becomes significant while proving Theorem 5.

A rewrite of the program may add new global variables or local variables to procedures, and/or may replace some statements by others. We apply two main techniques to rewrite statements: reduction and abstraction. Reduction [47] replaces a compound statement consisting of several atomic actions with a single atomic action if certain non-interference conditions hold. This allows a subsequent abstraction step to summarize the entire calculation in that statement locally. Abstraction replaces an atomic action with a more relaxed atomic action allowing more behaviors. This allows reduction to combine that action with others.

A rewrite of a program P_i is denoted $P_i[x \mapsto y]$ where an element x of the program is replaced with another element y of the same kind. For example, the proof step $(P_i, \mathcal{I}_i) \dashrightarrow (P_i[\textit{Global} \mapsto \textit{Global} \cup \{x\}, \alpha \mapsto \beta], \mathcal{I}_i)$ adds a new global variable x to the program and replaces *every occurrence of* an action α in procedure bodies with the action β .

13.1.3 Proving Assertions

A program P_1 is *proved correct* by a sequence of legal proof steps $(P_1, \mathcal{I}_1) \dashrightarrow (P_2, \mathcal{I}_2) \dots \dashrightarrow (P_n, \mathcal{I}_n)$ if the final program P_n is one in which all the gated actions are validated. A gated

action $\phi \triangleright \tau$ is *validated* by showing that $\mathcal{I}_n \Rightarrow \phi$ is valid, i.e., the action never goes wrong from $\mathcal{I}_n$, and replacing its gate with **true**. Since $\mathcal{I}_n$ is an inductive invariant of P_n , no run of P_n from $\mathcal{I}_n$ causes an assertion violation. In this case, our soundness theorem given below states that the initial program P_1 cannot go wrong starting from a state satisfying $\mathcal{I}_n$. We call the sequence of steps that proves a program correct a *successful proof*.

Having formalized the notion of a proof in QED, in the rest of this chapter, we present the QED proof rules, and finally state our soundness theorem.

13.2 Invariants

Each proof starts from the proof state $(P_1, \mathbf{true})$ where the invariant **true** indicates that no prior fact about the state space of the program is assumed. New program invariants are introduced by strengthening the current invariant by using the following rule:

$$\frac{\text{INVARIANT} \quad \models \mathcal{I}_2 \Rightarrow \mathcal{I}_1 \quad P \Leftarrow \mathcal{I}_2}{(P, \mathcal{I}_1) \dashrightarrow (P, \mathcal{I}_2)}$$

The rule requires that the new invariant $\mathcal{I}_2$ be stronger than $\mathcal{I}_1$ and be preserved by every action of the program. While INVARIANT specifies a general way of updating $\mathcal{I}$, a usual way of modifying the invariant is to conjoin it with a new state predicate ψ about the program. This is done using the following rule derived from INVARIANT:

$$\frac{\text{ADD-INVARIANT (INVARIANT)} \quad P \Leftarrow (\mathcal{I} \wedge \psi)}{(P, \mathcal{I}) \dashrightarrow (P, \mathcal{I} \wedge \psi)}$$

13.3 Abstraction

The basic idea in abstraction is to replace an action with another action that abstracts the former. The abstraction relation defined below enables us to decide when it is legal to replace a gated action with another.

Definition 6 (Abstraction between actions):

Let α, β be actions and $\mathcal{I}$ be a state predicate. β abstracts α from $\mathcal{I}$, written $\mathcal{I} \vdash \alpha \preceq \beta$, if both of the following are valid:

$$1. (\mathcal{I} \wedge \neg\phi_\alpha) \Rightarrow \neg\phi_\beta$$

$$2. (\mathcal{I} \wedge \tau_\alpha) \Rightarrow (\neg\phi_\beta \vee \tau_\beta)$$

■

Intuitively, for every state s satisfying $\mathcal{I}$, condition 1 states that if α goes wrong from s , then β can go wrong from s when substituted in place of α , and condition 2 states that if β does not go wrong from s , it simulates succeeding transitions of α from s . Notice that, there is no special treatment of exceptions in these conditions, since throwing an exception is still considered a successful transition. Therefore, if α throws an exception l when it terminates, β is required to either go wrong or make a successful transition that agrees with α on the pre- and post-states of the transition including that $exc = l$ at the post-state.

The proof rule **ABSTRACT** states that it is sound to replace every occurrence of an action α with another action β if the latter abstracts the former from the current invariant and preserves the invariant.

$$\frac{\text{ABSTRACT} \quad \mathcal{I} \vdash \alpha \preceq \beta \quad \beta \hookrightarrow \mathcal{I} \quad \text{Var}_\alpha = \text{Var}_\beta}{(P, \mathcal{I}) \dashrightarrow (P[\alpha \mapsto \beta], \mathcal{I})}$$

Common ways of rewriting a gated action α to β are the following:

- Strengthening the gate, such that $\models (\phi_\beta \Rightarrow \phi_\alpha)$. This corresponds to adding an assertion to the action, and thus making the action fail at more pre-states. A special case of this is explained in Section 13.3.1.
- Weakening the transition predicate, such that $\models (\tau_\alpha \Rightarrow \tau_\beta)$. This corresponds to adding more successful atomic transitions to the action, and thus making the action terminate at more post-states. A special case of this is explained in Section 13.3.2.
- Validating the gated action, such that $\phi_\beta = \text{true}$. This is explained in Section 13.3.3.

13.3.1 Borrowing Assertions

A special case of the abstraction proof rule is the annotation of gated actions with assertions. When a gated action $\phi_1 \triangleright \tau_1$ is annotated with an assertion ψ , it is transformed to the gated action $(\phi_1 \wedge \psi) \triangleright \tau_1$. This is stated by the following rule derived from **ABSTRACT**:

$$\begin{array}{c}
\text{ADD-ASSERTION (ABSTRACT)} \\
\frac{\text{fv}(\psi) \subseteq \text{Var}_\alpha}{(P, \mathcal{I}) \dashrightarrow (P[\alpha \mapsto (\phi_\alpha \wedge \psi) \triangleright \tau_\alpha], \mathcal{I})}
\end{array}$$

In the rule, we require that the new assertion only refers to the variables visible to the action α . In our approach, each such assertion introduces a case split where either (1) the assertion holds or (2) the program goes wrong. Case (1) allows us to further simplify the program. Our method allows program transformations to proceed while the verification of the introduced assertion is deferred to later proof stages. The soundness of the transformation steps ensures that if the program obtained after a sequence of transformations is free of assertion violations, all “(2)” branches of earlier case splits were vacuous and all assertions introduced earlier were in fact valid. Borrowing assertions in this manner (instead of proving them right away) is a powerful tool, since in the transformed program with larger atomic actions, validation of assertions can often be done automatically.

We use assertions to enable later applications of the abstraction rule. Consider the case where we would like to replace $\phi_1 \triangleright \tau_1$ with $\phi_2 \triangleright \tau_2$ but it is not the case that $\mathcal{I} \vdash \phi_1 \triangleright \tau_1 \preceq \phi_2 \triangleright \tau_2$ because the invariant $\mathcal{I}$ is not strong enough. Instead of having to strengthen $\mathcal{I}$ by reasoning about the entire program, we simply express our belief about the program state when $\phi_1 \triangleright \tau_1$ is reached using an assertion ψ . The new simulation check $\mathcal{I} \vdash (\phi_1 \wedge \psi) \triangleright \tau_1 \preceq (\phi_2 \wedge \psi) \triangleright \tau_2$ is more likely to pass.

13.3.2 Read and Write Abstractions

Here we give two proof rules derived from ABSTRACT for abstracting the read or write of a single variable. A *read abstraction* makes the gated action $\phi \triangleright \tau$ read a nondeterministic value from v at the beginning of the action, thus making the operation of the action independent of the initial value of v when its execution starts. For this purpose, we use ABSTRACT-READ-I and ABSTRACT-READ-II given below; the difference between the actions is whether the original action writes to v or not.

$$\begin{array}{c}
\text{ABSTRACT-READ-I (ABSTRACT)} \\
\frac{\alpha = \phi \triangleright \langle \tau \rangle_M \quad v \in \text{Var}_\alpha \quad v \in M \quad \beta = \phi \triangleright \langle \exists v. \tau \rangle_M \quad \beta \sqsubseteq \mathcal{I}}{(P, \mathcal{I}) \dashrightarrow (P[\alpha \mapsto \beta], \mathcal{I})}
\end{array}
\qquad
\begin{array}{c}
\text{ABSTRACT-READ-II (ABSTRACT)} \\
\frac{\alpha = \phi \triangleright \langle \tau \rangle_M \quad v \in \text{Var}_\alpha \quad v \notin M \quad \beta = \phi \triangleright \langle v' = v \wedge \exists v. \tau \rangle_M \quad \beta \sqsubseteq \mathcal{I}}{(P, \mathcal{I}) \dashrightarrow (P[\alpha \mapsto \beta], \mathcal{I})}
\end{array}$$

A *write abstraction* makes the gated action $\phi \triangleright \tau$ write a nondeterministic value to v at the end of the action. For this purpose, we use ABSTRACT-WRITE given below:

$$\begin{array}{c}
\text{ABSTRACT-WRITE (ABSTRACT)} \\
\frac{\alpha = \phi \triangleright \langle \tau \rangle_M \quad v \in \text{Var}_\alpha \quad \beta = \phi \triangleright \langle \exists v'. \tau \rangle_{M \cup \{v\}} \quad \beta \sqsubseteq \mathcal{I}}{(P, \mathcal{I}) \dashrightarrow (P[\alpha \mapsto \beta], \mathcal{I})}
\end{array}$$

By the definition of simulation, both read and write abstractions are sound by construction. Notice that we do not allow abstractions that violate the program invariant $\mathcal{I}$.

13.3.3 Validating Gated Actions

Recall that, the abstraction between two actions is checked under the context of an invariant. This allows us to justify the soundness of discharging an assertion at the gate of an action. To see how the validating a gated action as a special case of the rule ABSTRACT, consider the rule RELAX-I below.

$$\begin{array}{c}
\text{RELAX-I (ABSTRACT)} \\
\frac{\models (\mathcal{I} \Rightarrow \phi)}{(P, \mathcal{I}) \dashrightarrow (P[\phi \triangleright \tau \mapsto \text{true} \triangleright \tau], \mathcal{I})}
\end{array}$$

The rule replaces the gate ϕ of an action with **true** if the gate is implied by $\mathcal{I}$. While checking the abstraction between two actions, we only consider the program states that satisfy $\mathcal{I}$. Due to $\models \mathcal{I} \Rightarrow \phi$, the action $\phi \triangleright \tau$ never goes wrong from states satisfying $\mathcal{I}$. Therefore, $\mathcal{I} \vdash \phi \triangleright \tau \sqsubseteq \text{true} \triangleright \tau$ holds trivially, and we can omit the gate ϕ and replace it with **true**. The program is proved correct (i.e., good) when all the gates of $\text{Atoms}(P)$ are replaced with **true**.

Example. Suppose $\mathcal{I} = (0 \leq y \wedge y < x)$ and x, y are integer variables. RELAX allows one to replace $(1 \leq x) \triangleright \langle x' = x + 1 \rangle_x$ with the validated action $\mathbf{true} \triangleright \langle x' = x + 1 \rangle_x$ since $\models (0 \leq y \wedge y < x) \Rightarrow (1 \leq x)$. $\square$

The following derived rule allows to eliminate one of the assertions in a gated action.

$$\text{RELAX-II (ABSTRACT)} \quad \frac{\models (\mathcal{I} \Rightarrow \phi_1)}{(P, \mathcal{I}) \dashrightarrow (P[(\phi_1 \wedge \phi_2) \triangleright \tau \mapsto \phi_2 \triangleright \tau], \mathcal{I})}$$

As we will explain in Section 14.2, an action $\phi \triangleright \tau$ represents an atomic statement in QEDPL where the gate ϕ is obtained by propagating the assertions in the code block to the beginning of the block using the weakest pre-condition operator [79]. Therefore, the rules RELAX-I and RELAX-II emphasize the fact that validating assertions in an action, in essence, is nothing more than sequential reasoning where the invariant is used as the pre-condition.

13.4 Variable Addition and Hiding

Our proof system allows to add and hide variables to/from the program. Intuitively, variable introduction rewrites actions in the program so that these can refer to a new variable. Variable hiding is the dual of variable introduction; each action is rewritten so that it no longer refers to the hidden variable. Hiding a variable that appears in the invariant also requires quantifying out that variable in the invariant. In the following, we give the rules for adding auxiliary variables and hiding variables; we demonstrate typical uses of these rules in Chapter 16. The rules distinguish between global and local variables. We require that the signature of the program (Sig_P) is fixed makes during the proof, so the rules presented below cannot be used to add or hide procedure input and output variables, but only global variables and private variables of procedures. In order to ensure soundness, in both cases, we define a new abstraction relation for each rule. The new abstraction relations are used to relate actions over different sets of variables.

13.4.1 Adding Auxiliary Variables

The rules for adding new variables require that each action α_i be replaced by an action β_i that preserves the invariant and abstracts the former. This abstraction relation is expressed by the abstraction relation $\preceq_{+a}$ under variable addition, as defined below.

Definition 7 (Abstraction under variable addition):

Let α, β be actions and $\mathcal{I}$ be a state predicate. Let v be a fresh variable such that $v \notin \text{Var}_\alpha$ and $v \in \text{Var}_\beta$. $\mathcal{I} \vdash \alpha \preceq_{+v} \beta$ holds if the following are both valid:

1. $(\mathcal{I} \wedge \neg\phi_\alpha) \Rightarrow (\forall v. \neg\phi_\beta)$
2. $(\mathcal{I} \wedge \tau_\alpha) \Rightarrow (\forall v. (\neg\phi_\beta \vee (\exists v'. \tau_\beta)))$ ■

The definition above states that the new action β specifies how a is updated (or left unmodified), but preserves the effect of α on the existing variables. In particular, the introduction of a cannot cause β to block at a state s if the original action α did not. However, the new variable a can be used in the gate of β and can cause β to go wrong more often than α did.

The rules ADD-GLOBAL-VARIABLE and ADD-LOCAL-VARIABLE allow one to add a new variable, a , to the program and rewrite each action $\alpha \in \text{Atoms}(P)$ to a new action β so that they β refers to the new variable. This corresponds to annotating the program with *history* variables as in [42, 62], except that, we also allow the gates (assertions) to be updated along with the transitions.

ADD-GLOBAL-VARIABLE

$$\frac{\begin{array}{l} \text{Atoms}(P) = \{\alpha_1, \dots, \alpha_n\} \quad \beta_1 \triangleleft \mathcal{I} \quad \dots \quad \beta_n \triangleleft \mathcal{I} \\ a \notin \text{Global} \quad \mathcal{I} \vdash \alpha_1 \preceq_{+a} \beta_1 \quad \dots \quad \mathcal{I} \vdash \alpha_n \preceq_{+a} \beta_n \end{array}}{(P, \mathcal{I}) \dashrightarrow (P[\text{Global} \mapsto \text{Global} \cup \{a\}, \alpha_1 \mapsto \beta_1, \dots, \alpha_n \mapsto \beta_n, \mathcal{I}]}$$

ADD-LOCAL-VARIABLE

$$\frac{\begin{array}{l} \mathbf{p} \in \text{Proc} \quad \text{Atoms}(\text{Body}_{\mathbf{p}}) = \{\alpha_1, \dots, \alpha_n\} \quad \beta_1 \triangleleft \mathcal{I} \quad \dots \quad \beta_n \triangleleft \mathcal{I} \\ a \notin \text{Local}_{\mathbf{p}} \quad \mathcal{I} \vdash \alpha_1 \preceq_{+a} \beta_1 \quad \dots \quad \mathcal{I} \vdash \alpha_n \preceq_{+a} \beta_n \end{array}}{(P, \mathcal{I}) \dashrightarrow (P[\text{Private}_{\mathbf{p}} \mapsto \text{Private}_{\mathbf{p}} \cup \{a\}, \alpha_1 \mapsto \beta_1, \dots, \alpha_n \mapsto \beta_n, \mathcal{I}]}$$

13.4.2 Hiding Existing Variables

The rules for hiding existing variables require that each action α_i be replaced by an action β_i that preserves the invariant and abstracts the former. This relationship between α_i and β_i is expressed by the abstraction relation $\preceq_{-a}$ under variable hiding, as defined below.

Definition 8 (Abstraction under variable hiding):

Let α, β be actions and $\mathcal{I}$ be a state predicate. Let v be a variable such that $v \in \text{Var}_P$ and $v \notin \text{Var}_\beta$. $\mathcal{I} \vdash \alpha \preceq_{-v} \beta$ holds if the following are both valid:

1. $(\exists v. \mathcal{I} \wedge \neg \phi_\alpha) \Rightarrow \neg \phi_\beta$
2. $(\exists v, v'. \mathcal{I} \wedge \tau_\alpha) \Rightarrow (\neg \phi_\beta \vee \tau_\beta)$ ■

The definition above states that the new action β must preserve all transitions of α on the remaining variables. In particular, β must preserve all existing (successful or failing) transitions of α over the remaining variables.

The rules HIDE-GLOBAL-VARIABLE and HIDE-LOCAL-VARIABLE allow one to remove an existing variable, a , from the program and rewrite each action $\alpha \in \text{Atoms}(P)$ to a new action β so that β no longer refers to a .

HIDE-GLOBAL-VARIABLE

$$\frac{\begin{array}{c} \text{Atoms}(P) = \{\alpha_1, \dots, \alpha_n\} \quad \beta_1 \hookrightarrow \exists a. \mathcal{I} \quad \dots \quad \beta_n \hookrightarrow \exists a. \mathcal{I} \\ a \in \text{Global} \quad \mathcal{I} \vdash \alpha_1 \preceq_{-a} \beta_1 \quad \dots \quad \mathcal{I} \vdash \alpha_n \preceq_{-a} \beta_n \end{array}}{(P, \mathcal{I}) \dashrightarrow (P[\text{Global} \mapsto \text{Global} \setminus \{a\}, \alpha_1 \mapsto \beta_1, \dots, \alpha_n \mapsto \beta_n], \exists a. \mathcal{I})}$$

HIDE-LOCAL-VARIABLE

$$\frac{\begin{array}{c} \mathbf{p} \in \text{Proc} \quad \text{Atoms}(\text{Body}_{\mathbf{p}}) = \{\alpha_1, \dots, \alpha_n\} \quad \beta_1 \hookrightarrow \exists a. \mathcal{I} \quad \dots \quad \beta_n \hookrightarrow \exists a. \mathcal{I} \\ a \in \text{Private}_{\mathbf{p}} \quad \mathcal{I} \vdash \alpha_1 \preceq_{-a} \beta_1 \quad \dots \quad \mathcal{I} \vdash \alpha_n \preceq_{-a} \beta_n \end{array}}{(P, \mathcal{I}) \dashrightarrow (P[\text{Private}_{\mathbf{p}} \mapsto \text{Private}_{\mathbf{p}} \setminus \{a\}, \alpha_1 \mapsto \beta_1, \dots, \alpha_n \mapsto \beta_n], \mathcal{I})}$$

13.5 Reduction

Our reduction rules are based on Lipton's theory [47], and aim to combine multiple actions into a single action. For example, the following rule –to be explained later– allows us to make a single action from a sequential composition of two actions if the first action is a right mover.

REDUCE-SEQUENTIAL-RIGHT

$$\frac{(P, \mathcal{I}) \vdash \alpha_1 : \mathbb{R}}{(P, \mathcal{I}) \dashrightarrow (P[\alpha_1; \alpha_2 \mapsto \llbracket \alpha_1; \alpha_2 \rrbracket], \mathcal{I})}$$

where the antecedent of the rule indicates that α is a right mover, and $\llbracket \alpha_1; \alpha_2 \rrbracket$ denotes an atomic action representing the atomic execution of the statement $\alpha_1; \alpha_2$. In this section, we first define the atomic composition operator $\llbracket \cdot \rrbracket$ (Section 13.5.1), then formalize the notion of movers and introduce a static, local check for mover types (Section 13.5.2), and finally present the rules for reducing various compound statements and procedures to actions (Section 13.5.7-13.6).

13.5.1 Atomic Composition of Actions

The $\llbracket \cdot \rrbracket : Stmt \rightarrow Atomic$ operator takes a statement and produces a single action that represents the atomic execution of the given statement. We use this operator to specify the result of the reduction rules for sequential (;) and choice ($\square$) operators. Moreover, in Section 13.6, we make use of $\llbracket \cdot \rrbracket$ to compute verification conditions to check conformance between statements. The definition of $\llbracket \cdot \rrbracket$ uses the following two definitions:

Definition 9 (Weakest (liberal) pre-condition):

Let ϕ be a state predicate and τ be a transition predicate, and $V = Var_\tau$. The weakest (liberal) pre-condition of ϕ for τ , denoted $wp(\tau, \phi)$, is a state predicate specifying all the states that cannot reach a state that violates ϕ after executing τ without blocking. Formally,

$$wp(\tau, \phi) \triangleq \forall V'. (\tau \Rightarrow \phi[V'/V]) \quad (13.2)$$

■

Definition 10 (Atomic composition of transitions):

Let τ_1 and τ_2 be transition predicates, and $V = Var_{\tau_1} \cap Var_{\tau_2}$. The atomic composition of τ_1 and τ_2 , denoted $\tau_1 \cdot \tau_2$, is a transition predicate specifying the effect of applying first τ_1 and then τ_2 to the current program state. Formally,

$$\tau_1 \cdot \tau_2 \triangleq \exists V''. (\tau_1[V''/V'] \wedge \tau_2[V''/V]) \quad (13.3)$$

■

We define $\llbracket \cdot \rrbracket$ recursively in Figure 13.1. The figure specifies the composition of two gated actions by the “;” and “ $\square$ ” operators. Intuitively, $\llbracket S \rrbracket$ denotes a gated action that simulates (resembling the $\preceq$ relation in Definition 6) the sequential execution of the statement S . For example, if α or β can go wrong when they are executed consecutively from a state s , then the action $\llbracket \alpha; \beta \rrbracket$ can go wrong from a state s . If α or β make normal transitions when they are executed consecutively from s , then $\llbracket \alpha; \beta \rrbracket$ can either go wrong from s , or it has a transition

$$\boxed{\llbracket \cdot \rrbracket : Stmt \rightarrow Atomic}$$

$$\begin{aligned}
\alpha \# \beta &\triangleq (\phi_\alpha \wedge \mathbf{wp}(\tau_\alpha, \phi_\beta)) \triangleright (\tau_\alpha \cdot \tau_\beta) \\
\llbracket \alpha; l : \beta \rrbracket &\triangleq \alpha \# (exc \neq l \vee \phi_\beta) \triangleright \langle exc \neq l \rangle_\emptyset \vee (exc = l \wedge \tau_\beta) \\
\llbracket \alpha; * : \beta \rrbracket &\triangleq \alpha \# (exc = \mathbf{skip} \vee \phi_\beta) \triangleright \langle exc = \mathbf{skip} \rangle_\emptyset \vee (exc \neq \mathbf{skip} \wedge \tau_\beta) \\
\llbracket \alpha; \beta \rrbracket &\triangleq \llbracket \alpha; \mathbf{skip} : \beta \rrbracket \\
\llbracket \alpha \square \beta \rrbracket &\triangleq (\phi_\alpha \wedge \phi_\beta) \triangleright (\tau_\alpha \vee \tau_\beta)
\end{aligned}$$

Figure 13.1: Composing gated actions

that summarizes the consecutive transitions of α and β . Similarly, the action $\llbracket \alpha \square \beta \rrbracket$ can go wrong from a state s if α or β can go wrong from s ; otherwise, it makes a transition from s that is allowed by either α or β . We use $\llbracket \cdot \rrbracket$ to compose actions from the same procedure, so the scope is preserved by atomic composition, i.e., $Var_\alpha = Var_\beta = Var_{\llbracket \alpha; \beta \rrbracket} = Var_{\llbracket \alpha \square \beta \rrbracket}$. Note that, Var_α consists of the global variables of the program and the local variables of the containing procedure. In Figure 13.3 of Section 13.6, we will extend the definition of $\llbracket S \rrbracket$ for arbitrary compound statements.

13.5.2 Movers

Reduction classifies each action in the program as a mover. There are four types of movers: right mover ($\mathbb{R}$), left mover ($\mathbb{L}$), both mover ($\mathbb{B}$), and atomic non-mover ($\mathbb{A}$). Informally, an action α is a right mover (resp., left mover) if for every run of the program, commuting α to the right of the next action (resp. to the left of the previous action) either leads to the same state or leads to a violation. Below we give the formal definitions of right and left movers.

Definition 11 (Right mover):

An action α is a right mover in a proof state $(P, \mathcal{I})$, if for each run $s \xrightarrow{1} s_1 \xrightarrow{(t, \alpha)(u, \beta)} s_2$ of P from $\mathcal{I}$ such that $t \neq u$, there exists either (1) a run $s \xrightarrow{1} s_1 \xrightarrow{(u, \beta)(t, \alpha)} s_2$, (2) a violation $s \xrightarrow{1} s_1 \xrightarrow{(u, \beta)} \mathbf{wrong}$, or (3) a violation $s \xrightarrow{1} s_1 \xrightarrow{(u, \beta)(t, \alpha)} \mathbf{wrong}$. ■

Definition 12 (Left mover):

An action α is a left mover in a proof state $(P, \mathcal{I})$, if for each run $s \xrightarrow{1} s_1 \xrightarrow{(u, \beta)(t, \alpha)} s_2$ of P from $\mathcal{I}$ such that $t \neq u$, there exists either (1) a run $s \xrightarrow{1} s_1 \xrightarrow{(t, \alpha)(u, \beta)} s_2$, (2) a violation

(a) $P, \mathcal{I} \vdash S : m$

RIGHT-MOVER $\frac{\forall \beta \in \text{Atoms}(P). \mathcal{I} \vdash \mathcal{MC}(\alpha, \beta) \quad \forall \beta \in \text{Atoms}(P). \mathcal{I} \vdash \alpha \Rightarrow \beta}{(P, \mathcal{I}) \vdash \alpha : \mathbb{R}}$	LEFT-MOVER $\frac{\forall \beta \in \text{Atoms}(P). \mathcal{I} \vdash \mathcal{MC}(\beta, \alpha) \quad \text{Non-blocking}(\alpha)}{(P, \mathcal{I}) \vdash \alpha : \mathbb{L}}$	
ATOMIC-NON-MOVER $\frac{}{(P, \mathcal{I}) \vdash \phi \triangleright \tau : \mathbb{A}}$	BOTH-MOVER $\frac{(P, \mathcal{I}) \vdash S : \mathbb{L} \quad (P, \mathcal{I}) \vdash S : \mathbb{R}}{(P, \mathcal{I}) \vdash S : \mathbb{B}}$	SUP-MOVER $\frac{(P, \mathcal{I}) \vdash S : n \quad n \sqsubseteq m}{(P, \mathcal{I}) \vdash S : m}$
LABELED-MOVER $\frac{(P, \mathcal{I}) \vdash S : m \quad \dagger \in \text{Lbl} \cup \{\star\}}{(P, \mathcal{I}) \vdash (\dagger : S) : m}$	COMBINE-MOVERS $\frac{(P, \mathcal{I}) \vdash S_1 : m \quad (P, \mathcal{I}) \vdash S_2 : n \quad \natural \in \{;, \square\}}{(P, \mathcal{I}) \vdash \llbracket S_1 \natural S_2 \rrbracket : m \natural n}$	
CALL-MOVER $\frac{\mathcal{A}(\mathbf{p}) = (m, \phi, \tau)}{(P, \mathcal{I}, \mathcal{A}) \vdash \mathbf{p}() : m}$	LOOP-MOVER $\frac{P, \mathcal{I} \vdash S : m}{P, \mathcal{I} \vdash S^\odot : m^\odot}$	STATEMENT-MOVER $\frac{(P, \mathcal{I}) \vdash S_1 : m \quad (P, \mathcal{I}) \vdash S_2 : n \quad \natural \in \{;, \square, \parallel\}}{(P, \mathcal{I}) \vdash S_1 \natural S_2 : m \natural n}$

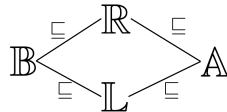
(b) Definitions of $m; n$, $m \square n$, $m \parallel n$, and $m^\odot$

$m; n$	$\mathbb{B}$	$\mathbb{R}$	$\mathbb{L}$	$\mathbb{A}$		$m \square n$	$\mathbb{B}$	$\mathbb{R}$	$\mathbb{L}$	$\mathbb{A}$		$m \parallel n$	$\mathbb{B}$	$\mathbb{R}$	$\mathbb{L}$	$\mathbb{A}$		$m^\odot$	
$\mathbb{B}$	$\mathbb{B}$	$\mathbb{R}$	$\mathbb{L}$	$\mathbb{A}$		$\mathbb{B}$	$\mathbb{B}$	$\mathbb{R}$	$\mathbb{L}$	$\mathbb{A}$		$\mathbb{B}$	$\mathbb{B}$	$\mathbb{R}$	$\mathbb{L}$	$\mathbb{A}$		$\mathbb{B}$	$\mathbb{B}$
$\mathbb{R}$	$\mathbb{R}$	$\mathbb{R}$	$\mathbb{A}$	$\mathbb{A}$		$\mathbb{R}$	$\mathbb{R}$	$\mathbb{R}$	$\mathbb{A}$	$\mathbb{A}$		$\mathbb{R}$	$\mathbb{R}$	$\mathbb{R}$	-	-		$\mathbb{R}$	$\mathbb{R}$
$\mathbb{L}$	$\mathbb{L}$	-	$\mathbb{L}$	-		$\mathbb{L}$	$\mathbb{L}$	$\mathbb{A}$	$\mathbb{L}$	$\mathbb{A}$		$\mathbb{L}$	$\mathbb{L}$	-	$\mathbb{L}$	-		$\mathbb{L}$	$\mathbb{L}$
$\mathbb{A}$	$\mathbb{A}$	-	$\mathbb{A}$	-		$\mathbb{A}$	$\mathbb{A}$	$\mathbb{A}$	$\mathbb{A}$	$\mathbb{A}$		$\mathbb{A}$	$\mathbb{A}$	-	-	-		$\mathbb{A}$	-

Figure 13.2: Rules for deciding movers

$s \xrightarrow{1} s_1 \xrightarrow{(t, \alpha)} \text{wrong}$, or (3) a violation $s \xrightarrow{1} s_1 \xrightarrow{(t, \alpha)(u, \beta)} \text{wrong}$. ■

A both mover is both a right mover and a left mover, and an atomic non-mover is neither a right mover nor a left mover. By definition, every action is an atomic non-mover. The partial order $\sqsubseteq$ defines a lattice over movers as follows:



We use the judgment $(P, \mathcal{I}) \vdash \alpha : m$ to indicate that α is m -mover in the proof state $(P, \mathcal{I})$, where $m \in \{\mathbb{L}, \mathbb{R}, \mathbb{B}, \mathbb{A}\}$.

Figure 13.2 shows the judgment rules for deciding the mover type of a statement. Notice that the rules in the figure are used not only for gated actions but also for compound statements. The rules RIGHT-MOVER and LEFT-MOVER in Figure 13.2 decide right and left movers for gated actions, as will be described below. ATOMIC-NON-MOVER states that every gated action is a non-mover, BOTH-MOVER specifies both movers, and SUP-MOVER determines mover types using the partial order $\sqsubseteq$. The rest of the rules are used to determine the mover type of composed actions (via $\llbracket \cdot \rrbracket$) and compound statements, as explained in Section 13.5.6 and Section 13.6.

13.5.3 The Checks for Movers

The above definition of movers requires a global reasoning over all executions of the program to decide whether a particular action is a mover or not. This is nearly impossible to do efficiently for large programs. Instead, we formulate a sufficient, yet precise, mechanism to label actions as movers. In the following, we first define a pairwise check, called the *mover check* between an action α and every other action in the program, using which we define our checks for right and left movers. We encode the mover check using the abstraction relation ($\preceq$) relation, which can be automatically verified using an SMT solver. In this way, we reduce the global reasoning over executions to a local reasoning in which we only consider consecutive execution of a pair of actions.

For the following definitions, fix a proof state $(P, \mathcal{I})$ and let α and β be two actions of the program P . We consider a consecutive execution of α and β by different threads. In order to express this statically, we do the following two transformations on the actions:

- We replace *tid* with distinct symbolic thread ids. Given an action α , define $\alpha[t]$ to be the action obtained by substituting the thread id t for every occurrence of *tid* in ϕ_α and τ_α .
- Since α and β are by different threads, the local variables in their scope must be distinct. While local variables of different procedures have different names, α and β may be from the same procedure. To indicate that α and β refer to distinct local

variables in this case, we rename the local variables in α and β with fresh variable names. Extending the definition of *Fresh* in Section 12.4.5, define $\text{Fresh}(\alpha)$ be the action obtained from α by renaming all the local variables in ϕ_α and τ_α to fresh variables.

In the following we define two operators that sequentially compose actions, say α and β , to a single action. These operators are similar to $\llbracket \cdot \rrbracket$ defined in Section 13.5.1, except that they combine actions that are executed by different threads. The renaming of the local variables ensures that V refers to *Global* in Equations (13.3) and (13.2). Thus, the state predicate $\phi_\alpha \wedge \mathbf{wp}(\tau_\alpha, \phi_\beta)$ and the transition predicate $\tau_\alpha \cdot \tau_\beta$ encode that α and β does not read from or write to the local variables of each other.

1. $\alpha \circ \beta$ produces a gated action satisfying the following:

$$\begin{aligned} - \phi_{\alpha \circ \beta} &= \phi_\alpha \wedge \mathbf{wp}(\tau_\alpha, \phi_\beta) \\ - \tau_{\alpha \circ \beta} &= (\phi_\alpha \wedge \tau_\alpha) \cdot (\phi_\beta \wedge \tau_\beta) \end{aligned}$$

2. $\alpha \bullet \beta$ produces a gated action satisfying the following:

$$\begin{aligned} - \phi_{\alpha \bullet \beta} &= (\phi_\alpha \Rightarrow \mathbf{wp}(\tau_\alpha, \phi_\beta)) \wedge (\neg \phi_\alpha \Rightarrow (\phi_\beta \wedge \mathbf{wp}(\tau_\beta, \text{false}))) \\ - \tau_{\alpha \bullet \beta} &= (\phi_\alpha \wedge \tau_\alpha) \cdot (\phi_\beta \wedge \tau_\beta) \end{aligned}$$

Definition 13 (Mover check):

Let α and β be two actions, $\tilde{\alpha} = \text{Fresh}(\alpha)$, and $\tilde{\beta} = \text{Fresh}(\beta)$. The mover check between α and β at a proof state $(P, \mathcal{I})$ succeeds, denoted by $\mathcal{I} \vdash \mathcal{MC}(\alpha, \beta)$, if the following holds:

$$\forall t \neq u. \quad \mathcal{I} \vdash (\tilde{\alpha}[t] \bullet \tilde{\beta}[u]) \preceq (\tilde{\beta}[u] \circ \tilde{\alpha}[t]) \quad (13.4)$$

Notice that, the abstraction check in $\mathcal{MC}(\alpha, \beta)$ is not symmetric; we use different composition operators on the left and right hand sides of Equation 13.4. In fact, the only difference between $\alpha \circ \beta$ and $\alpha \bullet \beta$ is that from states at which α goes wrong and β blocks, $\alpha \circ \beta$ goes wrong but $\alpha \bullet \beta$ blocks.

For actions α and β , the judgment $\mathcal{I} \vdash \mathcal{MC}(\alpha, \beta)$ dictates the following:

For every state s_1 such that $s_1 \models \mathcal{I}$, if $s_1 \xrightarrow{(t, \alpha)(u, \beta)} s_2$, then either (1) $s_1 \xrightarrow{(u, \beta)(t, \alpha)} s_2$, (2) $s_1 \xrightarrow{(u, \beta)} \text{wrong}$, or (3) $s_1 \xrightarrow{(u, \beta)(t, \alpha)} \text{wrong}$ holds. In words, whenever α

and β are executed consecutively by different threads, swapping the execution order of the actions yields either a run that leads to the same state after the actions or a violation.

13.5.4 Deciding Right and Left Movers

Recall the definition of right and left movers in Definitions 11 and 12. The abstraction relation in $\mathcal{MC}$ precisely captures the conditions in the definitions, except that $\mathcal{I}$ is used to approximate the set of states “ s_1 ” in the definition.

The rules RIGHT-MOVER and LEFT-MOVER in Figure 13.2 decide right and left movers of gated actions using the mover check ($\mathcal{MC}$) defined above. In words, for an action α to be a right mover, $\mathcal{I} \vdash \mathcal{MC}(\alpha, \beta)$ must hold for every action β in the program; similarly, for α to be a left mover, $\mathcal{I} \vdash \mathcal{MC}(\beta, \alpha)$ must hold for every action β in P .

In addition to the mover check, the rule RIGHT-MOVER also requires that α preserves the gates of all actions in the program, defined as follows:

Definition 14 (Preserving gates):

Let α and β be two actions, $\tilde{\alpha} = \text{Fresh}(\alpha)$, and $\tilde{\beta} = \text{Fresh}(\beta)$. Action α is said to preserve the gate of β , denoted $\mathcal{I} \vdash \alpha \rightleftharpoons \beta$, if from every state pair s_1, s_2 such that $s_1 \models (\mathcal{I} \wedge \phi_\beta)$ and $s_1 \xrightarrow{(t, \alpha)} s_2$, we have $s_2 \models \phi_\beta$. The judgment $\mathcal{I} \vdash \alpha \rightleftharpoons \beta$ holds if the following is valid:

$$\forall t \neq u. ((\mathcal{I} \wedge \phi_{\tilde{\beta}[u]}) \Rightarrow \text{wp}(\tau_{\tilde{\alpha}[t]}, \phi_{\tilde{\beta}[u]})) \quad (13.5)$$

The rule LEFT-MOVER also requires that the actions be non-blocking, i.e., they can always make a transition to a post-state from every state from which they do not go wrong.

Definition 15 (Non-blocking actions):

An action α is non-blocking, denoted $\text{Non-blocking}(\alpha)$, if the following is valid:

$$\phi_\alpha \Rightarrow (\exists \text{Var}'_\alpha. \tau_\alpha) \quad (13.6)$$

Notice that, apart from the extra conditions (preserving gates of other actions for right movers and being non-blocking for left movers) the judgments for the right and left movers both make use of the same mover check. The claims (1) α commutes to the right of β (while deciding right movers) and (2) β commutes to the left of α (while deciding left movers) are determined using the same commutativity check: $\mathcal{I} \vdash \mathcal{MC}(\alpha, \beta)$. Therefore, the result of

the check $\mathcal{MC}(\alpha, \beta)$ used to prove that α is a right mover can be reused while checking if β is a left mover. This means that an analysis to determine the right and left movers can use the same set of mover checks, i.e., $\mathcal{MC}(\alpha, \beta)$ for each pair (α, β) is performed only once. If there are n actions in $Atoms(P)$, then checking the mover type (right and left) of all the actions in the program requires only $\binom{n}{2} + n = \frac{n^2+n}{2}$ mover checks.

In addition, the mover check allows us to perform the reduction step in a modular way. Formally, if α and β are independent from each other (see Definition 2), then $\mathcal{I} \vdash \mathcal{MC}(\alpha, \beta)$ and $\mathcal{I} \vdash \mathcal{MC}(\beta, \alpha)$ both hold. Recall that, the formulation of the mover check already ensures that the local variables accessed by α and β are distinct. The modularity property states that the mover check $\mathcal{MC}(\alpha, \beta)$ will succeed if α and β read and write distinct global variables. This enables us to exploit the modular design of the program. Without loss of generality, let a module be defined as a collection of procedures. If two distinct modules M_A and M_B of the program are shown to access distinct set of global variables, then the mover types of actions in M_A can be checked omitting the actions in M_B , and vice versa. In addition, assume that the program did not contain the module M_B and we add M_B to the program after transforming the module M_A through a number of QED proof steps and proving assertions in it. Addition of M_B does not invalidate the soundness of the former steps of the proof on M_A , but only affects—requiring the former proof steps be repeated—the modules that share global variables with M_B .

Theorem 3 (The mover check $\mathcal{MC}$ is modular):

Given a proof state $(P, \mathcal{I})$, for any pair (α, β) of actions in P are independent from each other, $\mathcal{I} \vdash \mathcal{MC}(\alpha, \beta)$ holds. ■

PROOF.

- (1)1. We prove the claim by picking a state s_1 such that $s_1 \models \mathcal{I}$ and $s_1 \xrightarrow{(t, \alpha)} s_2 \xrightarrow{(u, \beta)} s_3$, and showing that either $s_1 \xrightarrow{(u, \beta)} s'_2 \xrightarrow{(t, \alpha)} s'_3$ exists such that $\sigma_{s'_3} = \sigma_{s_3}$ and $\theta_{s'_3} = \theta_{s_3}$.
- (1)2. By Definition 2, either $\text{Var}_\alpha \cap \text{Var}_\beta = \emptyset$ or α and β do not conflict on any common variables. Since the global variables are visible to all actions in the program, we will consider the latter case.
- (1)3. Let R_α and R_β be the sets of variables read by the actions in their gate and transition predicates and W_α and W_β be the sets of variables written by the actions.

- (1)4. By equation (12.3), $R_\alpha \cap W_\beta = R_\beta \cap W_\alpha = W_\alpha \cap W_\beta = \emptyset$. Thus, modification by one of the actions on the program state does not affect the operation of the other.
 (1)5. $\sigma_{s_1}|_{R_\alpha} = \sigma_{s'_2}|_{R_\alpha}$, by $R_\alpha \cap W_\beta = \emptyset$
 (1)6. α is not blocked at s'_2 since α is not blocked at s_1 and by (1)5
 (1)7. $\sigma_{s_2}|_{R_\beta} = \sigma_{s_1}|_{R_\beta}$, by $R_\beta \cap W_\alpha = \emptyset$
 (1)8. β is not blocked at s_1 since β is not blocked at s_2 and by (1)7
 (1)9. Since the operation of α only depends on the valuation of R_α , $\sigma_{s_2}|_{W_\alpha} = \sigma_{s'_3}|_{W_\alpha}$. Since $W_\alpha \cap W_\beta = \emptyset$, $\sigma_{s_2}|_{W_\alpha} = \sigma_{s_3}|_{W_\alpha}$. Thus $\sigma_{s_3}|_{W_\alpha} = \sigma_{s'_3}|_{W_\alpha}$.
 (1)10. Similarly to the previous step, since the operation of β only depends on the valuation of R_β , $\sigma_{s_3}|_{W_\beta} = \sigma_{s'_2}|_{W_\beta}$. Since $W_\alpha \cap W_\beta = \emptyset$, $\sigma_{s'_2}|_{W_\beta} = \sigma_{s'_3}|_{W_\beta}$. Thus $\sigma_{s_3}|_{W_\beta} = \sigma_{s'_3}|_{W_\beta}$.
 (1)11. Since $W_\alpha \cap W_\beta$, $\sigma_{s_3}|_{W_\alpha \cup W_\beta} = \sigma_{s'_3}|_{W_\alpha \cup W_\beta}$.
 (1)12. Since α and β only write to $W_\alpha \cup W_\beta$, $\sigma_{s_3} = \sigma_{s'_3}$.
 (1)13. Since $\sigma_{s_1}|_{R_\alpha} = \sigma_{s'_2}|_{R_\alpha}$ ((1)9) and $\sigma_{s_2}|_{R_\beta} = \sigma_{s_1}|_{R_\beta}$ ((1)10), $\theta_{s'_3} = \theta_{s_3}$.
 (1)14. The claim holds by (1)12 and (1)13.

□

13.5.5 Role of Assertions in Mover Checks

Adding an assertion to an action α is an abstraction, and replacing actions with more abstract actions is sound for assertion checking. A common way of using assertions is to eliminate apparent interference between actions which in reality does not exist due to synchronization mechanisms, such as mutual exclusion, used in the program. For example, by adding contradicting assertions to a pair (α, β) we indicate that α and β run at different synchronization states and may never immediately follow each other. Consider the following actions with two conflicting writes to the global variable x :

$$\alpha = (\text{lock-owner} = \text{tid}) \triangleright \langle x' = y \rangle_x$$

$$\beta = (\text{lock-owner} = \text{tid}) \triangleright \langle x' = z \rangle_x$$

Assume that we use the variable `lock-owner` to track the id of the thread holding a mutex lock. The assertion “`lock-owner = tid`” at the gates of the actions indicates that the actions may run only when the current thread is holding the lock. Recall that, the formulation of the mover check $\mathcal{MC}(\alpha, \beta)$ assumes different values of `tid` for α and β . Thus, the assertions in

α and β refer to different synchronization state of the mutex lock, where the lock is held by different threads. In this case, any consecutive execution of these actions leads to a violation and thus the simulation check 13.4 is satisfied trivially. In Chapter 18, we investigate several common synchronization idioms and provide a technique that automatically generates assertion annotations to show atomicity of code blocks using these idioms.

13.5.6 Deciding Movers After Reduction Steps

Starting with next section, we give rules for reducing composition of –usually two– actions to single actions. After the application of each rule, there are two ways to compute the mover type of the new action, through a fast and a slow path. In the following, assume that a reduction rule replaces a compound statement $\alpha; \beta$ with $\llbracket \alpha; \beta \rrbracket$. The reasoning below also applies to nondeterministic choice (where $\alpha \sqcap \beta$ is replaced with $\llbracket \alpha \sqcap \beta \rrbracket$), parallel composition (where $\alpha \parallel \beta$ is replaced with $\llbracket \alpha; \beta \rrbracket$).

The fast path is to use the judgment rule COMBINE-MOVERS in Figure 13.2. The rule COMBINE-MOVERS uses the mover types of α and β to compute a sound but imprecise estimate of the mover type of $\llbracket \alpha; \beta \rrbracket$ without performing any extra mover checks. The table in Figure 13.2 gives the resulting mover type of combining mover types. The symbol – in the table indicates that no conclusion about the mover type can be reached.

The slow path is to use the rules RIGHT-MOVER and LEFT-MOVER in Figure 13.2 to compute the mover type of $\llbracket \alpha; \beta \rrbracket$ precisely. This requires performing mover checks between $\llbracket \alpha; \beta \rrbracket$ and all other actions in the program and thus incurs extra cost. However, this may prove $\llbracket \alpha; \beta \rrbracket$ to be a right or left mover although the fast path concludes with a non mover ($\mathbb{A}$) type.

13.5.7 Reducing Sequential Composition

The rules REDUCE-SEQUENTIAL-RIGHT and REDUCE-SEQUENTIAL-LEFT make use of the $\llbracket \cdot \rrbracket$ operator defined above to make a single action from a sequential composition of two actions. The rules are applicable to $\alpha; \beta$ if either α is a right mover or β is a left mover.

REDUCE-SEQUENTIAL-RIGHT

$$\frac{(P, \mathcal{I}) \vdash \alpha : \mathbb{R}}{(P, \mathcal{I}) \dashrightarrow (P[\alpha; \beta \mapsto \llbracket \alpha; \beta \rrbracket], \mathcal{I})}$$

REDUCE-SEQUENTIAL-LEFT

$$\frac{(P, \mathcal{I}) \vdash \beta : \mathbb{L}}{(P, \mathcal{I}) \dashrightarrow (P[\alpha; \beta \mapsto \llbracket \alpha; \beta \rrbracket], \mathcal{I})}$$

In addition, we present two rules to reduce statements of the form $\alpha; l : \beta$ and $\alpha; * : \beta$ to a single gated action. The statement represents a **try-catch** statement where α is the **try**-body and β is the **catch**-body. The result of the rule is expressed using the $\llbracket \cdot \rrbracket$ operator introduced in Figure 13.1.

REDUCE-LABELED-RIGHT

$$\frac{(P, \mathcal{I}) \vdash \alpha : \mathbb{R} \quad \dagger \in Lbl \cup \{*\}}{(P, \mathcal{I}) \dashrightarrow (P[\alpha; \dagger : \beta \mapsto \llbracket \alpha; \dagger : \beta \rrbracket], \mathcal{I})}$$

REDUCE-LABELED-LEFT

$$\frac{(P, \mathcal{I}) \vdash \beta : \mathbb{L} \quad \dagger \in Lbl \cup \{*\}}{(P, \mathcal{I}) \dashrightarrow (P[\alpha; \dagger : \beta \mapsto \llbracket \alpha; \dagger : \beta \rrbracket], \mathcal{I})}$$

After the application of the rules above, the mover type of the new action can be computed as described in Section 13.5.6.

13.5.8 Reducing Nondeterministic Choice

The rule REDUCE-CHOICE makes use of the $\llbracket \cdot \rrbracket$ operator defined above to make a single action from a nondeterministic choice composition of two actions. The rule does not depend on the mover types of the actions.

REDUCE-CHOICE

$$\frac{}{(P, \mathcal{I}) \dashrightarrow (P[\alpha \square \beta \mapsto \llbracket \alpha \square \beta \rrbracket], \mathcal{I})}$$

After the application of the rule, the mover type of the new action $\llbracket \alpha \square \beta \rrbracket$ can be computed as described in Section 13.5.6.

13.5.9 Reducing Parallel Composition

We reduce parallel statements composed of gated actions to sequential compositions using three rules. The first rule, EXPAND-PARALLEL, eliminates the parallel composition by explicitly enumerating the two possible orderings of the gated actions. We say that X and Y are *disjointly-fresh for* $Local_{\mathbf{p}}$ to indicate that each of X and Y represent a fresh set of variables for $Local_{\mathbf{p}}$ (see the use of *fresh for* in Equation 12.7) and $X \cap Y = \emptyset$.

EXPAND-PARALLEL

$$\begin{array}{c}
\alpha_1, \beta_1 \in \text{Atoms}(\text{Body}_{\mathbf{p}}) \quad X \text{ and } Y \text{ disjointly-fresh for } \text{Local}_{\mathbf{p}} \\
\alpha_2 = \llbracket \alpha_1[\text{left}(\text{tid})/\text{tid}, X/\text{Local}_{\mathbf{p}}]; *:\text{skip} \rrbracket \quad \beta_2 = \llbracket \beta_1[\text{right}(\text{tid})/\text{tid}, Y/\text{Local}_{\mathbf{p}}]; *:\text{skip} \rrbracket \\
\hline
(P, \mathcal{I}) \dashrightarrow (P[\text{Private}_{\mathbf{p}} \mapsto \text{Private}_{\mathbf{p}} \cup X \cup Y, \alpha_1 \parallel \beta_1 \mapsto (\alpha_2; \beta_2) \square (\beta_2; \alpha_2)], \mathcal{I})
\end{array}$$

We make three modifications to the actions before sequentially composing them. These modifications reflect the operational semantics of the parallel composition to the sequential composition. First, we replace tid with $\text{left}(\text{tid})$ and $\text{right}(\text{tid})$ in order to express that they refer to different thread ids. Second, we replace the local variables in actions to fresh local variables X and Y in order to express that α and β refer to local variables distinct from each other and from their parent procedure. Third, we append the statement $*:\text{skip}$ at the end of each action to suppress all exceptions it throws before executing the other.¹

The rules REDUCE-PARALLEL-LEFT and REDUCE-PARALLEL-RIGHT exploit left and right movers to eliminate the choice operator in EXPAND-PARALLEL.

REDUCE-PARALLEL-LEFT (EXPAND-PARALLEL, REDUCE-SEQUENTIAL-LEFT)

$$\begin{array}{c}
P, \mathcal{I} \vdash \alpha_1 : \mathbb{L} \quad \alpha_1, \beta_1 \in \text{Atoms}(\text{Body}_{\mathbf{p}}) \quad X \text{ and } Y \text{ disjointly-fresh for } \text{Local}_{\mathbf{p}} \\
\alpha_2 = \llbracket \alpha_1[\text{left}(\text{tid})/\text{tid}, X/\text{Local}_{\mathbf{p}}]; *:\text{skip} \rrbracket \quad \beta_2 = \llbracket \beta_1[\text{right}(\text{tid})/\text{tid}, Y/\text{Local}_{\mathbf{p}}]; *:\text{skip} \rrbracket \\
\hline
(P, \mathcal{I}) \dashrightarrow (P[\text{Private}_{\mathbf{p}} \mapsto \text{Private}_{\mathbf{p}} \cup X \cup Y, \alpha_1 \parallel \beta_1 \mapsto \alpha_2; \beta_2], \mathcal{I})
\end{array}$$

REDUCE-PARALLEL-RIGHT (EXPAND-PARALLEL, REDUCE-SEQUENTIAL-RIGHT)

$$\begin{array}{c}
P, \mathcal{I} \vdash \alpha_1 : \mathbb{R} \quad \alpha_1, \beta_1 \in \text{Atoms}(\text{Body}_{\mathbf{p}}) \quad X \text{ and } Y \text{ disjointly-fresh for } \text{Local}_{\mathbf{p}} \\
\alpha_2 = \llbracket \alpha_1[\text{left}(\text{tid})/\text{tid}, X/\text{Local}_{\mathbf{p}}]; *:\text{skip} \rrbracket \quad \beta_2 = \llbracket \beta_1[\text{right}(\text{tid})/\text{tid}, Y/\text{Local}_{\mathbf{p}}]; *:\text{skip} \rrbracket \\
\hline
(P, \mathcal{I}) \dashrightarrow (P[\text{Private}_{\mathbf{p}} \mapsto \text{Private}_{\mathbf{p}} \cup X \cup Y, \alpha_1 \parallel \beta_1 \mapsto \beta_2; \alpha_2], \mathcal{I})
\end{array}$$

REDUCE-PARALLEL-MOVER (EXPAND-PARALLEL, REDUCE-SEQUENTIAL-LEFT/RIGHT)

$$\begin{array}{c}
P, \mathcal{I} \vdash \alpha : m \quad P, \mathcal{I} \vdash \beta : m \quad m \in \{\mathbb{R}, \mathbb{L}\} \\
\alpha_1, \beta_1 \in \text{Atoms}(\text{Body}_{\mathbf{p}}) \quad X \text{ and } Y \text{ disjointly-fresh for } \text{Local}_{\mathbf{p}} \\
\alpha_2 = \llbracket \alpha_1[\text{left}(\text{tid})/\text{tid}, X/\text{Local}_{\mathbf{p}}]; *:\text{skip} \rrbracket \quad \beta_2 = \llbracket \beta_1[\text{right}(\text{tid})/\text{tid}, Y/\text{Local}_{\mathbf{p}}]; *:\text{skip} \rrbracket \\
\hline
(P, \mathcal{I}) \dashrightarrow (P[\text{Private}_{\mathbf{p}} \mapsto \text{Private}_{\mathbf{p}} \cup X \cup Y, \alpha_1 \parallel \beta_1 \mapsto \llbracket \alpha_2; \beta_2 \rrbracket], \mathcal{I})
\end{array}$$

After the application of the rule REDUCE-PARALLEL-MOVER, the mover type of the new action $\llbracket \alpha; \beta \rrbracket$ can be computed as described in Section 13.5.6.

¹Note that skip is a sugar for the action $\text{true} \triangleright \langle \text{true} \rangle_{\emptyset}$.

13.5.10 Reducing loops

Our proof system contains three reduction rules for the purpose of replacing a loop with a single gated action. For the following rules, we make use of two special statements:²

- The statement $\ast : \text{block}$ blocks if executed from a state where $\text{exc} \neq \text{skip}$, otherwise, has no side effects. Therefore, the action $\llbracket \alpha ; \ast : \text{block} \rrbracket$ expresses the executions of α ending only with $\text{exc} = \text{skip}$. We use this below to model non-final iterations of a loop.
- The statement $\text{skip} : \text{block}$ blocks if executed from a state where $\text{exc} = \text{skip}$, otherwise, has no side effects. Therefore, the action $\llbracket \alpha ; \text{skip} : \text{block} \rrbracket$ expresses the executions of α ending with an exception other than skip . We use this below to model the final iteration of a loop.

The rules below also reasons about the reflexivity of an action. Informally, an action is said to be reflexive if from every state s where α does not go wrong, its execution may end at the same state s .

Definition 16 (Reflexive actions):

An action α is reflexive, denoted $\text{Reflexive}(\alpha)$, if the following is valid:

$$\phi_\alpha \Rightarrow \tau_\alpha[\text{Var}_\alpha / \text{Var}'_\alpha] \quad (13.7)$$

The rule REDUCE-LOOP reduces a loop $\alpha^\odot$ whose body is a single action and is a left or right mover:

$$\begin{array}{c} \text{REDUCE-LOOP} \\ \gamma = \llbracket \alpha ; \ast : \text{block} \rrbracket \quad (P, \mathcal{I}) \vdash \gamma : m \quad m \in \{\mathbb{R}, \mathbb{L}\} \\ \beta \Leftrightarrow \mathcal{I} \quad \text{Reflexive}(\beta) \quad \mathcal{I} \vdash \llbracket \beta ; \gamma \rrbracket \preceq \beta \\ \hline (P, \mathcal{I}) \dashrightarrow (P[\alpha^\odot \mapsto \beta ; \llbracket \alpha ; \text{skip} : \text{block} \rrbracket], \mathcal{I}) \end{array}$$

The rule eliminates the loop by replacing it with the statement an action $\beta ; (\alpha ; \text{skip} : \text{block})$. Intuitively, β represents the non-final iterations of the loop, and $\llbracket \alpha ; \text{skip} : \text{block} \rrbracket$ represents the final iteration of the loop, which terminates with an exception other than skip .

²Note that block is a sugar for the action $\text{true} \triangleright \text{false}$.

In the rule the non-final iterations of the loop are represented by the action γ , which is equivalent to $\llbracket \alpha; *:\text{block} \rrbracket$. This action blocks when α does not terminate with $\text{exc} = \mathbf{skip}$. We check that β simulates the zero or more iterations of γ . Intuitively, ϕ_β is a predicate that is true at the beginning of every iteration and τ_β specifies a relation between the beginning of the loop and the end of any non-final iteration. The conditions needed to apply this rule are the following:

1. Each non-final iteration (γ) is a right or left mover with respect to the other actions in the program. This requires performing additional mover checks ($\mathcal{MC}$) between γ and actions in $\text{Atoms}(P)$.
2. The new action β preserves the current invariant (expressed by $\beta \trianglelefteq \mathcal{I}$).
3. The new action β simulates by zero or more non-final iterations (expressed by $\text{Reflexive}(\beta)$ and $\mathcal{I} \vdash \llbracket \beta; \gamma \rrbracket \preceq \beta$).

The rule REDUCE-LOOP-RIGHT given below does not require that the loop body be a mover, but that the new action β be a right mover. Differently from REDUCE-LOOP, the action β exists in the program and establishes an initial state before starting the loop. Thus, we do not require that β be reflexive.

$$\begin{array}{c}
 \text{REDUCE-LOOP-RIGHT} \\
 \hline
 \gamma = \llbracket \alpha; *:\text{block} \rrbracket \quad (P, \mathcal{I}) \vdash \beta : \mathbb{R} \quad \mathcal{I} \vdash \llbracket \beta; \gamma \rrbracket \preceq \beta \\
 \hline
 (P, \mathcal{I}) \dashrightarrow (P[(\beta; \alpha^\circ) \mapsto \beta; \llbracket \alpha; \mathbf{skip}:\text{block} \rrbracket], \mathcal{I})
 \end{array}$$

Reducing Nondeterministic Loops

In addition to the core rules for infinite loops, we also provide rules for the sugared statement S^* .³ These rules are simpler in that they do not require separating the final iteration, and they only reason that a given gated action is a specification for the loop. For soundness, the rules require that α not throw any exception other than $\mathbf{skip}$; in other words, the termination of the loop must be due to only a nondeterministic throwing of $\mathbf{break}$ (see Equation 12.5). This requirement is expressed in the rules as the validity check $\models (\tau_\alpha \Rightarrow (\text{exc} = \mathbf{skip}))$.

³These rules correspond to the reduction rules for non-deterministic loops in [99].

$$\begin{array}{c}
\text{REDUCE-NONDET-LOOP} \\
\vdash (\tau_\alpha \Rightarrow (exc' = \mathbf{skip})) \\
(P, \mathcal{I}) \vdash \alpha : m \quad m \in \{\mathbb{R}, \mathbb{L}\} \\
\beta \Leftarrow \mathcal{I} \quad \text{Reflexive}(\beta) \quad \mathcal{I} \vdash \llbracket \beta; \alpha \rrbracket \preceq \beta \\
\hline
(P, \mathcal{I}) \dashrightarrow (P[\alpha^* \mapsto \beta], \mathcal{I})
\end{array}$$

$$\begin{array}{c}
\text{REDUCE-NONDET-LOOP-RIGHT} \\
\vdash (\tau_\alpha \Rightarrow (exc' = \mathbf{skip})) \\
(P, \mathcal{I}) \vdash \beta : \mathbb{R} \quad \mathcal{I} \vdash \llbracket \beta; \alpha \rrbracket \preceq \beta \\
\hline
(P, \mathcal{I}) \dashrightarrow (P[\beta; \alpha^* \mapsto \beta], \mathcal{I})
\end{array}$$

$$\begin{array}{c}
\text{REDUCE-NONDET-LOOP-LEFT} \\
\vdash (\tau_\alpha \Rightarrow (exc' = \mathbf{skip})) \\
(P, \mathcal{I}) \vdash \beta : \mathbb{L} \quad \mathcal{I} \vdash \llbracket \alpha; \beta \rrbracket \preceq \beta \\
\hline
(P, \mathcal{I}) \dashrightarrow (P[\alpha^*; \beta \mapsto \beta], \mathcal{I})
\end{array}$$

13.6 Reducing Procedures

In this section, we show how our proof method deals with procedures. The rule `INLINE-CALL` below is particularly simple; it simply inlines $Body_{\mathbf{p}}$, the body of the callee $\mathbf{p}$, at the call site in the body of the caller $\mathbf{q}$. The rule uses the *Inline* operation defined by Equation 12.7 in Section 12.4.6, which is also used in Section 12.4.5 to define the semantics of procedure calls.

$$\begin{array}{c}
\text{INLINE-CALL} \\
\overrightarrow{out} := \mathbf{p}(\overrightarrow{in}) \text{ in } Body_{\mathbf{q}} \quad \langle S, X \rangle = \text{Inline}(\overrightarrow{out} := \mathbf{p}(\overrightarrow{in})) \\
\hline
(P, \mathcal{I}) \dashrightarrow (P[\text{Private}_{\mathbf{q}} \mapsto \text{Private}_{\mathbf{q}} \cup X, \overrightarrow{out} := \mathbf{p}(\overrightarrow{in}) \mapsto S], \mathcal{I})
\end{array}$$

All the other rules discussed previously can be used to transform the body of a procedure iteratively making it smaller and simpler. In fact, in the limit, the procedure body can be transformed to a single atomic action. Once the procedure body has been simplified enough, it can be inlined at call sites without any significant increase in the program size.

13.6.1 Recursive Procedures

The rule `INLINE-CALL` is inadequate if there is recursion in the program. For example, when a recursive procedure is inlined, there will still remain calls to the same procedure. In this case, we use the proof rule `REDUCE-PROCEDURE`, which takes a set $\mathcal{M}$ of procedures and gives a mechanism whereby the body of each method in $\mathcal{M}$ is replaced by a gated action. While

we devised the rule REDUCE-PROCEDURE for recursive procedures, the rule can be applied to any set of non-recursive procedures in a sound manner.

REDUCE-PROCEDURE

$\mathcal{M} = \{\mathbf{p}_1, \dots, \mathbf{p}_n\}$ is closed under call

There exists an annotation function $\mathcal{A}$ where for each $\mathbf{p}_i \in \mathcal{M}$: $\mathcal{A}(\mathbf{p}_i) = (m_i, \alpha_i)$

$$\frac{\forall \mathbf{p}_i \in \mathcal{M} : \quad \alpha_i \trianglelefteq \mathcal{I} \quad (P, \mathcal{I}, \mathcal{A}) \vdash \text{Body}_{\mathbf{p}_i} : m_i \quad (P, \mathcal{I}, \mathcal{A}) \vdash \{\alpha_i\} \mathbf{p}_i \quad \text{Finalizable}(\mathbf{p}_i, \phi_{\alpha_i})}{(P, \mathcal{I}) \dashv\dashv (P[\text{Body}_{\mathbf{p}_1} \mapsto \alpha_1, \dots, \text{Body}_{\mathbf{p}_n} \mapsto \alpha_n], \mathcal{I})}$$

We require that $\mathcal{M}$ be closed under call (see Definition 3). To break circular dependencies, REDUCE-PROCEDURE requires (from the user) an annotation function $\mathcal{A}$ that provides a specification for each procedure $\mathbf{p}_i$ in $\mathcal{M}$ and for each loop occurring in a $\mathbf{p}_i$ as explained later in this section.

The rule REDUCE-PROCEDURE requires that the body $\text{Body}_{\mathbf{p}_i}$ of every procedure $\mathbf{p}_i \in \mathcal{M}$ be an *atomic body*. $\text{Body}_{\mathbf{p}_i}$ is an atomic body if $(P, \mathcal{I}, \mathcal{A}) \vdash \text{Body}_{\mathbf{p}_i} : \mathbb{A}$ can be derived using the judgment rules in Figure 13.2. Notice that in the judgment, the proof state is augmented with $\mathcal{A}$, since it provides the mover types of procedure calls. The rule CALL-MOVER allows us to assign the mover type specified by $\mathcal{A}(\text{proc}_i)$ to a procedure call to $\mathbf{p}_i \in \mathcal{M}$. The mover type of the actions are computed using mover checks as described in Section 13.5.4. The mover type of compound statements are computed using the judgment rules STATEMENT-MOVER, LOOP-MOVER, and LABELED-MOVER.

The function $\mathcal{A}$ is defined as follows:

- For each $\mathbf{p}_i \in \mathcal{M}$, $\mathcal{A}(\mathbf{p}_i)$ is a pair $(m_i, \phi_i \triangleright \tau_i)$, where $m_i \in \{\mathbb{A}, \mathbb{L}, \mathbb{R}, \mathbb{B}\}$ specifies the mover type of the body of $\mathbf{p}_i$, and the gated action $\phi_i \triangleright \tau_i$ gives a specification for $\mathbf{p}_i$: ϕ_i asserts a pre-condition for the call point of the procedure and τ_i specifies a post-condition, a relationship between the call and return points of the procedure. If the rule REDUCE-PROCEDURE succeeds, then it is sound to replace the body of each procedure $\mathbf{p}_i \in \mathcal{M}$ with $\phi_i \triangleright \tau_i$. For the soundness of the REDUCE-PROCEDURE rule, we require that every execution of $\mathbf{p}_i$ from ϕ_i generates a finite number of transitions. In the rule, we express this requirement using the predicate $\text{Finalizable}(\mathbf{p}_i, \phi_{\alpha_i})$, as formally defined below.

Definition 17 (Finalizable procedures):

A procedure $\mathbf{p}$ in a program P is finalizable from ϕ , denoted $\text{Finalizable}(\mathbf{p}, \phi)$, if for every state s_1 such that $\theta_{s_1}(t) = (\vec{\ell} : \neg \mathbf{p}(\vec{E}_{\vec{\ell}}))$ for some thread t and for every run $s_1 \xrightarrow{\vec{1}} s_2$ of P , $\vec{1}|_t$ contains a finite number of transition labels. ■

The condition $\text{Finalizable}(\mathbf{p}_i, \phi_{\alpha_i})$ is used in the proof of Lemma 4. We currently assume that the programmer has ensured that the requirement $\text{Finalizable}(\mathbf{p}_i, \phi_i)$ is met and do not provide a mechanical check for it in our tool. This condition is similar to a termination requirement, in fact, termination is a sufficient condition for being finalizable. Thus this check can be mechanized using tool like TERMINATOR [100]; we leave this study to future work.

- For each loop $S^{\odot}$ in $\text{Body}_{\mathbf{p}_i}$ for $\mathbf{p}_i \in \mathcal{M}$, $\mathcal{A}(S^{\odot})$ is a gated action $\phi \triangleright \tau$. Informally, ϕ is a loop invariant for $S^{\odot}$, and τ specifies the computations performed by any number of (possibly zero) *non-final* iterations of the loop.

Since the specifications above are written only for procedures with atomic bodies and for loop in these procedures (as required by REDUCE-PROCEDURE), the problem of deriving these specifications is the same as providing loop invariants and procedure specifications for sequential code. In the following, we will describe how to check that a procedure $\mathbf{p}_i$ satisfies the annotation $\mathcal{A}(\mathbf{p}_i)$ and a loop $S^{\odot}$ satisfies the annotation $\mathcal{A}(S^{\odot})$. Then, it will be clear that these checks correspond to the standard reasoning about procedure pre-/post-conditions and loop invariants in the verification of sequential programs [38, 41].

Given an annotation function $\mathcal{A}$, REDUCE-PROCEDURE checks each procedure $\mathbf{p}_i$ in three phases:

1. It checks that the specification of $\mathbf{p}_i$ indicates an action that preserves the invariant. In the rule, this is described by the condition $\alpha_i \Leftrightarrow \mathcal{I}$.
2. It checks that the body of $\mathbf{p}_i$ in P conforms to its mover type provided by $\mathcal{A}$ in the augmented proof state $(P, \mathcal{I}, \mathcal{A})$. In the statement of the rule, this is expressed with the condition $P, \mathcal{I}, \mathcal{A} \vdash \text{Body}_{\mathbf{p}_i} : m_i$. The rules in Figure 13.2 are used in this phase; for brevity, we have elided the implicit argument $\mathcal{A}$ to all the judgments.

$\llbracket \cdot \rrbracket : Stmt \rightarrow Atomic$	
$\llbracket l:S \rrbracket$	$\triangleq l: \llbracket S \rrbracket$
$\llbracket * : S \rrbracket$	$\triangleq * : \llbracket S \rrbracket$
$\llbracket S_1; S_2 \rrbracket$	$\triangleq \llbracket \llbracket S_1 \rrbracket; \llbracket S_2 \rrbracket \rrbracket$
$\llbracket S_1 \square S_2 \rrbracket$	$\triangleq \llbracket \llbracket S_1 \rrbracket \square \llbracket S_2 \rrbracket \rrbracket$
$\llbracket S_1 \parallel S_2 \rrbracket$	$\triangleq \llbracket (S_1[left(tid)/tid, X/Local_{\mathbf{p}}]; * : skip); (S_2[right(tid)/tid, Y/Local_{\mathbf{p}}]; * : skip) \rrbracket$ where X and Y are fresh for $Local \cup \{exc, tid\}$
$\llbracket S^\odot \rrbracket_{\mathcal{A}}$	$\triangleq \llbracket \alpha; S; \mathbf{skip} : \mathbf{block} \rrbracket_{\mathcal{A}}$ where $\mathcal{A}(S^\odot) = \alpha$
$\llbracket \overrightarrow{out} := \mathbf{p}(\overrightarrow{in}) \rrbracket_{\mathcal{A}}$	$\triangleq \alpha$ where $\mathcal{A}(\mathbf{p}) = (m, \alpha)$

Figure 13.3: Composing a gated action from a compound statement.

3. It verifies that the “atomic” body of $\mathbf{p}_i$ implements the specification described by the action α_i . This is described by the condition $(P, \mathcal{I}, \mathcal{A}) \vdash \{\alpha_i\} \mathbf{p}_i$. Formally, $(P, \mathcal{I}, \mathcal{A}) \vdash \{\alpha_i\} \mathbf{p}_i$ if the following holds:

$$\mathcal{I} \vdash \llbracket Body_{\mathbf{p}_i} \rrbracket_{\mathcal{A}} \preceq \alpha_i \quad (13.8)$$

The equation (13.8) uses the $\llbracket \cdot \rrbracket_{\mathcal{A}}$ operator defined in Figure 13.3 to check that $Body_{\mathbf{p}_i}$, the atomic body of the procedure $\mathbf{p}_i$, is abstracted by the given action in $\mathcal{A}$. Figure 13.3 complements the definition of $\llbracket S \rrbracket$ in Figure 13.1 for arbitrary statements. $\llbracket S \rrbracket_{\mathcal{A}}$ takes a statement possibly containing loops and procedure calls and produces a single atomic action representing the atomic computation performed by S . The computation of $\llbracket S \rrbracket_{\mathcal{A}}$ uses the specifications in $\mathcal{A}$ when a procedure call or a loops is seen. Since the body of $\mathbf{p}$, $Body_{\mathbf{p}}$, is shown to be at least an atomic non-mover in the previous phase, we can use the action $\llbracket Body_{\mathbf{p}} \rrbracket$ to model the atomic execution of the body. While computing $\llbracket Body_{\mathbf{p}} \rrbracket$, we use the action α_j given by $\mathcal{A}(\mathbf{p}_j)$ when we see a call to $\mathbf{p}_j$ (see the use of $\mathcal{A}$ in Figure 13.3). Notice also that, in Figure 13.3, our handling of the parallel composition operator is similar to the one for the sequential composition. Since $S_1 \parallel S_2$ is part of a statement that has been proved to be atomic, $S_1 \parallel S_2$ is atomic as well. Thus, every behavior of $S_1 \parallel S_2$ is equivalent to a behavior of the sequential execution of the statement after renaming tid and local variables as done in

Figure 13.3. The correctness of the VC generation for parallel composition crucially depends on this observation.

In addition, $(P, \mathcal{I}, \mathcal{A}) \vdash \{\alpha_i\} \mathbf{p}_i$ implicitly dictates that every loop $S^\odot$ in the body of $\mathbf{p}_i$ implements its specification $\mathcal{A}(S^\odot) = \alpha$ by checking the following proof obligations:

1. $\alpha \Leftrightarrow \mathcal{I}$
2. $\text{Reflexive}(\alpha)$
3. $\mathcal{I} \vdash \llbracket \alpha; \gamma \rrbracket \preceq \alpha$ where $\gamma = \llbracket S; * : \text{block} \rrbracket$

Condition 1 ensures that α preserves the invariant. Conditions 2 and 3 verify that α simulates zero or more non-final iterations of the loop. Note that while computing the atomic gate from this loop in Figure 13.3, the statement $\llbracket S; \text{skip} : \text{block} \rrbracket$ representing the final iteration is sequentially composed with the action α , so we only have to ensure that α simulates the non-final iterations of the loop.

Notice the resemblance of the conditions above to the ones in the reduction rule REDUCE-LOOP in Section 13.5.10. In fact, the loops in the procedures $\mathcal{M}$ can be handled differently following the other rules in Section 13.5.10. For example, if the loop is in the form of a nondeterministic loop (S^*) and S not throw any non-skip exception, then we can follow the rule REDUCE-NONDET-LOOP and check the specification $\mathcal{A}(S^*)$ as follows (using simpler conditions than above):

1. $\alpha \Leftrightarrow \mathcal{I}$
2. $\text{Reflexive}(\alpha)$
3. $\mathcal{I} \vdash \llbracket \alpha; S \rrbracket \preceq \alpha$

Moreover, instead of using $\llbracket S^\odot \rrbracket_{\mathcal{A}}$ in Figure 13.3, which combines α with S , we can simply use $\llbracket S^* \rrbracket$, as defined below:

$$\llbracket S^* \rrbracket_{\mathcal{A}} \triangleq \alpha \quad \text{where } \mathcal{A}(S^\odot) = \alpha \quad (13.9)$$

Each of the conditions above precisely formulates of a verification condition (VC) in FOL (see Section 12.1 for the expression language we use). In our implementation, the

check is performed by proving the validity of the corresponding VC using an automatic theorem prover. Due to the formulation of $\llbracket S \rrbracket_{\mathcal{A}}$ in Figure 13.1 and 13.3, $(P, \mathcal{I}, \mathcal{A}) \vdash \{\alpha\} \mathbf{p}_i$ corresponds to the VC generation technique for sequential programs in the ESC/Java [38] and Boogie verifiers [41]. The rule REDUCE-PROCEDURE replaces the body of every procedure $\mathbf{p}_i \in \mathcal{M}$ with α_i if all the checks succeed.

13.7 Soundness of the Proof System

In the following, we first introduce an abstraction relation between programs, and then state that each application of a rule in the proof system ensures that this abstraction relation exists between the programs before and after the rule is applied.

Definition 18 (Program abstraction relation):

Let P_1 and P_2 be two programs and $\mathcal{I}$ be a state predicate. Let $V = \text{Global}_{P_1} \cap \text{Global}_{P_2}$, $X_1 = \text{fv}(\mathcal{I}) \setminus \text{Global}_{P_1}$, and $X_2 = \text{fv}(\mathcal{I}) \setminus \text{Global}_{P_2}$. P_2 abstracts P_1 from $\mathcal{I}$, denoted $\mathcal{I} \vdash P_1 \preceq P_2$, if the following hold:

$$1. \text{Bad}|_V(P_1, \exists X_1. \mathcal{I}) \subseteq \text{Bad}|_V(P_2, \exists X_2. \mathcal{I})$$

$$2. \forall (\sigma_1, \sigma_2) \in \text{Good}|_V(P_1, \exists X_1. \mathcal{I}) :$$

$$(a) (\sigma_1, \sigma_2) \in \text{Good}|_V(P_2, \exists X_2. \mathcal{I}), \text{ or}$$

$$(b) \sigma_1 \in \text{Bad}|_V(P_2, \exists X_2. \mathcal{I})$$

■

Intuitively, condition 1 specifies that for every violation $s \rightarrow \text{wrong}$ of P_1 such that $s \models \exists X_1. \mathcal{I}$, P_2 has a violation $s' \rightarrow \text{wrong}$ such that $s' \models \exists X_2. \mathcal{I}$ and for *common global variables* V of both programs $\sigma_s|_V = \sigma_{s'}|_V$. Condition 2 specifies that for every successful run $s_1 \rightarrow s_2$ of P_1 such that $s_1 \models \exists X_1. \mathcal{I}$, there exists either (a) a successful run $s'_1 \rightarrow s'_2$ of P_2 such that $s'_1 \models \exists X_2. \mathcal{I}$ and $\sigma_{s_1}|_V = \sigma_{s'_1}|_V$ and $\sigma_{s_2}|_V = \sigma_{s'_2}|_V$, or (b) a violation $s'_1 \rightarrow \text{wrong}$ of P_2 such that $\sigma_{s_1}|_V = \sigma_{s'_1}|_V$. The set V is empty if P_1 and P_2 have no common variables, and the set X becomes empty when P_1 and P_2 have the same set of variables.

Notice that, $\text{Bad}|_V(P_1, \exists X_1. \mathcal{I})$ is never empty, unless $\text{Bad}(P_1, \exists X_1. \mathcal{I})$ is empty. This is true even though P_1 and P_2 have no common global variables ($V = \emptyset$): if $\text{Bad}(P_1, \exists X_1. \mathcal{I})$

is nonempty, then $Bad|_V(P_1, \exists X_1. \mathcal{I})$ is a singleton set containing an empty store. Similarly, $Good|_V(P_1, \exists X_1. \mathcal{I})$ is never empty, unless $Good(P_1, \exists X_1. \mathcal{I})$ is empty.

The relation $\preceq$ is used to connect two programs that appear in proof states separated by one or more proof steps. Our soundness theorem (Theorem 5) states that after a sequence of proof steps $(P_1, \mathcal{I}_1) \dashrightarrow \dots \dashrightarrow (P_n, \mathcal{I}_n)$, P_n abstracts P_1 . Before giving our soundness theorem, we formulate several helper lemmas to be used while proving the theorem.

Lemma 3 (Eliminating right movers from a violation):

Let $(P, \mathcal{I})$ be a proof state and $(P, \mathcal{I}) \vdash \alpha : \mathbb{R}$. Let $\rho = s_1 \xrightarrow{\vec{\Gamma}_1} s_i \xrightarrow{(t, \alpha)} s_{i+1} \xrightarrow{\vec{\Gamma}_2} \text{wrong}$ be a violation of P from $\mathcal{I}$, such that $\neg \text{Error}(s_{i+1})$, and for any $(u, \beta) \in \vec{\Gamma}_2$, $u \neq t$. Then, one of the following violations exists for some (strictly shorter) subsequence $\vec{\Gamma}'_2$ of $\vec{\Gamma}_2$:

1. $\rho' = s_1 \xrightarrow{\vec{\Gamma}_1} s_i \xrightarrow{\vec{\Gamma}'_2} \text{wrong}$
2. $\rho' = s_1 \xrightarrow{\vec{\Gamma}_1} s_i \xrightarrow{\vec{\Gamma}'_2(t, \alpha)} \text{wrong}$ ■

PROOF.

(1)1. Induction on the length n of $\vec{\Gamma}_2$.

(1)2. *Base case:* $n = 1$. Then the run is of the form $\rho = s_1 \xrightarrow{\vec{\Gamma}_1} s_i \xrightarrow{(t, \alpha)} s_{i+1} \xrightarrow{(u, \beta)} \text{wrong}$.

Since α is a right mover, then $\mathcal{I} \vdash \alpha \rightleftharpoons \beta$. Thus, we have $\rho' = s_1 \xrightarrow{\vec{\Gamma}_1} s_i \xrightarrow{(u, \beta)} \text{wrong}$, and the claim holds.

(1)3. *Inductive case:* $n > 1$. Assume that the claim holds for $\vec{\Gamma}_2$ of length $< n$.

(2)1. Assume a run that is of the form $\rho = s_1 \xrightarrow{\vec{\Gamma}_1} s_i \xrightarrow{(t, \alpha)(u, \beta)} s_{i+2} \xrightarrow{\vec{\Gamma}_3} \text{wrong}$ such that

$$\vec{\Gamma}_2 = (u, \beta) \vec{\Gamma}_3.$$

(2)2. By $(P, \mathcal{I}) \vdash \alpha : \mathbb{R}$, either (1) $\rho' = s_1 \xrightarrow{\vec{\Gamma}_1} s_i \xrightarrow{(u, \beta)(t, \alpha)} s_{i+2} \xrightarrow{\vec{\Gamma}_3} \text{wrong}$ or (2) $\rho' = s_1 \xrightarrow{\vec{\Gamma}_1} s_i \xrightarrow{(u, \beta)} \text{wrong}$ or (3) $\rho' = s_1 \xrightarrow{\vec{\Gamma}_1} s_i \xrightarrow{(u, \beta)(t, \alpha)} \text{wrong}$.

(3)1. If (1) exists, then by the inductive hypothesis for $\vec{\Gamma}_3$, there exists either $\rho'' = s_1 \xrightarrow{\vec{\Gamma}_1} s_i \xrightarrow{\vec{\Gamma}'_3} \text{wrong}$ or $\rho'' = s_1 \xrightarrow{\vec{\Gamma}_1} s_i \xrightarrow{\vec{\Gamma}'_3(t, \alpha)} \text{wrong}$ such that $\vec{\Gamma}'_3$ is a (strictly shorter) subsequence of $(u, \beta) \vec{\Gamma}_3$, and thus, of $\vec{\Gamma}_2$.

(3)2. If (2) or (3) happens, then the claim holds trivially, since (u, β) is a (strictly shorter) subsequence of $(u, \beta) \vec{\Gamma}_3$ and $\vec{\Gamma}_2$.

□

Lemma 4 (Moving right movers to the end of a run):

Let $(P, \mathcal{I})$ be a proof state and $(P, \mathcal{I}) \vdash \alpha : \mathbb{R}$. Let $\rho = s_1 \xrightarrow{\vec{I}_1} s_i \xrightarrow{(t, \alpha)} s_{i+1} \xrightarrow{\vec{I}_2} s_2$ be a run of P from $\mathcal{I}$ such that for any $(u, \beta) \in \vec{I}_2$, $u \neq t$. Then, one of the following exists:

1. $\rho' = s_1 \xrightarrow{\vec{I}_1} s_i \xrightarrow{\vec{I}_2(t, \alpha)} s_2$
2. $\rho' = s_1 \xrightarrow{\vec{I}_1} s_i \xrightarrow{\vec{I}_2(t, \alpha)} \text{wrong}$
3. $\rho' = s_1 \xrightarrow{\vec{I}_1} s_i \xrightarrow{\vec{I}'_2} \text{wrong}$ for a strictly shorter subsequence of $\vec{I}_2$
4. $\rho' = s_1 \xrightarrow{\vec{I}_1} s_i \xrightarrow{\vec{I}'_2(t, \alpha)} \text{wrong}$ for a strictly shorter subsequence of $\vec{I}_2$ ■

PROOF.

(1)1. Induction on the length n of $\vec{I}_2$.

(1)2. Base case: $n = 1$. Then the run is of the form $\rho = s_1 \xrightarrow{\vec{I}_1} s_i \xrightarrow{(t, \alpha)} s_{i+1} \xrightarrow{(u, \beta)} s_{i+2} \xrightarrow{\vec{I}_3} s_2$ such that $\vec{I}_2 = (u, \beta) \vec{I}_3$. Then there exists either $\rho' = s_1 \xrightarrow{\vec{I}_1} s_i \xrightarrow{(u, \beta)} s_{i+2} \xrightarrow{\vec{I}_3} s_2$ or $\rho' = s_1 \xrightarrow{\vec{I}_1} s_i \xrightarrow{(u, \beta)} \text{wrong}$. The claim holds in both cases.

(1)3. Inductive case: $n > 1$. Assume that the claim holds for $\vec{I}_2$ of length $< n$.

(2)1. In this case, the run is of the form $\rho = s_1 \xrightarrow{\vec{I}_1} s_i \xrightarrow{(t, \alpha)(u, \beta)} s_{i+2} \xrightarrow{\vec{I}_3} s_2$.

(2)2. By $(P, \mathcal{I}) \vdash \alpha : \mathbb{R}$, either (1) $\rho' = s_1 \xrightarrow{\vec{I}_1} s_i \xrightarrow{(u, \beta)(t, \alpha)} s_{i+2} \xrightarrow{\vec{I}_3} s_2$ or (2) $\rho' = s_1 \xrightarrow{\vec{I}_1} s_i \xrightarrow{(u, \beta)} \text{wrong}$ or (3) $\rho' = s_1 \xrightarrow{\vec{I}_1} s_i \xrightarrow{(u, \beta)(t, \alpha)} \text{wrong}$.

(3)1. If (1) exists, then by the inductive hypothesis for $\vec{I}_3$, there exists a run of the program that makes one of the conditions 1–4 hold.

(3)2. If (2) or (3) happens, then the claim holds trivially, since (u, β) is a (strictly shorter) subsequence of $(u, \beta) \vec{I}_3$, and thus, of $\vec{I}_2$.

□

Lemma 5 (Gathering atomic statements):

Let $(P, \mathcal{I})$ be a proof state and $\mathcal{S} = \{S_1, \dots, S_2\}$ be a set of atomic statements in P ($\forall S_i \in \mathcal{S}$, $(P, \mathcal{I}) \vdash S_i : \mathbb{A}$). For every run $\rho = s_1 \xrightarrow{\vec{I}} s_2$ of the program, there exists either $\rho' = s_1 \xrightarrow{\vec{I}'} s_2$ or $\rho' = s_1 \xrightarrow{\vec{I}'} \text{wrong}$ such that for every statement $S_i \in \mathcal{S}$, transitions that belong to an instance of S_i appear sequentially in ρ' . ■

PROOF.

- (1)1. Each instance of an atomic statement $S_i \in \mathcal{S}$ in a run ρ is represented by a sequence of transitions with labels $(t, \alpha_1)(t, \alpha_2)\dots(t, \alpha_n)$ such that one of the followings hold:
1. For all $i \in [1, n]$, $(P, \mathcal{I}) \vdash \alpha_i : \mathbb{R}$
 2. For all $i \in [1, n]$, $(P, \mathcal{I}) \vdash \alpha_i : \mathbb{L}$
 3. There exists some $k \in [1, n]$ such that for all $j \in [1, k - 1]$, $(P, \mathcal{I}) \vdash \alpha_j : \mathbb{R}$, $(P, \mathcal{I}) \vdash \alpha_k : \mathbb{A}$ ⁴, and for all $j \in [k + 1, n]$, $(P, \mathcal{I}) \vdash \alpha_j : \mathbb{L}$.
- In the execution, the transitions $(t, \alpha_1)(t, \alpha_2)\dots(t, \alpha_n)$ are possibly interleaved by transitions of other threads.
- (1)2. We assign each instance of S_i a mover type, either $\mathbb{R}$ or $\mathbb{L}$, and a commit point, a transition, as follows: If item 1 above is true, then the mover type of the instance is $\mathbb{R}$ and the commit point is (t, α_n) . If item 2 is true, then the mover type of the instance is $\mathbb{L}$ and the commit point is (t, α_1) . If item 3 is true, then the mover type of the instance is $\mathbb{L}$ and the commit point is (t, α_k) . We say that an instance is an $\mathbb{R}$ -instance, if it is given the mover type, $\mathbb{R}$, otherwise it is an $\mathbb{L}$ -instance.
- (1)3. Let $Commit_\rho : (Tid \times Atomic) \rightarrow \mathcal{N}$ be a (partial) function that maps each transition label $\mathbf{l}$ that belongs to an instance of a statement in $\mathcal{S}$ and returns the index of the commit point of instance that $\mathbf{l}$ belongs to.
- (1)4. Given a run ρ , let $Sorted_\rho : (Tid \times Atomic) \rightarrow \mathcal{N}$ be a function that defines a permutation of the transition labels in ρ as follows: For two transitions $\rho(i) = (t, \alpha)$ and $\rho(j) = (u, \beta)$ for $i < j$, $Sorted_\rho(t, \alpha) < Sorted_\rho(u, \beta)$ if $t = u$, or both transitions are commit transitions, or neither transitions belong to an instance of $\mathcal{S}$. In $Sorted_\rho$ -permuted run, all non-commit transitions of the instances appear sequentially around the commit transitions in the same order determined by ρ .
- (1)5. Define a distance function $Dist(\rho)$ that takes a run ρ and returns the number of pairs (i, j) such that $\rho(i) = (t, \alpha)$, $\rho(j) = (u, \beta)$, and $i < j$, but $Sorted_\rho(t, \alpha) > Sorted_\rho(u, \beta)$.
- (1)6. Take a run $\rho = s_1 \xrightarrow{\vec{1}} s_2$ of the program P and apply one of the following iteratively to obtain new runs of the program. At each iteration, we obtain a run ρ' from ρ such that either $Dist(\rho') < Dist(\rho)$, or ρ' is a violation that goes wrong earlier than ρ .

⁴ $(P, \mathcal{I}) \vdash \alpha_k : \mathbb{A}$ states that we do not have any restriction on the mover type of α_k ; it can also be a right and/or left mover.

- (2)1. Select a pair (i, j) such that $\rho(i) = (t, \alpha)$, $\rho(i + 1) = (u, \beta)$, and $i < j$, but $\text{Sorted}_\rho(t, \alpha) > \text{Sorted}_\rho(u, \beta)$.
- (2)2. Either $(P, \mathcal{I}) \vdash \alpha : \mathbb{R}$ or $(P, \mathcal{I}) \vdash \alpha : \mathbb{L}$.
- (2)3. Consider the run $\rho = s_1 \xrightarrow{\vec{I}_1} s_i \xrightarrow{(t, \alpha)(u, \beta)} s_{i+2} \xrightarrow{\vec{I}_2} s_2$. By the definition of right and left movers, there exists either (1) $\rho' = s_1 \xrightarrow{\vec{I}_1} s_i \xrightarrow{(u, \beta)(t, \alpha)} s_{i+2} \xrightarrow{\vec{I}_2} s_2$, or (2) $\rho' = s_1 \xrightarrow{\vec{I}_1} s_i \xrightarrow{(u, \beta)(t, \alpha)} \text{wrong}$, or (3) $\rho' = s_1 \xrightarrow{\vec{I}_1} s_i \xrightarrow{(u, \beta)} \text{wrong}$.
- (2)4. In (1) and (2), $\text{Dist}(\rho') < \text{Dist}(\rho)$. In (3), we obtain a run that goes wrong earlier than ρ .
- (1)7. The above iterations terminate since at the end of each iteration either $\text{Dist}(\rho') < \text{Dist}(\rho)$ holds, or we obtain a violation with fewer non-failing transitions. Let ρ' be the run after the iterations terminate. Either $\text{Dist}(\rho') = 0$ or ρ' goes wrong in zero transitions. In both cases, the claim holds.

□

The following theorem states that the application of each core proof rule ensures the abstraction relation $\preceq$ between the programs before and after the application of the rule. Therefore, failing runs of the original program are preserved after an application of any QED proof rule. In addition, from every state satisfying the invariant, the resulting programs *simulate* the succeeding runs of the original program as long as they do not go wrong from that state.

Theorem 4 (Preservation of $\preceq$ by the QED proof rules):

Let $(P_1, \mathcal{I}_1) \dashrightarrow (P_2, \mathcal{I}_2)$ be a legal proof step. Then $\mathcal{I}_2 \vdash P_1 \preceq P_2$ holds. ■

PROOF.

- (1)1. *Proof sketch:* Here we will prove a stronger claim that relates the runs of the programs. We show that for every run $\rho = s_1 \xrightarrow{\vec{I}} s_2$ of the program P_1 , there exists a run $\rho' = s'_1 \xrightarrow{\vec{I}'} s'_2$ of the program P_2 such that s_1 and s'_1 agree on the common variables of the programs, and either s_2 and s'_2 also agree on the common variables of the programs or ρ' is a violation, i.e., s'_2 is an error state. We call the run ρ' a *witness* for ρ . We case split on the proof rule applied during the step. In the following proof, we will ignore the θ component of the program state, thus two states are treated equivalent although the code maps are different.

ASSUME: $(P_1, \mathcal{I}_1) \dashrightarrow (P_2, \mathcal{I}_2)$

ASSUME: $V = \text{Global}_{P_1} \cap \text{Global}_{P_2}$

ASSUME: $X = (\text{Global}_{P_1} \cup \text{Global}_{P_2}) \setminus V$

ASSUME: A state s_1 of P_1 such that $s_1 \models \exists X.\mathcal{I}_2$ and a run $s_1 \xrightarrow{\vec{I}} s_2$ of P_1 .

PROVE: There exists a run ρ' of P_2 that is either in the form $\rho' = s'_1 \xrightarrow{\vec{I}'} s'_2$ or $\rho' = s'_1 \xrightarrow{\vec{I}'} \text{wrong}$ such that $s'_1 \models \exists X.\mathcal{I}_2$, $\sigma_{s_1}|_V = \sigma_{s'_1}|_V$, and $\sigma_{s_2}|_V = \sigma_{s'_2}|_V$.

CASE: INVARIANT

$\langle 3 \rangle 1$. In this case, $X = \emptyset$ and $P_1 = P_2$.

$\langle 3 \rangle 2$. We can take $\rho = \rho'$ since for each $i \in [1, |\rho|]$ $\rho_i \models \mathcal{I}_1$ implies $\rho_i \models \mathcal{I}_2$, by the antecedent $\models \mathcal{I}_1 \Rightarrow \mathcal{I}_2$ of the rule.

$\langle 3 \rangle 3$. Q.E.D.

CASE: ADD-INVARIANT

$\langle 3 \rangle 1$. This rule is a special case of INVARIANT, since $\mathcal{I}_2 = (\mathcal{I}_1 \wedge \psi)$ and $\models \mathcal{I}_1 \Rightarrow \mathcal{I}_2$. The claim holds trivially.

$\langle 3 \rangle 2$. Q.E.D.

CASE: ABSTRACT

$\langle 3 \rangle 1$. In this case, $X = \emptyset$ and $\mathcal{I}_1 = \mathcal{I}_2$, let us denote the invariant by $\mathcal{I}$.

$\langle 3 \rangle 2$. We prove the claim by induction on the length n of the run ρ .

$\langle 3 \rangle 3$. *Base case:* $n = 0$ so there is no transition. The claim holds trivially, by taking $\rho = \rho'$.

$\langle 3 \rangle 4$. *Inductive case:* $n > 0$. Assume that the claim holds for all runs of length $|\rho| < n$ and consider a run ρ of length n : $\rho = s_1 \xrightarrow{\vec{I}} s_{n-1} \xrightarrow{(t,\gamma)} s_n$.

$\langle 3 \rangle 5$. By the inductive hypothesis, there exists a run $s_1 \xrightarrow{\vec{I}'} s'_{n-1}$.

$\langle 3 \rangle 6$. If $\gamma \neq \alpha$, then take the witness run as $\rho' = s_1 \xrightarrow{\vec{I}'} s'_{n-1} \xrightarrow{(t,\gamma)} s_n$.

$\langle 3 \rangle 7$. If $\gamma = \alpha$, then the judgment $\mathcal{I} \vdash \alpha \preceq \beta$ implies that either $\rho' = s_1 \xrightarrow{\vec{I}'} s'_{n-1} \xrightarrow{(t,\beta)} s'_n$ or $\rho' = s_1 \xrightarrow{\vec{I}'} s'_{n-1} \xrightarrow{(t,\beta)} \text{wrong}$ is a witness for ρ .

$\langle 3 \rangle 8$. Q.E.D.

CASE: ADD-GLOBAL-VARIABLE

$\langle 3 \rangle 1$. In this case, $X = a$, $V = \text{Global}_{P_1}$ ($\text{Global}_{P_2} = V \cup \{a\}$) and $\mathcal{I}_1 = \mathcal{I}_2$, let us denote the invariant by $\mathcal{I}$.

$\langle 3 \rangle 2$. We prove the claim by induction on the length n of the run ρ .

⟨3⟩3. *Base case:* $n = 0$ so there is no transition. The claim holds trivially, by taking $\rho = \rho'$.

⟨3⟩4. *Inductive case:* $n > 0$. Assume that the claim holds for all runs of length $|\rho| < n$ and consider a run ρ of length n : $\rho = s_1 \xrightarrow{\vec{1}} s_{n-1} \xrightarrow{(t,\alpha)} s_n$.

⟨3⟩5. By the inductive hypothesis, there exists a run $s'_1 \xrightarrow{\vec{1}'} s'_{n-1}$ such that $\sigma_{s_1}|_V = \sigma_{s'_1}|_V$, and $\sigma_{s_{n-1}}|_V = \sigma_{s'_{n-1}}|_V$ or $\text{Error}(s'_{n-1})$.

PROVE: There exists a witness run $\rho' = s'_1 \xrightarrow{\vec{1}'} s'_{n-1} \xrightarrow{(t,\beta)} s'_n$ such that $\sigma_{s_n}|_V = \sigma_{s'_n}|_V$ or $\text{Error}(s'_n)$

⟨3⟩6. By the antecedent we know that $\mathcal{I} \vdash \alpha \preceq_{+a} \beta$.

⟨3⟩7. Assume $\sigma_{s_{n-1}}|_V = \sigma_{s'_{n-1}}|_V$. In this case, by $s_{n-1} \xrightarrow{(t,\alpha)} s_n$, $\sigma_{s_{n-1}}|_V = \sigma_{s'_{n-1}}|_V$, ⟨3⟩6, and Definition 7-Condition (2), either $s'_{n-1} \xrightarrow{(t,\beta)} \text{wrong}$ or $s'_{n-1} \xrightarrow{(t,\beta)} s'_n$ and $\sigma_{s_n}|_V = \sigma_{s'_n}|_V$. This proves the claim.

⟨3⟩8. Assume $\text{Error}(s'_{n-1})$. In this case, by $s_{n-1} \xrightarrow{(t,\alpha)} s_n$, $\sigma_{s_{n-1}}|_V = \sigma_{s'_{n-1}}|_V$, ⟨3⟩6, and Definition 7-Condition (1), either $s'_{n-1} \xrightarrow{(t,\beta)} \text{wrong}$. This proves the claim.

⟨3⟩9. Q.E.D.

CASE: ADD-LOCAL-VARIABLE

⟨3⟩1. The proof is similar to that of ADD-GLOBAL-VARIABLE.

⟨3⟩2. Q.E.D.

CASE: HIDE-GLOBAL-VARIABLE

⟨3⟩1. In this case, $X = a$, $V = \text{Global}_{P_2}$ ($\text{Global}_{P_1} = V \cup \{a\}$), and $\mathcal{I}_2 = \exists a.\mathcal{I}_1$.

⟨3⟩2. We prove the claim by induction on the length n of the run ρ .

⟨3⟩3. *Base case:* $n = 0$ so there is no transition. The claim holds trivially, by taking $\rho = \rho'$.

⟨3⟩4. *Inductive case:* $n > 0$. Assume that the claim holds for all runs of length $|\rho| < n$ and consider a run ρ of length n : $\rho = s_1 \xrightarrow{\vec{1}} s_{n-1} \xrightarrow{(t,\alpha)} s_n$.

⟨3⟩5. By the inductive hypothesis, there exists a run $s'_1 \xrightarrow{\vec{1}'} s'_{n-1}$ such that $\sigma_{s_1}|_V = \sigma_{s'_1}|_V$, and $\sigma_{s_{n-1}}|_V = \sigma_{s'_{n-1}}|_V$ or $\text{Error}(s'_{n-1})$.

PROVE: There exists a witness run $\rho' = s'_1 \xrightarrow{\vec{1}'} s'_{n-1} \xrightarrow{(t,\beta)} s'_n$ such that $\sigma_{s_n}|_V = \sigma_{s'_n}|_V$ or $\text{Error}(s'_n)$

⟨3⟩6. By the antecedent we know that $\mathcal{I} \vdash \alpha \preceq_{-a} \beta$.

⟨3⟩7. Assume $\sigma_{s_{n-1}}|_V = \sigma_{s'_{n-1}}|_V$. In this case, by $s_{n-1} \xrightarrow{(t,\alpha)} s_n$, $\sigma_{s_{n-1}}|_V = \sigma_{s'_{n-1}}|_V$,

$\langle 3 \rangle 6$, and Definition 8-Condition (2), either $s'_{n-1} \xrightarrow{(t,\beta)} \text{wrong}$ or $s'_{n-1} \xrightarrow{(t,\beta)} s'_n$ and $\sigma_{s_n}|_V = \sigma_{s'_n}|_V$. This proves the claim.

$\langle 3 \rangle 8$. Assume $\text{Error}(s'_{n-1})$. In this case, by $s_{n-1} \xrightarrow{(t,\alpha)} s_n$, $\sigma_{s_{n-1}}|_V = \sigma_{s'_{n-1}}|_V$, $\langle 3 \rangle 6$, and Definition 8-Condition (1), either $s'_{n-1} \xrightarrow{(t,\beta)} \text{wrong}$. This proves the claim.

$\langle 3 \rangle 9$. Q.E.D.

CASE: HIDE-LOCAL-VARIABLE

$\langle 3 \rangle 1$. The proof is similar to that of HIDE-GLOBAL-VARIABLE.

$\langle 3 \rangle 2$. Q.E.D.

CASE: REDUCE-SEQUENTIAL-RIGHT – REDUCE-SEQUENTIAL-LEFT

$\langle 3 \rangle 1$. In this case, $X = \emptyset$ and $\mathcal{I}_1 = \mathcal{I}_2$, let us denote the invariant by $\mathcal{I}$.

$\langle 3 \rangle 2$. Taking $\mathcal{S} = \{(\alpha; \beta)\}$ in Lemma 5, there exists either a run $\rho'' = s_1 \xrightarrow{\vec{\mathcal{I}'}} s_2$ or $\rho'' = s_1 \xrightarrow{\vec{\mathcal{I}'}} \text{wrong}$ such that every $(\alpha; \beta)$ appears sequentially in $\vec{\mathcal{I}'}$.

$\langle 3 \rangle 3$. We now obtain the witness run ρ' of P_2 from ρ'' as follows:

$\langle 4 \rangle 1$. As long as ρ'' is in the form $\rho'' = s_1 \xrightarrow{\vec{\mathcal{I}}_1} s_i \xrightarrow{(t,\alpha)} s_{i+1} \xrightarrow{\vec{\mathcal{I}}_2} \text{wrong}$ such that α has not matching β (since it is blocked), by Lemma 3, we obtain a run either in the form (1) $\rho'' = s_1 \xrightarrow{\vec{\mathcal{I}}_1 \vec{\mathcal{I}}'_2} \text{wrong}$ or (2) $\rho'' = s_1 \xrightarrow{\vec{\mathcal{I}}_1 \vec{\mathcal{I}}'_2(t,\alpha)} \text{wrong}$ where $\vec{\mathcal{I}}'_2$ is a strictly shorter subsequence of $\vec{\mathcal{I}}_2$.

$\langle 4 \rangle 2$. As long as ρ'' is in the form $\rho'' = s_1 \xrightarrow{\mathbf{l}_1} s_i \xrightarrow{(t,\alpha)(t,\beta)} s_{i+2} \xrightarrow{\mathbf{l}_2} s_2$ perform the following:

$\langle 5 \rangle 1$. Either (1) $s_i \xrightarrow{(t, \llbracket \alpha; \beta \rrbracket)} s_{i+2}$ or (2) $s_i \xrightarrow{(t, \llbracket \alpha; \beta \rrbracket)} \text{wrong}$ holds.

$\langle 5 \rangle 2$. If (1) holds, then we obtain the run $\rho'' = s_1 \xrightarrow{\mathbf{l}_1} s_i \xrightarrow{(t, \llbracket \alpha; \beta \rrbracket)} s_{i+2} \xrightarrow{\mathbf{l}_2} s_2$.

$\langle 5 \rangle 3$. If (2) holds, then we obtain $\rho'' = s_1 \xrightarrow{\mathbf{l}_1} s_i \xrightarrow{(t, \llbracket \alpha; \beta \rrbracket)} s_{i+2} \text{wrong}$. In this case, we obtain a run that goes wrong in fewer transitions than before.

$\langle 3 \rangle 4$. At the end of the above iteration, we obtain a run ρ'' either of the form $\rho'' = s_1 \xrightarrow{\vec{\mathcal{I}}} s_2$ or $\rho'' = s_1 \xrightarrow{\vec{\mathcal{I}}} \text{wrong}$.

$\langle 4 \rangle 1$. If the former holds, then take $\rho' = \rho''$ as the witness run of P_2 .

$\langle 4 \rangle 2$. If the latter holds, it is possible that ρ'' is of the form $s_1 \xrightarrow{\vec{\mathcal{I}}''(t,\alpha)} \text{wrong}$, i.e., with an unmatched transition (t, α) at the end of the run. In this case, we obtain $\rho' = s_1 \xrightarrow{\vec{\mathcal{I}}''(t, \llbracket \alpha; \beta \rrbracket)} \text{wrong}$ as the witness run of P_2 , since $\llbracket \alpha; \beta \rrbracket$ goes wrong from every state from which α goes wrong.

$\langle 3 \rangle 5$. Q.E.D.

CASE: REDUCE-LABELED-RIGHT – REDUCE-LABELED-LEFT

- ⟨3⟩1. Given $\dagger \in Lbl \cup \{*\}$, the action $\llbracket \alpha; \dagger; \beta \rrbracket$ is similar to $\llbracket \alpha; \beta \rrbracket$, except that whether to execute β after α depends on the value of *exc* after α terminates. Therefore, the proof of these rules are similar to those of the rules REDUCE-SEQUENTIAL-RIGHT and REDUCE-SEQUENTIAL-LEFT.

- ⟨3⟩2. Q.E.D.

CASE: REDUCE-CHOICE

- ⟨3⟩1. Induction on the length of runs of P_1 .
- ⟨3⟩2. *Base case*: $n = 0$ so there is no transition. The claim holds trivially, by taking $\rho = \rho'$.
- ⟨3⟩3. *Inductive case*: $n > 0$. Assume that the claim holds for all runs of length $|\rho| < n$ and consider a run ρ of length n : $\rho = s_1 \xrightarrow{\vec{I}} s_{n-1} \xrightarrow{(t, \gamma)} s_n$.
- ⟨3⟩4. By the inductive hypothesis, there exists a run $s_1 \xrightarrow{\vec{I}'} s'_{n-1}$.
- ⟨3⟩5. If $\gamma \neq \alpha$, then take the witness run as $\rho' = s_1 \xrightarrow{\vec{I}'} s'_{n-1} \xrightarrow{(t, \gamma)} s_n$.
- ⟨3⟩6. If $\gamma = \alpha$, then either $\rho' = s_1 \xrightarrow{\vec{I}'} s'_{n-1} \xrightarrow{\llbracket \alpha \square \beta \rrbracket} s'_n$ or $\rho' = s_1 \xrightarrow{\vec{I}'} s'_{n-1} \xrightarrow{\llbracket \alpha \square \beta \rrbracket} \text{wrong}$ is a witness for ρ .
- ⟨3⟩7. Q.E.D.

CASE: EXPAND-PARALLEL

- ⟨3⟩1. The claim holds trivially for this rule, by the operational semantics rules FORK, JOIN-DONE, JOIN-ERROR, and CHOOSE. Renaming the local variables and *tid* in the actions encodes the fact that α_2 and β_2 refer to distinct, fresh local variables of the procedure.

CASE: REDUCE-LOOP

- ⟨3⟩1. In this case, $X = \emptyset$ and $\mathcal{I}_1 = \mathcal{I}_2$, let us denote the invariant by $\mathcal{I}$.
- ASSUME: $\gamma = \llbracket \alpha; *; \text{block} \rrbracket$
- ⟨3⟩2. Every execution of α° yields a (possibly empty) sequence of non-final transitions of α ending with *exc* = **skip** followed by a final transition of α ending with *exc* $\neq$ **skip**. The rule separates the non-final transitions (represented by γ) from the final transition and replaces the former with β and the latter with $\llbracket \alpha; \text{skipblock} \rrbracket$.
- ⟨3⟩3. Since γ is either a right or a left mover, non-final iterations of α° can be treated as an atomic statement. Thus, by Lemma 5, there exists either a run $\rho'' = s_1 \xrightarrow{\vec{I}'} s_2$

or $\rho'' = s_1 \xrightarrow{\vec{1}'} \mathbf{wrong}$ such that for each instance of $\alpha^\odot$, if any, all the non-final transitions of that instance appear sequentially in $\vec{1}'$.

ASSUME: Without loss of generality $\rho'' = s_1 \xrightarrow{\vec{1}'} s_2$ of P_1 exists as given above.

PROVE: There exists a run ρ' of P_2 such that either $\rho' = s_1 \xrightarrow{\quad} s_2$ or $\rho' = s_1 \xrightarrow{\quad} \mathbf{wrong}$.

$\langle 4 \rangle 1$. We obtain ρ' of P_2 from ρ'' of P_2 as follows: For each instance of $\alpha^\odot$, we case split on whether the instance has any non-final transition or not.

$\langle 5 \rangle 1$. If the instance has no non-final transitions, we replace the final transition $s_i \xrightarrow{(t,\alpha)} s_{i+1}$ of the instance with $s_i \xrightarrow{(t,\beta)(t, \llbracket \alpha; \mathbf{skip} \mathbf{block} \rrbracket)} s_{i+1}$, since β is reflexive and $\llbracket \alpha; \mathbf{skip} : \mathbf{block} \rrbracket$ simulates all the transitions of α that ends with an exception other than **skip**.

$\langle 5 \rangle 2$. If the instance has some non-final transitions, then for any sequence of transitions $s_i \xrightarrow{(t,\alpha)(t,\alpha)\dots(t,\alpha)} s_{i+1}$, either $s_i \xrightarrow{(t,\beta)} s_{i+1}$ or $s_i \xrightarrow{(t,\beta)} \mathbf{wrong}$ exists, since β is reflexive and $\mathcal{I} \vdash \llbracket \beta; \gamma \rrbracket \preceq \beta$ where γ simulates transitions of α that end with $exc = \mathbf{skip}$.

$\langle 3 \rangle 4$. Q.E.D.

CASE: REDUCE-LOOP-RIGHT

$\langle 3 \rangle 1$. In this case, $X = \emptyset$ and $\mathcal{I}_1 = \mathcal{I}_2$, let us denote the invariant by $\mathcal{I}$.

ASSUME: $\gamma = \llbracket \alpha; * : \mathbf{block} \rrbracket$

$\langle 3 \rangle 2$. Every execution of $\alpha^\odot$ yields a (possibly empty) sequence of non-final transitions of α ending with $exc = \mathbf{skip}$ followed by a final transition of α ending with $exc \neq \mathbf{skip}$. The rule separates the non-final transitions (represented by γ) from the final transition and replaces the former with β and the latter with $\llbracket \alpha; \mathbf{skip} : \mathbf{block} \rrbracket$.

PROVE: For any run $\rho = s_1 \xrightarrow{\vec{1}} s_2$ of P_1 , there exists a run P_1 either in the form $s_1 \xrightarrow{\vec{1}'} s_2$ or $s_1 \xrightarrow{\vec{1}'} \mathbf{wrong}$ without any non-final transition of $\alpha^\odot$.

$\langle 3 \rangle 3$. We prove the claim by induction on the length n of the run ρ .

$\langle 3 \rangle 4$. *Base case:* $n = 0$. In this case, the run ρ has no transitions, so $\rho' = \rho$.

$\langle 3 \rangle 5$. *Inductive case:* $n > 0$. Assume that the claim holds for runs of length $< n$.

Consider a run $\rho = s_1 \xrightarrow{\vec{1}} s_n \xrightarrow{(t,\zeta)} s_{n+1}$. By inductive hypothesis, there exists a run of P_2 either in the form $s_1 \xrightarrow{\vec{1}'} s_n$ or $s_1 \xrightarrow{\vec{1}'} \mathbf{wrong}$. Without loss of generality, assume the former form. We do a case analysis on the transition (t, ζ) .

$\langle 4 \rangle 1$. If $\zeta \neq \alpha$, then $s_1 \xrightarrow{\vec{1}'} s_n \xrightarrow{(t,\zeta)} s_{n+1}$ is a witness run for the claim.

- $\langle 4 \rangle 2$. If $\zeta = \alpha$ and $s_n \xrightarrow{(t, \zeta)} s_{n+1}$ is the final transition of a α° instance, then $s_1 \xrightarrow{\vec{I}'}$
 $s_n \xrightarrow{(t, \zeta)} s_{n+1}$ is a witness run for the claim.
- $\langle 4 \rangle 3$. Assume now that $\zeta = \alpha$ and $s_n \xrightarrow{(t, \zeta)} s_{n+1}$ is a non-final transition of a α°
 instance. In this case, $\vec{I}'$ contains a transition (t, β) as the last transition of
 t before (t, ζ) . By Lemma 4, there exists either a run of P_1 either in the form
 (1) $s_1 \xrightarrow{\vec{I}''(t, \beta)} s_n$ or (2) $s_1 \xrightarrow{\vec{I}''} \text{wrong}$ or (3) $s_1 \xrightarrow{\vec{I}''(t, \beta)} \text{wrong}$ such that $\vec{I}''$ is
 a (strictly shorter) subsequence of $\vec{I}'$ without the transition (t, β) referred to
 above.
- $\langle 5 \rangle 1$. If (1) is true, since γ simulates all transitions of non-final transitions of α°
 and $\mathcal{I} \vdash \llbracket \beta; \gamma \rrbracket \preceq \beta$, then either $s_1 \xrightarrow{\vec{I}''(t, \beta)} s_{n+1}$ or $s_1 \xrightarrow{\vec{I}''(t, \beta)} \text{wrong}$ exists.
 Each of these runs is a witness run of the claim.
- $\langle 5 \rangle 2$. If (2) or (3) is true, then the corresponding violation is a witness run of P_2 .
- $\langle 3 \rangle 6$. We have proved that for any run $\rho = s_1 \xrightarrow{\vec{I}} s_2$ of P_1 , there exists a run P_1 either
 in the form $s_1 \xrightarrow{\vec{I}'} s_2$ or $s_1 \xrightarrow{\vec{I}'} \text{wrong}$ without any non-final transition of α° .
 Without loss of generality, assume the latter form. We now obtain a witness run
 of P_2 as follows: Replace every final transition $s_i \xrightarrow{(t, \alpha)} s_{i+1}$ of a α° instance with
 $s_i \xrightarrow{(t, \llbracket \alpha; \text{skip} : \text{block} \rrbracket)} s_{i+1}$, since $\llbracket \alpha; \text{skip} : \text{block} \rrbracket$ simulates all transitions of α that
 ends with $\text{exc} \neq \text{skip}$.
- $\langle 3 \rangle 7$. Q.E.D.

CASE: INLINE-CALL

- $\langle 3 \rangle 1$. The claim holds trivially for this rule, by the operational semantics rule PROC-
 CALL.

CASE: REDUCE-PROCEDURE

- $\langle 3 \rangle 1$. In this case, $X = \emptyset$ and $\mathcal{I}_1 = \mathcal{I}_2$, let us denote the invariant by $\mathcal{I}$.
- $\langle 3 \rangle 2$. Taking $\mathcal{S} = \{\text{Body}_{\mathbf{p}_i} \mid \mathbf{p}_i \in \mathcal{M}\}$ in Lemma 5, there exists either a run $\rho'' = s_1 \xrightarrow{\vec{I}'}$
 s_2 or $\rho'' = s_1 \xrightarrow{\vec{I}'} \text{wrong}$ such that every procedure from $\mathcal{M}$ appears sequentially
 in $\vec{I}'$. Since left movers are non-blocking, the $\mathbb{L}$ -instances of procedures in ρ''
 either goes wrong or terminates. If an $\mathbb{R}$ -instance has blocked, then ρ'' must be a
 violation. We eliminate the transitions of such instances from $\vec{I}'$ using Lemma 3
 and still obtain a violation from s_1 . Thus, assume that all the procedures have
 $\mathbb{L}$ -instances in ρ'' .

⟨3⟩3. We now obtain the witness run ρ' of P_2 from ρ'' as follows:

⟨4⟩1. As long as ρ'' is in the form $\rho'' = s_1 \xrightarrow{l_1} s_m \xrightarrow{(t, \beta_1) \dots (t, \beta_n)} s_{m+n} \xrightarrow{l_2} s_2$ such that $(t, \beta_1) \dots (t, \beta_n)$ are the transitions of an instance of a procedure $\mathbf{p}_i \in \mathcal{M}$ perform the following:

⟨5⟩1. By $(P, \mathcal{I}, \mathcal{A}) \vdash \{\alpha_i\} \mathbf{p}_i$, either (1) $s_m \xrightarrow{(t, \alpha_i)} s_{m+n}$ or (2) $s_m \xrightarrow{(t, \alpha_i)} \mathbf{wrong}$ holds.

⟨5⟩2. If (1) holds, then we obtain the run $\rho'' = s_1 \xrightarrow{l_1} s_m \xrightarrow{(t, \alpha_i)} s_{m+n} \xrightarrow{l_2} s_2$.

⟨5⟩3. If (2) holds, then we obtain $\rho'' = s_1 \xrightarrow{l_1} s_m \xrightarrow{(t, \alpha_i)} \mathbf{wrong}$. In this case, we obtain a run that goes wrong in fewer transitions than before.

⟨3⟩4. At the end of the above iteration, we obtain a run ρ'' of the form either $\rho'' = s_1 \xrightarrow{\dots} s_2$ or $\rho'' = s_1 \xrightarrow{\dots} \mathbf{wrong}$.

⟨4⟩1. If the former holds, then take $\rho' = \rho''$ as the witness run of P_2 .

⟨4⟩2. If the latter holds, it is possible that ρ'' is of the form $s_1 \xrightarrow{\vec{l}''(t, \beta_1) \dots (t, \beta_n)} \mathbf{wrong}$ such that $(t, \beta_1) \dots (t, \beta_n)$ are the transitions of an instance of a procedure $\mathbf{p}_i \in \mathcal{M}$. In this case, we obtain $\rho' = s_1 \xrightarrow{\vec{l}''(t, \alpha_i)} \mathbf{wrong}$ as the witness run of P_2 , since, by $(P, \mathcal{I}, \mathcal{A}) \vdash \{\alpha_i\} \mathbf{p}_i$, α_i goes wrong from every state from which a sequential execution of $\text{Body}_{\mathbf{p}_i}$ goes wrong.

⟨3⟩5. Q.E.D.

□

The following theorem generalizes the lemma above to an arbitrary number of proof steps and states the soundness of our proof method.

Theorem 5 (Soundness of the QED proof system):

Let $(P_1, \mathcal{I}_1) \dashrightarrow \dots \dashrightarrow (P_n, \mathcal{I}_n)$ be a sequence of legal proof steps. Let $\mathcal{I}$ be a state predicate such that for all $i \in [1, n]$, $\mathcal{I} \Rightarrow \mathcal{I}_i$. Then $\mathcal{I} \vdash P_1 \preceq P_n$ holds. ■

PROOF.

ASSUME: $(P_1, \mathcal{I}_1) \dashrightarrow \dots \dashrightarrow (P_n, \mathcal{I}_n)$

PROVE: For every run $\rho = s_1 \xrightarrow{\vec{l}} s_2$ of the program P_1 from $\exists X_1. \mathcal{I}$, there exists a run $\rho' = s'_1 \xrightarrow{\vec{l}'} s'_2$ or a violation $\rho' = s'_1 \xrightarrow{\vec{l}'} \mathbf{wrong}$ of the program P_n from $\exists X_n. \mathcal{I}$ such that $\sigma_{s_1}|_V = \sigma_{s'_1}|_V$, and $\sigma_{s_2}|_V = \sigma_{s'_2}|_V$ where $V = \text{Global}_{P_n} \cap \text{Global}_{P_1}$. We call ρ' a *witness* for ρ .

⟨1⟩1. In the following proof, we will ignore the θ component of the program state, thus two states are treated equivalent although the code maps are different.

- (1)2. Proof by induction on the length n of the proof.
- (1)3. *Base case:* $n = 0$. In this case there is no proof step, so $P_1 = P_n$. Thus, the claim holds trivially.
- (1)4. *Inductive case:* $n > 0$. Assume that the proof is of the form $(P_1, \mathcal{I}_1) \dashrightarrow \dots \dashrightarrow (P_n, \mathcal{I}_n) \dashrightarrow (P_{n+1}, \mathcal{I}_{n+1})$, and the claim holds for $(P_1, \mathcal{I}_1) \dashrightarrow \dots \dashrightarrow (P_n, \mathcal{I}_n)$.
- (1)5. Let $V = \text{Global}_{P_n} \cap \text{Global}_{P_1}$ and $W = \text{Global}_{P_{n+1}} \cap \text{Global}_{P_1}$.
- (1)6. Let $X_n = \text{fv}(\mathcal{I}) \setminus \text{Global}_{P_n}$ and $X_{n+1} = \text{fv}(\mathcal{I}) \setminus \text{Global}_{P_{n+1}}$.
- ASSUME: A state s_1 of P_1 such that $s_1 \models \mathcal{I}$ and a run $\rho = s_1 \xrightarrow{\vec{1}} s_2$ of P_1 .
- PROVE: There exists a run ρ' of P_{n+1} that is either in the form $\rho' = s'_1 \xrightarrow{\vec{1}'} s'_2$ or $\rho' = s'_1 \xrightarrow{\vec{1}'} \text{wrong}$ such that $s'_1 \models \exists X_{n+1}. \mathcal{I}$, $\sigma_{s_1}|_W = \sigma_{s'_1}|_W$, and $\sigma_{s_2}|_W = \sigma_{s'_2}|_W$.
- (3)1. We do a case analysis on the proof rule applied during the last step $(P_n, \mathcal{I}_n) \dashrightarrow (P_{n+1}, \mathcal{I}_{n+1})$.

CASE: INVARIANT

- (4)1. In this case, $P_n = P_{n+1}$ and $X_n = X_{n+1}$ and $V = W$.
- (4)2. By the inductive hypothesis for n , there exists a run ρ'' of P_n either in the form $\rho'' = s''_1 \xrightarrow{\vec{1}''} s''_2$ or $\rho'' = s''_1 \xrightarrow{\vec{1}''} \text{wrong}$ such that $s''_1 \models \exists X_n. \mathcal{I}$, $\sigma_{s_1}|_V = \sigma_{s''_1}|_V$, and $\sigma_{s_2}|_V = \sigma_{s''_2}|_V$.
- (4)3. $\sigma_{s_1}|_W = \sigma_{s''_1}|_W$ and $\sigma_{s_2}|_W = \sigma_{s''_2}|_W$ hold since $V = W$. Thus, taking ρ' as either form of ρ'' in (4)2 proves the claim.
- (4)4. Q.E.D.

CASE: ADD-GLOBAL-VARIABLE

- (4)1. In this case, $X_n = X_{n+1} \cup \{a\}$ and $V = W$.
- (4)2. By the inductive hypothesis for n , there exists a run ρ'' of P_n either in the form $\rho'' = s''_1 \xrightarrow{\vec{1}''} s''_2$ or $\rho'' = s''_1 \xrightarrow{\vec{1}''} \text{wrong}$ such that $s''_1 \models \exists X_n. \mathcal{I}$, $\sigma_{s_1}|_V = \sigma_{s''_1}|_V$, and $\sigma_{s_2}|_V = \sigma_{s''_2}|_V$.
- ASSUME: $Z = \text{Global}_{P_n}$.
- (4)3. $V \subseteq Z$ and $W \subseteq Z$.
- (4)4. Without loss of generality, assume $\rho'' = s''_1 \xrightarrow{\vec{1}''} s''_2$.
- (4)5. $s''_1 \models \mathcal{I}_{n+1}$ since $s''_1 \models X_n. \mathcal{I}$ and $\mathcal{I} \Rightarrow \mathcal{I}_{n+1}$ and $\text{Global}_{P_n} \cap X_n = \emptyset$.
- (4)6. By Theorem 4, there exists a run ρ' of P_{n+1} either in the form $\rho' = s'_1 \xrightarrow{\vec{1}'} s'_2$ or $\rho' = s'_1 \xrightarrow{\vec{1}'} \text{wrong}$ such that $s'_1 \models \mathcal{I}_{n+1}$, $\sigma_{s'_1}|_Z = \sigma_{s''_1}|_Z$ and $\sigma_{s'_2}|_Z = \sigma_{s''_2}|_Z$.

- ⟨4⟩7. $s'_1 \models \exists X_{n+1}.\mathcal{I}$, since $s''_1 \models \exists X_n.\mathcal{I}$, $X_n = X_{n+1} \cup \{a\}$, and by Theorem 4 (see the proof of ADD-GLOBAL-VARIABLE).
 ⟨4⟩8. $\sigma_{s_1}|_W = \sigma_{s'_1}|_W$, since (1) $\sigma_{s_1}|_V = \sigma_{s''_1}|_V$ and $V = W$, and (2) $\sigma_{s'_1}|_Z = \sigma_{s''_1}|_Z$ and $W \subseteq Z$.
 ⟨4⟩9. $\sigma_{s_2}|_W = \sigma_{s'_2}|_W$, since (1) $\sigma_{s_2}|_V = \sigma_{s''_2}|_V$ and $V = W$, and (2) $\sigma_{s'_2}|_Z = \sigma_{s''_2}|_Z$ and $W \subseteq Z$.
 ⟨4⟩10. ⟨4⟩7, ⟨4⟩8, and ⟨4⟩9 prove the existence of ρ' satisfying the claim.
 ⟨4⟩11. Q.E.D.

CASE: HIDE-GLOBAL-VARIABLE

- ⟨4⟩1. In this case, $X_n = X_{n+1}$.
 ⟨4⟩2. By the inductive hypothesis for n , there exists a run ρ'' of P_n either in the form $\rho'' = s''_1 \xrightarrow{\vec{1}''} s''_2$ or $\rho'' = s''_1 \xrightarrow{\vec{1}''} \text{wrong}$ such that $s''_1 \models \exists X_n.\mathcal{I}$, $\sigma_{s_1}|_V = \sigma_{s''_1}|_V$, and $\sigma_{s_2}|_V = \sigma_{s''_2}|_V$.
 ASSUME: $Z = \text{Global}_{P_{n+1}}$.
 ⟨4⟩3. $W \subseteq Z$.
 ⟨4⟩4. Without loss of generality, assume $\rho'' = s''_1 \xrightarrow{\vec{1}''} s''_2$.
 ⟨4⟩5. $s''_1 \models \mathcal{I}_{n+1}$ since $s''_1 \models X_n.\mathcal{I}$ and $\mathcal{I} \Rightarrow \mathcal{I}_{n+1}$ and $\text{Global}_{P_n} \cap X_n = \emptyset$.
 ⟨4⟩6. By Theorem 4, there exists a run ρ' of P_{n+1} either in the form $\rho' = s'_1 \xrightarrow{\vec{1}'} s'_2$ or $\rho' = s'_1 \xrightarrow{\vec{1}'} \text{wrong}$ such that $s'_1 \models \mathcal{I}_{n+1}$, $\sigma_{s'_1}|_Z = \sigma_{s''_1}|_Z$ and $\sigma_{s'_2}|_Z = \sigma_{s''_2}|_Z$.
 ⟨4⟩7. Either $a \in \text{Global}_{P_1}$ or not.

CASE: $a \in \text{Global}_{P_1}$

- ⟨5⟩1. In this case, $V = W \cup \{a\}$
 ⟨5⟩2. $s'_1 \models \exists X_{n+1}.\mathcal{I}$, since $s''_1 \models \exists X_n.\mathcal{I}$, $X_n = X_{n+1}$, $\sigma_{s'_1}|_Z = \sigma_{s''_1}|_Z$.
 ⟨5⟩3. $\sigma_{s_1}|_W = \sigma_{s'_1}|_W$, since (1) $\sigma_{s_1}|_V = \sigma_{s''_1}|_V$ and $W \subseteq V$, and (2) $\sigma_{s'_1}|_Z = \sigma_{s''_1}|_Z$ and $W \subseteq Z$.
 ⟨5⟩4. $\sigma_{s_2}|_W = \sigma_{s'_2}|_W$, since (1) $\sigma_{s_2}|_V = \sigma_{s''_2}|_V$ and $W \subseteq V$, and (2) $\sigma_{s'_2}|_Z = \sigma_{s''_2}|_Z$ and $W \subseteq Z$.
 ⟨5⟩5. ⟨5⟩2, ⟨5⟩3, and ⟨5⟩4 prove the existence of ρ' satisfying the claim.

CASE: $a \notin \text{Global}_{P_1}$

- ⟨5⟩1. In this case, $V = W$
 ⟨5⟩2. $s'_1 \models \exists X_{n+1}.\mathcal{I}$, since $s''_1 \models \exists X_n.\mathcal{I}$, $X_n = X_{n+1}$, $\sigma_{s'_1}|_Z = \sigma_{s''_1}|_Z$.

$\langle 5 \rangle 3$. $\sigma_{s_1}|_W = \sigma_{s'_1}|_W$, since (1) $\sigma_{s_1}|_V = \sigma_{s''_1}|_V$ and $V = W$, and (2) $\sigma_{s'_1}|_Z = \sigma_{s''_1}|_Z$ and $W \subseteq Z$.

$\langle 5 \rangle 4$. $\sigma_{s_2}|_W = \sigma_{s'_2}|_W$, since (1) $\sigma_{s_2}|_V = \sigma_{s''_2}|_V$ and $V = W$, and (2) $\sigma_{s'_2}|_Z = \sigma_{s''_2}|_Z$ and $W \subseteq Z$.

$\langle 5 \rangle 5$. $\langle 5 \rangle 2$, $\langle 5 \rangle 3$, and $\langle 5 \rangle 4$ prove the existence of ρ' satisfying the claim.

$\langle 4 \rangle 8$. Q.E.D.

CASE: Other proof rules

$\langle 4 \rangle 1$. In this case, $X_n = X_{n+1}$, $V = W$, and $\mathcal{I}_n = \mathcal{I}_{n+1}$.

$\langle 4 \rangle 2$. By the inductive hypothesis for n , there exists a run ρ'' of P_n either in the form $\rho'' = s''_1 \xrightarrow{\vec{1}''} s''_2$ or $\rho'' = s''_1 \xrightarrow{\vec{1}''} \text{wrong}$ such that $s''_1 \models \exists X_n.\mathcal{I}$, $\sigma_{s_1}|_V = \sigma_{s''_1}|_V$, and $\sigma_{s_2}|_V = \sigma_{s''_2}|_V$. Since $V = W$, $\sigma_{s''_2}|_V = \sigma_{s''_2}|_W$.

$\langle 4 \rangle 3$. $s''_1 \models \mathcal{I}_{n+1}$ since $s''_1 \models X_n.\mathcal{I}$ and $\mathcal{I} \Rightarrow \mathcal{I}_{n+1}$ and $\text{Global}_{P_n} \cap X_n = \emptyset$.

$\langle 4 \rangle 4$. Without loss of generality, assume $\rho'' = s''_1 \xrightarrow{\vec{1}''} s''_2$. By Theorem 4, there exists a run ρ' of P_{n+1} either in the form $\rho' = s'_1 \xrightarrow{\vec{1}'} s'_2$ or $\rho' = s'_1 \xrightarrow{\vec{1}'} \text{wrong}$ such that $s'_1 \models \mathcal{I}_{n+1}$, $\sigma_{s'_1} = \sigma_{s''_1}$ and $\sigma_{s'_2} = \sigma_{s''_2}$, since $\text{Global}_{P_n} = \text{Global}_{P_{n+1}}$ by the proof rules.

$\langle 4 \rangle 5$. $s'_1 \models \exists X_{n+1}.\mathcal{I}$, since $s''_1 \models \exists X_n.\mathcal{I}$, $X_n = X_{n+1}$, and $\sigma_{s'_1} = \sigma_{s''_1}$.

$\langle 4 \rangle 6$. $\sigma_{s_1}|_W = \sigma_{s'_1}|_W$, and $\sigma_{s_2}|_W = \sigma_{s'_2}|_W$, since $V = W$.

$\langle 4 \rangle 7$. $\langle 4 \rangle 5$ and $\langle 4 \rangle 6$ prove the existence of ρ' satisfying the claim.

$\langle 4 \rangle 8$. Q.E.D.

□

A corollary of this theorem is that if P_n is good from $\mathcal{I}_n$, then P_1 is good from $\mathcal{I}_n$. This means that one can prove the assertions in P_1 by gradually obtaining programs with coarser-grained concurrency using our proof rules. At any point during this process, the decreased level of granularity will simplify the checking of assertions. This corollary is formalized as follows:

Corollary 6 (Soundness of proving assertions using QED):

Let $(P_1, \text{true}) \dashrightarrow \dots \dashrightarrow (P_n, \mathcal{I}_n)$ be a sequence of proof steps, and $\mathcal{I}$ be a state predicate such that for all $i \in [1, n]$, $\mathcal{I} \Rightarrow \mathcal{I}_i$. Let $V = \text{Global}_{P_1} \cap \text{Global}_{P_n}$, $X_1 = \text{fv}(\mathcal{I}) \setminus \text{Global}_{P_1}$, and $X_n = \text{fv}(\mathcal{I}) \setminus \text{Global}_{P_n}$. If P_n is good from $\exists X_n.\mathcal{I}$, then (1) P_1 is good from $\exists X_1.\mathcal{I}$ and (2) $\text{Good}|_V(P_1, \exists X_1.\mathcal{I}) \subseteq \text{Good}|_V(P_n, \exists X_n.\mathcal{I})$. ■

In this chapter, we presented the core rules of the QED proof system. The ACTIONPL language and the proof rules define the fundamental approach using reduction-based atomicity and abstraction. In the next two chapters 14 and 15, we will show how this approach can be applied to a higher-level programming language, QEDPL. While doing this, we will justify the soundness of the transformations on QEDPL programs by associating these transformations with proof rules presented in this chapter.

Chapter 14

THE QEDPL PROGRAMMING LANGUAGE

Chapter 12 presented a low-level programming language, ACTIONPL, and Chapter 13 formalized the proof system and the verification approach on ACTIONPL programs. The purpose of this chapter and the next one are to demonstrate how the QED proof system can be applied to verify programs written in a higher-level programming language. For this, we will introduce a high-level programming language, called QEDPL and then propose high-level transformations on QEDPL programs. In order to distinguish the transformations on QEDPL programs from the transformations on ACTIONPL programs, which were called “proof rules”, we will call the former “proof tactic”. While presenting the proof tactics, we will establish a formal link between QEDPL and ACTIONPL programs and between proof rules and tactics. We obtain the former by giving a denotational semantics to QEDPL programs that compiles every QEDPL program down to an ACTIONPL program. Defining semantics of QEDPL through ACTIONPL helps us to obtain the latter link, and thus to justify the soundness of the proof tactics on QEDPL programs.

ACTIONPL and QEDPL share the formal representation of programs as explained in Section 12.2 and the syntax for the top-level elements (types, constants, functions, axioms, global variables and procedures). Similarly to an ACTIONPL program, a QEDPL program is represented by a tuple: $\hat{P} = \langle Type_{\hat{P}}, Function_{\hat{P}}, Constant_{\hat{P}}, Axiom_{\hat{P}}, Global_{\hat{P}}, Proc_{\hat{P}} \rangle$. We refer the reader to the pertaining subsections of Section 12.2 for the details about the elements of this tuple (the top-levels element of the program). From this point on, if a notation (e.g., P for program or S for statement) is shared by ACTIONPL and QEDPL, we use $\hat{hat}$ (e.g., $\hat{P}$ or $\hat{S}$) to indicate that it belongs to a QEDPL program.

QEDPL programs differ from ACTIONPL programs in the syntax of statements within procedure bodies. While ACTIONPL statements provide a compact way of expressing a computation, QEDPL statements provide a convenient (to the user) way of writing imperative programs and allow us to demonstrate the ideas in QED on a high-level language.

The syntax of QEDPL is based on that of the Boogie 2 language [52], so QEDPL strikes a reasonable compromise between (1) an expressive representation of computation and (2) a clear syntax. While (1) leads to translation of languages such as Java with richer features to QEDPL with precision, (2) leads to understandable code that very looks like the original version after this translation. Therefore, we chose the QEDPL language as the input language of our tool QED-VERIFIER, and in our experimental study, we proved the QEDPL versions of programs from the literature.

An alternative language to QEDPL is Dafny [101]. However, in this thesis, we opted for QEDPL, which is between ACTIONPL and Dafny and not as high-level as Dafny, in order to keep the higher-level language features minimal and focus on the core aspects of the QED approach. In this view, using Dafny would require to model complex aspects of the language such as object orientation, which is orthogonal to our core techniques. We foresee that further application of the proof system to Dafny and even more realistic languages, for example to Java or C# , will be possible by following the ideas in this thesis.

In this chapter, we will focus on explaining the syntax and semantics of QEDPL statements. In addition, we present the encoding of several extra language features, in terms of the core language features, which we frequently use in the benchmark programs. These features include conditional and nondeterministic while loops, maps and records: a structured data type that is analogous to a method-less class in Dafny or Java.

Outline. Section 14.1 presents the syntax for QEDPL statements along with their informal semantics. Section 14.2 defines the operational semantics of QEDPL statements by compiling them to ACTIONPL statements. Section 14.3 shows how we encode while loops, maps, and records using the core language features.

14.1 Syntax and Informal Semantics

Figure 14.1 shows the syntax of QEDPL statements. The syntax of expressions is as defined in Section 12.1. E refers to an expression of a standard (**int** or **bool**) or user-defined type, and B refers to a boolean predicate. Expressions and predicates whose free variables are all local are referred to by E_ℓ and B_ℓ , respectively. QEDPL makes use of the exception labels (Lbl) and the thread-local variables tid and exc , which were defined for ACTIONPL

$$\begin{array}{ll}
x \in \text{Global} & \ell \in \text{Local} \quad E \in \text{Expr} \quad B \in \text{BExpr} \\
\\
\text{Var} : & v ::= x \mid \ell \\
\text{Guard} : & g ::= * \mid B_\ell \\
\text{Lbl} : & l ::= \mathbf{break} \mid \mathbf{continue} \mid \mathbf{return} \mid \mathbf{exit} \\
\text{Cmd} : & C ::= \mathbf{assume} \, B \mid \mathbf{assert} \, B \mid v := E \mid \mathbf{havoc} \, v \mid \mathbf{throw} \, l \\
\text{StraightStmt} : & \bar{S} ::= C \mid \bar{S} ; \bar{S} \mid \text{if}(g) \{ \bar{S} \} \mid \text{if}(g) \{ \bar{S} \} \text{ else } \{ \bar{S} \} \\
& \quad \mid \text{try} \{ \bar{S} \} \text{ catch}(l) \{ \bar{S} \} \mid \text{try} \{ \bar{S} \} \text{ catch} \{ \bar{S} \} \\
\text{Atomic} : & \hat{\alpha} ::= C \mid \mathbf{atomic} \{ \bar{S} \} \\
\text{Stmt} : & \hat{S} ::= \hat{\alpha} \mid \mathbf{call} \, \ell^* := \mathbf{p}(E_\ell^*) \mid \hat{S}; \hat{S} \mid \mathbf{loop} \{ \hat{S} \} \mid \mathbf{dopar} \{ \hat{S} \} \text{ with } \{ \hat{S} \} \\
& \quad \mid \text{if}(g) \{ \hat{S} \} \mid \text{if}(g) \{ \hat{S} \} \text{ else } \{ \hat{S} \} \\
& \quad \mid \text{try} \{ \hat{S} \} \text{ catch}(l) \{ \hat{S} \} \mid \text{try} \{ \hat{S} \} \text{ catch} \{ \hat{S} \} \\
\text{Comment} \, \text{cmt} : & ::= // \text{ One-line text } \mid /* \text{ Multiple-line text } */
\end{array}$$

Figure 14.1: The syntax of the QEDPL language.

in Chapter 12.

The primitive statements are called *commands* (*Cmd*). The **assume** command is used to block the execution on a condition and the **assert** command is used to fail the execution on an unexpected condition. When the expression B evaluates to **true** at the current program state, both **assume** B and **assert** B are identical to skip, i.e., they do not cause any change in the program store (including *exc*). Otherwise, **assume** B blocks the current thread, while **assert** B causes the current thread (and also the current run) to go wrong. We will the command **skip** as a syntactic sugar for **assume true** and **block** as a syntactic sugar for **assume false**. Variable assignments ($v := E$) are interpreted in the standard way: The expression E is evaluated to a value of type $\text{Val}_{\mathbf{t}(v)}$ and the resulting value is assigned to the variable v . The command **havoc** v assigns a nondeterministic value to v from the set $\text{Val}_{\mathbf{t}(v)}$. We explain the command **throw** l while explaining the **try-catch** statement in Section 14.1.1.

An *atomic statement* (*Atomic*) is executed indivisibly by another thread. Commands (*Cmd*) are atomic statements by default. In addition, the statement **atomic**{ $\bar{S}$ } is used to execute the statement $\bar{S}$ sequentially, i.e., when $\bar{S}$ starts executing, no other thread is allowed

to run until $\bar{S}$ is completely evaluated. We only allow sequential composition ($;$), **if** and **try-catch** statements in $\bar{S}$. A statement under this restriction is called a *straight statement* (*StraightStmt*) and referred to by $\bar{S}$. As we will see in Section 14.2, this restriction is crucial for automatic compilation of atomic statements to gated actions in ACTIONPL. Let $\hat{\alpha}$ and $\hat{\beta}$ range over atomic statements (*Cmd* or **atomic** $\{\bar{S}\}$). Similarly to ACTIONPL, let $Atoms(\hat{P})$ return the set of the atomic statements in $\hat{P}$, as defined below:

$$\begin{aligned}
Atoms(\hat{\alpha}) &= \{\hat{\alpha}\} \\
Atoms(\text{call } \ell^* := \mathbf{p}(E_\ell^*)) &= \emptyset \\
Atoms(\hat{S}_1; \hat{S}_2) &= Atoms(\hat{S}_1) \cup Atoms(\hat{S}_2) \\
Atoms(\text{loop } \{\hat{S}\}) &= Atoms(\hat{S}) \\
Atoms(\text{if}(g) \{\hat{S}\}) &= Atoms(\hat{S}) \\
Atoms(\text{if}(g) \{\hat{S}_1\} \text{ else } \{\hat{S}_2\}) &= Atoms(\hat{S}_1) \cup Atoms(\hat{S}_2) \\
Atoms(\text{try } \{\hat{S}_1\} \text{ catch}(l)\{\hat{S}_2\}) &= Atoms(\hat{S}_1) \cup Atoms(\hat{S}_2) \\
Atoms(\text{try } \{\hat{S}_1\} \text{ catch}\{\hat{S}_2\}) &= Atoms(\hat{S}_1) \cup Atoms(\hat{S}_2) \\
Atoms(\text{dopar } \{\hat{S}_1\} \text{ with } \{\hat{S}_2\}) &= Atoms(\hat{S}_1) \cup Atoms(\hat{S}_2)
\end{aligned}$$

We extend the definition of *Atoms* to programs as follows:

$$Atoms(\hat{P}) = \bigcup_{\mathbf{p} \in Proc_{\hat{P}}} Atoms(Body_{\mathbf{p}}) \quad (14.1)$$

We define $Var(\hat{\alpha})$ be the set of variables visible to the atomic statement $\hat{\alpha}$. $Var(\hat{\alpha})$ is the union of global variables $Global_{\hat{P}}$ of the program, the local variables $Local_{\mathbf{p}}$ of the containing procedure $\mathbf{p}$, and the variables *tid* and *exc*. We syntactically enforce that all the free variables in the statement $\hat{\alpha}$ is contained in $Var(\hat{\alpha})$.

The *compound statements* (*Stmt*) of QEDPL are constructed by composing atomic statements. Conditional statements (**if**) and sequential composition ($;$) are interpreted in the standard way. The guard of a conditional statement, referred to by g , can be either a local boolean expression, or $*$. If the guard is B_ℓ , then the “then” branch is selected if B_ℓ evaluates to **true** in the current program state. If the guard is $*$ then whether to take the “then” branch is selected non-deterministically.

Our looping statement **loop** $\{\hat{S}\}$ causes its body $\hat{S}$ to be evaluated until an exception occurs. The iterations of the loop continue as long as the body terminates with $exc = \mathbf{skip}$

<code>return</code>	<code>throw return</code>
<code>break</code>	<code>throw break</code>
<code>continue</code>	<code>throw continue</code>
<code>exit</code>	<code>throw exit</code>

Figure 14.2: Syntactic sugars for commands throwing special exception labels.

and ends only when the body throws an exception other than **skip**. In Section 14.3.1 we present conditional and non-deterministic while loops as extra features of QEDPL encoded using the `loop` and `try-catch` statements.

The parallel composition statement `dopar{ $S_1$ }with{ $S_2$ }` is used to create new threads to execute S_1 and S_2 in parallel. The `dopar` statement provides a structured notion of parallelism in which the parent thread waits until both its children threads terminate.

14.1.1 Exception Handling

The exception handling mechanism in QEDPL uses the exception labels Lbl defined in Section 12.4.4 and l ranges over labels. QEDPL programs use two statements for throwing and handling exceptions: the `throw  $l$` command and the `try { $\hat{S}_1$ } catch( $l$ ){ $\hat{S}_2$ }` statement. While the former is used to throw an exception l by setting the `exc` variable to the label l , the latter is used to set up the handler $\hat{S}_2$ for the exception l if thrown in S_1 .

As long as no exception is thrown, `exc` stores the label **skip**. The labels **return**, **exit**, **break**, and **continue**, are used to implement features similar to ones described in Section 12.4.4. We refer to the `throw` commands using these labels with syntactic sugars in Figure 14.2. The command `return` is used to terminate the execution of the currently executing procedure; the exception **return** is caught and suppressed before the control flow passes to the caller. The command `exit` throws the exception **exit**; this exception is assumed to be never caught, and thus, causes the currently executing thread to skip the rest of its statement and terminate. The command `break` and `continue` are used to redirect control flow in the while loops, as explained in Section 14.3.1.

The statement `try { $\hat{S}_1$ } catch( $l$ ){ $\hat{S}_2$ }` is used to define a handler for the exception labeled l . If the execution of $\hat{S}_1$ ends with `exc =  $l$` , then the `catch` part suppresses this exception by

setting *exc* to **skip** and starts evaluating the statement $\hat{S}_2$; otherwise the **catch** part of the statement is skipped and *exc* keeps the value *l*. For a sequence of exceptions $l_1, \dots, l_n$ we use the syntactic sugar `try { $S_1$ } catch( $l_1, l_2$ ) { $S_2$ }` for the following statement:

```

1 try {
2   try {
3      $S_1$ 
4   } catch( $l_1$ ) { $S_2$ }
5 } catch( $l_2$ ) { $S_2$ }
```

where it is assumed that S_2 will not throw l_1 or l_2 . This encoding is extended to an arbitrary number of labels: `try { $S_1$ } catch( $l_1, \dots, l_n$ ) { $S_2$ }`. The statement `try { $\hat{S}_1$ } catch { $\hat{S}_2$ }` is used to define a handler for any exception other than **skip**, so $\hat{S}_2$ is executed as long as $\hat{S}_1$ does not end with *exc* = **skip**.

14.2 Denotational Semantics

In this section, we give a formal, denotational semantics to QEDPL programs: We translate every QEDPL program $\hat{P}$ to an ACTIONPL program P . Then, the execution model, program states, and runs, of $\hat{P}$ are explained by those of P . In summary, every run of the program contains a set of threads (*Tid*). The run starts with the single thread t_{main} calling the procedure *Main*. The new threads are created by a structured parallelism operator. Every run either blocks, or terminates at a final or error state. See Section 12.4.5 for more information on the operational semantics of ACTIONPL.

As ACTIONPL and QEDPL programs share the top-level elements and use a common syntax for them in the program text, the difference remains is in the syntax of the statements in procedure bodies. In the following, we define the compilation operator $Compile(\cdot) : \hat{Stmt} \rightarrow Stmt$, which takes a QEDPL statement and returns an ACTIONPL statement. This is also extended to programs. In particular, so $Compile(\hat{P})$, the translation of a QEDPL program $\hat{P} = \langle Type_{\hat{P}}, Function_{\hat{P}}, Constant_{\hat{P}}, Axiom_{\hat{P}}, Global_{\hat{P}}, Proc_{\hat{P}} \rangle$ to ACTIONPL, yields the program $P = \langle Type_{\hat{P}}, Function_{\hat{P}}, Constant_{\hat{P}}, Axiom_{\hat{P}}, Global_{\hat{P}}, \tilde{Proc}_P \rangle$ where $\tilde{Proc}_P$ is obtained by replacing $Body_{\mathbf{p}}$ with $Compile(Body_{\mathbf{p}})$ for every procedure $\mathbf{p} \in Proc_{\hat{P}}$. We

explain $Compile(\cdot)$ in two phases: We first show how to translate an atomic statement to a gated action in Section 14.2.1, using which we then show the translation procedure for compound statements in Section 14.2.2.

14.2.1 Translating Atomic Statements

Given an atomic statement $\hat{\alpha}$, which is either a command c or $\mathbf{atomic}\{\bar{S}\}$, $Compile(\hat{\alpha})$ produces a gated action as defined in Section 12.3.1. Figure 14.3 shows the translation of an atomic statement $\mathbf{atomic}\{\bar{S}\}$ to a gated action and Figure 14.4 gives example translations. As for a command C , $Compile(C)$ is equivalent to $Compile(\mathbf{atomic}\{C\})$.

The gated action is computed in two phases. The first part of Figure 14.3 computes $\mathbf{gt}(\hat{\alpha})$, the gate of the action by propagating the assertions in the atomic statement back to the beginning of the statement. In the figure, $\mathbf{wp}(\bar{S}, \Phi, L)$ is the weakest-liberal pre-condition of a straight statement $\bar{S}$ with respect to a state predicate Φ . L is used for capturing the exception handling in the atomic statement and is explained below.

The second part of Figure 14.3 computes $\mathbf{tr}(\hat{\alpha})$, the transition predicate of the gated action. The transition predicate of a straight statement $\bar{S}$ is computed using the $\mathbf{xp}$ operator. $\mathbf{xp}(\bar{S}, \mathcal{T}, L)$ is the strongest transition expressing the composition of $\bar{S}$ and the transition predicate $\mathcal{T}$ under the assumption that all the assertions in the statement are satisfied. Because of this assumption, the definition of $\mathbf{xp}$ for **assume** and **assert** are equivalent.

While computing $\mathbf{gt}(\bar{S})$ and $\mathbf{tr}(\bar{S})$, in order to capture the control flow due to throwing and handling exception, we use a function $L : Lbl \rightarrow BExpr$. L is given as an argument to $\mathbf{wp}$ and $\mathbf{xp}$. L maps exception labels to state predicates when we are computing $\mathbf{wp}$, and to transition predicates when we are computing $\mathbf{xp}$. $L(l)$ is used when a **throw** l command is seen. Unless no **throw** command is seen, L is not involved in the computation of $\mathbf{wp}$ and $\mathbf{xp}$.

We express L in the lambda notation: $\lambda x.B$ defines a function that will be evaluated to $B[y/x]$ when called as $(\lambda x.B)(y)$. The expression $\mathbf{ite}(B, E_1, E_2)$ evaluates to E_1 if B evaluates to true, and to E_2 otherwise.

The weakest pre-condition ($\mathbf{wp}$) of a statement $\hat{S}$ is computed by backward reasoning in which assertions in the sub-statements of $\hat{S}$ are propagated towards the beginning of $\hat{S}$. At every point during this computation, (1) Φ represents the state predicate computed for the part of $\hat{S}$ that come (in the control flow) after the current sub-statement of $\hat{S}$, given as the

$$\text{Compile}(\text{atomic } \{\bar{S}\}) = \text{gt}(\bar{S}) \triangleright \text{tr}(\bar{S})$$

Computing the gate predicate:

$$\begin{aligned} \text{gt}(\bar{S}) &= \text{wp}(\bar{S}, \text{true}, \lambda x. \text{true}) \\ \text{wp}(\bar{S}_1; \bar{S}_2, \Phi, L) &= \text{wp}(\bar{S}_1, \text{wp}(\bar{S}_2, \Phi, L), L) \\ \text{wp}(\text{assume } B, \Phi, L) &= B \Rightarrow \Phi \\ \text{wp}(\text{assert } B, \Phi, L) &= B \wedge \Phi \\ \text{wp}(v := E, \Phi, L) &= \Phi[E/v] \\ \text{wp}(\text{havoc } v, \Phi, L) &= \forall v. \Phi \\ \text{wp}(\text{if}(g) \{\bar{S}\} \Phi, L) &= \text{wp}(\text{if}(g) \{\bar{S}\} \text{ else } \{\text{skip}\}, \Phi, L) \\ \text{wp}(\text{if}(\ast) \{\bar{S}_1\} \text{ else } \{\bar{S}_2\}, \Phi, L) &= \text{wp}(\bar{S}_1, \Phi, L) \wedge \text{wp}(\bar{S}_2, \Phi, L) \\ \text{wp}(\text{if}(B) \{\bar{S}_1\} \text{ else } \{\bar{S}_2\}, \Phi, L) &= (B \Rightarrow \text{wp}(\bar{S}_1, \Phi, L)) \wedge (\neg B \Rightarrow \text{wp}(\bar{S}_2, \Phi, L)) \\ \text{wp}(\text{try } \{\bar{S}_1\} \text{ catch}(l) \{\bar{S}_2\}, \Phi, L) &= \text{wp}(\bar{S}_1, \Phi, \lambda x. \text{ite}(x = l, \text{wp}(\bar{S}_2, \Phi, L), L(x))) \\ \text{wp}(\text{try } \{\bar{S}_1\} \text{ catch } \{\bar{S}_2\}, \Phi, L) &= \text{wp}(\bar{S}_1, \Phi, \lambda x. \text{ite}(x \neq \mathbf{skip}, \text{wp}(\bar{S}_2, \Phi, L), L(x))) \\ \text{wp}(\text{throw } l, \Phi, L) &= L(l) \end{aligned}$$

Computing the transition predicate:

$$\begin{aligned} \text{tr}(\bar{S}) &= \text{xp}(\bar{S}, \langle \text{exc}' = \mathbf{skip} \rangle_{\text{exc}}, \lambda x. \langle \text{exc}' = x \rangle_{\text{exc}}) \\ \text{xp}(\bar{S}_1; \bar{S}_2, \mathcal{T}, L) &= \text{xp}(\bar{S}_1, \text{xp}(\bar{S}_2, \mathcal{T}, L), L) \\ \text{xp}(\text{assume } B, \mathcal{T}, L) &= B \wedge \mathcal{T} \\ \text{xp}(\text{assert } B, \mathcal{T}, L) &= B \wedge \mathcal{T} \\ \text{xp}(v := E, \mathcal{T}, L) &= \mathcal{T}[E/v] \\ \text{xp}(\text{havoc } v, \mathcal{T}, L) &= \exists v. \mathcal{T} \\ \text{xp}(\text{if}(g) \{\bar{S}\}, \mathcal{T}, L) &= \text{xp}(\text{if}(g) \{\bar{S}\} \text{ else } \{\text{skip}\}, \mathcal{T}, L) \\ \text{xp}(\text{if}(\ast) \{\bar{S}_1\} \text{ else } \{\bar{S}_2\}, \mathcal{T}, L) &= \text{xp}(\bar{S}_1, \mathcal{T}, L) \vee \text{xp}(\bar{S}_2, \mathcal{T}, L) \\ \text{xp}(\text{if}(B) \{\bar{S}_1\} \text{ else } \{\bar{S}_2\}, \mathcal{T}, L) &= (B \wedge \text{xp}(\bar{S}_1, \mathcal{T}, L)) \vee (\neg B \wedge \text{xp}(\bar{S}_2, \mathcal{T}, L)) \\ \text{xp}(\text{try } \{\bar{S}_1\} \text{ catch}(l) \{\bar{S}_2\}, \mathcal{T}, L) &= \text{xp}(\bar{S}_1, \mathcal{T}, \lambda x. \text{ite}(x = l, \text{xp}(\bar{S}_2, \mathcal{T}, L), L(x))) \\ \text{xp}(\text{try } \{\bar{S}_1\} \text{ catch } \{\bar{S}_2\}, \mathcal{T}, L) &= \text{xp}(\bar{S}_1, \mathcal{T}, \lambda x. \text{ite}(x \neq \mathbf{skip}, \text{xp}(\bar{S}_2, \mathcal{T}, L), L(x))) \\ \text{xp}(\text{throw } l, \mathcal{T}, L) &= L(l) \end{aligned}$$

Figure 14.3: Compiling atomic statements to gated actions.

$$\begin{aligned}
\text{Compile}(\text{assert } B) &= B \triangleright \langle \text{true} \rangle_{\emptyset} \\
\text{Compile}(\text{assert false}) &= \text{false} \triangleright \langle \text{true} \rangle_{\emptyset} \\
\text{Compile}(\text{assume } B) &= \text{true} \triangleright \langle B \rangle_{\emptyset} \\
\text{Compile}(\text{assume true}) &= \text{true} \triangleright \langle \text{true} \rangle_{\emptyset} \\
\text{Compile}(v := E) &= \text{true} \triangleright \langle v' = E \rangle_v \\
\text{Compile}(\text{havoc } v) &= \text{true} \triangleright \langle \text{true} \rangle_v \\
\text{Compile}(\text{atomic } \{\text{assert } B; v := E\}) &= B \triangleright \langle v' = E \rangle_v \\
\text{Compile}(\text{atomic } \{\text{assume } B; v := E\}) &= \text{true} \triangleright \langle B \wedge v' = E \rangle_v \\
\text{Compile}(\text{atomic } \{v := E; \text{assert } B\}) &= B[E/v] \triangleright \langle v' = E \rangle_v \\
\text{Compile}(\text{atomic } \{v := E; \text{assume } B\}) &= \text{true} \triangleright \langle v' = E \wedge B[v'/v] \rangle_v
\end{aligned}$$

Figure 14.4: Example translations from QEDPL to ACTIONPL.

first argument of **wp**, and (2) $L(l)$ stores the state predicate to be used (as explained below) when a **throw** l command is seen during the computation. Thus, L is not used as long as no **throw** command is seen. For all $l \in Lbl$, $L(l)$ is initialized to **true** and recomputed, using S_2 and the current value of Φ , when processing a statement **try** $\{\bar{S}_1\}$ **catch**(l) $\{\bar{S}_2\}$. When we see **throw** l , which indicates a redirection in the control flow towards a **catch** statement for l , the current value of Φ is ignored and the pre-computed predicate in $L(l)$ is propagated as the result of the **wp** computation for the current sub-statement.

The transition predicate (**xp**) of a statement $\hat{S}$ is also computed by backward reasoning similarly to the computation of **wp**. At every point $\mathcal{T}$ represents the transition predicate computed for the sub-statements that come after the current sub-statement. $L(l)$ stores the transition predicate to be used when a **throw** l is seen. For all $l \in Lbl$, $L(l)$ is initialized to $\langle exc' = x \rangle_{exc}$, which represents a transition that sets exc to x and leaves all other variables unchanged.

14.2.2 Translating Compound Statements

Figure 14.5 shows the translation of a QEDPL compound statement to an ACTIONPL statement. We translate **if** statements to non-deterministic choice ($\square$), **dopar** to parallel composition ($\parallel$) and **loop** to S° . The one-to-one correspondence between ACTIONPL and

$$\begin{aligned}
\text{Compile}(\text{call } \ell^* := \mathbf{p}(E_\ell^*)) &= \ell^* := \mathbf{p}(E_\ell^*) \\
\text{Compile}(S_1; S_2) &= \text{Compile}(S_1) ; \text{Compile}(S_2) \\
\text{Compile}(\text{if}(\ast)\{S_1\}\text{else}\{S_2\}) &= \text{Compile}(S_1) \sqcap \text{Compile}(S_2) \\
\text{Compile}(\text{if}(\ast)\{S\}) &= \text{Compile}(\text{if}(\ast)\{S\}\text{else}\{\text{skip}\}) \\
\text{Compile}(\text{if}(B)\{S_1\}\text{else}\{S_2\}) &= (\text{Compile}(\text{assume } B); \text{Compile}(S_1)) \\
&\quad \sqcap (\text{Compile}(\text{assume } \neg B); \text{Compile}(S_2)) \\
\text{Compile}(\text{if}(B)\{S\}) &= \text{Compile}(\text{if}(B)\{S\}\text{else}\{\text{skip}\}) \\
\text{Compile}(\text{loop}\{S\}) &= \text{Compile}(S)^\odot \\
\text{Compile}(\text{try}\{S_1\}\text{catch}(l)\{S_2\}) &= \text{Compile}(S_1) ; l : \text{Compile}(S_2) \\
\text{Compile}(\text{try}\{S_1\}\text{catch}(l_1, \dots, l_n)\{S_2\}) &= \text{Compile}(S_1) ; l_1 : \text{Compile}(S_2) ; \dots ; l_n : \text{Compile}(S_2) \\
\text{Compile}(\text{try}\{S_1\}\text{catch}\{S_2\}) &= \text{Compile}(S_1) ; \ast : \text{Compile}(S_2) \\
\text{Compile}(\text{dopar}\{S_1\}\text{with}\{S_2\}) &= \text{Compile}(S_1) \parallel \text{Compile}(S_2)
\end{aligned}$$

Figure 14.5: Compiling compound statements to ACTIONPL statements.

QEDPL statements makes this translation straightforward. A $\text{try}\{\hat{S}_1\}\text{catch}(l)\{\hat{S}_2\}$ statement is translated using labeled statements: The translation of the body $\hat{S}_1$ is followed by the translation of the handler $\hat{S}_2$ labeled with the exception label l . A $\text{try}\{\hat{S}_1\}\text{catch}\{\hat{S}_2\}$ statement is translated similarly now using the $\ast : S$ statement.

Let $\text{Compile}(\hat{P}) = P$. The above translation procedure indicates that there is a bijection between the atomic statements in $\text{Atoms}(\hat{P})$ and the gated actions in $\text{Atoms}(P)$. In addition, there is a trivial correspondence between the compound statements in $\hat{P}$ and P . In other words, each statement in $\hat{P}$ is translated to a statement in P that will exhibit the informal semantics of the statement described in Section 14.1.

14.3 Extra Features

Having presented the syntax and semantics of QEDPL, we now show the encoding of three additional features: while loops, maps, and method-less classes. We encode these features using syntactic sugars for the core language constructs, and thus, they do not add to the semantics of the language.

It is also possible to add these features to ACTIONPL. However, they do not add any benefit to the formal explanation of the techniques in QED, but they are for simplifying to write programs in practice. In addition, we found it easier to describe the encoding and usage of these features with the syntax of QEDPL statements.

14.3.1 While Loops

Conditional and non-deterministic while loops are more convenient to use than the looping construct `loop`. In the following, we model while loops using `loop`.

The syntax for the conditional while loop is as follows:

$$\text{while}(B_\ell) \{\hat{S}\}$$

At every iteration (including the first one) it is checked that whether B evaluates to `true`. If so, the next iteration starts, otherwise the execution of the loop terminates. Given that, $\text{while}(\text{true})\{\hat{S}\}$ is equivalent to $\text{loop}\{\hat{S}\}$.

The syntax for the non-deterministic while loop is as follows:

$$\text{while}(\ast) \{\hat{S}\}$$

At the end of every iteration, it is non-deterministically decided to continue with the next iteration or to terminate the loop.

The commands `break` (sugar for `throw break`) and `continue` (sugar for `throw continue`) are used to change the control flow in the while loops unconditionally. The former causes the loop to terminate, and the latter causes only the current iteration to terminate and directs the control flow to the beginning of the loop. The while loops are encoded considering the use of these commands. We syntactically enforce that `break` and `continue` are not used outside the body of loops.

Encoding While Loops

The statement $\text{while}(B_\ell) \{\hat{S}\}$ is equivalent to the following (desugared) statement:

```

1 try {
2   loop {
```

```

3    try {
4        if( $\neg B_\ell$ ) {throw break} else { $\hat{S}$ }
5    } catch(continue) {skip}
6 }
7 } catch(break) {skip}

```

In the desugared form, the body of **loop** iterates until the guard condition is not satisfied or $\hat{S}$ throws an exception other than **continue** (for example, **break** or **return**) (line 4). If S throws **break**, the outer exception handler at line 7 catches and suppresses this exception, and the execution continues with the next statement. Note that **skip** is equivalent to **assume true**, which has no side effect. If S executes **continue**, the exception **continue** is caught by the inner handler at line 5. In this case the loop continues to execute with the next iteration. If S throws an unhandled exception that is neither **break** nor **continue** (for example **return**), that exception is not treated specially and not suppressed within the encoding above. Thus, the exception causes the loop to terminate and is propagated to the statement next to the while loop in the control flow.

The desugared form of the statement **while**($*$) { $\hat{S}$ } is similar to the conditional while loop, except that the conditional if statement at line 4 is replaced by a non-deterministic if:

```

1 try {
2   loop {
3     try {
4       if( $*$ ) {throw break} else { $\hat{S}$ }
5     } catch(continue) {skip}
6   }
7 } catch(break) {skip}

```

14.3.2 Maps

Maps and records allow us to represent data structures in a more readable form. Here, we show encoding of maps, and we express records using maps. A program implicitly defines

map types over the existing types:

$$\text{Type : } \mathbf{t} ::= \mathbf{int} \mid \mathbf{bool} \mid [\vec{\mathbf{t}}]\mathbf{t} \mid \text{Custom type name}$$

A map type of arity n is expressed as $[\mathbf{t}_1, \dots, \mathbf{t}_n]\mathbf{t}$, where the i^{th} index ranges over $\mathbf{t}_i$ and each entry contains a value of type $\mathbf{t}$. A variable v of type $[\mathbf{t}_1, \dots, \mathbf{t}_n]\mathbf{t}$ is declared as follows:

$$\text{var } v : [\mathbf{t}_1, \dots, \mathbf{t}_n]\mathbf{t};$$

Given the above declaration, a map access expression $v[E_1, \dots, E_n]$ (given that $E_1, \dots, E_n$ evaluate to indices of type $\mathbf{t}_1, \dots, \mathbf{t}_n$, respectively) evaluates to a value of type $\mathbf{t}$. A map assignment command $v[E_1, \dots, E_n] := E$ assigns to v a new map that agrees with the old value of v on all sequences of indices except for the indices whose values are given by $E_1, \dots, E_n$ at the moment of the assignment.

Encoding Maps

For each map type $[\mathbf{t}_1, \dots, \mathbf{t}_n]\mathbf{t}$ that appears in the program, the following top level elements are added to the program (let $\#$ denote the type $[\mathbf{t}_1, \dots, \mathbf{t}_n]\mathbf{t}$):

```

type       $\mathbf{t}_\#$ ;
function   $\text{select}_\#(\mathbf{t}_\#, \mathbf{t}_1, \dots, \mathbf{t}_n)$  returns  $(\mathbf{t})$ ;
function   $\text{store}_\#(\mathbf{t}_\#, \mathbf{t}_1, \dots, \mathbf{t}_n, \mathbf{t})$  returns  $(\mathbf{t}_\#)$ ;
axiom      $(\forall M : \mathbf{t}_\#, i_1 : \mathbf{t}_1, \dots, i_n : \mathbf{t}_n, x : \mathbf{t}. \text{select}_\#(\text{store}_\#(M, i_1, \dots, i_n, x), i_1, \dots, i_n) = x)$ ;
axiom      $(\forall M : \mathbf{t}_\#, i_1 : \mathbf{t}_1, \dots, i_n : \mathbf{t}_n, j_1 : \mathbf{t}_1, \dots, j_n : \mathbf{t}_n, x : \mathbf{t}. (i_1 \neq j_1 \vee \dots \vee i_n \neq j_n) \\ \Rightarrow \text{select}_\#(\text{store}_\#(M, i_1, \dots, i_n, x), j_1, \dots, j_n) = \text{select}_\#(M, j_1, \dots, j_n))$ ;

```

The type $\mathbf{t}_\#$ allows us to represent a map as a plain value. The functions $\text{select}_\#$ and $\text{store}_\#$ and the two axioms are used to formalize an (incomplete) logic of maps [102]. Intuitively, the function $\text{select}_\#$ takes a map M as its first argument and a sequence of indices $\vec{i}$ and returns the value stored at $M[\vec{i}]$; the function $\text{store}_\#$ takes a map M , a sequence of indices $\vec{i}$, and a value to be stored at $M[\vec{i}]$, and returns the updated map $M[\vec{i} \mapsto x]$. The axioms state that the plain type $\mathbf{t}_\#$ correctly represents a map of type $[\mathbf{t}_1, \dots, \mathbf{t}_n]\mathbf{t}$.

Given the definitions above, variable definitions of map type, map access expressions, and assignments to maps are encoded as follows:

$$\text{var } v : [\mathbf{t}_1, \dots, \mathbf{t}_n] \mathbf{t}; \triangleq \text{var } v : \mathbf{t}_{\#}; \quad (14.2)$$

$$v[E_1, \dots, E_n] \triangleq \text{select}_{\#}(v, E_1, \dots, E_n) \quad (14.3)$$

$$v[E_1, \dots, E_n] := E \triangleq v := \text{store}_{\#}(v, E_1, \dots, E_n, E) \quad (14.4)$$

14.3.3 Records

Records extend the programming language with structured data types. A record is defined as a top-level element using the following syntax:

Record : *record* ::= **record** **r** { *field** }

Field : *field* ::= **f** : **t**;

Field read : *fread* ::= *E.f*

Field write : *fwrite* ::= *E.f* := *E*

Given a record definition, **record** **r** {**f**₁ : **t**₁, ..., **f**_n : **t**_n}, **r** is the name and **f**₁, ..., **f**_n are the fields of the record.

Once defined, the record **r** exports a new type with the same name **r**. A global variable, for example, that points to a record of type **r** can be defined as follows:

var *x* : **r**;

Fields of a record can be read from and written to using the dotted notation. A field access expression is denoted *E.f* where *E* evaluates to a value of a record type **r**, and **f** is a field defined in **r**.

The following example demonstrates a *field read* from a record pointed by **y** field and a *field write* to a record pointed by **x**:

```
record R { data: int; }
procedure foo(x,y: R) {
  x.data := y.data + 1;
}
```

We do not support procedures/methods above as members of records. Records in QEDPL are analogous to a method-less class in Dafny or Java, in the following manners:

- Each record models a dynamically allocated region on memory. Declaring a variable ℓ of type $\mathbf{r}$ does not allocate space for the record. The variable does not represent the record itself, but a reference to a record. Thus, more than one variable of type $\mathbf{r}$ may refer to the same record. An assignment $\ell_1 := \ell_2$ where both ℓ_1 and ℓ_2 are of type $\mathbf{r}$ makes the former to point to the same record pointed by the latter.
- We provide two primitives to allocate and free memory regions for records: $\ell := \text{New } \mathbf{r}$ allocates a new record of type $\mathbf{r}$ and assigns its reference to the variable ℓ (of type $\mathbf{r}$). $\text{Free } \ell$ deallocates the region pointed by the variable ℓ . At any point, a variable of $\mathbf{r}$ may store a reference to an allocated or unallocated record, or *null*.
- A procedure may have an input variable of type $\mathbf{r}$, and records are passed to procedures *by reference*. In the following example, the variables `r1` and `r2` dereference the same record, say `r`. Thus when the call to `bar` returns, the variable `l` is written the value 2, which was written to `r.data` via the reference `r2`.

```
record R { data: int; }
procedure foo(r1: R) {
  r1.data := 1;
  call bar(r1);
  l := r1.data;
}
procedure bar(r2: R) {
  r2.data := 2;
}
```

Encoding Records

Each record definition `record  $\mathbf{r}$  { $\mathbf{f}_1 : \mathbf{t}_1, \dots, \mathbf{f}_n : \mathbf{t}_n$ }` implicitly creates a type $\mathbf{r}$ and n global variables:

```

type  r;
var   f1 : [r]t1;
...   ...
var   fn : [r]tn;

```

Each global variable $\mathbf{f}_i$ represents a separate map, which takes a variable of type $\mathbf{r}$ and returns the value of the $\mathbf{f}_i$ field of the record. This models the case that the storage space for each field $\mathbf{f}_i$ is separate from other fields of $\mathbf{r}$. Therefore, the assignment $x.\mathbf{f}_i := E$ (where x is of type $\mathbf{r}$) may not modify other fields of x and other records that is pointed by x . In this sense, we rule out possible aliasing relationships between different fields of the same record. For simplicity we assume that the field names of a record are unique among all other variables and records (otherwise, we can also rename the field maps to make them unique).

We add to each record $\mathbf{r}$ a field *alloc* of type *AllocType* in order to encode the allocation/deallocation of a record $\mathbf{r}$. The type *AllocType* is defined as follows to represent the set $\{Alloc, Dealloc, Null\}$:

```

type  AllocType;
const  unique Alloc : AllocType;
const  unique Dealloc : AllocType;
const  unique Null : AllocType;
axiom  ( $\forall a : \text{AllocType}. (a = \text{Alloc}) \vee (a = \text{Dealloc}) \vee (a = \text{Null})$ );

```

Consider a variable x of type $\mathbf{r}$. At any point, x may store a reference to an allocated or unallocated record, or *null*. This information is stored in the *alloc* field of the record. If $x.\text{alloc} = \text{Alloc}$ then x points to an allocated record, and if $x.\text{alloc} = \text{Dealloc}$, then x points to a deallocated record. The condition $x.\text{alloc} = \text{Null}$ represents that x is not associated with a valid record. Notice that, the field access is a total operation for all variables of type $\mathbf{r}$, so one can access the fields of an unallocated records and even of *null*, although these should be considered a bug in the program.

Given the *alloc* field, the **New** and **Free** command are desugared to the following atomic statements:

$$v := \text{New } \mathbf{r} \triangleq \text{atomic } \{\text{havoc } v ; \text{assume } v.\text{alloc} = \text{Dealloc} ; v.\text{alloc} := \text{Alloc}\} \quad (14.5)$$

$$\text{Free } v \triangleq \text{atomic } \{\text{assert } v.\text{alloc} = \text{Alloc} ; v.\text{alloc} := \text{Dealloc}\} \quad (14.6)$$

The **New** command first assigns an unallocated record reference of v , and then makes this record allocated for the program by setting its *alloc* field. The **Free** command takes a variable that points to an allocated record, and then it makes the record unallocated by setting its *alloc* field.

Since our type system does not allow type polymorphism, we have to model *null* separately for each record $\mathbf{r}$. We do this by declaring a unique constant of type $\mathbf{r}$ whose *alloc* field is *Null* as follows:

```
record MyRec { ... }
const unique Null_MyRec : MyRec;
invariant (Null_Rec.alloc == Null);
invariant (forall x : MyRecord :: (x.alloc == Null) ==> (x == Null_MyRec));
```

The first invariant above states that *Null_MyRec* points to a *null* record, and the second invariant states that the *null* record pointed by *Null_MyRec* is unique, i.e., there is no other record instance whose *alloc* field is *Null*.

We allow the code to access x only when $x.\text{alloc} = \text{Alloc}$. In order to ensure this, we atomically insert the assertion **assert** $v.\mathbf{f} = \text{Alloc}$ before every access to $v.\mathbf{f}$. For example, we implicitly rewrite the assignment $v.\mathbf{f} := E$ to the following atomic statement:

$$\text{atomic } \{\text{assert } v.\mathbf{f} = \text{Alloc} ; v.\mathbf{f} := E\}$$

Therefore, the user also have to discharge the assertions above to ensure that every field access is made to an allocated record.

Finally, we desugared a field read expression $E.\mathbf{f}$ to the map access expression $\mathbf{f}[E]$, and an assignment to field $E_1.\mathbf{f} = E_2$ to the map assignment $\mathbf{f}[E_1] = E_2$.

This chapter presented the syntax of the QEDPL language and gave semantics to QEDPL statements by compiling them to ACTIONPL statements. In the next chapter, we will present

the proof transformations over QEDPL programs. The relationship between QEDPL and ACTIONPL will help to describe these transformations by connecting each to a proof rule given in Chapter 13.

Chapter 15

THE PROOF TACTICS FOR QEDPL

Having introduced the QEDPL language in Chapter 14, in this chapter, we formalize the notion of proving a program written in QEDPL. For this, we propose *proof tactics*, high-level transformations on QEDPL programs. Each proof tactic specifies an application of a QED proof rule presented in Chapter 13 at the QEDPL level. We prove a QEDPL program by applying a sequence of tactics to transform the program and discharging the assertions within the atomic statements of the final program. The soundness of tactic-based proofs is stated similarly to the theorems for proofs in ACTIONPL stated in Section 13.7. We reason about the soundness of each tactic using the underlying proof rule. While explaining each tactic, we point to the associated proof rule from Chapter 13. This chapter also presents several tactics that are not directly related to the core proof rules, but perform useful code transformations to simplify the application of the other tactics.

Outline. Section 15.1 revisits our verification approach and formalizes the proof of a QEDPL program. Section 15.2 introduces the proof tactics. Section 15.3 presents additional tactics that are not related to any proof rules but we found useful in our experience with proving benchmark programs. Section 15.4 gives the proof of a theorem that states the soundness of the approach by connecting proof tactics with the core rules of the QED proof system.

15.1 Proof Transformations on QEDPL

In this section, we revisit the formalization of a QED proof. A QED proof consists of rewriting the input program, denoted $\hat{P}_1$, iteratively so that, a program $\hat{P}_n$ that can be verified by sequential reasoning methods is obtained. A *proof state* is a pair $(\hat{P}, \mathcal{I})$, where $\hat{P}$ is a program written in QEDPL, and $\mathcal{I}$ is an *inductive invariant* of the program. $\mathcal{I}$ represents the set of program states at which the program is expected to start and run correctly. We

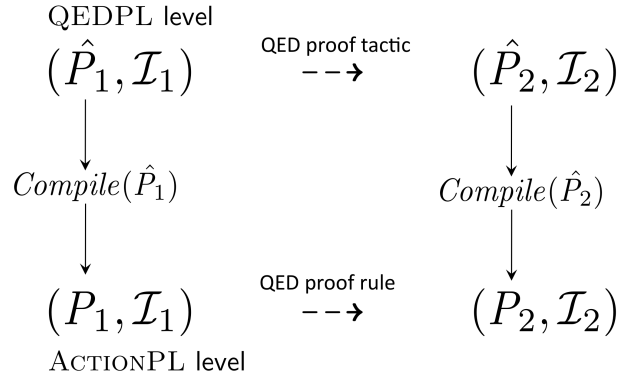


Figure 15.1: The relationship between QED proof tactics and rules.

require that for every proof state $(\hat{P}, \mathcal{I})$, all the atomic statement in $Atoms(P)$ preserve $\mathcal{I}$ (see Section 15.2.1 for the formal definition).

Formally, a proof is expressed as $(\hat{P}_1, \text{true}) \dashrightarrow (\hat{P}_2, \mathcal{I}_2) \dashrightarrow \dots \dashrightarrow (\hat{P}_n, \mathcal{I}_n)$. Each step $(\hat{P}_i, \mathcal{I}_i) \dashrightarrow (\hat{P}_{i+1}, \mathcal{I}_{i+1})$, is governed by one of the proof tactics presented in this chapter. The proof aims to discharge assertions in atomic actions, which requires that the assertions be proven valid under the current invariant. The proof succeeds when all the assertions in $\hat{P}_n$ are validated and eliminated.

Soundness

We now lift the soundness theorem of the proof calculus given in Section 13.7 to proofs based on tactics. Before giving the theorems, we introduce an abstraction relation between two QEDPL programs. For this, we make use of the operator $Compile(\cdot)$ defined in Section 14.2, and the abstraction relation $(\preccurlyeq)$ between ACTIONPL programs (Definition 18).

The soundness of the proof tactics is stated similarly to Theorem 4.

Theorem 7 (Preservation of $\preccurlyeq$ by the proof tactics):

Let $(\hat{P}_1, \mathcal{I}_1) \dashrightarrow (\hat{P}_2, \mathcal{I}_2)$ be a legal proof step. Then $\mathcal{I}_2 \vdash Compile(\hat{P}_1) \preccurlyeq Compile(\hat{P}_2)$ holds. ■

We give the proof of this theorem in Section 15.4. The proof connects the proof tactics presented in this chapter to the rules of the QED proof system presented in Chapter 13. Given the application of tactic in $(\hat{P}_1, \mathcal{I}_1) \dashrightarrow (\hat{P}_2, \mathcal{I}_2)$, we argue that there exists a proof step

$(\text{Compile}(\hat{P}_1), \mathcal{I}_1) \dashrightarrow (\text{Compile}(\hat{P}_2), \mathcal{I}_2)$ governed by a QED proof rule. This relationship between proof steps at the QEDPL and ACTIONPL levels is depicted in Figure 15.1. While presenting each tactic, we indicate the corresponding proof rule that is used to justify the soundness of the tactic.

The following theorem generalizes the above theorem to an arbitrary number of proof steps.

Theorem 8 (Soundness theorem of QED for QEDPL programs):

Let $(\hat{P}_1, \mathcal{I}_1) \dashrightarrow \dots \dashrightarrow (\hat{P}_n, \mathcal{I}_n)$ be a sequence of legal proof steps. Let $\mathcal{I}$ be a state predicate over $\text{Global}_{\hat{P}_1}$ such that for all $i \in [1, n]$, $\mathcal{I} \Rightarrow \mathcal{I}_i$. Then $\mathcal{I} \vdash \text{Compile}(\hat{P}_1) \preceq \text{Compile}(\hat{P}_2)$ holds. ■

PROOF. The proof follows from Theorem 7 and Theorem 5. □

15.2 Proof Tactics

In this section we introduce proof tactics for the QEDPL language. In contrast with the presentation of proof rules, we define each tactic in a less formal style in the following form:

Tactic X (Usage – RULE) *Description of the operation of the tactic.* ■

In the definition, **X** is the unique id of the tactic. The RULE following the usage indicates the QED proof rule from Chapter 13 this tactic is based on. The body of the tactic describes the operation of the tactic on the current proof state referred to by $(\hat{P}, \mathcal{I})$ and under which conditions (on $(\hat{P}, \mathcal{I})$) the tactic must be applied. The tactic describes the new form of the program and the invariant after the transformation by referring to elements of the proof state $(\hat{P}, \mathcal{I})$ before the transformations are applied. We use the following conventions in this description:

- We denote atomic statements in the form **atomic** $\{\bar{S}\}$. Application of tactics can be adapted to individual commands C by considering C as **atomic** $\{C\}$.
- Replacing the current invariant $\mathcal{I}_1$ with $\mathcal{I}_2$ refers to changing the existing top-level definition, **invariant** $\mathcal{I}_1$ with **invariant** $\mathcal{I}_2$ in the program text.
- Replacing the current invariant $\mathcal{I}$ with $\mathcal{I} \wedge \phi$ refers to adding the new top-level definition, **invariant** ϕ to the program text.

- Adding a new global variable x to $Global_P$ refers to adding the top-level definition $\text{var } x : \mathbf{t}$ to the program text. We assume that the type $\mathbf{t}$ is given along with the variable.
- Adding a new local variable ℓ to $Private_P$ refers to adding the definition $\text{var } \ell : \mathbf{t}$ to the definition of the procedure $\mathbf{p}$. We assume that the type $\mathbf{t}$ is given along with the variable.
- Hiding an existing global variable x from $Global_P$ refers to removing the top-level definition $\text{var } x : \mathbf{t}$ from the program text.
- Hiding an existing local variable ℓ from $Private_P$ refers to removing the definition $\text{var } \ell : \mathbf{t}$ from the definition of the procedure $\mathbf{p}$.

Figure 15.2 lists all the proof tactics introduced in this chapter.

Tactic 1	Add new invariant – ADD-INVARIANT	230
Tactic 2	Abstraction – ABSTRACT	230
Tactic 3	Add assertion – ADD-ASSERTION	231
Tactic 4	Read abstraction of v 1 – ABSTRACT-READ-I	232
Tactic 5	Read abstraction of v 2 – ADD-LOCAL-VARIABLE	232
Tactic 6	Write abstraction of v – ABSTRACT-WRITE	233
Tactic 7	Discharge all assertions – RELAX-I	233
Tactic 8	Discharge an assertion – RELAX-II	233
Tactic 9	Add global variable x – ADD-GLOBAL-VARIABLE	235
Tactic 10	Add local variable ℓ to $\mathbf{p}$ – ADD-LOCAL-VARIABLE	235
Tactic 11	Hide global variable x – HIDE-GLOBAL-VARIABLE	235
Tactic 12	Hide local variable ℓ from $\mathbf{p}$ – HIDE-LOCAL-VARIABLE	235
Tactic 13	Reduce sequential – REDUCE-SEQUENTIAL-RIGHT/LEFT	236
Tactic 14	Reduce conditional – REDUCE-CHOICE	237
Tactic 15	Reduce conditional with else – REDUCE-CHOICE	237
Tactic 16	Reduce try-catch – REDUCE-LABELED-RIGHT/LEFT	237
Tactic 17	Reduce try-catch any – REDUCE-LABELED-RIGHT/LEFT	238
Tactic 18	Reduce try-catch unconditionally	238
Tactic 19	Reduce try-catch any unconditionally	238
Tactic 20	Expand dopar statement – EXPAND-PARALLEL	239

Tactic 21	Reduce dopar statement – REDUCE-PARALLEL-LEFT/RIGHT	239
Tactic 22	Reduce loop – REDUCE-LOOP	240
Tactic 23	Reduce loop right – REDUCE-LOOP-RIGHT	241
Tactic 24	Reduce while – REDUCE-NONDET-LOOP	242
Tactic 25	Reduce while right – REDUCE-NONDET-LOOP-RIGHT	242
Tactic 26	Reduce loop left – REDUCE-NONDET-LOOP-LEFT	243
Tactic 27	Inline call – INLINE-CALL	243
Tactic 28	Reduce recursive procedures – REDUCE-PROCEDURE	244
Tactic 29	Peel out the final iteration of loop	245
Tactic 30	Peel out the final iteration of while (*)	246
Tactic 31	Peel out the final iteration of while (B_ℓ)	247
Tactic 32	Inject prefix to if (*)-else	250
Tactic 33	Inject prefix to if (*)	250
Tactic 34	Inject prefix to if (B_ℓ)-else	250
Tactic 35	Inject prefix to if (B_ℓ)	251
Tactic 36	Split if (B_ℓ)	251
Tactic 37	Extract assumption	252

Figure 15.2: The proof tactics for QEDPL

15.2.1 Invariant Introduction

The second component of the proof state, $\mathcal{I}$, is a state predicate that specifies the reachable states of the program $\hat{P}$. We say that $\mathcal{I}$ is an (inductive) invariant for $\hat{P}$, denoted $\hat{P} \hat{\trianglelefteq} \mathcal{I}$, if every atomic statement $\hat{\alpha}$ in $\hat{P}$ preserves $\mathcal{I}$. An atomic statement **atomic**{ $\bar{S}$ } preserves $\mathcal{I}$, denoted **atomic**{ $\bar{S}$ } $\hat{\trianglelefteq} \mathcal{I}$ if $\text{Compile}(\text{atomic}\{\bar{S}\}) \trianglelefteq \mathcal{I}$.

Each proof starts from the invariant **true**. A program invariant ϕ is introduced by strengthening the current invariant $\mathcal{I}$ and checking if $(\mathcal{I} \wedge \phi)$ is still an invariant for the program. The proof rule **INVARIANT** states the general form of updating $\mathcal{I}$, replacing it with a stronger invariant, and rule **ADD-INVARIANT** states a usual way of updating the invariant by conjoining a given predicate with the existing invariant. Here we give a tactic that formalizes the latter.

Tactic 1 (Add new invariant – ADD-INVARIANT):

Replace invariant $\mathcal{I}$ with $\mathcal{I} \wedge \phi$, if for every atomic statement $\text{atomic}\{\bar{S}\}$ in $\text{Atoms}(\hat{P})$ $\text{atomic}\{\bar{S}\} \hat{\hookrightarrow} \mathcal{I} \wedge \phi$ holds. ■

15.2.2 Abstraction

The abstraction tactics replace an atomic statement with another atomic statement that simulates the former. Note that, the purpose of abstraction in our approach is to rewrite the program with new actions that in principle leads to less interference with other threads than the existing actions. Thus, replacing an atomic statement with an abstract version of it is an essential proof step, which enables further application of reduction steps. The abstraction relation between atomic statements allows us to decide when we can replace an atomic statement with another.

Tactic 2 allows to replace an atomic statement $\text{atomic}\{\bar{S}_1\}$ with a more abstract version of the statement $\text{atomic}\{\bar{S}_2\}$. We require that the new atomic statement $\text{atomic}\{\bar{S}_2\}$ preserve the current invariant and that $\text{atomic}\{\bar{S}_2\}$ abstract $\text{atomic}\{\bar{S}_1\}$ from $\mathcal{I}$. The latter is denoted $\mathcal{I} \vdash \text{atomic}\{\bar{S}_1\} \hat{\approx} \text{atomic}\{\bar{S}_2\}$, and checked using the gated action translations of the atomic statements as explained below.

Recall that $\text{Compile}(\text{atomic}\{\bar{S}\})$ translates an atomic statement to a gated action in ACTIONPL. Using this operation and the relation $\preceq$ in Definition 6, we define the abstraction relation $\hat{\approx}$ between atomic statements of QEDPL:

Definition 19 (Abstraction between atomic statements):

Let $\text{atomic}\{\bar{S}_1\}$, $\text{atomic}\{\bar{S}_2\}$ be atomic statements and $\mathcal{I}$ be a state predicate. $\text{atomic}\{\bar{S}_2\}$ abstracts $\text{atomic}\{\bar{S}_1\}$ from $\mathcal{I}$, written $\mathcal{I} \vdash \text{atomic}\{\bar{S}_1\} \hat{\approx} \text{atomic}\{\bar{S}_2\}$, if $\mathcal{I} \vdash \text{Compile}(\text{atomic}\{\bar{S}_1\}) \preceq \text{Compile}(\text{atomic}\{\bar{S}_2\})$ holds. ■

Tactic 2 (Abstraction – ABSTRACT):

Replace occurrences of the atomic statement $\text{atomic}\{\bar{S}_1\}$ with the new statement $\text{atomic}\{\bar{S}_2\}$ if both of the following hold:

1. $\text{Var}(\text{atomic}\{\bar{S}_1\}) = \text{Var}(\text{atomic}\{\bar{S}_2\})$
2. $\text{atomic}\{\bar{S}_2\} \hat{\hookrightarrow} \mathcal{I}$
3. $(P, \mathcal{I}) \vdash \text{atomic}\{\bar{S}_1\} \hat{\approx} \text{atomic}\{\bar{S}_2\}$ ■

Common ways of abstracting an atomic statement are annotating the statement with assertions and weakening the transition predicate of the statement. In addition, abstraction is used to justify the soundness of the tactics for discharging assertions in atomic statements. In the following we give simpler tactics derived from Tactic 2 for these purposes.

Borrowing an Assertion

We use assertions to express our beliefs about the program state when $\text{atomic}\{\bar{S}\}$ is reached. Our method allows program transformations to proceed while the verification of the introduced assertion is deferred to later proof stages. Borrowing assertions in this manner (instead of proving them right away) is a powerful tool, since in the transformed program with larger atomic actions, validation of assertions can often be done automatically.

Let $\text{Compile}(\text{atomic}\{\bar{S}\}) = \phi \triangleright \tau$. When we transform $\text{atomic}\{\bar{S}\}$ to the statement $\text{atomic}\{\text{assert } \psi; \bar{S}\}$, the corresponding gated action is transformed to $(\phi \wedge \psi) \triangleright \tau$: the transformation strengthens the gate and does not modify the transition predicate. By the definition of the abstraction relation $\mathcal{I} \vdash \phi \triangleright \tau \preceq (\phi \wedge \psi) \triangleright \tau$ holds, and thus, replacing $\text{atomic}\{\bar{S}\}$ with $\text{atomic}\{\text{assert } \psi; \bar{S}\}$ is sound. Generalizing this reasoning, we now introduce a tactic to borrow an assertion anywhere in an atomic statement. To express the addition of an assertion to a statement, we define a statement context as follows:

Definition 20 (Context):

A context $\mathbf{C}$ is a statement with a hole and is recursively defined as follows:

$$\begin{aligned} \mathbf{C} ::= & [\] \mid \bar{S}; \mathbf{C} \mid \mathbf{C}; \bar{S} \mid \text{if}(g) \{ \bar{S} \} \text{ else } \{ \mathbf{C} \} \mid \text{if}(g) \{ \mathbf{C} \} \text{ else } \{ \bar{S} \} \\ & \mid \text{if}(g) \{ \mathbf{C} \} \mid \text{try } \{ \mathbf{C} \} \text{ catch}(l) \{ \bar{S} \} \mid \text{try } \{ \bar{S} \} \text{ catch}(l) \{ \mathbf{C} \} \\ & \mid \text{try } \{ \mathbf{C} \} \text{ catch} \{ \bar{S} \} \mid \text{try } \{ \bar{S} \} \text{ catch} \{ \mathbf{C} \} \end{aligned}$$

Then every straight statement $\bar{S}$ can be rewritten as $\mathbf{C}[\text{skip}]$ where skip indicates a place in the statement to insert an assertion.

Tactic 3 (Add assertion – ADD-ASSERTION):

Replace occurrences of the atomic statement $\text{atomic } \{ \mathbf{C}[\text{skip}] \}$ with the new statement $\text{atomic } \{ \mathbf{C}[\text{assert } B] \}$ where B is a state predicate over $\text{Var}(\text{atomic } \{ \mathbf{C}[\text{skip}] \})$. ■

Two example applications of the tactic –where the statement in (a) is replaced by the statement in (b)– are given below:

<code>atomic {</code>	<code>atomic {</code>	<code>atomic {</code>	<code>atomic {</code>
<code> x := x + 1;</code>	<code> assert x > 0;</code>	<code> if(x > 0){</code>	<code> if(x > 0){</code>
<code>}</code>	<code> x := x + 1;</code>	<code> o.f := x + 1;</code>	<code> assert(o.alloc==Alloc);</code>
	<code>}</code>	<code>}</code>	<code> o.f := x + 1;</code>
		<code>}</code>	<code>}</code>
			<code>}</code>
(a)	(b)	(a)	(b)

Read and Write Abstractions

Read and write abstractions are simple ways of abstracting an atomic statement by abstracting the value of a variable at the beginning or the end of the statement. By Definition 19, $\mathcal{I} \vdash \text{atomic}\{\bar{S}_1\} \hat{\approx} \text{atomic}\{\bar{S}_2\}$ holds after applying read and/or write abstractions presented below on $\text{atomic}\{\bar{S}_1\}$ to obtain $\text{atomic}\{\bar{S}_2\}$. In addition, we do not allow abstractions that violate the program invariant $\mathcal{I}$, thus $\text{atomic}\{\bar{S}_2\}$ is also required to preserve $\mathcal{I}$.

A read abstraction makes modifies an atomic statement $\text{atomic}\{\bar{S}\}$ so that all of the reads in $\bar{S}$ from a variable v return a nondeterministic value. Thus, $\text{atomic}\{\bar{S}\}$ becomes independent of the value of v when its execution starts. The following tactics are used for this purpose. Tactic 4 performs read abstraction in the trivial case that every execution of $\text{atomic}\{\bar{S}\}$ writes to v at least once and no assertion in $\text{atomic}\{\bar{S}\}$ refers to v . Tactic 5 uses a fresh local variable to represent the nondeterministic value to v until the first assignment to v .

Tactic 4 (Read abstraction of v 1 – ABSTRACT-READ-I):

Let $\mathbf{p}$ be the procedure containing $\text{atomic}\{\bar{S}\}$. Replace the atomic statement $\text{atomic}\{\bar{S}\}$ with $\text{atomic}\{\text{havoc } \ell; \bar{S}\}$ if all of the following hold:

1. $\text{atomic}\{\text{havoc } \ell; \bar{S}[\ell/v]\} \hat{\approx} \mathcal{I}$
2. Every execution path in $\bar{S}$ assigns to v .
3. $v \notin \text{fv}(B)$ for all assertions $\text{assert } B$ in $\bar{S}$. ■

Tactic 5 (Read abstraction of v 2 – ADD-LOCAL-VARIABLE):

Let $\mathbf{p}$ be the procedure containing $\text{atomic}\{\bar{S}\}$. Add a fresh local variable ℓ to $\text{Private}_{\mathbf{p}}$. Replace the atomic statement $\text{atomic}\{\bar{S}_1\}$ with $\text{atomic}\{\text{havoc } \ell; \bar{S}_2\}$ if $\text{atomic}\{\text{havoc } \ell; \bar{S}_2\} \hat{\approx} \mathcal{I}$, where

$\bar{S}_2 = \bar{S}_1[\ell \downarrow v]$, and $\bar{S}_1[\ell \downarrow v]$ denotes the substitution, in $\bar{S}_1$, of ℓ for every occurrences of v in all statements over all paths of up to (and including the right-hand-side of) the first assignment to v . ■

A write abstraction makes an atomic statement $\text{atomic}\{\bar{S}\}$ write a nondeterministic value to v before it finishes its execution. The following tactic is used for this purpose.

Tactic 6 (Write abstraction of v – ABSTRACT-WRITE):

Replace the atomic statement $\text{atomic}\{\bar{S}\}$ with $\text{atomic}\{\bar{S}; \text{havoc } v\}$ if $\text{atomic}\{\bar{S}; \text{havoc } v\} \hat{=} \mathcal{I}$. ■

Discharging an Assertion

As explained in Section 13.3.3, we make use of the abstraction mechanism to justify the soundness of discharging assertions in atomic statements. Notice that the way $\text{Compile}(\cdot)$ propagates the assertions to the beginning of the statement essentially corresponds to the sequential verification assertions in [38] and [41]. The tactics below proves the validity of one or more assertions by performing a sequential check and remove them from the statement.

The following tactic discharges all the assertions in an atomic statement if the current invariant implies the gate of the corresponding action.

Tactic 7 (Discharge all assertions – RELAX-I):

Replace every assertion $\text{assert } B$ in $\text{atomic}\{\bar{S}\}$ with skip , if $\mathcal{I} \Rightarrow \text{gt}(\text{atomic}\{\bar{S}\})$ is valid. ■

Recall the definition of context $\mathbf{C}$ (Definition 20). We can rewrite a straight statement $\bar{S}$ with an assertion $\text{assert } B$ using a context such that $\bar{S} = \mathbf{C}[\text{assert } B]$. The tactic below removes the assertion and rewrites the statement as $\mathbf{C}[\text{skip}]$.

Tactic 8 (Discharge an assertion – RELAX-II):

Let $\bar{S} = \mathbf{C}[\text{assert } B]$. Replace occurrences of $\text{atomic}\{\bar{S}\}$ with $\text{atomic}\{\mathbf{C}[\text{skip}]\}$ if $\mathcal{I} \vdash \text{atomic}\{\mathbf{C}[\text{assert } B]\} \hat{=} \text{atomic}\{\mathbf{C}[\text{skip}]\}$ holds. ■

15.2.3 Variable Introduction and Hiding

Intuitively, variable introduction rewrites some actions in the program so that these can refer to a new variable. Variable hiding is the dual of variable introduction; each action is rewritten so that it no longer refers to the hidden variable. Hiding a variable also requires quantifying out the variable in the invariant.

The variable addition is useful to augment the program state with variables keeping track of extra information, for example, the sum of two variables or some status information. Moreover, in Chapter 16, coupling variable introduction and hiding plays a key role in our new strategy for proving linearizability, in which we transform a data structure implementation to a desired specification. While we use reduction and abstraction to increase the granularity of atomic actions, we syntactically relate the representation of the implementation to that of the specification by hiding the variables of the former and introducing variables of the latter.

In order to ensure soundness, in both cases, we need a relation between actions over different sets of variables. For this, we use the abstraction relations $\hat{\preceq}_{+v}$ and $\hat{\preceq}_{-v}$ that are defined using the abstraction relations $\preceq_{+v}$ and $\preceq_{-v}$ in Section 13.4.

Definition 21 (Abstraction for atomic statements under variable addition):

Let $\text{atomic}\{S_1\}$, $\text{atomic}\{S_2\}$ be atomic statements and $\mathcal{I}$ be a state predicate. Let v be a fresh variable such that $v \notin \text{Var}(\text{atomic}\{S_1\})$ and $v \in \text{Var}(\text{atomic}\{S_2\})$. $\mathcal{I} \vdash \text{atomic}\{S_1\} \hat{\preceq}_{+v} \text{atomic}\{S_2\}$ holds if $\mathcal{I} \vdash \text{Compile}(\text{atomic}\{S_1\}) \preceq_{+v} \text{Compile}(\text{atomic}\{S_2\})$ holds. ■

Definition 22 (Abstraction for atomic statements under variable hiding):

Let $\text{atomic}\{S_1\}$, $\text{atomic}\{S_2\}$ be atomic statements and $\mathcal{I}$ be a state predicate. Let v be a variable such that $v \in \text{Var}(\text{atomic}\{S_1\})$ and $v \notin \text{Var}(\text{atomic}\{S_2\})$. $\mathcal{I} \vdash \text{atomic}\{S_1\} \hat{\preceq}_{-v} \text{atomic}\{S_2\}$ holds if $\mathcal{I} \vdash \text{Compile}(\text{atomic}\{S_1\}) \preceq_{-v} \text{Compile}(\text{atomic}\{S_2\})$ holds. ■

The relations $\hat{\preceq}_{+v}$ and $\hat{\preceq}_{-v}$ require that the new atomic statement preserve the violations of the existing statement, and that transitions over the common variables are either preserved or additional violations are introduced. In addition, we enforce that the signatures of procedures be preserved by the QED transformations, so the rules below cannot be used to add or hide an input or output variable of a procedure.

Variable Introduction

The following tactic allow one to annotate the program with new variables. Note that, this corresponds to annotating the program with *history* variables as in [42, 62], except that, we also allow the gates (assertions) to be updated along with the transitions. The new variable can be added as a global or a procedure-local variable.

Tactic 9 (Add global variable x – ADD-GLOBAL-VARIABLE):

Add the new variable x to $\text{Global}_{\mathbf{p}}$ and replace every atomic statement $\text{atomic}\{\bar{S}_1\}$ in $\text{Atoms}(\hat{P})$ with a statement $\text{atomic}\{\bar{S}_2\}$ if the following hold:

$$1. \text{atomic}\{S_2\} \hat{=} \mathcal{I}$$

$$2. \mathcal{I} \vdash \text{atomic}\{\bar{S}_1\} \hat{=}_{+x} \text{atomic}\{\bar{S}_2\} \quad \blacksquare$$

Tactic 10 (Add local variable ℓ to $\mathbf{p}$ – ADD-LOCAL-VARIABLE):

Let $\mathbf{p} \in \text{Proc}$. Add the new variable ℓ to $\text{Private}_{\mathbf{p}}$ and replace every atomic statement $\text{atomic}\{\bar{S}_1\}$ in $\text{Atoms}(\text{Body}_{\mathbf{p}})$ with a statement $\text{atomic}\{S_2\}$ if the following hold:

$$1. \text{atomic}\{S_2\} \hat{=} \mathcal{I}$$

$$2. \mathcal{I} \vdash \text{atomic}\{\bar{S}_1\} \hat{=}_{+\ell} \text{atomic}\{\bar{S}_2\} \quad \blacksquare$$

Variable Hiding

The following tactics allow one to remove a variable from the program and replace atomic statements with new ones that no longer refer to the new variable.

Tactic 11 (Hide global variable x – HIDE-GLOBAL-VARIABLE):

Remove the existing variable x from $\text{Global}_{\mathbf{p}}$, replace the current invariant $\mathcal{I}$ with $(\exists x.\mathcal{I})$ and replace every atomic statement $\text{atomic}\{\bar{S}_1\}$ in $\text{Atoms}(\hat{P})$ with a statement $\text{atomic}\{S_2\}$ if the following hold:

$$1. \text{atomic}\{S_2\} \hat{=} (\exists x.\mathcal{I})$$

$$2. \mathcal{I} \vdash \text{atomic}\{\bar{S}_1\} \hat{=}_{-x} \text{atomic}\{\bar{S}_2\} \quad \blacksquare$$

Tactic 12 (Hide local variable ℓ from $\mathbf{p}$ – HIDE-LOCAL-VARIABLE):

Let $\mathbf{p} \in \text{Proc}$. Remove the existing variable ℓ from $\text{Private}_{\mathbf{p}}$ and replace every atomic statement $\text{atomic}\{\bar{S}_1\}$ in $\text{Atoms}(\text{Body}_{\mathbf{p}})$ with a statement $\text{atomic}\{S_2\}$ if the following hold:

$$1. \text{atomic}\{S_2\} \hat{=} \mathcal{I}$$

$$2. \mathcal{I} \vdash \text{atomic}\{\bar{S}_1\} \hat{=}_{-\ell} \text{atomic}\{\bar{S}_2\} \quad \blacksquare$$

15.2.4 Reduction

The reduction tactics resemble the reduction rules presented in Section 13.5: Every atomic statement in a QEDPL program is classified as a mover, and combined with others to form larger atomic statements. In order to compute the mover types, we use the static mover check specified in Definition 13. We lift the definition of the mover check judgment to atomic statements by using the translation of these statements to gated actions as follows.

Definition 23 (Mover check for atomic statements):

The mover check between two atomic statements $\text{atomic}\{\bar{S}_1\}$ and $\text{atomic}\{\bar{S}_2\}$ at a proof state $(\hat{P}, \mathcal{I})$ succeeds, denoted by $\mathcal{I} \vdash \mathcal{MC}(\text{atomic}\{\bar{S}_1\}, \text{atomic}\{\bar{S}_2\})$, if $\mathcal{I} \vdash \mathcal{MC}(\text{Compile}(\text{atomic}\{\bar{S}_1\}), \text{Compile}(\text{atomic}\{\bar{S}_2\}))$ holds. ■

The definition of the mover check above indicates that the movers in a QEDPL program $\hat{P}$ are computed by (1) translating $\hat{P}$ to an ACTIONPL program P such that $\text{Compile}(\hat{P}) = P$, (2) computing the mover type of every gated action in P , and (3) propagating the resulting mover types back to the corresponding atomic statements in $\hat{P}$. (3) is done considering the bijection between $\text{Atoms}(\hat{P})$ and $\text{Atoms}(P)$ ensured by the formulation of $\text{Compile}(\cdot)$. During (2), the derivation rules in Figure 13.2 are used to decide the mover types of the actions. Therefore, the judgment $(\hat{P}, \mathcal{I}) \vdash \text{atomic}\{\bar{S}\} : m$ dictates that $\text{Compile}(\text{atomic}\{\bar{S}\})$ is m -mover in the proof state $(P, \mathcal{I})$.

The tactics for reducing sequentially composed statements and conditional (if) statements in QEDPL are as follows:

Tactic 13 (Reduce sequential – REDUCE-SEQUENTIAL-RIGHT/LEFT):

Replace occurrences of

$$\text{atomic}\{\bar{S}_1\} ; \text{atomic}\{\bar{S}_2\}$$

with

$$\text{atomic}\{\bar{S}_1; \bar{S}_2\}$$

if at least one of the following hold:

- $(\hat{P}, \mathcal{I}) \vdash \text{atomic}\{\bar{S}_1\} : \mathbb{R}$
- $(\hat{P}, \mathcal{I}) \vdash \text{atomic}\{\bar{S}_2\} : \mathbb{L}$

■

The following tactics put an if statement inside an atomic statement unconditionally, only if both branches of the statement are atomic statements.

Tactic 14 (Reduce conditional – REDUCE-CHOICE):

Replace occurrences of

$$\text{if}(g) \{ \text{atomic}\{\bar{S}\} \}$$

with

$$\text{atomic}\{\text{if}(g) \{ \bar{S} \} \}$$

Tactic 15 (Reduce conditional with else – REDUCE-CHOICE):

Replace occurrences of

$$\text{if}(g) \{ \text{atomic}\{\bar{S}_1\} \} \text{ else } \{ \text{atomic}\{\bar{S}_2\} \}$$

with

$$\text{atomic}\{\text{if}(g) \{ \bar{S}_1 \} \text{ else } \{ \bar{S}_2 \} \}$$

Reducing Try-Catch Statements

The following tactics are used to make a **try-catch** statement atomic. Both tactics require that the main body and the catch part be both atomic statements. The first tactic makes use of the mover type of the atomic statements that constitute the **try-catch** statement. The second tactic considers the special case that the main body may not throw an exception that the catch part is interested in.

Tactic 16 (Reduce try-catch – REDUCE-LABELED-RIGHT/LEFT):

Replace occurrences of

$$\text{try} \{ \text{atomic}\{\bar{S}_1\} \} \text{ catch}(l) \{ \text{atomic}\{\bar{S}_2\} \}$$

with

$$\text{atomic}\{\text{try} \{ \bar{S}_1 \} \text{ catch}(l) \{ \bar{S}_2 \} \}$$

if at least one of the following hold:

- $(\hat{P}, \mathcal{I}) \vdash \text{atomic}\{\bar{S}_1\} : \mathbb{R}$
- $(\hat{P}, \mathcal{I}) \vdash \text{atomic}\{\bar{S}_2\} : \mathbb{L}$

■

Tactic 17 (Reduce try-catch any – REDUCE-LABELED-RIGHT/LEFT):*Replace occurrences of*

$$\text{try } \{\text{atomic}\{\bar{S}_1\}\} \text{ catch}\{\text{atomic}\{\bar{S}_2\}\}$$

with

$$\text{atomic}\{\text{try } \{\bar{S}_1\} \text{ catch}\{\bar{S}_2\}\}$$

if at least one of the following hold:

- $(\hat{P}, \mathcal{I}) \vdash \text{atomic}\{\bar{S}_1\} : \mathbb{R}$
- $(\hat{P}, \mathcal{I}) \vdash \text{atomic}\{\bar{S}_2\} : \mathbb{L}$ ■

Tactic 18 (Reduce try-catch unconditionally):*Replace occurrences of*

$$\text{try } \{\text{atomic}\{\bar{S}_1\}\} \text{ catch}(l)\{\text{atomic}\{\bar{S}_2\}\}$$

with

$$\text{atomic}\{\text{try } \{\bar{S}_1\} \text{ catch}(l)\{\bar{S}_2\}\}$$

if $\bar{S}_1$ does not contain the statement throw l and does not make a call to a procedure that contains throw l . ■

Tactic 19 (Reduce try-catch any unconditionally):*Replace occurrences of*

$$\text{try } \{\text{atomic}\{\bar{S}_1\}\} \text{ catch}\{\text{atomic}\{\bar{S}_2\}\}$$

with

$$\text{atomic}\{\text{try } \{\bar{S}_1\} \text{ catch}\{\bar{S}_2\}\}$$

if $\bar{S}_1$ does not contain any throw statements and does not make a call to a procedure that contains a throw statements. ■

Reducing Parallel Statements

The tactics in this section correspond to the proof rules in Section 13.5.9, which rewrite a parallel composition ($\parallel$) in terms of sequential composition ($;$) and nondeterministic choice

(□). Similarly the tactics rewrite a **dopar** statement to a statement using sequential composition and if statements.

Recall that while giving semantics to QEDPL programs we translate a **dopar** statement into a parallel composition ($\parallel$) in ACTIONPL, and the semantics of $S_1 \parallel S_2$ is to create two threads each with a fresh copy of the parent thread's local store initialized to arbitrary values. In order to express that the children of the parent thread access distinct local variables, we use the function *Fresh* to create fresh variables, and using those, rename the existing local variables in the branches of the **dopar** statement. In addition, the exceptions (other than **skip**) thrown inside a child thread is not propagated to parent thread. To express this, while sequentially compositing the branches of the **dopar** statement, we use **try-catch** statements to suppress the exceptions thrown by these branches.

The first tactic for **dopar** statements expands the statement if both of its branches are atomic.

Tactic 20 (Expand dopar statement – EXPAND-PARALLEL):

Replace occurrences of

$$\text{dopar } \{\text{atomic}\{\bar{S}_1\}\} \text{ with } \{\text{atomic}\{\bar{S}_2\}\}$$

with the following, where $\mathbf{p}$ is the containing procedure, and X and Y are disjointly-fresh for $\text{Local}_{\mathbf{p}}$ ($X \cap Y = \emptyset$):

```

1 if( *) {
2   atomic{try{ $\bar{S}_1[X/\text{Local}_{\mathbf{p}}]}$ catch{skip}}} ;
3   atomic{try{ $\bar{S}_2[Y/\text{Local}_{\mathbf{p}}]}$ catch{skip}}}
4 } else {
5   atomic{try{ $\bar{S}_2[Y/\text{Local}_{\mathbf{p}}]}$ catch{skip}}} ;
6   atomic{try{ $\bar{S}_1[X/\text{Local}_{\mathbf{p}}]}$ catch{skip}}}
7 }
```

■

The following rule transforms a **dopar** statement using the mover types of its branches, similarly to the REDUCE-PARALLEL-LEFT and REDUCE-PARALLEL-RIGHT rules.

Tactic 21 (Reduce dopar statement – REDUCE-PARALLEL-LEFT/RIGHT):

Replace occurrences of

$$\text{dopar } \{\text{atomic}\{\bar{S}_1\}\} \text{ with } \{\text{atomic}\{\bar{S}_2\}\}$$

with the following, where $\mathbf{p}$ is the containing procedure, and X and Y are sets of fresh variables for $\text{Local}_{\mathbf{p}}$:

```
1  atomic{try{ $\bar{S}_1[X/\text{Local}_{\mathbf{p}}]$ }catch{skip}}};
2  atomic{try{ $\bar{S}_2[Y/\text{Local}_{\mathbf{p}}]$ }catch{skip}}
```

if at least one of the following hold:

- $(\hat{P}, \mathcal{I}) \vdash \text{atomic}\{\bar{S}_1\} : \mathbb{L}$
- $(\hat{P}, \mathcal{I}) \vdash \text{atomic}\{\bar{S}_2\} : \mathbb{R}$ ■

Reducing Loops

Reducing a loop means to replacing the loop with a statement without a loop. Note that, the QEDPL language do not allow loops in atomic statements. This restriction is important for us to be able to compute the gated action representation of an atomic statement (via $\text{Compile}(\cdot)$) automatically. If the loops were allowed in the atomic statement, still automating this process will require the user to manually provide loop invariants. Instead, we provide tactics to explicitly eliminate loops by replacing them with loop-free statements. In order to apply the reduction tactics presented above on a statement with loops, one first needs to reduce loops using the tactics presented below. The tactics for reducing loops require from the user an atomic statement that will represent, similarly to a loop invariant, the non-final iterations of the loop.

The following tactic corresponds to the rule REDUCE-LOOP in Section 13.5.10 and replaces a loop whose body is an atomic statement with a loop-free statement.

Tactic 22 (Reduce loop – REDUCE-LOOP):

Replace occurrences of

$$\text{loop}\{\text{atomic}\{\bar{S}_1\}\}$$

with

$$\text{atomic}\{\bar{S}_2\}; \text{atomic}\{\bar{S}_1; \text{block}\}$$

if the following hold, where $\hat{S} = \text{try}\{\bar{S}_1\}\text{catch}\{\text{block}\}$:

1. $(\hat{P}, \mathcal{I}) \vdash \text{atomic}\{\hat{S}\} : m$ such that $m \in \{\mathbb{R}, \mathbb{L}\}$

$$2. \text{atomic}\{\bar{S}_2\} \hat{=} \mathcal{I}$$

$$3. \text{Reflexive}(\text{Compile}(\text{atomic}\{\bar{S}_2\}))$$

$$4. (\hat{P}, \mathcal{I}) \vdash \text{atomic}\{\bar{S}_2; \hat{S}\} \hat{\approx} \text{atomic}\{\bar{S}_2\} \quad \blacksquare$$

Intuitively, the above tactic handles the final iteration of the loop from separately from the non-final iterations. Condition 1 states that the non-final iterations of the loop are either right or left movers, and condition 3 and 4 state if (zero or more) non-final iterations of the loop are simulated by the statement $\text{atomic}\{\bar{S}_2\}$ playing the role of a loop invariant. In these checks, the statement $\text{atomic}\{\hat{S}\}$ models the executions of the loop body ending with the post-condition $\text{exc} = \mathbf{skip}$, and thus represents a non-final iteration of the loop. If all the conditions hold, the rule replaces the rule with the sequential composition of $\text{atomic}\{\bar{S}_2\}$, simulating the non-final iterations of the loop and $\text{atomic}\{\bar{S}_1; \mathbf{block}\}$ simulating the final iteration of the loop. The statement $\mathbf{block}$ in the latter indicates that $\text{atomic}\{\bar{S}_1; \mathbf{block}\}$ terminates only by throwing an exception other than $\mathbf{skip}$.

We also provide the following tactic where the loop specification has been introduced before.

Tactic 23 (Reduce loop right – REDUCE-LOOP-RIGHT):

Replace occurrences of

$$\text{atomic}\{\bar{S}_2\}; \text{loop}\{\text{atomic}\{\bar{S}_1\}\}$$

with

$$\text{atomic}\{\bar{S}_2; \bar{S}_1; \mathbf{block}\}$$

if the following hold, where $\hat{S} = \text{try}\{\bar{S}_1\}\text{catch}\{\mathbf{block}\}$:

$$1. (\hat{P}, \mathcal{I}) \vdash \text{atomic}\{\bar{S}_2\} : \mathbb{R}$$

$$2. (\hat{P}, \mathcal{I}) \vdash \text{atomic}\{\bar{S}_2; \hat{S}\} \hat{\approx} \text{atomic}\{\bar{S}_2\} \quad \blacksquare$$

In addition to the core tactics above, we provide tactics specialized to **while** statements. Note that, in Section 14.3.1, we added **while** statements as syntactic sugars for a particular use of the **loop** statement using the **continue** and **break** statements. Thus, the tactics below can be encoded as special applications of Tactic 22 and 23. The tactics for while loops below

are easier to use since one does not have to think separately about the non-final and final iterations of a loop. In our experience we needed only these tactics to handle the loops in the benchmarks.

Tactic 24 (Reduce while – REDUCE-NONDET-LOOP):

Replace occurrences of

$$\text{while}(\ast)\{\text{atomic}\{\bar{S}_1\}\}$$

with

$$\text{atomic}\{\bar{S}_2\}$$

if the following hold:

1. $\bar{S}_1$ cannot throw any exception other than **skip**
2. $(\hat{P}, \mathcal{I}) \vdash \text{atomic}\{\bar{S}_1\} : m$ such that $m \in \{\mathbb{R}, \mathbb{L}\}$
3. $\text{atomic}\{\bar{S}_2\} \hat{=} \mathcal{I}$
4. $\text{Reflexive}(\text{Compile}(\text{atomic}\{\bar{S}_2\}))$
5. $(\hat{P}, \mathcal{I}) \vdash \text{atomic}\{\bar{S}_2; \bar{S}_1\} \hat{=} \text{atomic}\{\bar{S}_2\}$ ■

The following tactics correspond to the rules REDUCE-LOOP-RIGHT and REDUCE-LOOP-LEFT in Section 13.5.10.

Tactic 25 (Reduce while right – REDUCE-NONDET-LOOP-RIGHT):

Replace occurrences of

$$\text{atomic}\{\bar{S}_2\}; \text{while}(\ast)\{\text{atomic}\{\bar{S}_1\}\}$$

with

$$\text{atomic}\{\bar{S}_2\}$$

if the following hold:

1. $\bar{S}_1$ cannot throw any exception other than **skip**
2. $(\hat{P}, \mathcal{I}) \vdash \text{atomic}\{\bar{S}_2\} : \mathbb{R}$
3. $(\hat{P}, \mathcal{I}) \vdash \text{atomic}\{\bar{S}_2; \bar{S}_1\} \hat{=} \text{atomic}\{\bar{S}_2\}$ ■

Tactic 26 (Reduce loop left – REDUCE-NONDET-LOOP-LEFT):

Replace occurrences of

$$\text{while}(\ast)\{\text{atomic}\{\bar{S}_1\}\}; \text{atomic}\{\bar{S}_2\}$$

with

$$\text{atomic}\{\bar{S}_2\}$$

if the following hold:

1. $\bar{S}_1$ cannot throw any exception other than **skip**
2. $(\hat{P}, \mathcal{I}) \vdash \text{atomic}\{\bar{S}_2\} : \mathbb{L}$
3. $(\hat{P}, \mathcal{I}) \vdash \text{atomic}\{\bar{S}_1; \bar{S}_2\} \hat{\approx} \text{atomic}\{\bar{S}_2\}$ ■

Reducing Procedures

For procedures without recursion, we apply Tactic 27 below to inline procedures at call points. This tactic inlines the body of the callee at the call point and adds new local variables to the caller. Following Equation 12.7, $\text{Inline}(\text{call } \overrightarrow{\text{out}} := \mathbf{p}(\overrightarrow{\text{in}}))$ returns a pair $\langle \hat{S}, V \rangle$ where $\hat{S}$ is the inlined body of the procedure $\mathbf{p}$ and V is a set of fresh local variables used while generating the statement $\hat{S}$. The inlined body $\hat{S}$, which catches and suppresses the exception **return**, is defined as follows:

```

1 try {
2    $\text{Body}_{\mathbf{p}}[\overrightarrow{\text{in}}/\overrightarrow{\text{In}}_{\mathbf{p}}, \overrightarrow{\text{out}}/\overrightarrow{\text{Out}}_{\mathbf{p}}, V/\text{Private}_{\mathbf{p}}]$ 
3 } catch(return) {skip}

```

Tactic 27 (Inline call – INLINE-CALL):

Given a procedure call $\text{call } \overrightarrow{\text{out}} := \mathbf{p}(\overrightarrow{\text{in}})$ in the body of $\mathbf{p}'$ let $\langle \hat{S}, V \rangle = \text{Inline}(\text{call } \overrightarrow{\text{out}} := \mathbf{p}(\overrightarrow{\text{in}}))$. Replace $\text{call } \overrightarrow{\text{out}} := \mathbf{p}(\overrightarrow{\text{in}})$ with $\hat{S}$ and add V to $\text{Private}_{\mathbf{p}'}$ of the containing procedure $\mathbf{p}'$. ■

When there is recursion in the program, we use Tactic 28 given below. The tactic uses an annotation function $\mathcal{A}$, as defined in Section 13.6.1, that maps every procedure $\mathbf{p}_i \in \mathcal{M}$ to a pair $(m_i, \text{atomic}\{\bar{S}_i\})$ and every loop $\text{while}(g)\{\hat{S}\}$ to a statement $\text{atomic}\{\bar{S}_j\}$. The first

element of the pair $(m_i, \bar{S}_i)$ assigns a mover type to $\mathbf{p}_i$ and the second element gives an atomic specification for sequential executions of $\text{Body}_{\mathbf{p}_i}$. As for loops, $\text{atomic}\{\bar{S}_j\}$ gives an atomic specification for the non-final iterations of the loop assuming that the loop is executed sequentially till its final iteration. Conditions of the tactic resembles those of the proof rule REDUCE-PROCEDURE except that gated actions are replaced by atomic statements. We assume that nested loops are processed starting from the inner one, so when processing a loop, its body does not contain any loops.

Tactic 28 (Reduce recursive procedures – REDUCE-PROCEDURE):

Take a set of procedures $\mathcal{M} = \{\mathbf{p}_1, \dots, \mathbf{p}_n\}$ such that $\mathcal{M} \subseteq \text{Proc}$ and $\mathcal{M}$ is closed under call (see Definition 3), and an annotation function $\mathcal{A}$. Replace the body of every procedure in $\mathcal{M}$ with the given specification in $\mathcal{A}$, if the following hold for each procedure $\mathbf{p}_i \in \mathcal{M}$ such that $\mathcal{A}(\mathbf{p}_i) = (m_i, \text{atomic}\{\bar{S}_i\})$:

1. $\text{atomic}\{\bar{S}_i\} \hat{\hookrightarrow} \mathcal{I}$
2. $(P, \mathcal{I}, \mathcal{A}) \vdash \text{Body}_{\mathbf{p}_i} : m_i$
3. $\mathcal{I} \vdash \llbracket \text{Compile}(\text{Body}_{\mathbf{p}_i}) \rrbracket_{\mathcal{A}} \hat{\approx} \text{atomic}\{\bar{S}_i\}$
4. $\text{Finalizable}(\mathbf{p}_i, \text{gt}(\text{atomic}\{\bar{S}_i\}))$
5. For every loop $\text{loop}\{\hat{S}\}$ in $\text{Body}_{\mathbf{p}_i}$ such that $\mathcal{A}(\text{loop}\{\hat{S}\}) = \text{atomic}\{\bar{S}_j\}$:
 - (a) $\text{atomic}\{\bar{S}_j\} \hat{\hookrightarrow} \mathcal{I}$
 - (b) $\text{Reflexive}(\text{Compile}(\text{atomic}\{\bar{S}_j\}))$
 - (c) $\mathcal{I} \vdash \text{atomic}\{\bar{S}_j; \text{try}\{\hat{S}\}\text{catch}\{\text{block}\}\} \hat{\approx} \text{atomic}\{\bar{S}_j\}$ ■

15.3 Code Transformations

In this section, we provide several tactics that are not related to any proof rule, but that we find useful to simplify the reduction steps.

15.3.1 Peeling Out the Last Iteration of While Loops

The tactics 24, 25, and 26 provide the simplest ways of reducing loops. However, they operate on nondeterministic while loops whose bodies do not throw any exceptions other than **skip**. Because of this, in practice, we follow the following strategy that enables us to use tactics 29–31 for arbitrary while loops.

1. Using tactics 29, 30, and 31 presented below, we separate the non-final iterations of the loop from its final iteration. In the resulting code, the non-final iterations are represented by a nondeterministic while loop whose body does not throw an exception, and is suitable to reduce using the tactics 24, 25 and 26.
2. Using reduction tactics, we reduce the body of nondeterministic while loop obtained in item 1 to a single atomic statement. This can be done by an aggressive use of read and write abstractions when the modifications on the program state are carried out in the final iteration.
3. When the body of the nondeterministic while loop becomes atomic, the loop is reduced to a single action using one of the tactics 24, 25 and 26.

During the first phase of the strategy above, we use tactics 29, 30, and 31, each of which peels out the final iteration of either a **loop**, **while(*)** or **while(B_ℓ)** statement and represents the non-final iterations as a **while(*)** loop. The tactics, using **try-catch** statements, ensure that the body of **while(*)** in the resulting statement does not terminate with an exception other than **skip**.

Tactic 29 (Peel out the final iteration of loop):

Replace occurrences of

$$\text{loop}\{\hat{S}\}$$

with

```

1 while(*) {
2   try {
3      $\hat{S}$ 
4   } catch {block}
```

```

5 };
6  $\hat{S}$  ; block

```

■

In the tactic above, the nondeterministic while loop spanning lines 1-5 represent the non-final iterations of the loop. The **try-catch** statement at lines 2-5 ensures that every execution of $\hat{S}$ at line 3 ending with an exception other than **skip** blocks. At line 6 $\hat{S}$ is followed by **block**; whenever $\hat{S}$ throws a non-skip exception, this makes the current thread block. Note that, in our method, the blocking executions of threads (unless others go wrong) are not considered as part of the program's behavior. Thus, terminating executions of line 6 (with a non-skip exception) model the final iteration of the loop.

Tactic 30 (Peel out the final iteration of **while(*)):**

Replace occurrences of

while(*){ $\hat{S}$ }

with

```

1 while(*) {
2   try {
3     try {
4        $\hat{S}$ 
5     } catch(continue) {skip}
6   } catch{block};
7 } ;
8 if(*){
9   try {
10    try {
11       $\hat{S}$ ;
12    } block
13  } catch(continue) {block}
14 } catch(break){skip};
15 }

```

■

In the tactic above, the nondeterministic while loop spanning lines 1-7 represents the non-final iterations of the loop. The **try-catch** statement at lines 3-5 suppresses the exception

continue that may be thrown by $\hat{S}$ at line 4, and the **try-catch** statement at lines 2-6 makes the current thread block if $\hat{S}$ at line 4 throws another (non-skip) exception. The if statement at lines 8-15 represents the final iteration of the loop. Differently from lines 3-5, the **try-catch** statement at lines 10-13 blocks if $\hat{S}$ throws **continue**. In addition, **block** at line 12 blocks all terminations of $\hat{S}$ at line 11 ending with $exc = \mathbf{skip}$. If $\hat{S}$ at line 11 throws **break**, this is suppressed by the **catch** statement at line 14.

Tactic 31 (Peel out the final iteration of $\mathbf{while}(B_\ell)$):

Replace occurrences of

$\mathbf{while}(B_\ell)\{\hat{S}\}$

with

```

1 while(*) {
2   try {
3     try {
4       assume  $B_\ell$ ;
5        $\hat{S}$ 
6     } catch(continue) {skip}
7   } catch{block};
8 } ;
9 if( $B_\ell$ ){
10  try {
11    try {
12       $\hat{S}$ ;
13      block
14    } catch(continue) {block}
15  } catch(break){skip};
16 }
```

■

Recall that B_ℓ is a predicate whose free variables are only local variables. The tactic above is similar to Tactic 30 except for the following:

- We insert the **assume** command at line 4 in order to model the case that any non-final iteration is enabled only when B_ℓ is satisfied at the current state.

- “if(B_ℓ)” at line 9 is needed since the loop may terminate because either B_ℓ is not satisfied or the body $\hat{S}$ at line 12 throws a non-skip exception. If the former happens, then the body of the if statement at lines 10-15 is skipped. The latter case is modeled by entering into the body of the if statement and having $\hat{S}$ at line 12 throw an exception. If this exception is **break**, it is suppressed by the **catch** statement at line 15.

(1) Initial version of the while loop <pre> found := false; i := 0; while(i < N) { if (A[i] >= key) { found := true; break; } i := i + 1; } </pre> (2) After applying Tactic 31 <pre> found := false; i := 0; while(*) { try { try { assume i < N; if (A[i] >= key) { found := true; break; } } i := i + 1; } catch(continue) {skip} } catch {block} } if(i < N) { try { try { if (A[i] >= key) { found := true; break; } i := i + 1; block; } catch(continue) {block} } catch(break) {skip} } </pre>	(3) After applying reduction tactics <pre> atomic { found := false; i := 0; } while(*) { atomic { assume i < N; assume A[i] < key; i := i + 1; } } atomic { if(i < N) { assume A[i] >= key; found := true; } } </pre> (4) After replacing the first atomic action with the specification of the loop using Tactic 2 <pre> atomic { // Right mover found := false; havoc i; assume 0 <= i && i <= N; assume (forall j:int :: 0 <= j && j < i ==> A[j] < key); } while(*) { atomic { assume i < N; assume A[i] < key; i := i + 1; } } atomic { if(i < N) { assume A[i] >= key; found := true; } } </pre>

Figure 15.3: The first part of the loop-reduction example.

(5) After reducing the nondeterministic while

```

loop using Tactic 25
atomic { // Right mover
  found := false;
  havoc i;
  assume 0 <= i && i <= N;
  assume (forall j:int ::
    0 <= j && j < i ==> A[j] < key);
}

atomic {
  if(i < N) {
    assume A[i] >= key;
    found := true;
  }
}

```

(7) After simplifying the final atomic statement using Tactic 2

```

atomic {
  if(*) {
    i := N;
    assume (forall j:int ::
      0 <= j && j < N ==> A[j] < key);
    found := false;
  }
  else {
    havoc i;
    assume 0 <= i && i <= N;
    assume (forall j:int ::
      0 <= j && j < i ==> A[j] < key);
    assume A[i] >= key;
    found := true;
  }
}

```

(6) After combining the atomic statements

```

atomic {
  found := false;
  havoc i;
  assume 0 <= i && i <= N;
  assume (forall j:int ::
    0 <= j && j < i ==> A[j] < key);

  if(i < N) {
    assume A[i] >= key;
    found := true;
  }
}

```

Figure 15.4: The second part of the loop-reduction example.

Figures 15.3 and 15.4 demonstrate our strategy for reducing while loops described above. The first four steps of the example are shown in Figure 15.3, and the last three steps of the example are shown in Figure 15.4. The given code traverses a sorted array and detects the first element of the array that is greater than or equal to the given `key`. The example shows the key roles of the tactics 31, 2, and 25 in reducing the initial code to a single atomic statement. The final atomic statement represents the overall operation of the loop.

15.3.2 Injecting Prefix to If Statements

The following tactics take an atomic statement `atomic{ $\bar{S}$ }` followed by an if statement, and inject `atomic{ $\bar{S}$ }` to the beginning of both branches (if the `else` branch does not exist, they create one).

Suppose that we would like to reduce the statement `atomic{ $\bar{S}$ } if(*){ $S_1$ }else{ $S_2$ }` to an

atomic statement. We find this code transformation helpful at two cases:

- , When the reduction can be done by applying an abstraction tactic on $\text{atomic}\{\bar{S}\}$, but we need different abstractions on $\text{atomic}\{\bar{S}\}$ for each branch of the IF statement. In this case, each abstraction can be performed on separate copies of $\text{atomic}\{\bar{S}\}$ within the branches of the if statement.
- When $\text{atomic}\{\bar{S}\}$ is a right mover, but the if statement cannot be reduced to a single action; thus, reduction tactics cannot enlarge $\text{atomic}\{\bar{S}\}$. In this case, $\text{atomic}\{\bar{S}\}$ can be combined with the actions within the branches of the if statement after its injection.

Tactic 32 (Inject prefix to if(*)-else):

Replace occurrences of

$$\text{atomic}\{\bar{S}\} \text{ if } (*)\{S_1\}\text{else}\{S_2\}$$

with the following

$$\text{if } (*)\{\text{atomic}\{\bar{S}\}; S_1\}\text{else}\{\text{atomic}\{\bar{S}\}; S_2\}$$

Tactic 33 (Inject prefix to if(*)):

Replace occurrences of

$$\text{atomic}\{\bar{S}\} \text{ if } (*)\{S_1\}$$

with the following

$$\text{if } (*)\{\text{atomic}\{\bar{S}\}; S_1\}\text{else}\{\text{atomic}\{\bar{S}\}\}$$

For the following tactics, recall that B_ℓ is a predicate whose free variables are only local variables.

Tactic 34 (Inject prefix to if(B_ℓ)-else):

Replace occurrences of

$$\text{atomic}\{\bar{S}\} \text{ if } (B_\ell)\{S_1\}\text{else}\{S_2\}$$

with the following

$$\text{if } (*)\{\text{atomic}\{\bar{S}\}; \text{assume } B_\ell; S_1\}\text{else}\{\text{atomic}\{\bar{S}\}; \text{assume } \neg B_\ell; S_2\}$$

Tactic 35 (Inject prefix to $\text{if}(B_\ell)$):*Replace occurrences of*

$$\text{atomic}\{\bar{S}\} \text{ if}(B_\ell)\{S_1\}$$

with the following

$$\text{if}(\ast)\{\text{atomic}\{\bar{S}\}; \text{assume } B_\ell; S_1\} \text{ else } \{\text{atomic}\{\bar{S}\}; \text{assume } \neg B_\ell\}$$

15.3.3 Splitting If Statements

The following tactics are needed for some case in which we cannot reduce an statement $\text{if}(B_\ell)\{\text{atomic}\{\bar{S}_1\}; \text{atomic}\{\bar{S}_2\}\}$ into a single action, because neither $\text{atomic}\{\bar{S}_1\}$ nor $\text{atomic}\{\bar{S}_2\}$ is a mover. In such cases, we split the body of the if statement into two if statements $\text{if}(B_\ell)\{\text{atomic}\{\bar{S}_1\}\}; \text{if}(B_\ell)\{\text{atomic}\{\bar{S}_2\}\}$. After this transformation, we can transform each if statement to an atomic statement (using tactics 14 or 15) and then combine each atomic statement with previous or next actions.

Tactic 36 (Split $\text{if}(B_\ell)$):*Replace occurrences of*

$$\text{if}(B_\ell)\{S_1; S_2\}$$

with the following

$$\text{if}(B_\ell)\{S_1\}; \text{if}(B_\ell)\{S_2\}$$

if S_1 does not write to any variable in B_ℓ . ■**15.3.4 Extracting Assumptions**

The following tactic is used for a special case: in the statement $\text{atomic}\{\bar{S}_1\}; \text{atomic}\{\bar{S}_2\}$ we need to propagate a post-condition, say B , of $\bar{S}_1$ to $\bar{S}_2$ but we cannot do this by applying a reduction tactic since neither of the actions is a mover. In this case, we apply the following transformation to propagate the post-condition to the second statement.

1. Using the tactic below, we extract the post-condition B from $\text{atomic}\{\bar{S}_1\}$ as an assumption. The result is $\text{atomic}\{\bar{S}_1\}; \text{assume } B; \text{atomic}\{\bar{S}_2\}$.

2. Using Tactic 13, we prove that `assume B` is a right mover and combine it with the second statement. The result is `atomic{ $\bar{S}_1$ }; atomic{
assumecmd B; stmT2}`.

Tactic 37 (Extract assumption):

Replace the atomic statement

`atomic{ $\bar{S}$ }`

with

`atomic{ $\bar{S}$ }; assume B`

if $(\mathcal{I} \wedge \text{gt}(\text{atomic}\{\bar{S}\}) \wedge \text{tr}(\text{atomic}\{\bar{S}\})) \Rightarrow B'$ is valid. ■

15.4 Proof of Theorem 7

In this section, we argue that given the application of tactic in $(\hat{P}_1, \mathcal{I}_1) \dashrightarrow (\hat{P}_2, \mathcal{I}_2)$, there exists a proof step $(\text{Compile}(\hat{P}_1), \mathcal{I}_1) \dashrightarrow (\text{Compile}(\hat{P}_2), \mathcal{I}_2)$ governed by the QED proof rule. This proof rule was indicated while defining the tactic in this chapter. We do through a case analysis on the proof tactic applied.

For the following proof, for two actions $\phi_1 \triangleright \tau_1$ and $\phi_2 \triangleright \tau_2$, we write $\phi_1 \triangleright \tau_1 \cong \phi_2 \triangleright \tau_2$ to state that (1) $\models \phi_1 \Leftrightarrow \phi_2$, i.e., the gates of the actions are satisfied by the same set of states, and (2) $\models \phi_2 \Rightarrow (\tau_1 \Leftrightarrow \tau_2)$, i.e., for every state pair (s_1, s_2) , if $s_1 \models \phi_2$, then $(s_1, s_2) \models \tau_1$ iff $(s_1, s_2) \models \tau_2$. Therefore, if $\phi_1 \triangleright \tau_1 \cong \phi_2 \triangleright \tau_2$, then $\phi_1 \triangleright \tau_1$ and $\phi_2 \triangleright \tau_2$ exhibit exactly the same set of atomic transitions.

PROOF.

CASE: Tactic 1

$\langle 2 \rangle 1$. In this case, the program is not modified. So $\hat{P}_1 = \hat{P}_2$ and thus $\text{Compile}(\hat{P}_1) = \text{Compile}(\hat{P}_2)$. The proof rule ADD-INVARIANT justifies the proof step.

$\langle 2 \rangle 2$. Q.E.D.

$\langle 1 \rangle 1$. In the following, except for the proof of Tactic 11, the invariant $\mathcal{I}_1$ and $\mathcal{I}_2$ are the same. We will denote this invariant simply by *Inv*.

CASE: Tactic 2

$\langle 2 \rangle 1$. By the tactic, $\text{atomic}\{\bar{S}_2\} \hat{\Leftarrow} \mathcal{I}$. Thus, $\text{Compile}(\text{atomic}\{\bar{S}_2\}) \Leftarrow \mathcal{I}$ holds.

- ⟨2⟩2. By the tactic, $\mathcal{I} \vdash \text{atomic } \{\bar{S}_1\} \hat{\approx} \text{atomic } \{\bar{S}_2\}$ holds. Therefore, by Definition 19, $\mathcal{I} \vdash \text{Compile}(\text{atomic } \{\bar{S}_1\}) \preceq \text{Compile}(\text{atomic } \{\bar{S}_2\})$ holds.
- ⟨2⟩3. By the definition of $\text{Compile}(\cdot)$, for all $\mathbf{p} \in \text{Proc}$, $\text{Compile}(\hat{P}_2)$ agrees with $\text{Compile}(\hat{P}_1)$ on $\text{Body}_{\mathbf{p}}$ except that every $\text{Compile}(\text{atomic}\{\bar{S}_1\})$ in $\mathbf{p} \in \text{Proc}$ is replaced by $\text{Compile}(\text{atomic}\{\bar{S}_2\})$.
- ⟨2⟩4. By ⟨2⟩1, ⟨2⟩2, and ⟨2⟩3, the proof rule ABSTRACT justifies the proof step.
- ⟨2⟩5. Q.E.D.

CASE: Tactic 3

- ⟨2⟩1. Let $\text{Compile}(\text{atomic } \{\mathbf{C}[\text{skip}]\}) = \phi_1 \triangleright \tau_1$. By the definition of $\text{Compile}(\cdot)$ for atomic statements, $\text{Compile}(\text{atomic } \{\mathbf{C}[\text{assert } B]\}) = \phi_2 \triangleright \tau_2$ such that $\phi_2 \Rightarrow \phi_1$. In this case, there exists a formula ψ such that $\phi_2 = \phi_1 \wedge \psi$.
- ⟨2⟩2. Although τ_1 and τ_2 are not equivalent, the difference is only on the contribution of the assertion in the computation of B . Since τ_2 is evaluated only when ϕ_2 is satisfied, τ_1 and τ_2 represent the same set of successful transitions from states satisfying ϕ_2 .
- ⟨2⟩3. By ⟨2⟩1 and ⟨2⟩2, the proof rule ADD-ASSERTION justifies the proof step.
- ⟨2⟩4. Q.E.D.

CASE: Tactic 4

- ⟨2⟩1. By the tactic, $\text{atomic}\{\text{havoc } v; \bar{S}\} \hat{\Leftarrow} \mathcal{I}$. Thus, $\text{Compile}(\text{atomic}\{\text{havoc } v; \bar{S}\}) \Leftarrow \mathcal{I}$ holds.
- ⟨2⟩2. Let $\text{Compile}(\text{atomic}\{\bar{S}\}) = \phi \triangleright \tau$. By the definition of $\text{Compile}(\cdot)$, $\text{Compile}(\text{atomic}\{\text{havoc } v; \bar{S}\}) = \phi \triangleright \exists v. \tau$. This is because no assertion refers to v , so the gate of the statement does not change, and the `havoc` v command makes the transition predicate τ of the old statement existentially quantified on v .
- ⟨2⟩3. By ⟨2⟩1 and ⟨2⟩2 the proof rule ABSTRACT-READ-I justifies the proof step.
- ⟨2⟩4. Q.E.D.

CASE: Tactic 6

- ⟨2⟩1. By the tactic, $\text{atomic}\{\bar{S}; \text{havoc } v\} \Leftarrow \mathcal{I}$. Thus, $\text{Compile}(\text{atomic}\{\bar{S}; \text{havoc } v\}) \Leftarrow \mathcal{I}$ holds.
- ⟨2⟩2. Let $\text{Compile}(\text{atomic}\{\bar{S}\}) = \phi \triangleright \tau$. By the definition of $\text{Compile}(\cdot)$, $\text{Compile}(\text{atomic}\{\bar{S}; \text{havoc } v\}) = \phi \triangleright \exists v'. \tau$. This is because the final write to v by `havoc` v does not affect the gate of the statement, which is computed for only

considering $\bar{S}$, and the **xp** computation for the **havoc** v command at the end gives $\exists v.(v' = v)$ and propagating this to the beginning of the statement results in a transition predicate equivalent to $\exists v'.\tau$.

$\langle 2 \rangle 3$. By $\langle 2 \rangle 1$ and $\langle 2 \rangle 2$ the proof rule **ABSTRACT-WRITE** justifies the proof step.

$\langle 2 \rangle 4$. Q.E.D.

CASE: Tactic 7

$\langle 2 \rangle 1$. Since $\mathcal{I} \Rightarrow \mathbf{gt}(\mathbf{atomic}\{\bar{S}\})$ is valid, the action $\mathbf{Compile}(\mathbf{atomic}\{\bar{S}\})$ never goes wrong from $\mathcal{I}$. Thus the proof rule **RELAX-I** justifies the proof step.

$\langle 2 \rangle 2$. Q.E.D.

CASE: Tactic 8

$\langle 2 \rangle 1$. The soundness of this tactic is justified by both Tactic 2 and the proof rule **RELAX-II**.

$\langle 2 \rangle 2$. Q.E.D.

CASE: Tactic 9

$\langle 2 \rangle 1$. By the tactic, $\mathbf{atomic}\{\bar{S}_2\} \hat{=}_{\mathcal{I}} \mathcal{I}$. Thus, $\mathbf{Compile}(\mathbf{atomic}\{\bar{S}_2\}) \hat{=} \mathcal{I}$ holds.

$\langle 2 \rangle 2$. By the tactic, $\mathcal{I} \vdash \mathbf{atomic}\{\bar{S}_1\} \hat{\preceq}_{+x} \mathbf{atomic}\{\bar{S}_2\}$ holds. Therefore, by Definition 21, $\mathcal{I} \vdash \mathbf{Compile}(\mathbf{atomic}\{\bar{S}_1\}) \preceq_{+x} \mathbf{Compile}(\mathbf{atomic}\{\bar{S}_2\})$ holds.

$\langle 2 \rangle 3$. By the definition of $\mathbf{Compile}(\cdot)$, for all $\mathbf{p} \in \mathbf{Proc}$, $\mathbf{Compile}(\hat{P}_2)$ agrees with $\mathbf{Compile}(\hat{P}_2)$ on $\mathbf{Body}_{\mathbf{p}}$ except that every $\mathbf{Compile}(\mathbf{atomic}\{\bar{S}_1\})$ in $\mathbf{p} \in \mathbf{Proc}$ is replaced by $\mathbf{Compile}(\mathbf{atomic}\{\bar{S}_2\})$.

$\langle 2 \rangle 4$. By $\langle 2 \rangle 1$, $\langle 2 \rangle 2$, and $\langle 2 \rangle 3$, the proof rule **ADD-GLOBAL-VARIABLE** justifies the proof step.

$\langle 2 \rangle 5$. Q.E.D.

CASE: Tactic 10

$\langle 2 \rangle 1$. By a similar reasoning to the one for Tactic 9, the proof rule **ADD-LOCAL-VARIABLE** justifies the proof step.

$\langle 2 \rangle 2$. Q.E.D.

CASE: Tactic 11

$\langle 2 \rangle 1$. The tactic ensures that $\mathcal{I}_2 = \exists x.\mathcal{I}_1$. Let $\mathcal{I} = \mathcal{I}_1$ for this part of the proof.

$\langle 2 \rangle 2$. By the tactic, $\mathbf{atomic}\{\bar{S}_2\} \hat{=}_{(\exists x.\mathcal{I})} \mathcal{I}$. Thus, $\mathbf{Compile}(\mathbf{atomic}\{\bar{S}_2\}) \hat{=} (\exists x.\mathcal{I})$ holds.

$\langle 2 \rangle 3$. By the tactic, $\mathcal{I} \vdash \mathbf{atomic}\{\bar{S}_1\} \hat{\preceq}_{-x} \mathbf{atomic}\{\bar{S}_2\}$ holds. Therefore, by Definition 22, $\mathcal{I} \vdash \mathbf{Compile}(\mathbf{atomic}\{\bar{S}_1\}) \preceq_{-x} \mathbf{Compile}(\mathbf{atomic}\{\bar{S}_2\})$ holds.

$\langle 2 \rangle 4$. By the definition of $\mathbf{Compile}(\cdot)$, for all $\mathbf{p} \in \mathbf{Proc}$, $\mathbf{Compile}(\hat{P}_2)$ agrees with $\mathbf{Compile}(\hat{P}_2)$

on $Body_{\mathbf{p}}$ except that every $Compile(\mathbf{atomic}\{\bar{S}_1\})$ in $\mathbf{p} \in Proc$ is replaced by $Compile(\mathbf{atomic}\{\bar{S}_2\})$.

$\langle 2 \rangle 5$. By $\langle 2 \rangle 1$, $\langle 2 \rangle 2$, $\langle 2 \rangle 3$, and $\langle 2 \rangle 4$, the proof rule HIDE-GLOBAL-VARIABLE justifies the proof step.

$\langle 2 \rangle 6$. Q.E.D.

CASE: Tactic 12

$\langle 2 \rangle 1$. By a similar reasoning to the one for Tactic 11, the proof rule HIDE-LOCAL-VARIABLE justifies the proof step.

$\langle 2 \rangle 2$. Q.E.D.

CASE: Tactic 13

$\langle 2 \rangle 1$. Assume that $(\hat{P}, \mathcal{I}) \vdash \mathbf{atomic}\{\bar{S}_1\} : \mathbb{R}$.

$\langle 2 \rangle 2$. Since $(\hat{P}, \mathcal{I}) \vdash \mathbf{atomic}\{\bar{S}_1\} : \mathbb{R}$, $(Compile(\hat{P}), \mathcal{I}) \vdash Compile(\mathbf{atomic}\{\bar{S}_1\}) : \mathbb{R}$ holds.

$\langle 2 \rangle 3$. The definitions of $Compile(\cdot)$ and $\llbracket \cdot \rrbracket$ ensure that

$Compile(\mathbf{atomic}\{\bar{S}_1; \bar{S}_2\}) \cong \llbracket Compile(\mathbf{atomic}\{\bar{S}_1\}); Compile(\mathbf{atomic}\{\bar{S}_2\}) \rrbracket$. This follows from (1) the relationship between $\mathbf{wp}(\tau, \phi)$ defined by Equation 13.2 and $\mathbf{wp}(\bar{S}_1; \bar{S}_2, \Phi, L)$ defined in Figure 14.3, and (2) the relationship between $\tau_1 \cdot \tau_2$ defined by Equation 13.3 and $\mathbf{xp}(\bar{S}_1; \bar{S}_2, \mathcal{T}, L)$ defined in Figure 14.3.

$\langle 2 \rangle 4$. By $\langle 2 \rangle 2$ and $\langle 2 \rangle 3$, the proof rule REDUCE-SEQUENTIAL-RIGHT justifies the proof step. The proof step when $(\hat{P}, \mathcal{I}) \vdash \mathbf{atomic}\{\bar{S}_2\} : \mathbb{L}$ is similarly justified by the proof rule REDUCE-SEQUENTIAL-LEFT.

$\langle 2 \rangle 5$. Q.E.D.

CASE: Tactic 15

$\langle 2 \rangle 1$. Without loss of generality, assume that $g = B$ where B is a state predicate whose free variables are all local.

$\langle 2 \rangle 2$. $Compile(\mathbf{if}(B)\{\mathbf{atomic}\{S_1\}\}\mathbf{else}\{\mathbf{atomic}\{S_2\}\}) =$
 $(Compile(\mathbf{assume } B); Compile(\mathbf{atomic}\{S_1\})) \square (Compile(\mathbf{assume } \neg B); Compile(\mathbf{atomic}\{S_2\})).$
 Since all the free variables in B are local, the gated actions corresponding to $\mathbf{assume } B$ and $\mathbf{assume } \neg B$ are both movers. Thus, $Compile(\mathbf{if}(B)\{\mathbf{atomic}\{S_1\}\}\mathbf{else}\{\mathbf{atomic}\{S_2\}\}) =$
 $Compile(\mathbf{atomic}\{\mathbf{assume } B; S_1\}) \square Compile(\mathbf{atomic}\{\mathbf{assume } \neg B; S_2\}).$

$\langle 2 \rangle 3$. By $\langle 2 \rangle 2$, the proof rule REDUCE-CHOICE justifies the proof step.

$\langle 2 \rangle 4$. Q.E.D.

CASE: Tactic 14

- ⟨2⟩1. By a similar reasoning to the one for Tactic 15, the proof rule REDUCE-CHOICE justifies the proof step.
- ⟨2⟩2. Q.E.D.

CASE: Tactic 16

- ⟨2⟩1. Assume that $(\hat{P}, \mathcal{I}) \vdash \text{atomic}\{\bar{S}_1\} : \mathbb{R}$.
- ⟨2⟩2. Since $(\hat{P}, \mathcal{I}) \vdash \text{atomic}\{\bar{S}_1\} : \mathbb{R}$, $(\text{Compile}(\hat{P}), \mathcal{I}) \vdash \text{Compile}(\text{atomic}\{\bar{S}_1\}) : \mathbb{R}$ holds.
- ⟨2⟩3. The definitions of $\text{Compile}(\cdot)$ and $\llbracket \cdot \rrbracket$ ensure that

$$\text{Compile}(\text{atomic}\{\text{try}\{\bar{S}_1\}\text{catch}(l)\{\bar{S}_2\}\}) \cong \llbracket \text{Compile}(\text{atomic}\{\bar{S}_1\}); l : \text{Compile}(\text{atomic}\{\bar{S}_2\}) \rrbracket.$$
- ⟨2⟩4. By ⟨2⟩2 and ⟨2⟩3, the proof rule REDUCE-LABELED-RIGHT justifies the proof step. The proof step when $(\hat{P}, \mathcal{I}) \vdash \text{atomic}\{\bar{S}_2\} : \mathbb{L}$ is similarly justified by the proof rule REDUCE-LABELED-LEFT.
- ⟨2⟩5. Q.E.D.

CASE: Tactic 17

- ⟨2⟩1. By a similar reasoning to the one for Tactic 16, the proof rule REDUCE-LABELED-RIGHT and REDUCE-LABELED-LEFT justify the proof step.
- ⟨2⟩2. Q.E.D.

CASE: Tactic 20

- ⟨2⟩1. By the definition of $\text{Compile}(\cdot)$, $\text{Compile}(\text{dopar}\{S_1\}\text{with}\{S_2\}) = \text{Compile}(S_1) \parallel \text{Compile}(S_2)$.
- ⟨2⟩2. The definitions of $\text{Compile}(\cdot)$ and $\llbracket \cdot \rrbracket$ ensure that

$$\text{Compile}(\text{atomic}\{\text{try}\{\bar{S}_1[X/\text{Local}_{\mathbf{p}}]\}\text{catch}\{\text{skip}\}\}) \cong \llbracket \alpha[\text{left}(\text{tid})/\text{tid}, X/\text{Local}_{\mathbf{p}}]; * : \text{skip} \rrbracket, \text{ where } \alpha = \text{Compile}(\text{atomic}\{\bar{S}_1\}).$$
- ⟨2⟩3. By a similar argument to ⟨2⟩2, $\text{Compile}(\text{atomic}\{\text{try}\{\bar{S}_2[Y/\text{Local}_{\mathbf{p}}]\}\text{catch}\{\text{skip}\}\}) \cong \beta[\text{left}(\text{tid})/\text{tid}, Y/\text{Local}_{\mathbf{p}}]; * : \text{skip}$.
- ⟨2⟩4. By the definition of $\text{Compile}(\cdot)$,

$$\text{Compile}(\text{if}(*)\{\text{atomic}\{S_1\}\}\text{else}\{\text{atomic}\{S_2\}\}) = \text{Compile}(\text{atomic}\{S_1\}) \sqcap \text{Compile}(\text{atomic}\{S_2\})$$
- ⟨2⟩5. By ⟨2⟩1, ⟨2⟩2, ⟨2⟩3, and ⟨2⟩4 the proof rule EXPAND-PARALLEL justifies the proof step.
- ⟨2⟩6. Q.E.D.

CASE: Tactic 21

- ⟨2⟩1. By a similar reasoning to the one for Tactic 20, the proof rule REDUCE-PARALLEL-LEFT and REDUCE-PARALLEL-RIGHT justify the proof step.

⟨2⟩2. Q.E.D.

CASE: Tactic 22

⟨2⟩1. The definitions of $Compile(\cdot)$ and $\llbracket \cdot \rrbracket$ ensure that

$$Compile(\text{atomic}\{\text{try}\{\bar{S}_1\}\text{catch}\{\text{block}\}\}) \cong \llbracket \alpha; *:\text{block} \rrbracket, \text{ where } \alpha = Compile(\text{atomic}\{\bar{S}_1\}).$$

⟨2⟩2. By a similar argument to ⟨2⟩1, $Compile(\text{atomic}\{\text{atomic}\{\bar{S}_2; \text{try}\{\bar{S}_1\}\text{catch}\{\text{block}\}\}\}) \cong \llbracket \beta; \alpha; *:\text{block} \rrbracket$, where $\beta = Compile(\text{atomic}\{\bar{S}_2\})$.

⟨2⟩3. By the definition of $Compile(\cdot)$, $Compile(\text{atomic}\{\bar{S}_2\}; \text{atomic}\{\bar{S}_1; \text{block}\}) = \beta; \llbracket \alpha; \text{skip}:\text{block} \rrbracket$, since $\alpha = Compile(\text{atomic}\{\bar{S}_1\})$ and $\beta = Compile(\text{atomic}\{\bar{S}_2\})$.

⟨2⟩4. By ⟨2⟩1, ⟨2⟩2, and ⟨2⟩3, the proof rule REDUCE-LOOP justifies the proof step.

⟨2⟩5. Q.E.D.

CASE: Tactic 27

⟨2⟩1. The proof rule INLINE-CALL justifies the proof step trivially.

⟨2⟩2. Q.E.D.

□

In this chapter, we introduced the notion of proving a QEDPL program and the proof tactics for doing so. We demonstrated, using the tactics, the practical application of the conceptual transformations in the core QED proof system on a high-level language. Proof tactics also form the building blocks of proof commands in our tool QED-VERIFIER presented in Chapter 17. QED-VERIFIER combines proof tactics into coarser-grained proof steps, driven by concise proof commands, in order to further simplify the application of the QED proof system in practice. While introducing our proof commands in Chapter 17, we will refer to the tactics in this chapter.

Chapter 16

PROVING LINEARIZABILITY

The previous chapters elaborated on proving assertions using the QED proof system. In this chapter, we focus on linearizability. Linearizability is a well-known correctness criterion for concurrent data-structure implementations [36]. A concurrent implementation, denoted *Impl*, is said to be *linearizable* with respect to a sequential specification, denoted *Spec*, if every concurrent operation *op* of *Impl* takes effect atomically between its call and return points, where the correct effect is described by a sequential operation *op'* in *Spec*.

The typical proof of linearizability establishes an *abstraction map*, from *Impl*-states to *Spec*-states [62], and shows that only one action of *op*, called the *commit action*, is mapped to *op'*, and other actions are mapped to stuttering (identity) transitions in *Spec*. Under fine-grained concurrency control, constructing such a proof requires considerable expertise. First, identifying the commit action becomes nontrivial when *op* is written in terms of many small actions that make visible changes to the state. It further complicates the analysis when the commit point is determined at runtime depending on thread interleavings. Second, while the abstraction map relates *Impl*-states to *Spec*-states, it must also filter out the effects of the partially completed operations of *Impl* on the state except for the commit action. This includes completing partial operations or rolling back the effects of these operations back to a clean state [103]. Under fine-grained concurrency and complicated concurrency control, defining this abstraction map requires considerable expertise.

In this chapter, we present a new method, based on the QED proof system, for proving linearizability of programs with fine-grained concurrency. Our method permits more tractable proofs by eliminating the above difficulties of constructing an abstraction map and not requiring the identification of the commit points. In Chapters 12-15, we showed that by interleaving reduction with abstraction, we can increase atomicity to the point that assertions in a concurrent program can be verified by sequential (local) reasoning. In this part of the thesis, we argue that program rewriting guided by atomicity is an effective method for

proving linearizability and present a sound proof system and a supporting tool that realize this method in a formal and practical setting.

We prove that *Impl* is linearizable with respect to *Spec*, by transforming *Impl* to *Spec* via a sequence of phases. In a *reduction* phase, we alternate reduction and abstraction to mark a set of sequentially composed actions as atomic. These actions are collected together, and the effects of thread interleavings are eliminated. In a *refinement* phase, we couple variable introduction and variable hiding, in order to make the code closer to the specification. These techniques provide us with the ability to *syntactically* relate implementation of a data structure to a specification with a different representation. We also provide the soundness guarantee that, the proof transformations, while they grow atomic blocks, preserve the behaviors of the original program. Thus, one can simplify the program by growing atomic blocks and continue the linearizability proof with another method, e.g., rely/guarantee [87] or separation logic [104]. For example, eliminating the reasoning about the interference within lock-protected blocks will simplify the proof obligations in a rely/guarantee-based proof.

Interleaving reduction and refinement phases supports the incremental construction of the abstraction map. By increasing atomicity, a reduction phase enables a following refinement phase to implicitly establish a simple and clean abstraction map towards the specification. A refinement phase also helps to improve a following reduction phase by eliminating *superficial conflicts*. A superficial conflict happens when two actions by different threads interfere with each other over the implementation variables, but the their interference does not affect the end result of the computation (here return values of procedures). This is an important limitation for reduction, and interestingly, our standard notion of abstraction on the existing variables (Section 13.3) does not help in this situation. Our solution to eliminating the conflict is to hide the variables on which the conflict happens; but, differently from the standard abstraction, introducing new variables, which will carry enough (semantic) information from the hidden variables and will not cause conflicts. This strategy indirectly introduces a semantic hierarchy into mover checks in reduction, which is not particular to linearizability and is likely to be useful in any kind of reduction proof. Sections 19.10 and 19.11 illustrate how coupling variable introduction and hiding helps us to cope with superficial conflicts while proving a non-blocking queue and a mutex-lock implementation, respectively.

We demonstrate the effectiveness and practicality of our method by proving linearizability of several tricky examples from the literature. All proofs were mechanized using the QED-VERIFIER verifier; the proof scripts are available online and reproducible using QED-VERIFIER. We explain these examples and the key points in their proofs in Chapter 19.

Outline. Section 16.1 points out strengths of our approach over classical proofs of linearizability using an example. Section 16.2 formalizes the proof of linearizability and states a related soundness theorem. Section 16.3 concludes this chapter by discussing related work.

16.1 Motivation and Overview

Our running example is a multiset of integers. Figure 16.1 shows the concurrent implementation (*Impl*), and the sequential specification (*Spec*), of *InsertPair* and *LookUp* operations¹. The instruction **assume** ϕ blocks until ϕ becomes **true**, and **havoc** x assigns a nondeterministic value to x . Our goal is to verify linearizability of *Impl* with respect to *Spec*.

Spec uses the variable *S*, which maps each integer to its cardinality in the multiset. Initially, *S* is empty, so $S[x] == 0$ for every integer x .

Impl contains an array *M* of *N* slots. For each slot, the **elt** field stores an integer, and the **stt** field indicates the status of the slot. The atomic *FindSlot* operation² allocates an empty slot by setting its **stt** field to **reserved**, and returns its index. *FindSlot* fails and returns -1 if it cannot find any empty slot. The lock of each slot is acquired and released separately by **lock** and **unlock** operations, respectively.

A typical linearizability proof establishes an abstraction map that relates the slots *M* of *Impl* to the map *S* of *Spec* and, using that map, shows that *Impl* behaves as if it is *Spec*. Let $|A|$ denote the cardinality of the set *A*. The following abstraction map expresses the programmer's design intent clearly:

$$S[x] == |\{ i \mid 0 \leq i < N \ \&\& \ M[i].elt == x \ \&\& \ M[i].stt == full \}|$$

¹We omit the *Insert* operation to simplify the explanation.

²The original implementation of *FindSlot* uses fine-grain locking, and traverses the array using a loop similar to that of *LookUp*. In order to simplify the explanation, we use a version of *FindSlot* that has already been transformed using our proof steps.

In words, for each integer x , the number of slots i in *Impl* with $M[i].elt == x$ and $M[i].stt == full$ represents $S[x]$ in *Spec*. When the proof is done at finest granularity of concurrency, more complicated variants of this abstraction map has to be used. In the following, we envision such a proof, and highlight common difficulties. We then illustrate how our proposed approach alleviates these difficulties and allows the proof to use the above map.

Implementation (*Impl*)

```

enum Status = { empty,reserved,full };
record Slot { elt: int, stt: Status };
var M: array[0..N-1] of Slot

LookUp(x:int) returns(r:bool)
  var i: int;
1 for (i := 0; i < N; i++) {
2   lock(M[i]);
3   if (M[i].elt==x && M[i].stt==full){
4     unlock(M[i]);
5     r := true; return;
6   } else unlock(M[i]);
7 }
8 r := false;

atomic FindSlot(x:int) returns (r:int)
1 if (forall 0<=i<N. M[i].stt != empty) {
2   r := -1;
3 } else {
4   assume (0<=r<N && M[r].stt==empty);
5   M[r].stt := reserved;
6 }

InsertPair(x:int, y:int) returns (r:bool)
  var i,j: int;
1 i := FindSlot(x);
2 if (i == -1) {
3   r := false; return;
4 }
5 j := FindSlot(y);
6 if (j == -1) {
7   M[i].stt := empty;
8   r := false; return;
9 }
10 M[i].elt := x;
11 M[j].elt := y;
12 lock(M[i]);
13 lock(M[j]);
14 M[i].stt := full;
15 M[j].stt := full;
16 unlock(M[i]);
17 unlock(M[j]);
18 r := true;

```

Specification (*Spec*)

```

var S: array [int] of int;

atomic LookUp(x:int) returns (r:bool)
  r := (S[x] > 0);

atomic InsertPair(x:int, y:int) returns (r:bool)
  if(r) { S[x] := S[x] + 1; S[y] := S[y] + 1; }

```

Figure 16.1: The concurrent implementation and the sequential specification of multiset

16.1.1 Challenges in a Typical Refinement Proof for Multiset

Abstraction Maps and Commit Points

Many techniques work by first selecting a commit point in every operation. The most likely choice for the commit point for `InsertPair` is line 16, since releasing the first lock makes the inserted element `M[i].elt` visible to other threads. Consider an abstraction map from *Impl* to *Spec* and suppose that line 16 of `InsertPair` is executed by *Impl*. This transition must be mapped to a single transition that increments `S[x]` and `S[y]` atomically. As a first try, let us consider the simple abstraction map introduced above:

$$S[x] == |\{ i \mid 0 \leq i < N \ \&\& \ M[i].elt == x \ \&\& \ M[i].stt == full \} |$$

This map does not work with this choice of commit point, because when lines 14 and 15 of `InsertPair` are executed, `S[x]` and `S[y]` are incremented, but the execution has not reached the commit point yet. In addition, the updates that are propagated to `S` are not atomic. Our next, slightly more sophisticated map below does not update `S[x]` and `S[y]` while the locks to these cells are held. (`HeldBy(M[i], t)` is `true` when thread `t` is holding the lock of `M[i]`):

$$S[x] == |\{ i \mid 0 \leq i < N \ \&\& \ M[i].elt == x \ \&\& \ M[i].stt == full \ \&\& \ !HeldBy(M[i], t) \} |$$

The problem with this map is that every slot locked by a thread would be excluded from `S`. As a result, at line 16 (the commit point) the map would increment `S[x]` but not `S[y]` since `M[j]` is still locked. Thus, this map still does not accomplish the atomic specification state update we are after. The right map has to complete this partial update at the commit point by incrementing `S[y]` as well although the lock of `M[j]` is still held.

We next try different selections of commit points: lines 14, 15 or 17. For each of these choices, in order to produce the intended specification state and avoid non-atomic updates to it, an abstraction map must “roll back” effects of executions of `InsertPair` that have not reached their commit point, and must “complete” the effects of others that are past their commit point but have not yet finished. To accomplish this, the map must refer to not only the locking state but also the program counters of all threads.

Non-fixed Commit Points

Another issue that complicates the linearizability proof for multiset is that the commit action of **LookUp** is not fixed, but depends on the concurrently executing insertions by other threads. Consider a run in which **LookUp**(**x**) executes. If **LookUp**(**x**) returns **true** its commit action is at line 3, where it finds out that the slot being visited contains **x** and is valid. When **LookUp**(**x**) fails, its commit point must be chosen as the first read of a slot it performs or earlier. This is because, in the absence of a **Delete** operation, it is possible that **x** gets inserted into a slot **M**[**i**], after **LookUp** visits the i^{th} slot and fails to find **x**, therefore, the commit point cannot be past the first read of a slot. Techniques that depend on the existence of a fixed commit point would be ineffective in such situations [87].

16.1.2 Proof by Reduction and Abstraction

Our approach uses atomicity as a proof tool to simplify the above reasoning about the abstraction map and commit points. Observe that the code blocks between lines 12-17 of **InsertPair** is *atomic*, i.e., any execution in which the actions of this block are interleaved with actions from other threads can be transformed into one in which actions of the commit block are contiguous. The technique we present allows us to express this fact and use it in a sound manner in a refinement or linearizability proof. Being able to treat the commit block as a single atomic action eliminates all of the potential difficulties outlined above.

In our method the proof is constructed by transforming *Impl* to *Spec*, both shown in Figure 16.1. This is done through a reduction phase followed by a refinement phase. In the reduction phase, we reduce the bodies of **InsertPair** and **LookUp** to single atomic actions. This phase is guided by a simple hint about the locking discipline (see [105] for details of automating reduction).

In order to handle the non-fixed commit points of **LookUp**, we apply a transformation to separate its succeeding and failing executions. Since each failing iteration of **LookUp** is a left mover, the failing branch of **LookUp** trivially reduces to a single action. For the successful branch, we apply an abstraction to the failing iterations that makes them also right movers, and combine the abstracted iterations with the final, successful iteration. This reduces the successful branch into an atomic action. At the end, we obtain **LookUp** as a single atomic action that summarizes both successful and failing executions of the original code.

After transforming `InsertPair` and `LookUp` to single atomic actions, the locking state becomes unnecessary. We use variable hiding to clean up the calls to `lock` and `unlock`. Finally, we arrive at the representation of the multiset in *Spec* in three proof steps. First, we introduce the *Spec* variable `S` to the current version of the program. Then, we add (and prove) the following invariant, which links the new variable to the array `M`:

$$S[x] == |\{ i \mid 0 \leq i < N \ \&\& \ M[i].elt == x \ \&\& \ M[i].stt == full \} |$$

Recall that the above invariant establishes the simplest abstraction map that reflects the programmer's design intent. The invariant allows us to add the assignments `S[x] := S[x] + 1`; and `S[y] := S[y] + 1`; at the end of `InsertPair`. We follow the introduction of `S` with a variable hiding step in which we replace the bodies of `InsertPair` and `LookUp` with the corresponding bodies in *Spec* (Figure 16.1). Our soundness theorem given in Section 16.2 guarantees that transforming *Impl* to *Spec* using our rules implies the linearizability of *Impl*.

What is noteworthy about the proof we outlined is that it handles two separate concerns in separate proof steps: 1) concurrency control using locking and the `stt` field, and 2) relating the array-based representation of *Impl* to the representation in *Spec*. This example does not illustrate the use of variable hiding to eliminate superficial conflicts. Sections 19.10 and 19.11 provide two examples from our benchmarks that do.

16.2 The Verification Approach and Its Soundness

In this section, we formally describe the verification approach and state its soundness. In this approach, the implementation of a data structure is represented by a program denoted *Impl* and the specification of the data structure is represented by an “atomic” program denoted *Spec*. A program *P* is called an *atomic program* if for every $\mathbf{p} \in Proc_P$, $Body_{\mathbf{p}}$ is an atomic action. The claim “*Impl* is linearizable with respect to *Spec*” is proved by performing a sequence of proof transformations $(P_1, true) \dashrightarrow (P_2, \mathcal{I}_2) \cdots \dashrightarrow (P_{n-1}, \mathcal{I}_{n-1}) \dashrightarrow (P_n, \mathcal{I}_n)$ such that $P_1 = Impl$, $P_n = Spec$. Here we use notations for ACTIONPL programs and every step is guided by a proof rule. However, the same reasoning can be applied to proofs over QEDPL programs using proof tactics.

To simplify the presentation, we make the following assumptions without loss of generality:

- Each procedure has one input and one output variable. Let us call these variables $in_{\mathbf{p}}$ and $out_{\mathbf{p}}$, respectively, so $\vec{In}_{\mathbf{p}} = \{in_{\mathbf{p}}\}$ and $\vec{Out}_{\mathbf{p}} = \{out_{\mathbf{p}}\}$. The reasoning in this section holds when there are more input and output variables, having $in_{\mathbf{p}}$ and $out_{\mathbf{p}}$ represent multiple variables. The case where a procedure has no input or output variables can also be modeled by allowing $in_{\mathbf{p}}$ and $out_{\mathbf{p}}$ carry arbitrary values.
- Each thread calls one procedure $\mathbf{p}$ from $Proc$, and terminates when $\mathbf{p}$ returns. Scenarios in which a thread makes multiple calls can be emulated by using extra threads and scheduling restrictions.

In the following, we establish a link between programs $Impl$ and $Spec$ in the context of proving linearizability. For this, we first give the generic notion of linearizability. Then we define a special kind of simulation, called *behavioral simulation*, that relates two programs through their observable behaviors over procedure input and output values, and argue that at every point during a QED proof, this relationship between the initial and the current program holds. We finally establish a connection between linearizability and behavioral simulation and show that proving linearizability using QED transformations is sound.

16.2.1 Linearizability

We now formalize the generic notion of linearizability in [36]. In this notion a run of the program is described by a sequence of meta-actions *invoke* and *response*. The sequence of these meta-actions is called the *history* of the run. The history of the run ρ is denoted $\mathbf{H}_{\rho}$, and $\mathbf{H}(i)$ denotes the i^{th} meta-action in $\mathbf{H}$. Let $Tid_{\mathbf{H}}$ refer to the set Tid_{ρ} of thread ids where $\mathbf{H}$ is a history of the run ρ . Let $\mathbf{H}|t$ be the sub-history of $\mathbf{H}$ projected on the thread t , i.e., the meta-actions executed by t .

An *invoke* action $inv\langle\mathbf{p}, x, t\rangle$ describes a call to a procedure $\mathbf{p}$ with the arguments x (for $in_{\mathbf{p}}$) by thread t . A *response* action $res\langle\mathbf{p}, y, t\rangle$ describes a return from the procedure $\mathbf{p}$ with return values y (via $in_{\mathbf{p}}$) in thread t . We will often omit the elements of the meta-actions and refer to them using the dotted notation (e.g., $inv.\mathbf{p}$).

We say that (inv, res) is a *pair of matching invoke and response* in $\mathbf{H}$, if $\mathbf{H}|t(i) = inv$ and $\mathbf{H}|t(i+1) = res$ for some thread t and index i . We say that a history $\mathbf{H}$ is *sequential*, if for each matching pair (inv, res) in $\mathbf{H}$, $\mathbf{H}(i) = inv$ and $\mathbf{H}(i+1) = res$ for some index i .

Here we focus on successful runs, so for each ρ , $\mathbf{H}_\rho$ *complete*: it contains only invocations and their matching responses. In addition, we consider only *well-formed* histories: $\mathbf{H}$ is well-formed if for each $t \in \text{Tid}_{\mathbf{H}}$ $\mathbf{H}|t$ is sequential.

We say $\mathbf{H}$ and $\mathbf{H}'$ are *equivalent*, denoted by $\mathbf{H} \cong \mathbf{H}'$, if $\text{Tid}_{\mathbf{H}} = \text{Tid}_{\mathbf{H}'}$ and $\forall t \in \text{Tid}_{\mathbf{H}}. \mathbf{H}|t = \mathbf{H}'|t$.

Given a history $\mathbf{H}$, we define *response-invoke preservation relation* $\preceq_{\mathbf{H}}$ as follows:

Two matching invoke-response pairs (inv, res) and $(\text{inv}', \text{res}')$ in $\mathbf{H}$ are ordered, denoted, $(\text{inv}, \text{res}) \preceq_{\mathbf{H}} (\text{inv}', \text{res}')$, if $\mathbf{H}(i) = \text{res}$ and $\mathbf{H}(j) = \text{inv}'$ for some $i < j$.

A history $\mathbf{H}$ is linearizable to a sequential history $\mathbf{H}'$ if $\mathbf{H} \cong \mathbf{H}'$ and $\preceq_{\mathbf{H}} \subseteq \preceq_{\mathbf{H}'}$. We say P is linearizable to P' from $\mathcal{I}$ to restrict the definition of linearizability to runs of P and P' from $\mathcal{I}$.

Definition 24 (Linearizability):

Let P be a program, P' an atomic program, and $\mathcal{I}$ an assert predicate. Let $X_1 = \text{fv}(\mathcal{I}) \setminus \text{Var}_P$ and $X_2 = \text{fv}(\mathcal{I}) \setminus \text{Var}_{P'}$. A program P is linearizable to P' from $\mathcal{I}$ if $\text{Sig}(P) = \text{Sig}(P')$ and for every successful run ρ of P from $\exists X_1.\mathcal{I}$, there exists a successful run r' of P' from $\exists X_2.\mathcal{I}$ such that $\mathbf{H}_\rho$ is linearizable to $\mathbf{H}_{r'}$. ■

16.2.2 Behavioral Simulation

Let $r = s_1 \xrightarrow{1} s_n$ be a successful run of the program. Let $\mathbf{p}$ be the procedure executed by t . We call the tuple $(t, \mathbf{p}, \sigma_{s_1}(t, \text{in}_{\mathbf{p}}), \sigma_{s_n}(t, \text{out}_{\mathbf{p}}))$ the behavior of t in ρ and denote it by $\text{beh}(\rho, t)$. The behavior includes the name of the procedure called by t , along with the values of the input and the output variables of the procedure³. We write $\text{Beh}(\rho)$ to denote $\{\text{beh}(\rho, t) \mid t \in \text{Tid}(\rho)\}$. We lift the definition to programs; $\text{Beh}(P, \mathcal{I})$ is the collection of all behaviors of the program from invariant $\mathcal{I}$.

We define $\text{fst}(\rho, t)$ and $\text{lst}(\rho, t)$ be the indices of first and the last actions of t in ρ . Formally, with $L = \{i \mid r(i) = (t, \alpha)\}$, $\text{fst}(\rho, t) = \min(L)$ and $\text{lst}(\rho, t) = \max(L)$. Let $\ll_\rho$ be a partial order over Tid_ρ ordering threads that do not execute concurrently: $t \ll_\rho u$ if $\text{lst}(\rho, t) < \text{fst}(\rho, u)$.

³Notice that the first and the last states of the run provide us the values of $\overrightarrow{\text{In}}_{\mathbf{p}}$ and $\overrightarrow{\text{Out}}_{\mathbf{p}}$, respectively.

Definition 25 (Behavioral simulation between programs):

Let P_1 and P_2 be two programs with $\text{Sig}(P_1) = \text{Sig}(P_2)$, and let $\mathcal{I}$ be a state predicate. Let $X_1 = \text{fv}(\mathcal{I}) \setminus \text{Global}_{P_1}$ and $X_2 = \text{fv}(\mathcal{I}) \setminus \text{Var}_{P_2}$. P_2 behaviorally-simulates P_1 from $\mathcal{I}$, denoted $\mathcal{I} \vdash P_1 \trianglelefteq P_2$ if for each successful run ρ of program P_1 from $\exists X_1.\mathcal{I}$, there exists a run ρ' of P_2 from $\exists X_2.\mathcal{I}$ such that:

1. $\text{Beh}(\rho) = \text{Beh}(\rho')$ and

2. $\ll_{\rho} \subseteq \ll_{\rho'}$ ■

In the following, we will prove that each good program reached during a QED-based proof behaviorally simulates the initial program.

We define new notation for state transitions as follows:

- $\sigma_1 \xrightarrow{l} \sigma_2$ holds if there exists ϵ_1, ϵ_2 such that $(\sigma_1, \epsilon_1) \xrightarrow{l} (\sigma_2, \epsilon_2)$
- $\epsilon_1 \xrightarrow{l} \epsilon_2$ holds if there exists σ_1, σ_2 such that $(\sigma_1, \epsilon_1) \xrightarrow{l} (\sigma_2, \epsilon_2)$

Note that, $\sigma_1 \xrightarrow{l} \sigma_2$ and $\epsilon_1 \xrightarrow{l} \epsilon_2$ together imply $(\sigma_1, \epsilon_1) \xrightarrow{l} (\sigma_2, \epsilon_2)$.

Definition 26 (Behaviorally-simulating runs):

Let P and P' be two programs with $\text{Sig}(P) = \text{Sig}(P')$, and let $\mathcal{I}$ be a state predicate. Let ρ and ρ' be runs of P and P' , respectively. The run ρ' is said to behaviorally-simulate ρ , denoted $\rho \blacktriangleleft \rho'$, if 1) $\text{Beh}(\rho) = \text{Beh}(\rho')$, and 2) $\ll_{\rho} \subseteq \ll_{\rho'}$. ■

For statement c , we define $c[\beta/\alpha]$ be the statement in which every occurrence of α is replaced by β . In addition, $\theta[\beta/\alpha]$ and $P[\beta/\alpha]$ are defined to be θ and P in which the same replacement is applied to every $\theta(t)$ and $\text{Body}(\mathbf{p})$, respectively.

The following lemma states that after replacing α with β in the program, the sequences of θ -transitions remain the same.

Lemma 6 (Behavioral simulation and code maps):

Let P be a program. The following holds.

- For $t \neq u$, if $\theta_1 \xrightarrow{(t,\alpha)(u,\beta)} \theta_2$ be a run of P , then $\theta_1 \xrightarrow{(u,\beta)(t,\alpha)} \theta_2$ is a run of P .
- If $\theta_1 \xrightarrow{(t,\alpha)} \theta_2$ be a run of P , then $\theta_1[\beta/\alpha] \xrightarrow{(t,\alpha)} \theta_2[\beta/\alpha]$ is a run of $P[\beta/\alpha]$.

- If $\theta_1 \xrightarrow{(t,\alpha)(t,\beta)} \theta_2$ be a run of P , then $\theta_1[\gamma/(\alpha;\beta)] \xrightarrow{(t,\gamma)} \theta_2[\gamma/(\alpha;\beta)]$ is a run of $P[\gamma/(\alpha;\beta)]$.
- If $\theta_1 \xrightarrow{(t,\alpha)} \theta_2$ be a run of P , then $\theta_1[\gamma/(\alpha \square \beta)] \xrightarrow{(t,\gamma)} \theta_2[\gamma/(\alpha \square \beta)]$ is a run of $P[\gamma/(\alpha;\beta)]$.
- If $\theta_1 \xrightarrow{(t,\beta)} \theta_2$ be a run of P , then $\theta_1[\gamma/(\alpha \square \beta)] \xrightarrow{(t,\gamma)} \theta_2[\gamma/(\alpha \square \beta)]$ is a run of $P[\gamma/(\alpha;\beta)]$.
- If $\theta_1 \xrightarrow{(t,\alpha_1)\dots(t,\alpha_n)} \theta_2$ be a run of P , then $\theta_1[\beta/\alpha^\odot] \xrightarrow{(t,\beta)} \theta_2[\beta/\alpha^\odot]$ is a run of $P[\beta/\alpha^\odot]$. ■

PROOF. The claims of the lemma hold by the operational semantics of the language given in Section 12.4.5. $\square$

We will extensively use the following properties of $\blacktriangleleft$ in our proofs.

Lemma 7 ($\blacktriangleleft$ is reflexive and transitive):

$\blacktriangleleft$ is reflexive and transitive. ■

PROOF. Equivalence on Beh and $\subseteq$ relation on $\ll$ are both reflexive and transitive. Thus, $\blacktriangleleft$ is trivially reflexive and transitive. $\square$

The following lemma states that final state of a run determines the behavior of that run.

Lemma 8 (Final state determines the behavior):

Let $(P, \mathcal{I})$ be a proof state such that P is good from $\mathcal{I}$. Let $\rho = \rho_1 \xrightarrow{1} \rho_n$ and $\rho' = \rho'_1 \xrightarrow{1'} \rho'_m$ are two runs of P from $\mathcal{I}$ such that $Tid(\rho) = Tid(\rho')$ and every thread calls the same procedure in ρ and ρ' . If $\sigma_{\rho_n} = \sigma_{\rho'_m}$, then $\text{Beh}(\rho) = \text{Beh}(\rho')$. ■

PROOF. The behaviors are encoded by input and output variables of the threads, which are all local. For each thread $t \in Tid(\rho)$ let $\mathbf{p}$ be the procedure called by t . The input variables $\overrightarrow{In}_{\mathbf{p}}$ are read-only, and output variables $\overrightarrow{Out}_{\mathbf{p}}$ are only written by t . Thus, the behavior $\text{beh}(\rho, t)$ can be determined by only looking at the values $\overrightarrow{In}_{\mathbf{p}}$ and $\overrightarrow{Out}_{\mathbf{p}}$ at ρ_n and ρ_m . Thus, the equivalence of the final states implies the equivalence of the behaviors. $\square$

Lemma 9 (Behavioral simulation and right movers):

Let $(P, \mathcal{I})$ be a proof state such that P is good from $\mathcal{I}$, and α be action that is right mover. Let $\rho = \rho_1 \xrightarrow{1_1} \rho_i \xrightarrow{(t,\alpha)1} \rho_j \xrightarrow{(t,\gamma)1_2} r_n$ be a run of P from $\mathcal{I}$, such that $1 = (u_1, \beta_1)(u_2, \beta_2) \dots (u_m, \beta_m)$ and $t \notin \{u_1, \dots, u_m\}$. There exists a run $\rho' = \rho_1 \xrightarrow{1_1} \rho_i \xrightarrow{1(t,\alpha)} \rho_j \xrightarrow{(t,\gamma)1_2} r_n$ of P from $\mathcal{I}$, such that $\rho \blacktriangleleft \rho'$. ■

PROOF.

ASSUME: $(P, \mathcal{I}) \vdash \alpha : \mathbb{R}$

$\langle 1 \rangle 1$. Induction on the length (m) of the sequence $\mathbf{l} = (u_1, \beta_1)(u_2, \beta_2) \cdots (u_m, \beta_m)$

CASE: Base case

$\langle 2 \rangle 1$. The claim trivially holds when $m = 0$, by taking $\rho = \rho'$.

CASE: Inductive case

$\langle 2 \rangle 1$. Consider the run $\rho = \rho_1 \xrightarrow{\mathbf{l}_1} \rho_i \xrightarrow{(t, \alpha)\mathbf{l}} \rho_k \xrightarrow{(u_{m+1}, \beta_{m+1})} \rho_j \xrightarrow{(t, \gamma)\mathbf{l}_2} \rho_n$ such that $t \notin \{u_1, \dots, u_m, u_{m+1}\}$.

$\langle 2 \rangle 2$. By the inductive hypothesis, there exists a run $\rho'' = \rho_1 \xrightarrow{\mathbf{l}_1} \rho_i \xrightarrow{\mathbf{l}(t, \alpha)} \rho_k \xrightarrow{(u_{m+1}, \beta_{m+1})} \rho_j \xrightarrow{(t, \gamma)\mathbf{l}_2} \rho_n$ such that $\rho'' \triangleleft \rho$.

PROVE: There exists $\rho' = \rho_1 \xrightarrow{\mathbf{l}_1} \rho_i \xrightarrow{\mathbf{l}(u_{m+1}, \beta_{m+1})(t, \alpha)} \rho_j \xrightarrow{(t, \gamma)\mathbf{l}_2}$.

$\langle 3 \rangle 1$. Let ρ'' in $\langle 2 \rangle 2$ be as follows: $\rho'' = \rho_1 \xrightarrow{\mathbf{l}_1} \rho_i \xrightarrow{\mathbf{l}} \rho_z \xrightarrow{(t, \alpha)} \rho_k \xrightarrow{(u_{m+1}, \beta_{m+1})} \rho_j \xrightarrow{(t, \gamma)\mathbf{l}_2} \rho_n$

$\langle 3 \rangle 2$. $\sigma_{\rho_z} \models \phi_{\beta_{m+1}} \wedge \text{wp}(\tau_{\beta_{m+1}}, \phi_\alpha)$, since P is good from $\mathcal{I}$.

$\langle 3 \rangle 3$. $\sigma_{\rho_z} \xrightarrow{(u_{m+1}, \beta_{m+1})(t, \alpha)} \sigma_{\rho_j}$ holds, since $(P, \mathcal{I}) \vdash \alpha : \mathbb{R}$, and $\tau_{\beta_{m+1}} \cdot \tau_\alpha \Rightarrow \tau_\alpha \cdot \tau_{\beta_{m+1}}$.

$\langle 3 \rangle 4$. $\theta_{\rho_z} \xrightarrow{(u_{m+1}, \beta_{m+1})(t, \alpha)} \theta_{\rho_j}$ holds, by $t \neq u_{m+1}$ and Lemma 6.

$\langle 3 \rangle 5$. $\rho_z \xrightarrow{(u_{m+1}, \beta_{m+1})} \rho_s \xrightarrow{(t, \alpha)} \rho_j$ for some ρ_s by $\langle 3 \rangle 3$ and $\langle 3 \rangle 4$.

PROVE: $\rho \triangleleft \rho'$

$\langle 3 \rangle 1$. $\rho \triangleleft \rho''$ by the inductive hypothesis ($\langle 2 \rangle 2$).

PROVE: $\rho'' \triangleleft \rho'$

$\langle 4 \rangle 1$. Since the final states of ρ'' and ρ' are equivalent, $\text{Beh}(\rho'') = \text{Beh}(\rho')$ by Lemma 8.

$\langle 4 \rangle 2$. The only way to break a relation in $\ll_{\rho''}$, by commuting (t, α) with (u_{m+1}, β_{m+1}) is when α is the last action of t and β_{m+1} is the first action of u_{m+1} . However, this contradicts with the existence of γ . Thus $\ll_{\rho''} \subseteq \ll_{\rho'}$

$\langle 3 \rangle 2$. $\rho \triangleleft \rho'$ by $\langle 3 \rangle 1$, $\rho'' \triangleleft \rho'$, and $\triangleleft$ is transitive.

□

Lemma 10 (Behavioral simulation and left movers):

Let $(P, \mathcal{I})$ be a proof state such that P is good from $\mathcal{I}$, and γ be action that is left mover.

Let $\rho = \rho_1 \xrightarrow{\mathbf{l}_1(t, \alpha)} \rho_i \xrightarrow{\mathbf{l}(t, \gamma)} \rho_j \xrightarrow{\mathbf{l}_2} r_n$ be a successful, run of P from $\mathcal{I}$, such that $\mathbf{l} = (u_1, \beta_1)(u_2, \beta_2) \cdots (u_m, \beta_m)$ and $t \notin \{u_1, \dots, u_n, u_{n+1}\}$. There exists a run $\rho' = \rho_1 \xrightarrow{\mathbf{l}_1(t, \alpha)} \rho_i \xrightarrow{(t, \gamma)\mathbf{l}} \rho_j \xrightarrow{\mathbf{l}_2} r_n$ of P from $\mathcal{I}$, such that $\rho \triangleleft \rho'$. ■

PROOF. The proof is similar to that of Lemma 9. $\square$

We now state and prove the main soundness theorem of this chapter.

Theorem 9 (Preservation of $\leq$ by the QED proof rules):

Let $(P_1, \mathcal{I}_1) \dashrightarrow \dots \dashrightarrow (P_n, \mathcal{I}_n)$ be a sequence of proof steps and $\mathcal{I}$ be a state predicate such that for all $i \in [1, n]$, $\mathcal{I} \Rightarrow \mathcal{I}_i$. Let $V = \text{Global}_{P_1} \cap \text{Global}_{P_n}$, $X_1 = \text{fv}(\mathcal{I}) \setminus \text{Global}_{P_1}$, and $X_n = \text{fv}(\mathcal{I}) \setminus \text{Global}_{P_n}$. If P_n is good from $\exists X_n. \mathcal{I}$, then for all $1 \leq i \leq n$, $\mathcal{I} \vdash P_1 \leq P_i$ holds. $\blacksquare$

PROOF.

ASSUME: Since P_n is good from $\exists X_n. \mathcal{I}$, for all $1 \leq i \leq n$, P_i is good from $\exists X_1. \mathcal{I}$, by Theorem 5.

$\langle 1 \rangle 1$. Proof by induction on the length of the proof, i.e. the number of proof states reached during the proof.

CASE: Base case

$\langle 2 \rangle 1$. There is only one proof state $(P_1, \mathcal{I}_1)$.

$\langle 2 \rangle 2$. The claim holds trivially for $(P_1, \mathcal{I}_1)$, since $P_1 \leq_{\mathcal{I}_1} P_1$, by definition of $\leq$ and Lemma 7.

CASE: Inductive case

$\langle 2 \rangle 1$. Consider part of proof: $(P_1, \mathcal{I}_1) \dashrightarrow \dots \dashrightarrow (P_i, \mathcal{I}_i) \dashrightarrow (P_{i+1}, \mathcal{I}_{i+1}) \dashrightarrow \dots \dashrightarrow (P_n, \mathcal{I}_n)$

ASSUME: For all $1 \leq j \leq i$, $P_1 \leq_{\mathcal{I}_n} P_j$ holds, by inductive hypothesis.

PROVE: $P_1 \leq_{\mathcal{I}_n} P_{i+1}$.

$\langle 3 \rangle 1$. For any $1 \leq k \leq n$, let $X_k = \text{Var}_{P_k} \setminus \text{fv}(\mathcal{I}_n)$.

ASSUME: A run $\rho = \rho_1 \xrightarrow{1} \rho_N$ of P_1 from $\exists X_1. \mathcal{I}_n$.

PROVE: There exists a run ρ' of P_{n+1} from $\exists X_{i+1}. \mathcal{I}_n$ such that $\rho \blacktriangleleft \rho'$.

$\langle 4 \rangle 1$. Case split by the proof rule.

CASE: rule INVARIANT: Replace invariant $\mathcal{I}_1$ with $\mathcal{I}_2$ if $\alpha \rightleftharpoons \mathcal{I}_2$ for all the actions α in P , and $\mathcal{I}_2 \Rightarrow \mathcal{I}_1$.

$\langle 5 \rangle 1$. $P_i = P_{i+1}$, so taking $\rho' = \rho$ makes the claim hold, since $\rho \blacktriangleleft \rho$ by Lemma 7.

$\langle 5 \rangle 2$. Q.E.D.

CASE: Rule ABSTRACT: Replace the action α with action β if $\beta \rightleftharpoons \mathcal{I}$ and $(P, \mathcal{I}) \vdash \alpha \preceq \beta$.

ASSUME: $(P_i, \mathcal{I}_i) \vdash \alpha \preceq \beta$.

⟨5⟩1. $\mathcal{I}_{n+1} = \mathcal{I}_n$, by proof rule.

⟨5⟩2. $X_i = X_{i+1}$, since $\text{Var}_{P_i} = \text{Var}_{P_{i+1}}$, by proof rule.

⟨5⟩3. Construct ρ' from ρ inductively in the following way.

CASE: Base case

⟨6⟩1. Take $\rho'_1 = (\sigma_{\rho_1}, \theta_{\rho_1}[\beta/\alpha])$.

CASE: Inductive case

ASSUME: $\rho'_1 \longrightarrow r'_j$ so that $\forall 1 \leq k \leq j. r'_k = (\sigma_{\rho_k}, \theta_{\rho_k}[\beta/\alpha])$.

ASSUME: Transition $\rho_j \xrightarrow{(t,\gamma)} \rho_{j+1}$.

PROVE: $(\sigma_{\rho_j}, \theta_{\rho_j}[\beta/\alpha]) \xrightarrow{(t,\gamma)} (\sigma_{\rho_{j+1}}, \theta_{\rho_{j+1}}[\beta/\alpha])$

⟨7⟩1. If $\gamma \neq \alpha$, the claim trivially holds by operational semantics of atomic actions, and Lemma 6.

⟨7⟩2. Let $\gamma = \alpha$.

⟨7⟩3. $\sigma_{\rho_j} \xrightarrow{(t,\beta)} \sigma_{\rho_{j+1}}$, since $(P, \mathcal{I}) \vdash \alpha \preceq \beta$, and P is good from $\mathcal{I}$.

⟨7⟩4. $\theta_{\rho_j}[\beta/\alpha] \xrightarrow{(t,\beta)} \theta_{\rho_{j+1}}[\beta/\alpha]$, by Lemma 6.

⟨7⟩5. $(\sigma_{\rho_j}, \theta_{\rho_j}[\beta/\alpha]) \xrightarrow{(t,\alpha)} (\sigma_{\rho_{j+1}}, \theta_{\rho_{j+1}}[\beta/\alpha])$ by ⟨7⟩3 and ⟨7⟩4.

⟨7⟩6. Q.E.D.

PROVE: $\rho \blacktriangleleft \rho'$

⟨6⟩1. By the above construction, $\text{Pid}(\rho) = \text{Pid}(\rho')$, and every procedure executes the same procedure in both ρ and ρ' .

⟨6⟩2. By the above construction, $\forall 1 \leq j \leq N. r_j = r'_j$.

⟨6⟩3. $\text{Beh}(\rho) = \text{Beh}(\rho')$ by ⟨6⟩1, ⟨6⟩2, and Lemma 8.

⟨6⟩4. For every thread $t \in \text{Pid}(\rho)$, $\text{fst}(\rho, t) = \text{fst}(\rho', t)$ and $\text{lst}(\rho, t) = \text{lst}(\rho', t)$.

Thus $\ll_{\rho} = \ll_{\rho'}$

⟨6⟩5. Q.E.D.

⟨5⟩4. By ⟨5⟩2, $\rho_1 \models \exists X_n. \mathcal{I}_n$ implies $\rho'_1 \models \exists X_{i+1}. \mathcal{I}_n$. Thus ρ' is a run of P_{i+1} from $\exists X_{i+1}. \mathcal{I}_n$ such that $\rho \blacktriangleleft \rho'$.

⟨5⟩5. Q.E.D.

CASE: Rule REDUCE-SEQUENTIAL: Replace occurrences of $\alpha ; \gamma$ with $\llbracket \alpha ; \gamma \rrbracket$ if either

$(P, \mathcal{I}) \vdash \alpha : \mathbb{R}$ or $(P, \mathcal{I}) \vdash \gamma : \mathbb{L}$.

⟨5⟩1. $\mathcal{I}_{n+1} = \mathcal{I}_n$ by proof rule

⟨5⟩2. $X_i = X_{i+1}$, since $\text{Var}_{P_i} = \text{Var}_{P_{i+1}}$ by proof rule.

⟨5⟩3. We do the proof for the cases in which α is right mover and γ is left mover separately.

CASE: $(P_i, \mathcal{I}_i) \vdash \alpha : \mathbb{R}$

⟨6⟩1. Construct ρ'' , a run of P_i from ρ , such that $\rho \blacktriangleleft \rho''$ inductively (on the number of (α, γ) pairs not appearing adjacent in ρ).

CASE: Base case

⟨8⟩1. Take $\rho'' = \rho$, so $\rho \blacktriangleleft \rho''$.

CASE: Inductive case

ASSUME: $\longrightarrow \rho'' = \rho_1 \xrightarrow{\mathbf{l}_1} \rho_i \xrightarrow{(t, \alpha)\mathbf{l}(t, \gamma)} \rho_j \xrightarrow{\mathbf{l}_2} r_N$, such that $\mathbf{l} = (u_1, \beta_1)(u_2, \beta_2) \cdots (u_m, \beta_m)$, $t \notin \{u_1, \dots, u_m, u_{m+1}\}$, and $\rho \blacktriangleleft \rho''$

⟨8⟩1. $\rho''' = \rho_1 \xrightarrow{\mathbf{l}_1} \rho_i \xrightarrow{\mathbf{l}(t, \alpha)(t, \gamma)} \rho_j \xrightarrow{\mathbf{l}_2} r_n$ exists such that $\rho'' \blacktriangleleft \rho'''$, by Lemma 9.

⟨8⟩2. The number of pairs α, γ in ρ''' is one less than ρ'' .

⟨7⟩1. Output run ρ'' such that for every $\rho''(k) = (t, \alpha)$ for some thread t , $\rho''(k+1) = (t, \gamma)$, i.e., all (α, γ) pairs appear adjacent to each other. As a result of the above induction, $\rho \blacktriangleleft \rho''$.

⟨7⟩2. Q.E.D.

⟨6⟩2. Construct ρ' , a run of P_{i+1} from ρ'' inductively (on the length of the run).

CASE: Base case

⟨8⟩1. Take $\rho'_1 = \rho''_1$.

CASE: Inductive case

ASSUME: $\rho' = \rho''_1 \xrightarrow{\mathbf{l}} \rho''_k$ for some $\mathbf{l}$.

ASSUME: Transition $\rho''_k \xrightarrow{(t, \alpha)} \rho''_{k+1} \xrightarrow{(t, \gamma)} \rho''_{k+2}$.

PROVE: $(\sigma_{\rho''_k}, \theta_{\rho''_k}[\llbracket \alpha; \gamma \rrbracket / \alpha; \beta]) \xrightarrow{(t, \llbracket \alpha; \gamma \rrbracket)} (\sigma_{\rho''_{k+2}}, \theta_{\rho''_{k+2}}[\llbracket \alpha; \gamma \rrbracket / \alpha; \gamma])$.

⟨9⟩1. $\llbracket \alpha; \gamma \rrbracket = \text{assert}(\phi_\alpha \wedge \text{wp}(\tau_\alpha, \phi_\gamma)); (\tau_\alpha \cdot \tau_\gamma)$, by definition of $\llbracket \cdot \rrbracket$.

⟨9⟩2. Since the run ρ is successful, $\rho''_k \models \phi_\alpha$, and in addition $\rho''_{k+1} \models \phi_\gamma$.

⟨9⟩3. $\rho''_k \models \text{wp}(\tau_\alpha, \phi_\gamma)$

⟨9⟩4. $(\tau_\alpha(\rho_k, r''_{k+1}) \wedge \tau_\gamma(\rho''_{k+1}, r''_{k+2})) \Rightarrow (\tau_\alpha \cdot \tau_\gamma)(\rho''_k, r''_{k+2})$, by definition of

$\cdot$.

⟨9⟩5. $\sigma_{\rho''_k} \xrightarrow{(t, \llbracket \alpha; \gamma \rrbracket)} \sigma_{\rho''_{k+2}}$

⟨9⟩6. $\theta_{\rho''_k}[\llbracket \alpha; \gamma \rrbracket / \alpha; \gamma] \xrightarrow{(t, \llbracket \alpha; \gamma \rrbracket)} \theta_{\rho''_{k+2}}[\llbracket \alpha; \gamma \rrbracket / \alpha; \gamma]$, by Lemma 6.

⟨9⟩7. Q.E.D.

PROVE: $\rho'' \blacktriangleleft \rho'$

⟨8⟩1. By the above construction, $Tid(\rho'') = Tid(\rho')$, and every procedure executes the same procedure in both ρ'' and ρ' .

⟨8⟩2. By the above construction, $\rho''_N = \rho'_N$.

⟨8⟩3. $Beh(\rho'') = Beh(\rho')$ by ⟨8⟩1, ⟨8⟩2, and Lemma 8.

⟨8⟩4. $\ll_{\rho''} = \ll_{\rho'}$, since the above construction does not introduce in ρ' any more interleavings than that of ρ'' between two threads.

⟨8⟩5. Q.E.D.

⟨6⟩3. $\rho'_1 \models \exists X_{i+1}. \mathcal{I}_n$, by $\sigma_{\rho_1} = \sigma_{\rho''_1} = \sigma_{\rho'_1}$, $\rho_1 \models \exists X_i. \mathcal{I}_n$, and ⟨5⟩2. Thus ρ' is a run of P_{i+1} from $\exists X_{i+1}. \mathcal{I}_n$.

⟨6⟩4. $\rho \blacktriangleleft \rho'$, by $\rho \blacktriangleleft \rho''$, $\rho'' \blacktriangleleft \rho'$, and Lemma 7.

⟨6⟩5. Q.E.D.

CASE: $(P_i, \mathcal{I}_i) \vdash \gamma : \mathbb{L}$

⟨6⟩1. This case is proved similar to the former case, using Lemma 10.

⟨6⟩2. Q.E.D.

⟨5⟩4. Q.E.D.

CASE: Rule REDUCE-CHOICE: Replace occurrences of $\alpha \square \gamma$ with $\llbracket \alpha \square \gamma \rrbracket$.

⟨5⟩1. $\mathcal{I}_{n+1} = \mathcal{I}_n$, by proof rule

⟨5⟩2. $X_i = X_{i+1}$, since $Var_{P_i} = Var_{P_{i+1}}$, by proof rule.

⟨5⟩3. Construct ρ' , a run of P_{i+1} from ρ inductively (on the length of the run).

CASE: Base case

⟨7⟩1. Take $\rho'_1 = \rho_1$.

CASE: Inductive case

ASSUME: $\rho' = \rho_1 \xrightarrow{1} \rho_k$ for some $\mathbf{l}$.

ASSUME: $\rho_k \xrightarrow{(t, \beta)} \rho_{k+1}$

PROVE: $(\sigma_{\rho_k}, \theta_{\rho_k}[\llbracket \alpha \square \gamma \rrbracket / \alpha; \gamma]) \xrightarrow{(t, \beta)} (\sigma_{\rho_{k+1}}, \theta_{\rho_{k+1}}[\llbracket \alpha \square \gamma \rrbracket / \alpha \square \gamma])$

⟨8⟩1. If $\beta \neq \alpha$ and $\beta \neq \gamma$, then the claim trivially holds by operational semantics of atomic actions, and Lemma 6.

⟨8⟩2. Let $\beta = \alpha$ (where the transition is due to $\alpha \square \gamma$).

⟨8⟩3. $\llbracket \alpha \square \gamma \rrbracket = \text{assert}(\phi_\alpha \wedge \phi_\gamma); (\tau_\alpha \vee \tau_\gamma)$, by definition of $\llbracket \cdot \rrbracket$.

- ⟨8⟩4. Since P_i is good from $\exists X_i.\mathcal{I}$, $\rho_k \models \phi_\gamma$, and in addition to $\rho_k \models \phi_\alpha$.
 ⟨8⟩5. $\tau_\alpha(\rho_k, r_{k+1}) \Rightarrow (\tau_\alpha \vee \tau_\gamma)(\rho_k, r_{k+1})$.
 ⟨8⟩6. $\sigma_{\rho_k} \xrightarrow{(t, \llbracket \alpha \Box \gamma \rrbracket)} \sigma_{\rho_{k+1}}$
 ⟨8⟩7. $\theta_{\rho_k}[\llbracket \alpha \Box \gamma \rrbracket / \alpha \Box \gamma] \xrightarrow{(t, \llbracket \alpha \Box \gamma \rrbracket)} \theta_{\rho_{k+1}}[\llbracket \alpha \Box \gamma \rrbracket / \alpha \Box \gamma]$, by Lemma 6.
 ⟨8⟩8. Similar construction is done if $\beta = \gamma$ (where the transition is due to $\alpha \Box \gamma$).
 ⟨8⟩9. Q.E.D.

PROVE: $\rho \blacktriangleleft \rho'$

- ⟨7⟩1. By the above construction, $Tid(\rho) = Tid(\rho')$, and every procedure executes the same procedure in both ρ and ρ' .
 ⟨7⟩2. By the above construction, $\rho_N = \rho'_N$.
 ⟨7⟩3. $\text{Beh}(\rho) = \text{Beh}(\rho')$ by ⟨7⟩1, ⟨7⟩2, and Lemma 8.
 ⟨7⟩4. $\ll_\rho = \ll_{\rho'}$, since the above construction does not introduce in ρ' any more interleavings than that of ρ between two threads.
 ⟨7⟩5. Q.E.D.
 ⟨5⟩4. $\rho'_1 \models \exists X_{i+1}.\mathcal{I}_n$, by $\sigma_{\rho_1} = \sigma_{\rho'_1} = \sigma_{\rho'_1}$, $\rho_1 \models \exists X_i.\mathcal{I}_n$, and ⟨5⟩2. Thus ρ' is a run of P_{i+1} from $\exists X_{i+1}.\mathcal{I}_n$.
 ⟨5⟩5. Q.E.D.

CASE: Rule REDUCE-LOOP

- ⟨5⟩1. $\mathcal{I}_{n+1} = \mathcal{I}_n$
 ⟨5⟩2. $X_i = X_{i+1}$, since $\text{Var}_{P_i} = \text{Var}_{P_{i+1}}$.

CASE: $(P_i, \mathcal{I}_i) \vdash \alpha : \mathbb{R}$

- ⟨6⟩1. Construct ρ'' , a run of P_i from ρ in the following way: For each occurrence of the pair α, γ , derive a new run from the current one, by moving every α (due to $\alpha^\odot$) to the right, until it becomes adjacent to the last occurrence of α .

PROVE: $\rho \blacktriangleleft \rho''$

- ⟨7⟩1. By induction on the number of pairs α^1, α^2 , where α^2 is last iteration of the loop, that do not appear adjacent in ρ .
 ⟨7⟩2. In the base case we take $\rho'' = \rho$, so $\rho \blacktriangleleft \rho''$.
 ⟨7⟩3. In the inductive case, we take the run $\rho = \rho_1 \xrightarrow{\mathbf{l}_1} \rho_i \xrightarrow{(t, \alpha^1) \mathbf{l}(t, \alpha^2)} \rho_j \xrightarrow{\mathbf{l}_2} r_n$

- such that $\mathbf{I} = (u_1, \beta_1)(u_2, \beta_2) \cdots (u_m, \beta_m)$ and $t \notin \{u_1, \dots, u_n, u_{n+1}\}$, and construct the run $\rho'' = \rho_1 \xrightarrow{\mathbf{I}_1} \rho_i \xrightarrow{\mathbf{I}(t, \alpha^1)(t, \alpha^2)} \rho_j \xrightarrow{\mathbf{I}_1} r_n$. By Lemma 9, $\rho \blacktriangleleft \rho''$ holds. In addition, the number of pairs α, γ in ρ'' is less than ρ .
- (7)4. Output ρ'' such that for every $\rho''(k) = (t, \alpha)$ for some thread t , either $\rho''(k)$ is the last iteration, or $\rho''(k+1) = (t, \alpha)$. As a result of the above induction, $\rho \blacktriangleleft \rho''$.
- (7)5. Q.E.D.
- (6)2. Construct ρ' , a run of P_{i+1} from ρ'' in the following way: Replace every occurrence of $\rho''_k \xrightarrow{(t, \alpha)^*} \rho''_{k+m}$ with $(\sigma_{\rho''_k}, \theta_{\rho''_k}[\alpha^\circ / \beta]) \xrightarrow{(t, \beta)} (\sigma_{\rho''_{k+m}}, \theta_{\rho''_{k+m}}[\alpha^\circ / \beta])$.
- (7)1. Since τ_β is reflexive, in the case of $m = 0$, we yield the transition $(\sigma_{\rho''_k}, \theta_{\rho''_k}[\alpha^\circ / \beta]) \xrightarrow{(t, \beta)} (\sigma_{\rho''_k}, \theta_{\rho''_k}[\alpha^\circ / \beta])$
- (7)2. In the case of $m > 0$, by $(P_i, \mathcal{I}_i) \vdash \llbracket \beta; \alpha \rrbracket \preceq \beta$, and since P_i is good from $\exists X_i. \mathcal{I}_n, \sigma_{\rho''_k} \xrightarrow{(t, \llbracket \alpha; \gamma \rrbracket)} \sigma_{\rho''_{k+m}}$ holds.
- (7)3. $\theta_{\rho''_k}[\alpha^\circ / \beta] \xrightarrow{(t, \beta)} \theta_{\rho''_{k+m}}[\alpha^\circ / \beta]$, by operational semantics.
- (7)4. Q.E.D.

PROVE: $\rho'' \blacktriangleleft \rho'$

- (7)1. $Tid(\rho'') = Tid(\rho')$.
- (7)2. Since the final states of ρ'' and ρ' are equivalent, $\text{Beh}(\rho'') = \text{Beh}(\rho')$ by Lemma 8.
- (7)3. For every thread $t \in Tid(\rho'')$, replacing consecutive transitions of t with again transitions of t does not effect relative ordering of t with other threads. Thus $\ll_{\rho} = \ll_{\rho'}$
- (7)4. Q.E.D.
- (6)3. $\rho_1 = \rho''_1$, by construction and Lemma 9.
- (6)4. $\rho'_1 \models \exists X_{i+1}. \mathcal{I}_n$, by $\sigma_{\rho'_1} = \sigma_{\rho''_1}$, $\rho_1 \models \exists X_i. \mathcal{I}_n$, and (5)2. Thus ρ' is a run of P_{i+1} from $\exists X_{i+1}. \mathcal{I}_n$.
- (6)5. $\rho \blacktriangleleft \rho'$, by $\rho \blacktriangleleft \rho''$ and $\rho'' \blacktriangleleft \rho'$.
- (6)6. Q.E.D.

CASE: $(P_i, \mathcal{I}_i) \vdash \alpha : \mathbb{L}$

- (6)1. This case is proved similar to the former case, using Lemma 10.
- (6)2. Q.E.D.

⟨5⟩3. Q.E.D.

CASE: Rule ADD-VARIABLE: Add the new variable v to Var_P , and replace every action α with β whenever $(P, \mathcal{I}) \vdash \alpha \preceq_{+v} \beta$, which holds if the following are both valid:

$$\mathbf{A1.} \quad (\mathcal{I} \wedge \neg\phi_\alpha) \Rightarrow (\forall v. \neg\phi_\beta) \quad \mathbf{A2.} \quad (\mathcal{I} \wedge \tau_\alpha) \Rightarrow (\forall v. \neg\phi_\beta \vee (\exists v'. \tau_\beta))$$

⟨5⟩1. $\mathcal{I}_{n+1} = \mathcal{I}_n$, by proof rule.

⟨5⟩2. If v is a local variable, then $X_i = X_{i+1}$; otherwise $X_i = X_{i+1} \cup \{v\}$, since $Var_{P_{i+1}} = Var_{P_i} \cup \{v\}$.

⟨5⟩3. Construct ρ' from ρ inductively (inductive on the number of states in ρ) way.

CASE: Base case

⟨7⟩1. If $v \in Local_{\mathbf{p}}$ for some procedure $\mathbf{p}$, then take $\rho'_1 = (\bar{\sigma}_{\rho_1}, \theta_{\rho_1}[\beta/\alpha])$, where $\bar{\sigma}_{\rho_1} = \sigma_{\rho_1}[t, v \mapsto x]$ and x is some value, for every thread t calling $\mathbf{p}$.

⟨7⟩2. If v is global, then take $\rho'_1 = (\bar{\sigma}_{\rho_1}, \theta_{\rho_1}[\beta/\alpha])$, where $\bar{\sigma}_{\rho_1} = \sigma_{\rho_1}[v \mapsto x]$ such that $\bar{\sigma}_{\rho_1} \models \exists X_{i+1}. \mathcal{I}_n$. There exists x since $\sigma_{\rho_1} \models X_i. \mathcal{I}_n$ and $X_i = X_{i+1} \cup \{v\}$.

CASE: Inductive case

ASSUME:

$\rho' = \rho'_1 \xrightarrow{1} \rho'_k$ so that for all $1 \leq j \leq k$, $\rho'_j = (\bar{\sigma}_{\rho_j}, \theta_{\rho_j}[\beta/\alpha])$, where $\bar{\sigma}_{\rho_j} = \sigma_{\rho_j}[v \mapsto x]$, with x chosen during the construction properly.

ASSUME: Transition $\rho_k \xrightarrow{(t, \alpha)} \rho_{k+1}$ exists.

PROVE: $\rho'_k \xrightarrow{(t, \alpha)} (\sigma_{\rho_{k+1}}, \theta_{\rho_{k+1}}[\beta/\alpha])$

⟨8⟩1. $\bar{\sigma}_{\rho_k} \xrightarrow{(t, \beta)} \bar{\sigma}_{\rho_{k+1}}$, where $\bar{\sigma}_{\rho_{k+1}} = \sigma_{\rho_{k+1}}[v \mapsto x]$ such that $\tau_\beta(\bar{\sigma}_{\rho_k}, \bar{\sigma}_{\rho_{k+1}})$ holds. There exists x since $\tau_\alpha(\sigma_{\rho_k}, \sigma_{\rho_{k+1}})$, and by condition **A2** of $\preceq_{+v}$.

⟨8⟩2. $\theta_{\rho_k}[\beta/\alpha] \xrightarrow{(t, \beta)} \theta_{\rho_{k+1}}[\beta/\alpha]$, by Lemma 6.

PROVE: $\rho \blacktriangleleft \rho'$

⟨6⟩1. $Tid(\rho) = Tid(\rho')$.

⟨6⟩2. Let $V = Var_{P_i} \cap Var_{P_{i+1}}$. The above construction ensures that $\sigma_{\rho_N}|_V = \sigma_{\rho'_N}|_V$, and $\forall \mathbf{p}. v \notin Local_{\mathbf{p}}$, thus $Beh(\rho) = Beh(\rho')$ by Lemma 8.

⟨6⟩3. For every thread $t \in Tid(\rho)$, $fst(\rho, t) = fst(\rho', t)$ and $lst(\rho, t) = lst(\rho', t)$.

Thus $\ll_{\rho} \ll_{\rho'}$.

⟨6⟩4. Q.E.D.

⟨5⟩4. If v is local, then $\rho'_1 \models \exists X_{i+1}.\mathcal{I}_n$ since $\rho_1 \models \exists X_i.\mathcal{I}_n$ and $X_i = X_{i+1}$.

⟨5⟩5. If v is global, $\rho'_1 \models \exists X_{i+1}.\mathcal{I}_n$ is ensured by the above construction, choosing $\sigma_{\rho'_1}(v)$ properly to satisfy $\exists X_i.\mathcal{I}_n$.

⟨5⟩6. Thus ρ' is a run of P_{i+1} from $\exists X_{i+1}.\mathcal{I}_n$.

⟨5⟩7. Q.E.D.

CASE: Rule HIDE-VARIABLE: Remove the existing variable v from the program, and replace the invariant $\mathcal{I}$ with $\exists v.\mathcal{I}$. Replace every action α with β whenever $(P, \mathcal{I}) \vdash \alpha \preceq_{-v} \beta$, which holds if the following are both valid:

$$\mathbf{H1.} \quad (\exists v.\mathcal{I} \wedge \neg\phi_\alpha) \Rightarrow \neg\phi_\beta \quad \mathbf{H2.} \quad (\exists v, v'.\mathcal{I} \wedge \tau_\alpha) \Rightarrow (\neg\phi_\beta \vee \tau_\beta)$$

⟨5⟩1. $\mathcal{I}_{n+1} = \exists v.\mathcal{I}_n$, by proof rule.

⟨5⟩2. $X_i = X_{i+1}$, by proof rule.

⟨5⟩3. Construct ρ' from ρ inductively (inductive on the number of states in ρ) way.

CASE: Base case

⟨7⟩1. Take $\rho'_1 = (\bar{\sigma}_{\rho_1}, \theta_{\rho_1}[\beta/\alpha])$, where $\bar{\sigma}_{\rho_1} = \sigma_{\rho_1}|_{\text{Var}_{P_i}}$. $\rho_1 \models X_i.\mathcal{I}_n$, ⟨5⟩1 and ⟨5⟩2 imply that $\rho'_1 \models X_{i+1}.\mathcal{I}_n$.

CASE: Inductive case

ASSUME:

$\rho' = \rho'_1 \xrightarrow{1} \rho'_k$ so that for all $1 \leq j \leq k$, $\rho'_j = (\bar{\sigma}_{\rho_j}, \theta_{\rho_j}[\beta/\alpha])$, where $\bar{\sigma}_{\rho_j} = \sigma_{\rho_j}|_{\text{Var}_{P_i}}$.

ASSUME: Transition $\rho_k \xrightarrow{(t, \alpha)} \rho_{k+1}$ exists.

PROVE: $\rho'_k \xrightarrow{(t, \alpha)} (\sigma_{\rho_{k+1}}, \theta_{\rho_{k+1}}[\beta/\alpha])$

⟨8⟩1. $\bar{\sigma}_{\rho_k} \xrightarrow{(t, \beta)} \bar{\sigma}_{\rho_{k+1}}$, where $\bar{\sigma}_{\rho_{k+1}} = \sigma_{\rho_{k+1}}|_{\text{Var}_{P_i}}$, since condition **H2** of $\preceq_{-v}$ implies that if $\sigma_{\rho_k} \xrightarrow{(t, \alpha)} \sigma_{\rho_{k+1}}$ holds then $\sigma_{\rho_k}|_{\text{Var}_{P_i}} \xrightarrow{(t, \beta)} \sigma_{\rho_{k+1}}|_{\text{Var}_{P_i}}$ holds.

⟨8⟩2. $\theta_{\rho_k}[\beta/\alpha] \xrightarrow{(t, \beta)} \theta_{\rho_{k+1}}[\beta/\alpha]$, by Lemma 6.

PROVE: $\rho \blacktriangleleft \rho'$

⟨6⟩1. $\text{Tid}(\rho) = \text{Tid}(\rho')$.

⟨6⟩2. Let $V = \text{Var}_{P_i} \cap \text{Var}_{P_{i+1}}$. The above construction ensures that $\sigma_{\rho_N}|_V = \sigma_{\rho'_N}|_V$, and $\forall \mathbf{p}. v \notin \text{Local}_{\mathbf{p}}$, thus $\text{Beh}(\rho) = \text{Beh}(\rho')$ by Lemma 8.

⟨6⟩3. For every thread $t \in \text{Tid}(\rho)$, $\text{fst}(\rho, t) = \text{fst}(\rho', t)$ and $\text{lst}(\rho, t) = \text{lst}(\rho', t)$.

Thus $\ll_{\rho} = \ll_{\rho'}$.

$\langle 6 \rangle 4$. Q.E.D.

$\langle 5 \rangle 4$. Since $\rho_1 \models \exists X_i. \mathcal{I}_n$ and $X_i = X_{i+1}$, the above construction ensures that

$\rho'_1 \models \exists X_{i+1}. \mathcal{I}_n$ holds. Thus ρ' is a run of P_{i+1} from $\exists X_{i+1}. \mathcal{I}_n$.

$\langle 5 \rangle 5$. Q.E.D.

$\langle 1 \rangle 2$. Q.E.D.

□

16.2.3 Connecting Behavioral Simulation to Linearizability

Up to this point, we defined behavioral simulation and showed that every program in a QED proof behaviorally simulates the initial program. The following theorem connects behavioral simulation to the generic notion of linearizability, and thus, allows us to reason that the linearizability of the initial program is still preserved after a sequence of QED transformations.

Theorem 10 (Equivalence of linearizability and behavioral simulation):

Let P' be an atomic program that is good from $\mathcal{I}$. A program P is linearizable to P' from $\mathcal{I}$ iff $\mathcal{I} \vdash P \preceq P'$. ■

PROOF.

CASE: $\rightarrow$

ASSUME: P is linearizable to P' from $\mathcal{I}$

PROVE: $P \preceq_{\mathcal{I}} P'$

$\langle 3 \rangle 1$. Take a run ρ of P from $\mathcal{I}$. Consider its history $\mathbf{H}_{\rho}$. By assumption, there exists a sequential history $\mathbf{H}'$ of P' from $\mathcal{I}$ such that $\mathbf{H}$ is linearizable to $\mathbf{H}'$. Then there exists a run r' of P' from which $\mathbf{H}'$ is obtained from, so that $Tid(\mathbf{H}) = Tid(\mathbf{H}') = Tid(\rho) = Tid(\rho')$. In addition, $\text{Beh}(\rho') = \text{Beh}(\rho)$ since the input and output values from $\mathbf{H}$ and $\mathbf{H}'$ are equivalent. In addition, $\preceq_{\mathbf{H}} \subseteq \preceq_{\mathbf{H}'}$ implies $\ll_{\rho} \subseteq \ll_{\rho'}$. Thus for every ρ , we can find a run r' that will allow us to show that $P \preceq_{\mathcal{I}} P'$.

CASE: $\leftarrow$

PROVE: $P \preceq_{\mathcal{I}} P'$

ASSUME: P is linearizable to P' from $\mathcal{I}$

⟨3⟩1. Let $\mathbf{H}$ be history of program P . We have a run ρ of P from which we obtain $\mathbf{H}$. Since $P \leq_{\mathcal{I}} P'$, there exists a run ρ' of P' such that $\text{Beh}(\rho) = \text{Beh}(\rho')$ and $\ll_{\rho} \subseteq \ll_{\rho'}$. We first construct sequential history $\mathbf{H}'$ of ρ' of atomic program P' as described in the definition of a history by means of invocation and return of procedures in ρ' . Since $\text{Beh}(\rho) = \text{Beh}(\rho')$ we know that $\text{Tid}(\rho) = \text{Tid}(\rho')$ and hence $\text{Tid}(\mathbf{H}) = \text{Tid}(\mathbf{H}')$ and $\forall t \in \text{Tid}(\rho)$, $\text{beh}(\rho, t) = \text{beh}(\rho', t)$. Thus, $\forall t \in \text{Tid}(\mathbf{H})$ $\mathbf{H}|t = \mathbf{H}'|t$. That is, $\mathbf{H} \cong \mathbf{H}'$. We also have $\ll_{\rho} \subseteq \ll_{\rho'}$ which implies $\preceq_{\mathbf{H}} \subseteq \preceq_{\mathbf{H}'}$ since we do not change order of actions when deriving the history of a run. Hence, $\mathbf{H}$ is linearizable to $\mathbf{H}'$, and P is linearizable to P' .

□

Theorems 9 and 10 provide two options for proving linearizability of Impl to the intended specification, represented by an atomic program Spec , from $\mathcal{I}$. First, one can complement another proof method with ours, by first performing the proof $(\text{Impl}, \text{true}) \dashrightarrow \dots \dashrightarrow (P_k, \mathcal{I})$, and then applying her method to prove that P_k is linearizable to Spec . Once the proof passes, this implies that Impl is also linearizable to Spec , since our transformations preserve all the behaviors of the program relevant to linearizability. Alternatively, s/he can keep transforming $(P_k, \mathcal{I})$ up to $(\text{Spec}, \mathcal{I})$, and complete the full proof of linearizability in our system. Note that, for the theorems to ensure soundness in these cases, s/he must also prove that P_k (resp. Spec) is good from $\mathcal{I}$. The latter is formalized by the following.

Corollary 11 (Soundness of proving linearizability using QED):

Let $(\text{Impl}, \mathcal{I}_1) \dashrightarrow \dots \dashrightarrow (\text{Spec}, \mathcal{I}_n)$ be a sequence of proof steps and $\mathcal{I}$ be a state predicate such that for all $i \in [1, n]$, $\mathcal{I} \Rightarrow \mathcal{I}_i$. Let $X_1 = \text{fv}(\mathcal{I}) \setminus \text{Global}_{\text{Impl}}$, and $X_n = \text{fv}(\mathcal{I}) \setminus \text{Global}_{\text{Spec}}$. If Spec is good from $\exists X_n. \mathcal{I}$, then Impl is linearizable to Spec from $\exists X_1. \mathcal{I}$. ■

16.3 Related Work in Proving Linearizability

Refinement between a concurrent program and its sequential specification is well-studied [62, 106, 107, 108]. Previous work showed that, under certain conditions, auxiliary variables enable construction of an abstraction map to prove refinement [62, 106]. However, in practice writing an abstraction map for programs with fine-grained concurrency remains a challenge since there are a large number of cases to consider. [103] used a complex abstraction map, called *aggregation function*, that completes the atomic transactions that are committed but

not yet finished. The refinement proofs in [36, 109, 110, 111], despite being supported by automated proof checkers, all require manual guidance for the derivation of the proof, requiring the user to manage low-level logical reasoning. On the other hand, in our method the user guides the proof via code transformations at the programming language level. Recently, [104] provided a tool that automates the derivation of the proof using shape abstraction. To our knowledge, its automating ability is limited to linked-list based data structures and it still requires identification of the possible commit points.

Owicki-Gries [42, 36] and rely-guarantee [87] methods have been used in refinement proofs. However, in the case of fine-grained concurrency, deriving the proof obligations in both approaches requires expertise. The idea of local reasoning is exploited by separation logic [112] which is not particularly useful for shared objects with high level of interference. In these cases, we show that abstraction is an important tool to reduce the effects of interference.

Wang and Stoller [51] statically prove linearizability of the program using its sequentially executed version as the specification. Their notion of atomicity is defined over a fixed set of primitives, which is limited in the case of superficial conflicts. On the other hand, our notion of atomicity is more general and supported by abstraction to prove atomicity even under high level of interference. They provided hand-crafted proofs for several non-blocking algorithms, and our proofs are mechanically checked. In [113], Groves gives a hand-proof of the linearizability of the non-blocking queue, by reducing executions the fine-grained program to its sequential version. His use of reduction is non-incremental, and must consider the commutativity of each action by doing a global reasoning, while our reasoning is local.

[107] provide a verification method where they prove refinement between two systems by introducing *proof by transduction*. In their method, a *transducer* maps the concrete system to the abstract system, by introducing delays where necessary. This delay is used to eliminate the use of prophecy variables or backward simulation. [108] suggest using network invariants to abstract parameterized systems that will simplify the verification of correctness LTL formulas, such as liveness.

In this chapter we showed how the key techniques in QED significantly simplify the reasoning about linearizability: While reduction and abstraction increase the granularity of atomic actions, variable introduction and hiding syntactically relate the representation of the im-

plementation to that of the specification. In the resulting approach the abstraction map is constructed incrementally, and the need to reason about the location of commit points is eliminated. In Chapter 19 we will explain several linearizability proofs performed using QED and demonstrate how we eliminate superficial conflicts in these proofs.

Chapter 17

THE IMPLEMENTATION AND PROOF COMMANDS

The previous chapters introduced the first two levels of our contributions. The first level consists of the ACTIONPL language (Chapter 12) and the QED proof rules over ACTIONPL programs (Chapter 13). The second level consists of the QEDPL language (Chapter 14) and proof tactics over QEDPL programs (Chapter 15). In this chapter, we start presenting the third level, which consists of our implementation of the first two levels in our interactive tool called QED-VERIFIER. While proof rules and tactics are low-level building blocks of a proof, in this chapter, we will present *proof commands*, intuitive-to-use, concise instructions to order QED-VERIFIER to perform one or multiple proof tactics for a specific purpose, for example, to apply reduction tactics iteratively in a given procedure. In this view, proof commands are high-level building blocks of proofs in QED-VERIFIER. Each command preserves the soundness of assertion checking by construction; the application of the command is justified by the soundness of the proof tactics it triggers.

Outline. Section 17.1 presents our implementation QED-VERIFIER and explains its features. Section 17.2 gives the proof strategy we followed while proving our benchmarks with QED-VERIFIER. Section 17.3 introduces the proof commands provided by our tool.

17.1 Our Software Tool: QED-VERIFIER

We provide an implementation of the QED proof system as an interactive tool called QED-VERIFIER. QED-VERIFIER accepts as input a multithreaded program written in the QEDPL language and a proof script. The *proof script* contains a sequence of proof commands. The tool runs the proof script by applying these commands in the given order. Each proof command triggers the application of one or multiple proof tactics presented in Chapter 15 to transform the current program. The program after the application of each command is available for the user to examine; at this point, QED-VERIFIER allows the user

to intercept the proof by giving a new command.

Recall that each tactic in Chapter 15 specifies a set of conditions that should be satisfied by the current proof state, and if so, describes a set of transformations that are performed on the proof state. For each proof command, the tool automatically proves the conditions of the proof tactics triggered. For this, QED-VERIFIER translates atomic statements to gated actions (using the $\llbracket \cdot \rrbracket$ operator), using these gated actions, generates the verification conditions (VC) necessary to prove these conditions, and checks the validity of the VCs using Z3, a state-of-the-art SMT solver from Microsoft Research [53]. A command is rejected if the checking of a necessary condition fails. If all the conditions are proved, QED-VERIFIER automatically applied the program transformations required by the tactics. In Table 17.1, we list the checks that are performed automatically using Z3.

Notation	Description	Definition
$\alpha \sqsubseteq \mathcal{I}$	Action preserving an invariant	Equation 13.1
$\mathcal{I} \vdash \alpha \Rightarrow \beta$	Preserving the gate of another action	Definition 14
$\mathcal{I} \vdash \alpha \preceq \beta$	Abstraction between actions	Definition 18
$\mathcal{I} \vdash \alpha \preceq_{+v} \beta$	Abstraction under variable addition	Definition 7
$\mathcal{I} \vdash \alpha \preceq_{-v} \beta$	Abstraction under variable hiding	Definition 8
$\mathcal{I} \vdash \mathcal{MC}(\alpha, \beta)$	Mover check	Definition 13
$(P, \mathcal{I}, \mathcal{A}) \vdash \{\alpha\} \mathbf{p}$	Procedure conforms to its sequential specification	Equation 13.8
$\text{Reflexive}(\alpha)$	Reflexive actions	Definition 16
$\text{Non-blocking}(\alpha)$	Non-blocking actions	Definition 15

Table 17.1: Checks performed in QED-VERIFIER using the Z3 SMT solver

The input language of QED-VERIFIER is the QEDPL language presented in Chapter 14. The syntax of QEDPL is based on that of Boogie 2 [52], an intermediate language developed at Microsoft Research for verifying sequential programs written in C and Spec#. QEDPL and Boogie 2 share most of the sequential constructs: the syntax of expressions, the top-level elements given in Figure 12.2, primitive commands (assignment, `assume`, `assert`, `havoc`), sequential composition (`;`), conditional and nondeterministic `if` and `while` statements, and maps. Boogie 2 is used in static verification to simplify the complexity of generating the verification conditions (VC) required to verify a Spec# program. While Boogie 2 lacks the

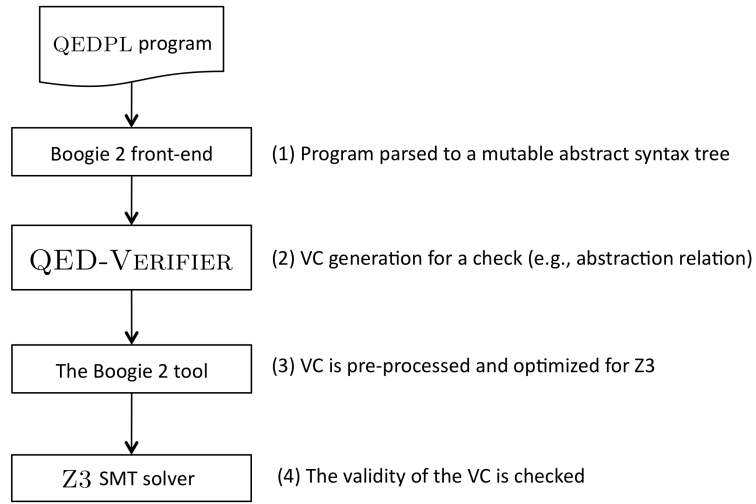


Figure 17.1: The interaction between QED-VERIFIER, the Boogie tool, and Z3.

complex, object-oriented features of *Spec#*, it provides programming constructs (constants, function symbols, axioms, non-deterministic control-flow) that are closer to the logics used by SMT solvers, allowing an effective and customizable VC generation framework. In this framework, the VCs are generated in two phases: First, the *Spec#* program is translated into a Boogie 2 program; the complex features of *Spec#* are encoded in terms of the low-level constructs (constants, functions, axioms, etc.) of Boogie 2. Second, the VCs are generated from the Boogie 2 program and checked using an SMT solver. The failing VCs are propagated back as a trace of the *Spec#* program for debugging. The tool that parses and checks Boogie 2 programs is also called *Boogie* [41].

Figure 17.1 shows the interaction between QED-VERIFIER, the Boogie tool and Z3. QED-VERIFIER uses a modified version of the Boogie 2 front-end (part of the Boogie tool) to parse, type check, and manipulate QEDPL programs ((1) in Figure 17.1). We modified the front-end to parse and type check atomic statements (`atomic{·}`), exception throwing (`throw`) and handling (`try-catch`), parallel statements (`dopar`), and records. Other features of QEDPL are encoded in terms of the core QEDPL syntax. For example, we defined the exception labels by declaring the type `Exception` and unique constants of type `Exception`.

While generating the verification conditions required for the checks shown in Table 17.1, QED-VERIFIER interacts with the Boogie tool for the tasks common to VC generation for

both QEDPL and Boogie 2. For example, we represent QEDPL programs in the block-based form of Boogie programs and apply several optimizations (e.g., unifying return points of each procedure) on this form, which significantly simplify the VC generation process and allow us to obtain compact VCs. We refer the reader to [41] and [114] for an elaborate explanation of these techniques. QED-VERIFIER passes the generated VC back to the Boogie tool for further pre-processing and optimizations required for the SMT solver ((3) in Figure 17.1). For example, variables are replaced by untyped variables, and the typing information is encoded using functions and axioms (see [115] for details) in order to make the VCs more precise.

While not discussed in Chapter 14, QED-VERIFIER supports the use of the Boogie 2 type system [115] in QEDPL programs. The Boogie 2 type system allows a rich set of features including complex type definitions such as parameterized type synonyms and polymorphic types. The type system is orthogonal to our technique and do not affect the soundness of our approach, since the typing information is processed after QED-VERIFIER passes the VCs back to the Boogie tool (step (3) in Figure 17.1).

When loading a QEDPL program, we pre-process the program as follows:

1. For ease of writing programs, we allow, in the input program, guards of conditional statements in QEDPL (if and while) to refer to global variables. However, while loading the program into QED-VERIFIER, we rewrite such statements as follows:

- For every $\text{if}(B)\{\hat{S}_1\}\text{else}\{\hat{S}_2\}$ where B refers to global variables, we create a local, boolean variable ℓ and add it to $\text{Private}_{\mathbf{p}}$ of the containing procedure $\mathbf{p}$. Then we replace $\text{if}(B)\{\hat{S}_1\}\text{else}\{\hat{S}_2\}$ with

$$\ell := B; \text{ if}(\ell)\{\hat{S}_1\}\text{else}\{\hat{S}_2\}$$

- For every $\text{while}(B)\{\hat{S}\}$ where B refers to global variables, we create a local, boolean variable ℓ and add it to $\text{Private}_{\mathbf{p}}$ of the containing procedure $\mathbf{p}$. Then we replace $\text{while}(B)\{\hat{S}\}$ with

$$\ell := B; \text{ while}(\ell)\{\hat{S}\}$$

2. For every QEDPL command C that accesses multiple global variables but is not surrounded by $\text{atomic}\{\cdot\}$, we split C into a sequence of QEDPL commands so that

each resulting command accesses at most one global variable. Let $V = \{x_1, \dots, x_n\}$ be the set of global variables accessed by C . For each variable $x_i \in V$, we create a local variable ℓ_i and add it to $Private_{\mathbf{p}}$ of the containing procedure $\mathbf{p}$. Then we replace C with the statement

`dopar{ $\ell_1 := x_1$ }with{ $\dots$ dopar{ $\ell_{n-1} := x_{n-1}$ }with{ $\ell_{n-1} := x_{n-1}$ }};  $C[\ell_1/x_1, \dots, \ell_n/x_n]$`

In the statement above, the global variables are read inside a parallel statement, allowing an arbitrary evaluation order for expressions.

3. We replace each QEDPL command C that is not surrounded by `atomic{·}` by `atomic{C}`.

We divide the procedures in every QEDPL program into two disjoint sets: *Public* or *Private*. We omit the procedures in *Private* from $\text{Sig}_{\hat{P}}$, the signature of the program (see Section 12.2.6). In order to reduce the complexity of the mover checks, when a procedure $\mathbf{p} \in Private$ is inlined at *all* of its call points, we also remove $\mathbf{p}$ from $Proc_{\hat{P}}$; this does not affect $\text{Sig}_{\hat{P}}$, and thus, is sound.

Public and private procedures are declared using the *attribute* construct provided by the Boogie 2 parser. We use the attribute `{:ispublic}` as follows:

```
procedure {:ispublic true} APublicProcedure(...) { ... }
procedure {:ispublic false} APrivateProcedure(...) { ... }
```

We allow the user to declare a procedure as atomic using the `{:isatomic}` attribute as follows:

```
procedure {:isatomic true} AnAtomicProcedure(...) { ... }
```

While loading a declared-atomic procedure, our tool automatically surrounds its body with `atomic{·}`, if not done already.

17.2 Proof Strategy

We view a proof of a concurrent program as a sequence of steps, each applying a proof command presented in this section. We have found the strategy sketched in Figure 17.2 to be a good start. Our proof strategy informally defines a general proof script template. The proof scripts of our benchmarks given in Chapter 19 follow this template.

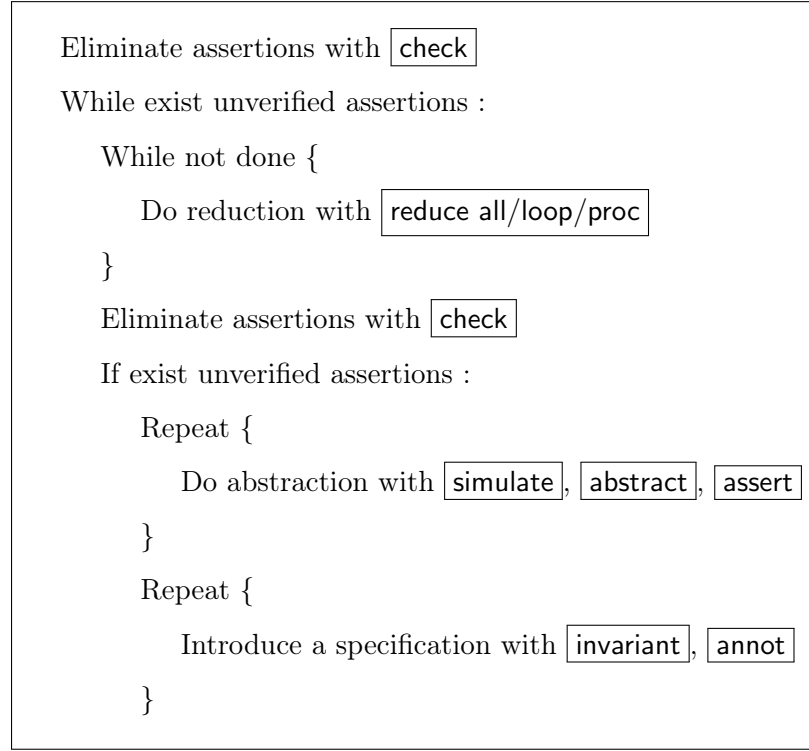


Figure 17.2: The proof strategy for verifying a program using proof commands

In each iteration we first apply reduction and then verify assertions within the atomic statements of the program. When there are still unverified assertions, this means that the atomicity level of the program is not sufficient to discharge the assertions sequentially, so we need more reduction. In this case, we do abstractions, for example, by a read or write abstraction or adding assertions to actions, and introduce specifications; this allows reduction to obtain coarser atomic actions at the next iteration.

17.3 Proof Commands

In this section we will present the proof commands provided by our tool QED-VERIFIER. Table 17.2 summarizes all the commands provided by QED-VERIFIER, and thus, defines the language for writing proof scripts. These commands restrict the users attention to applying a restricted set of tactics in a predetermined way, though, we found them general and adequate to prove all the benchmarks presented in Chapter 19. Nevertheless, QED-VERIFIER provides an extensible platform to write new proof commands. If one needs to

apply a new tactic or apply an existing tactic in a different way, s/he can easily add a new proof command to the tool by extending the `ProofCommand` class in the implementation.

Each proof command consists of one or more keywords followed by a sequence of parameters separated by commas, e.g., `abstract read  $\alpha, \bar{S}$` . In order to refer to the statements of a program in the parameters, we assign each atomic statement a unique label. Every non-atomic statement is also referred to by using the label of an atomic statement relative to the former; for example, we refer to a `while( $g$ ){ $\hat{S}$ }` statement through the label of the first atomic statement in $\hat{S}$. A parameter that must refer to a statement is given the label of that statement. While denoting the syntax of proof commands (for example, in Table 17.2), we overload the following notations:

- α or β refer to the label of an atomic statement.
- S refers to the label of a statement.
- `loop{ $S$ }` and `while( $g$ ){ $S$ }` refer to the label of a loop.
- $\mathbf{p}$ refers to the name of a procedure.
- v refers to the name of a (global or local) variable.
- B refers to a string specifying a boolean expression in the syntax given in Section 12.1.
- $\bar{S}$ refers to a code block specifying a straight statement (without loops and procedure calls) in the syntax given in sections 14.1 and Section 14.3.
- m refers to a mover type from the set $\{\mathbf{R}, \mathbf{L}, \mathbf{B}, \mathbf{A}\}$

If a proof command has arguments of incorrect types, QED-VERIFIER rejects the application of that command and stops evaluating the current proof script.

Proof command	Usage and related tactic
<code>invariant B</code>	Add a new invariant to the proof state. Tactic 1

– Continued on the next page –

Proof command	Usage and related tactic
<code>annot proc, $\mathbf{p}$, $\bar{S}$</code>	Introduce a specification <code>atomic{$\bar{S}$}</code> for the procedure <code>$\mathbf{p}$</code> .
<code>annot mover $\mathbf{p}$, m</code>	Specify that the body of procedure <code>$\mathbf{p}$</code> is of mover type m .
<code>annot loop while(g){S}, $\bar{S}$</code>	Introduce a specification <code>atomic{$\bar{S}$}</code> for the loop <code>while(g){S}</code> .
<code>mover all</code>	Compute the mover types of all atomic statements and store them in Δ . ¹
<code>mover proc $\mathbf{p}$</code>	Compute the mover types of the atomic statements in <code>$\mathbf{p}$</code> and store them in Δ .
<code>mover α</code>	Compute the mover types of the atomic statement α and store them in Δ .
<code>merge all</code>	Apply reduction tactics to all procedure bodies. Tactics 13, 14, 16, and 21
<code>merge proc $\mathbf{p}$</code>	Apply reduction tactics to the body of the procedure <code>$\mathbf{p}$</code> .
<code>merge stmt S</code>	Apply reduction tactics to a particular statement.
<code>reduce all</code>	Apply reduction iteratively to all the procedure bodies. Tactics 13, 14, 16, 21
<code>reduce proc $\mathbf{p}$</code>	Apply reduction iteratively to the body of the procedure <code>$\mathbf{p}$</code> . Tactics 13, 14, 16, 21
<code>reduce loop while(g){S}, $\bar{S}$</code>	Reduce the loop to the given specification <code>atomic{$\bar{S}$}</code> . Tactics 24, 25, and 26.
<code>peelout while(g){S}</code>	Peel out the final iteration of the loop. Tactics 29, 30, and 31.
<code>reduce recursive $\mathbf{p}_1, \dots, \mathbf{p}_n$</code>	Reduce the procedures to their existing specifications and do inlining. Tactics 28 and 27
<code>inline $\mathbf{p}$</code>	Inline the body of the procedure <code>$\mathbf{p}$</code> at all call sites. Tactic 27
<code>inline all</code>	Inline all procedures with atomic bodies at all call sites. Tactic 27
<code>simulate α, $\bar{S}$</code>	Abstract the atomic statement α with the new body <code>atomic{$\bar{S}$}</code> . Tactic 2
<code>abstract read v, α</code>	Abstract the value of v at the entry of the atomic statement α . Tactic 4 and 5
<code>abstract write v, α</code>	Abstract the value of v at the exit of the atomic statement α . Tactic 6
<code>assert α, B</code>	Add new assertion B to the entry of the atomic statement α . Tactic 3

– Continued on the next page –

¹The use of *Delta* is explained in Section 17.3.4.

Proof command	Usage and related tactic
<code>check all</code>	Validate assertions in all the atomic statements using sequential analysis. Tactics 7 and 8
<code>check proc p</code>	Validate assertions in the body of procedure p using sequential analysis. Tactics 7 and 8

Table 17.2: The proof commands provided by QED-VERIFIER

17.3.1 Introducing Invariants

The command `invariant B` introduces a new invariant B into the proof state by applying Tactic 1. The current invariant $\mathcal{I}$ is replaced with the conjunction $\mathcal{I} \wedge B$. The command fails to change the invariant if any atomic statement in the program does not preserve $\mathcal{I} \wedge B$. Upon the failure, QED-VERIFIER reports the atomic statement that does not preserve the invariant along with the minimum subset of formulas $\{\phi_1, \dots, \phi_n\}$ that is not preserved, given that B is in the form of $\phi_1 \wedge \dots \wedge \phi_n$.

17.3.2 Abstraction

Abstraction is applied by adding either extra transitions or extra assertions to a gated action. We use the commands `simulate`, `abstract`, and `assert` for abstracting atomic statements.

The `simulate` command provides the most general way of abstracting an atomic statement. The command `simulate  $\alpha, \bar{S}$` replaces the atomic statement α with the new statement `atomic $\{\bar{S}\}$` after proving that the pre-conditions of Tactic 2 hold. This allows the user to provide arbitrary code in $\bar{S}$ to replace α as long as `atomic $\{\bar{S}\}$` abstracts α .

The read and write abstractions described in Section 15.2.2 are applied by the `abstract read` and `abstract write` commands. The command `abstract read  $v, \alpha$` applies tactics 4 and 5 and makes the atomic statement α read a nondeterministic value from the variable v at the beginning of the statement, thus making the operation of the action independent of the initial value of v when its execution starts. The command `abstract write  $v, \alpha$` applies Tactic 6 and makes the atomic statement α write a nondeterministic value to the variable v at the end of the statement.

We use the command `assert  $B, \alpha$` to add the assertion B to the beginning of the atomic statement α , using Tactic 3. When one needs to add the assertion inside α (i.e., via a context $\mathbf{C}$ as in Tactic 3), s/he must use the `simulate` command and give the new body of α with the assertion inserted.

17.3.3 Variable Addition and Hiding

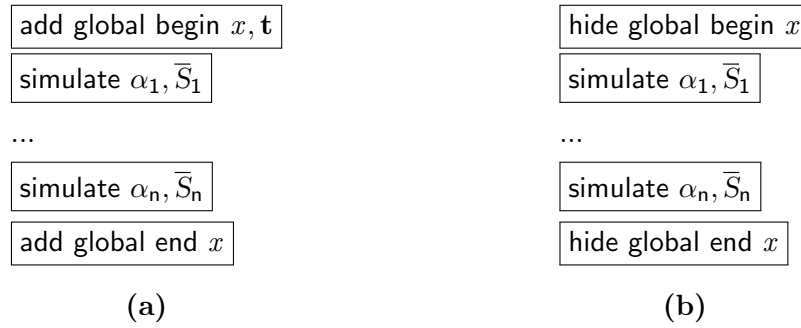


Figure 17.3: Commands for adding (a) and hiding (b) a global variable

We add or hide a variable through a sequence of proof commands. Figure 17.3 shows common patterns for adding and hiding a global variable x . The first and last commands mark the beginning and end of introducing v . In addition to the name of the new variable, we also provide the type of the variable as an argument to `add global begin`. Each command `simulate  $\alpha_i, \bar{S}_i$` between `add global begin` and `add global end` checks the relation $\preceq_{+x}$ (as specified by Tactic 9) between the atomic statements α_i and the newly provided statement `atomic{ $\bar{S}_i$ }`. If the abstraction relation holds, then α_i is replaced by `atomic{ $\bar{S}_i$ }`. A local variable is added similarly except that (1) the commands used at the beginning and the end of the command sequence are named `add local begin` and `add local end`, and (2) `add local begin` takes a procedure name as argument.

Hiding a global (resp. local) variable is similar, except that (1) the command sequence starts with `hide global begin` (resp. `hide local begin`), and ends with `hide global end` (resp. `hide local end`), and (2) the `simulate` commands check the relation $\preceq_{-x}$ as specified by Tactic 11 (resp. Tactic 12). If a global variable is hidden, the command `hide global end` also transforms the invariant $\mathcal{I}$ to $\exists x. \mathcal{I}$.

17.3.4 Reduction

The command **reduce all** is used to obtain coarser atomic statements in the program by iteratively computing mover types of atomic statements and applying reduction rules for sequential composition, as well as **if**, **try-catch**, and **dopar** statements. To enable iterative application of the reduction tactics, QED-VERIFIER keeps in the proof state a map $\Delta : \text{Atomic} \rightarrow \{\mathbb{R}, \mathbb{L}, \mathbb{B}, \mathbb{A}\}$. The command **reduce all** uses two proof commands **mover** and **merge**. The command performs the following steps iteratively and finishes when no more reduction can be done.

1. Compute the mover type for each atomic statement in the program using the rules in Figure 13.2. For this step we provide the command **mover**. `mover  $\alpha$` computes the mover type of the atomic statement α and stores the result in $\Delta(\alpha)$. This step calls `mover all`, which triggers `mover  $\alpha$` for every α in the program and keeps the results of the mover checks in Δ .
2. Run the command **merge all**, which performs a sequence of reduction operations governed by the tactics 13, 14, 16, 21 using the mover types in Δ . Command **merge all** applies the tactics iteratively as follows:

```
Repeat {
  While tactic is applicable { Apply Tactic 13 }
  While tactic is applicable { Apply Tactic 14 }
  While tactic is applicable { Apply Tactic 16 }
  While tactic is applicable { Apply Tactic 21 }
}
```

Go to step 3 when none of the tactics above is applicable on the current program.

- (a) While applying the reduction tactics, for each atomic statement α , $(P, \mathcal{I}) \vdash \alpha : m$ is decided if $\Delta(\alpha) = m$.

- (b) When a new atomic statement β is obtained as a result of a reduction tactic, the mover type of β is computed using the rule COMBINE-MOVERS in Figure 13.2, and then $\Delta(\beta)$ is initialized accordingly. Note that, $\Delta(\beta)$ is at least $\mathbb{A}$ (according to $\sqsubseteq$).
3. At this point, we decide whether to continue the iteration from step 1 or terminate the operation of `reduce all`.
- (a) If at least one tactic has been successfully applied in 2, go back to step 1.
 - (b) If no tactic was applicable during step 2, we end the execution of the proof command.

The rule COMBINE-MOVERS in Figure 13.2 provides a conservative way of inferring the mover types of atomic statements produced at step 2. Therefore, applying at least one reduction tactic in step 2 causes the iteration to go back to step 1, in which `mover all` possibly recomputes mover types of the new atomic statements more precisely than COMBINE-MOVERS.

QED-VERIFIER provides various versions of the commands `reduce`, `mover`, and `merge` to apply mover check and reduction tactics on a particular atomic statement, a compound statement, or procedure body; they are listed in Table 17.2.

Reducing Loops

The command `reduce all` does not apply tactics for reducing loops. Proof commands for transforming loops to loop-free statements require the user to provide an extra atomic statement that specifies the non-final iterations of the loop.

We use two commands, `peelout` and `reduce loop` to reduce a *while* loop to an atomic statement. The command `peelout while( $g$ ){ $S_1$ }` applies Tactic 30 and 31 depending on whether g is $*$ or B . The command separates the final iteration of the loop from the non-final iterations. The non-final iterations are encoded as a loop of the form `while( $*$ ){ $S_2$ }`, which is eliminated using the command `reduce loop while( $*$ ){ $S_2$ },  $\bar{S}$` . This command applies Tactic 24 and uses `atomic{ $\bar{S}$ }` as the specification of the loop. We also provide the command `reduce loop2 while( $*$ ){ $S_2$ }`, which applies Tactic 25 and uses the atomic statement before

the loop as its specification. If `reduce loop` and `reduce loop2` need to reason about the mover type of the loop body, they first query the pre-computed mover types in Δ , and if Δ does not give the expected mover type, they trigger a `mover` command to perform the precise mover check.

Reducing Procedures

At any time during a proof, a procedure can be inlined at call points. For this, we provide the `inline p` and `inline all` commands. The former replaces all the calls to `p` with its body after doing proper substitutions for formals. The latter performs the inlining for all procedures whose bodies are single atomic statements.

We recommend following the strategy sketched below for a procedure `p`:

1. Reduce Body_p to an atomic statement α by applying a sequence of abstraction and reduction commands.
2. Apply the command `simulate  $\alpha \bar{S}$` to prove that the reduced body α implements a given specification `atomic{ $\bar{S}$ }` and to replace α with `atomic{ $\bar{S}$ }`.
3. Use the command `inline p` to replace calls to `p` with its new body `atomic{ $\bar{S}$ }` given in step 2. If `p` is in the set *Private*, `p` is also removed from the program.

When there is recursion, step 1 of the strategy above will not work due to the recursive procedure calls. In this case, the tactic `reduce recursive  $p_1, \dots, p_n$` is used to reduce the bodies of procedures $p_1, \dots, p_n$ to single atomic actions. The command `reduce recursive` performs the checks specified in Tactic 28 where $\mathcal{M} = \{p_1, \dots, p_n\}$. The annotation function $\mathcal{A}$ required by Tactic 28 is entered by the user before the application of `reduce recursive` using the commands described below. The command `reduce recursive` fails if a procedure or loop does not implement its given specification, or the body of a procedure is not of the given mover type. In this case, the command indicates the failing procedures. If all the checks pass, the command replaces all the calls to $p_1, \dots, p_n$ with their corresponding specifications given in $\mathcal{A}$.

The specifications for loops and procedures required by `reduce recursive` are introduced beforehand using the commands `annot loop`, `annot proc`, and `annot mover` in order to con-

struct the annotation function $\mathcal{A}$ and to reduce procedures as described in Section 13.6. The specifications can be introduced partially while the proof progresses. Newly introduced specifications are conjoined with the existing specifications. The command `annot loop loop{ $S$ },  $\bar{S}$` adds a new specification `atomic{ $\bar{S}$ }` to a loop with the given label. The command `annot proc  $\mathbf{p}$ ,  $\bar{S}$` adds a new specification `atomic{ $\bar{S}$ }` to the procedure $\mathbf{p}$. The command `annot mover  $\mathbf{p}$ ,  $m$` specifies that the body of the procedure $\mathbf{p}$ is of mover type m .

In this chapter, we presented our implementation of the QED proof system as the QED-VERIFIER tool. We described the formats and operations of our proof commands for different purposes and introduced common patterns of applying these commands during the proof. In the next chapter, we will present even higher-level proof commands. These commands take hints from the user about a synchronization idiom used in the program and automatically annotate (via assertions and assignments) atomic statements with auxiliary variables. Each annotation indicates a fact about the synchronization state at which the containing atomic statement is executed. We will also present proof templates making use of these commands to automatically identify the code blocks whose atomicity is ensured by a common synchronization idiom.

Chapter 18

PROOF TEMPLATES FOR SYNCHRONIZATION IDIOMS

This chapter introduces two key concepts, proof templates and synchronization idioms, for improving the usability of the QED proof system. A QED proof consists of scenarios in which the user indicates a fact about a synchronization discipline in the program, for example, that a variable is always protected by a lock. Such scenarios may require the application of multiple, low-level proof rules that are difficult for programmers to use. Based on the observation that the proof rules applied for every instance of a scenario follow a common pattern, we have developed proof templates to document these patterns and mechanize their application. A *proof template* defines (possibly parameterized forms of) the low-level rules required for a proof scenario. The user completes the template by providing simple and application-specific hints and runs the completed script in the QED tool automatically.

The issue of proving non-interference between certain program actions is central to our verification method, which checks atomicity of code blocks. A programmer who wants to show that a certain code block is atomic must (1) make use of synchronization mechanisms that ensure the desired non-interference in his program, and (2) provide the program annotations needed to convince the atomicity checker that the desired non-interference has indeed been accomplished. The first task is aided by the use of well-known synchronization idioms. In this work, we formally model the functionality of common synchronization idioms, including mutex and reader/writer locks and events. The second task is often more difficult and error prone. One typically needs to introduce auxiliary variables, program invariants, and assertions in order to reason about programs using these idioms. It is easily possible to write annotations that are too strong or too weak, and, when the check fails, it is difficult to diagnose whether the program contains synchronization bugs or whether the annotations are not satisfactory. We address this issue by automating the generation of program annotations required for these scripts to succeed.

We present an annotation assistant implemented inside QED-VERIFIER that, given sim-

ple hints about the use of a synchronization idiom, generates annotations for identifying atomic blocks whose atomicity is ensured by the synchronization idiom and reduce these blocks to atomic statements. The annotation generation technique makes use of proof templates to automate the process. There are two key goals of the study in this chapter. First, we aim to reduce and simplify user interaction with QED-VERIFIER during the proof as much as possible by, for example, constructing a proof only in terms of proof templates. Second, we aim to help the programmer to diagnose incorrect or insufficient uses of a synchronization idiom in a program, by examining the generated annotations and the atomic blocks inferred using these annotations.

In our view, synchronization idioms serve as contracts and proof hints to a static verification tool. For each synchronization idiom we explored, we provide an abstract semantics expressed in terms of predicates, which serves as the connection between the idiom and its realization within the program. In addition, for each idiom we provide a proof template. The programmer indicates which synchronization idioms he makes use of and supplies the predicates required by the template in terms of program variables. For example, for the readers/writer lock idiom, we require two predicates in terms of program variables indicating when the readers' (alternatively, writer's) lock is held. Based on the idiom and the predicates, we produce a proof script from the associated proof template. While being evaluated, the proof script triggers the annotation assistant to generate the program annotations required to make the script succeed. If the proof script succeeds, the assertions in the original program have been validated. Otherwise, the way in which the script fails allows us to distinguish between i) incorrect implementation of an idiom, for example, a mutex-lock implementation allowing more than one threads to acquire a lock, ii) incorrect use of the idiom, for example, releasing a mutex-lock before acquiring it, or iii) the idiom not being the appropriate one to guarantee the assertions in the program. From this point of view, the annotation generation serves as a debugging tool for the concurrency protocol of the program. While we opted for implementing this technique inside QED-VERIFIER, our theoretical results do not actually depend on which tool to use as long as the tool is based on Lipton's reduction theory.

Outline. Section 18.1 gives motivating examples that demonstrate the essence of the annotations while proving atomicity and debugging synchronization idioms. Section 18.2

presents our modeling of synchronization idioms and the annotation generation techniques. In Section 18.4, we give examples from the literature that can be handled by our method. We discuss the related work in Section 18.5.

18.1 Motivation and Overview

In Section 11.2 we gave an overview of how annotating programs helps in proving atomicity. This section gives motivation for the techniques presented in the following sections, through an overview of how this annotation procedure can be utilized to debug the synchronization in the program. Our annotation technique improves the approach demonstrated in Section 11.2 in two directions. First, using similar kinds of annotations, we can handle a variety of idioms including mutual exclusion, readers/writer locks and events. Second, the annotation generation scheme for an idiom, such as mutual exclusion, can be applied to different implementations of that idiom in a common way automatically. In the following we will give more information about our method on a larger example.

The sample codes in this chapter are given in QEDPL. For simplicity of presentation, we allow procedure calls in atomic blocks, where such calls are evaluated without being interleaved with other threads. However, we will describe the formal aspects of our approach using the ActionPL notations. Therefore, we will often use the term “action” to refer to either a gated action in ACTIONPL or an atomic statement in QEDPL; the meaning of the term should be clear from the context.

18.1.1 The example

Throughout this chapter, we will use several versions of the program given in Figure 18.1 to demonstrate our techniques. We denote parallel composition with “ $\parallel$ ”. Dashed lines are used to visually separate sequential statements. Similarly to the examples in Chapter 11, in this chapter, we will surround atomic actions with “[...]”. Note that, [...] corresponds to `atomic{...}` in QEDPL.

In the example, the global variables x and v stand for a shared data variable and its version number, respectively. Thread T1 initializes x and v and forks T2–T5. When T2–T5 all terminate, T1 checks whether the snapshots of the (x, v) pair read by T4 and T5 are

```

T1: [x := 0; v := 0;]
-----
T2:          T3:          T4:          T5:
Update(1); || Update(2); || (x1,v1):= Read(); || (x2,v2):= Read();
-----
T1: assert (v1 == v2) ==> (x1 == x2);

procedure Update(k : int) {    procedure Read() returns(lx:int, lv:int) {
1: x := k;                    1: lx := x;
2: lv := v;                   2: lv := v;
3: v := lv + 1;               }
}

```

Figure 18.1: Threads accessing the (x, v) pair

consistent. The `assert` statement makes the execution go wrong if the given condition evaluates to `false`.

Procedure `Update` writes to `x` and increases `v`, and procedure `Read` takes a snapshot of the (x, v) pair. The variables `lx` and `lv` are procedure-local. Each line in the procedures denotes a separate atomic action. The procedures are written so that each action does at most one global variable read or write.

If the bodies of `Update` and `Reads` in Figure 18.1 were executed sequentially, it would be trivial to prove that the assertion at the end of the program is not violated in any execution. However, in a concurrent execution, the lines of these procedures may get interleaved. In order to ensure that every call to `Read` returns the correct version number for `x`, the programmer must guarantee that `Update` and `Read` by different threads do not interfere with each other, i.e. `Update` and `Read` are atomic. In other words, no `Read` is allowed to execute between the writes to `x` and `v` by `Update`, and no `Update` is allowed to execute between reads of `x` and `v` by `Read`. If this is the case, then `Update` and `Read` can be analyzed as if each runs sequentially and the assertion can be verified with little effort.

In order to ensure the aforementioned non-interference requirement, the programmer employs the readers/writer (r/w) lock idiom. Figure 18.2 shows the new versions of `Update`

and **Read** using the r/w lock idiom. The figure also shows the implementation of the r/w lock, which uses the mutual exclusion (mutex) idiom to obtain the correct functionality of a r/w lock.

While proving that **Update** and **Read** are atomic, we show that the bodies of **Update** and **Read** are composed of actions of certain mover types. When the code in Figure 18.2 is considered as is, the conflicting actions accessing x and v cannot be shown to be movers. However, the r/w lock ensures that these lines cannot run consecutively in any execution, i.e., they never interfere with each other. Since the mover analysis involves local commutativity checks on individual actions, it misses this global information. By annotating these conflicting actions with assertions, we encode that these actions run at different synchronization states of the readers/writer lock. For example, suppose that one of the actions (α) in the pairwise mover check is executed when the writer lock is held, and the other action (β) in the check is executed when the readers lock is held. By adding to α and β assertions expressing these facts, we indicate that they cannot be executed immediately following each other in any execution. In fact, any such pair of α and β must be separated by release and acquire operations.

We prove that **Update** and **Read** are atomic in two steps. First, by generating annotations for the mutual exclusion idiom, we prove that procedures **AcqRead/RelRead** and **AcqWrite/RelWrite**, which implement the r/w lock operations, are composed of certain movers, thus are atomic. Second, we reason about the r/w lock to show that **Update** and **Read** are atomic. At each step, our method generates the assertions for the regarding idiom automatically, given a simple hint about how the idiom is implemented in the program. In the following, we will first introduce the implementations of these idioms, and then explain how the annotation engine is informed of these idioms through simple hints.

18.1.2 Synchronization Idioms

Figure 18.2 shows the procedures that implement the readers/writer and mutex lock idioms. While the former idiom is used to access the (x, v) pair atomically in **Update** and **Read**, the latter is used to implement the readers/writer lock correctly.

```

procedure Update(k : int) {
1: call AcqWrite();
2: x := k;
3: lv := v;
4: v := lv + 1;
5: call RelWrite();
}

procedure AcqWrite() {
1: call AcqMutex();
2: assume reads==0 && write==false;
3: write := true;
4: call RelMutex();
}

procedure RelWrite() {
1: call AcqMutex();
2: write := false;
3: call RelMutex();
}

procedure AcqMutex() {
1: [assume held == false;
2: held == true;]
}

procedure Read() returns(lx:int, lv:int) {
1: call AcqRead();
2: lx := x;
3: lv := v;
4: RelRead();
}

procedure AcqRead() {
1: call AcqMutex();
2: assume write==false;
3: reads := reads + 1;
4: call RelMutex();
}

procedure RelRead() {
1: call AcqMutex();
2: reads := reads - 1;
3: call RelMutex();
}

procedure RelMutex() {
1: [held := false;]
}

```

Figure 18.2: The procedures used in Figure 18.1

Readers/writer Lock

Procedures **AcqRead** and **RelRead** (alternatively, **AcqWrite** and **RelWrite**) are used to acquire and release the read (write) lock, respectively. The implementation ensures that no code between **AcqRead** and **RelRead** is executed concurrently with code between **AcqWrite** and **RelWrite**. However, code between **AcqRead** and **RelRead** can be executed by multiple threads concurrently. Thus, the r/w lock ensures that **Update** and **Read** may not run concurrently, and allows concurrent execution of multiple **Read** calls.

The integer variable **reads** stores the number of currently reading threads, and the boolean variable **write** indicates whether a thread is currently writing. The **assume** state-

ment models waiting on a condition; **assume** *e* terminates when *e* holds, otherwise it blocks¹.

The following invariant expresses the implementation of the r/w lock in the program: While **read**>0 and **write**==**false**, the read lock is held, and while **reads**==0 and **write**==**true** one thread has held the write lock. For the program to implement readers/writer correctly, the pair (**reads**,**write**) must be accessed atomically. The programmer ensures this using the mutual exclusion idiom.

Mutual exclusion

Procedures **AcqMutex** and **RelMutex** are used to acquire and release the mutex lock. We assume that **AcqMutex** and **RelMutex** have atomic bodies. The implementations of **AcqMutex** and **RelMutex** ensure that the code blocks in **AcqRead/RelRead** and **AcqWrite/RelWrite** between the calls to **AcqMutex** and **RelMutex** are executed without being interleaved with each other.

The programmer uses the boolean **held** variable to indicate whether the lock is currently held by a thread or not. The following invariant expresses the implementation of the mutual exclusion in the program: While **held**==**false**, the lock is free, and while **held**==**true** one thread has acquired the lock. The lock is acquired and released by setting **held** to **true** and **false**, respectively.

18.1.3 Annotating Procedures

Figure 18.3 shows the version of the procedures in Figure 18.2 annotated by our technique. We now describe the annotations generated by our tool and explain how the assertions indicate noninterference while proving that the procedures **Update** and **Read** are atomic.

Let us first consider how mutual exclusion is expressed in the annotations. The programmer provides the annotation tool the predicate **held**==**true**. The predicate indicates at which states the lock is held. The tool automatically determines the acquire and the release operations and generates the annotations as assignments to an auxiliary variable **m** and assertions referring to **m**.

¹The uses of **assume** at line 2 of **AcqWrite** and **AcqRead** in Figure 18.2 may cause deadlocks. We deliberately used **assume** to simplify the presentation. A deadlock-free implementation can be obtained by replacing **assume** *e* in both lines with **while**(!*e*){**RelMutex**();**AcqMutex**();}


```

procedure Update(k : int) {
1: [call AcqWrite(); w := tid;]
2: [assert w==tid; x := k;]
3: [assert w==tid; lv := v;]
4: [assert w==tid; v := lv + 1;]
5: [assert w==tid; call RelWrite();
   w := 0;]
}

procedure Read() returns(lx:int, lv:int) {
1: [call AcqRead(); w := 0;
   r[tid] := true;]
2: [assert w==0; lx := x;]
3: [assert w==0; lv := v;]
4: [assert w==0 && r[tid]==true;
   call RelRead(); r[tid] := false;]
}

procedure AcqWrite() {
1: [call AcqMutex(); m := tid;]
2: [assert m==tid;
   assume reads==0 && write==false;]
3: [assert m==tid;
   write := true;]
4: [assert m==tid; call RelMutex();
   m := 0;]
}

procedure AcqRead() {
1: [call AcqMutex(); m := tid;]
2: [assert m==tid;
   assume write==false;]
3: [assert m==tid;
   reads := reads + 1;]
4: [assert m==tid; call RelMutex();
   m := 0;]
}

procedure RelWrite() {
1: [call AcqMutex(); m := tid;]
2: [assert m==tid;
   write := false;]
3: [assert m==tid; call RelMutex();
   m := 0;]
}

procedure RelRead() {
1: [call AcqMutex(); m := tid;]
2: [assert m==tid;
   reads := reads - 1;]
3: [assert m==tid; call RelMutex();
   m := 0;]
}

```

Figure 18.3: The annotated version of the procedures in Figure 18.2

First, notice that lines 1 and 4 of `AcqRead` and `AcqWrite` are annotated with assignments to `m`. This is because the annotation engine detects that these lines acquire and release the mutex lock by calling `AcqMutex` and `RelMutex`, respectively. Thus, `m` must be updated to preserve the correspondence between the value of `m` and the lock state. Then, observe that the assertions at lines 2-3 of `AcqRead` and lines 2-3 of `AcqWrite` run when the thread executing these lines is holding the mutex lock. To see how this makes it possible to prove actions to be movers, consider a mover check between the following action pair:

<pre> AcqRead(): 2: [assert m==tid; </pre>	<pre> AcqWrite(): 3: [assert m==tid; </pre>

```
assume write==false;]           write := true;]
```

During the mover check, we assume that the actions above are executed by different threads whose `tid` variables are distinct. Thus, `tid` in each of the actions is substituted by distinct thread id. The assertions express that this pair of actions cannot run consecutively, since they are executed when the mutex lock is held by different threads. By using the assertions, we prove that the actions in `AcqRead` and `AcqWrite` that write to and read from the `(reads,write)` pair do not interfere with each other, thus these actions are movers. Consequently, `AcqRead` and `AcqWrite` are both atomic. A similar reasoning holds for `RelRead` and `RelWrite`. Once this is provided, we can now reason about the use of the readers/writer lock in `Update` and `Read`.

In order to generate the annotations for the r/w lock, the programmer provides the predicates `reads>0 && write==false` and `reads==0 && write==true`. These predicates indicate at which states the read and write lock are held, respectively. The annotation engine introduces the variable `w`, with domain $Tid \cup \{0\}$, and the boolean variable `r` to the program. These variables are used to add auxiliary information about the state of the readers/writer lock. The variable `w` indicates whether the write lock is being held and by which thread (`w==tid`), or the write lock is free (`w==0`). The variable `r` indicates whether the read lock is being held by the current thread (`r==true`) or not (`r==false`). The lines that read from and write to the `(x,v)` pair are annotated with assertions that indicate whether the write and read locks are free or not, respectively.

By considering the proved-atomic bodies of `AcqWrite` and `AcqRead`, the first lines of `Update` and `Read` are detected to acquire write and the read lock, respectively, so are annotated with assignments to `w` and `r`. Proper annotations are done for the actions that release the locks. Now consider lines 2-3 of `Read` and lines 2-4 of `Update`. While the former run while the read lock is held, the latter run while the write lock is held. Therefore these lines run at different synchronization states and thus do not interfere with each other. To see why these actions are movers, consider the following action pair executed by different threads:

<pre>Update(k : int): 2: [assert w==tid; x := k;]</pre>	<pre>Read(): 2: [assert w==0; lx := x;]</pre>

Notice that since the thread id stored in `tid` cannot have the value 0 and neither action assigns to `w`, the assertions in the action above conflict with each other. Therefore, this pair of actions cannot get executed consecutively.

In addition, the write lock provides mutual exclusion for concurrent runs of `Update`. To see how this is expressed through assertions, consider the following pair of actions, again executed by different threads:

<pre>Update(k : int): 2: [assert w==tid; x := k;]</pre>	<pre>Update(k : int): 2: [assert w==tid; x := k;]</pre>

During the mover analysis, this action pair is considered as if `tid` in each action refers to a different thread id. By a similar reasoning to mutual exclusion above, the assertions make this action pair trivially satisfy the mover check.

18.1.4 Debugging The Synchronization Idiom

While helping to prove atomicity, the generated annotations also indicate for which parts of the program the synchronization idiom is effective, and thus are useful for debugging applications of the idiom. We will now show several bug examples and explain how our annotation method helps to detect and diagnose these bugs.

Incorrect Use of the Idiom

Although the programmer correctly implements the data structures and operations in a synchronization idiom, s/he may not use these operations in the correct way required by the idiom. For example, two usage rules required by the readers/writer lock idiom are that (1) a thread must first acquire the write lock before using it, and (2) every release must match a previous acquire by the same thread. If the program contains a path that violates one of these rules, our technique is able to detect this. We do this by encoding each usage rule in two conditions: $\mathcal{R}$ (a rely condition) and $\mathcal{G}$ (a guarantee condition). These $(\mathcal{R}, \mathcal{G})$ pairs are generated automatically by the annotation engine given simple hints from the user about the synchronization idiom. During the check, $\mathcal{R}$ expresses an assumption that no other thread running concurrently with the procedure under the check violates the usage rule, while $\mathcal{G}$ asserts that every action of the current procedure obeys the rule. For example,

$\mathcal{R}$ for the r/w lock idiom expresses that if the lock is held by the current thread, no other thread may acquire it before the current one releases it. The corresponding $\mathcal{G}$ asserts that an action does not acquire a lock held by another thread. We check each procedure in the program against the usage rule, by generating a verification condition (VC) using $\mathcal{R}$ and $\mathcal{G}$ and checking its validity (using an SMT solver). The validity of the VC indicates that the procedure obeys the usage rule. If the checking of a VC fails, our annotation engine rejects to annotate the program and informs the programmer about the actions violating the usage rule. For example, if there is an execution path in a procedure that releases a lock before acquiring it, the annotation engine indicates the actions executed along this path.

Missing Synchronization

If the mover check for an action α protected by a synchronization idiom fails, there are two possible reasons. First, an annotation may be incorrect or too weak to show the desired non-interference. Section 18.1.5 gives an example where the annotations are weak. However, our annotation engine generates strong enough assertions to prove the atomicity of code blocks protected by a synchronization idiom. The second reason is that there is true interference between two actions in the program, as the programmer forgets to protect these actions using the idiom. In this case, the actual use of the idiom does not fit the intended use of the idiom.

For example, suppose that the example in Figure 18.1, using the procedures defined in Figure 18.2, contains an extra thread T6, in parallel to T2-T5, that updates the (x, v) pair without calling **AcqWrite** and **RelWrite**. Since T6 completely ignores using the r/w lock, the actions in T6 will not be annotated with assertions, and this indicates missing synchronization in T6. As a result, the code surrounded by **AcqWrite** and **RelWrite** will not get reduced to atomic blocks, since there is interference between the actions in this code and the actions of T6; the assertions in actions surrounded by **AcqWrite** and **RelWrite** do not help in ruling out the interference. The failing mover checks and the missing assertions help the user to identify the missing synchronization.

Incorrect Implementation of the Idiom

Missing annotations indicate either an incorrect implementation of the idiom, or insufficient atomicity level for the program. For example, suppose that the readers/writer lock implementation in Figure 18.2 does not call `AcqMutex` and `RelMutex`. As a result, the bodies of `AcqRead/RelRead` and `AcqWrite/RelWrite` will not be reduced to single atomic actions, and the pair `(reads, write)` is not read and updated atomically. However, for the annotation engine to identify the acquire and release operations on the readers/write lock, these operations should appear as atomic actions in the program. Thus, the annotation engine will not be able to find these operations and will not annotate the program. The missing annotations in `Update` and `Read`, although they call `AcqRead/RelRead` and `AcqWrite/RelWrite`, indicate that the procedures `AcqRead/RelRead` and `AcqWrite/RelWrite` do not implement the intended lock operations correctly. In this case, the programmer should first ensure that the bodies of `AcqRead/RelRead` and `AcqWrite/RelWrite` can be reduced to single atomic actions, and identified by the annotation engine the operations of the readers/write lock idiom.

18.1.5 Naive Attempts

We conclude this section by showing two failing attempts to annotating our example program despite the fact that the example is correct. These illustrate that coming up with the right program annotations is non-trivial, and indicates the essence of automating the annotation process.

<pre> procedure AcqWrite() { 1: [call AcqMutex(); m := true;] 2: [assert m==true; assume reads==0 && write==true;] 3: [assert m==true; write := true; 4: [assert m==true; call RelMutex(); m := false;] } </pre>	<pre> procedure AcqRead() { 1: [call AcqMutex(); m := true; 2: [assert m==true; assume write==false;] 3: [assert m==true; reads := reads + 1; 4: [assert m==true; call RelMutex(); m := false;] } </pre>

Figure 18.4: The annotated procedures with weaker assertions

Assertions Too Weak

Let us now use a boolean auxiliary variable `m` to perform the annotation for the mutual exclusion idiom as shown in Figure 18.4. Predicates `m==true` (alternatively, `m==false`) indicate the mutex lock is being held (alternatively, free). In the new scheme, the lines that call `AcqMutex` and `RelMutex` assign `true` and `false` to `m`, respectively.

Although the assertions are valid, these annotations are not enough to show that actions of `AcqRead` and `AcqWrite` are not simultaneously enabled, since both actions seem to run at the same synchronization state, where `m==true` holds for both actions. Thus, the mover check for these actions fails in this case, since the assertions do not suffice to express the fact that the lock is held by different threads. By contrast, the assertions `m==tid` inserted by our annotation scheme are strong enough to express this fact properly.

```

procedure Update(k : int) {
1: [call AcqWrite(); w := tid;]
2: [assert w==tid; x := k;]
3: [assert w==tid; lv := v;]
4: [assert w==tid; v := lv + 1;]
5: [assert w==tid; RelWrite();
   w := 0;]
}

procedure Read() returns(lx:int, lv:int) {
1: [call AcqRead(); w := tid;]
2: [assert w==tid; lx := x;]
3: [assert w==tid; lv := v;]
4: [assert w==tid; RelRead();
   w := 0;]
}

```

Figure 18.5: The annotated procedures with stronger assertions

Assertions Too Strong

Figure 18.5 gives an annotation scheme where `Update` and `Read` are both annotated with the same assertion `w==tid`. This makes pairs of actions having elements from `Update` and `Read` trivially satisfy the mover check.

However the assertions `w==tid` in `Read` indicate that no two thread may run `Read` simultaneously. In fact, these assertions are not valid as the readers/writer lock allows multiple instances of `Read` to run concurrently, violating the assertion. Consequently, it will not be possible to discharge these assertions in later stages of the proof. If this happens, it is difficult to distinguish whether the program contains a bug or whether the annotations

contained erroneous reasoning. Our annotation engine always generates valid assertions, which can be discharged later in the proof, preventing this difficulty. In fact, we will show in Section 18.2 that the assertions added by our tool to help in a subsequent reduction step are all discharged after the critical regions protected by the idiom are proved to be atomic.

18.2 The Annotation Method

Recapitulating Section 18.1, one of the main objectives of the work presented in this chapter is to have the program text and annotations reflect the synchronization idiom employed to control concurrency. Such an objective necessarily starts with the formal specification of the idiom. The specification provides the type of the idiom and some information on how each program line relates to the implementation of this idiom. The outcome of this step is a template which can be used to correctly annotate the program. The annotation can either be done by the user manually, following a proof script or automatically by a theorem prover. In this section, we will explain each step in detail, illustrating the main ideas via the running example.

18.2.1 Describing Synchronization Idioms

A synchronization idiom stands for any well-known pattern to restrict the amount of concurrency in the execution of a program. For instance, readers/writer lock is an idiom denoting the collection of implementation methods which distinguish the lines of code, the critical sections, for reading and writing part of memory that cannot be executed concurrently. We formalize the idea of a synchronization idiom by specifying a set of *idiom formulas*. For the mutual exclusion idiom, this set contains one formula, ϕ , over program variables. Intuitively, ϕ is the predicate which becomes **true** exactly when there is one thread that has the mutex lock. For the readers/writer lock idiom, the set contains two formulas: ϕ_R and ϕ_W both of which are predicates over program variables. While ϕ_R becomes **true** only when there is at least one thread that has a reader's lock, ϕ_W becomes **true** exactly when there is (exactly) one thread holding the writer's lock.

The idiom formulas express how the idiom is implemented in the program. Recalling our running example, the use of the mutual exclusion in Figure 18.2 is specified by the predicate $\phi = (\text{held} == \text{true})$. This means the program contains atomic actions that acquire and release

the lock by setting and resetting ϕ to **true** and **false**, respectively. The procedures **AcqMutex** and **RelMutex** contain these actions. Our method is able to recognize these actions and do the required annotation. Similarly, the use of the r/w lock is specified by the predicates $\phi_R = (\text{reads} > 0 \wedge \text{write} == \text{false})$ and $\phi_W = (\text{write} == \text{true})$. Each of the procedures **AcqRead**, **RelRead**, **AcqWrite** and **RelWrite** performs an operation that corresponds to either acquiring or releasing either the read or the write lock.

$$\begin{array}{l}
\boxed{\mathcal{S}_{\text{rwlock}}(\phi_R, \phi_W)} \\
\\
\mathbb{V} = \{r, w\} \quad \text{Dom}(w) = \text{Tid} \cup \{0\} \\
\quad \text{Dom}(r) = \text{PowSet}(\text{Tid} \times \{\text{true}, \text{false}\}) \\
\mathbb{I} = \bigwedge \left(\begin{array}{l} \phi_W \Rightarrow \neg \phi_R \\ \neg \phi_W \Leftrightarrow w = 0 \\ \phi_R \Leftrightarrow \exists t. r(t) = \text{true} \end{array} \right) \\
\mathbb{A} = \bigwedge \left(\begin{array}{l} \forall t \in \text{Tid}. \quad t \neq \text{tid} \Rightarrow r'(t) = r(t) \\ (\phi_W \Leftrightarrow \phi'_W) \Rightarrow w' = w \\ (\phi_R \Leftrightarrow \phi'_R) \Rightarrow r(\text{tid})' = r(\text{tid}) \\ (\neg \phi_W \wedge \phi'_R) \Rightarrow r'(\text{tid}) = \text{true} \\ (\neg \phi_R \wedge \phi'_W) \Rightarrow w' = \text{tid} \wedge r'(\text{tid}) = \text{false} \\ (\phi_W \wedge \neg \phi'_W) \Rightarrow w' = 0 \end{array} \right) \\
\mathbb{P} = \{ w = \text{tid}, w = 0, w \neq 0, w \neq \text{tid}, \\ \quad r[\text{tid}] = \text{true}, r[\text{tid}] = \text{false} \} \\
\\
\mathcal{R} = \bigwedge \left(\begin{array}{l} (w = \text{tid}) \Rightarrow (w' = w) \\ r(\text{tid}) = r'(\text{tid}) \end{array} \right) \\
\mathcal{G} = \bigwedge \left(\begin{array}{l} (w \neq 0 \vee w \neq \text{tid}) \Rightarrow w' = w \\ \forall t. t \neq \text{tid} \Rightarrow r(t) = r'(t) \end{array} \right)
\end{array}$$

Figure 18.6: The idiom template for the r/w lock idiom

18.2.2 Relating the Idiom to the Program

The idiom formulas divide the state space of the program. For example, the mutual exclusion idiom divides it into two, states in which the lock is held and states in which the lock is

free. In order to apply static proof methods to the program, the structure of the state space must be linked to the program text. We introduce now a structure, the *idiom template*, for establishing this link.

The idiom template is a tuple $\langle \mathbb{V}, \mathbb{I}, \mathbb{A}, \mathbb{P} \rangle$. The first component $\mathbb{V}$ is the set of auxiliary variables used to relate the idiom formulas to the gated actions of the program. $\mathbb{I}$ is the global invariant that is expected to hold if the idiom is correctly implemented and used. $\mathbb{A}$ defines the transition relation of the auxiliary variables in $\mathbb{V}$. This transition relation is defined using the truth values of the idiom formulas. Finally, $\mathbb{P}$ is a set of predicates over variables in $\mathbb{V}$. Each predicate in $\mathbb{P}$ is a candidate for the final annotation of the actions in the program due to the idiom under consideration. These predicates over the program variables and the variables in $\mathbb{V}$ encode the restrictions on concurrent executions enforced by the idiom.

The template for the r/w lock will be denoted by $\mathcal{S}_{\text{rwlock}}(\phi_R, \phi_W)$. The subscript and the parameters used for naming the template show the idiom and its formulas. The user has to specify both the idiom, e.g., that it is the r/w lock, and its associated formulas, ϕ_R and ϕ_W for the r/w lock idiom. Figure 18.6 shows the elements of this idiom template.

The set of the auxiliary variables, $\mathbb{V}$, contains two variables: w and r . The variable w can hold any value in the set $Tid \cup \{0\}$. Intuitively, its value defines the state of the writer's lock: if 0, no thread has the lock; if t , then the thread with id t holds the lock. The variable r is a unary predicate over Tid . Intuitively, the value $r(t)$ is true only when the thread with id t holds a reader's lock. As can be seen, these variables depict whether a lock is held and if so, by which thread.

The invariant $\mathbb{I}$, and the transition predicate $\mathbb{A}$ that annotates each action in the program together establish the link between the auxiliary variables, program state and transitions by making use of the idiom formulas ϕ_W, ϕ_R .

$\mathbb{I}$, is a conjunction of three terms. The first conjunct specifies that whenever the condition for the writer lock is satisfied, the condition for the reader lock is not. The other conjuncts relate the auxiliary variables to the idiom formulas, formalizing what we have described in the previous paragraph. The variable w is 0 iff the writer lock condition ϕ_W is not satisfied. The last conjunct states that the reader lock condition ϕ_R is satisfied iff there is at least one t such that $r(t)$ holds.

Whereas in $\mathbb{I}$, state invariants were given, $\mathbb{A}$ will provide the transition counterpart; that is, how auxiliary variables w , r change their values based on the change in the values of the idiom formulas ϕ_W , ϕ_R . For example, the fourth conjunct states that whenever the current thread acquires the readers lock, expressed by ϕ_W being false in the current state and ϕ_R being true in the next state, the auxiliary variable valuation should reflect this by setting the next value of $r(tid)$ to **true**. Similarly, the fifth conjunct states that whenever the current thread acquires the writers lock, expressed by ϕ_R being false in the current state and ϕ_W being true in the next state, the auxiliary variables should reflect this by setting the next state value of w is set to tid .

The candidate annotation set $\mathbb{P}$ contains four predicates. Each predicate represents some information about the writers lock. The first two are precise: $w = tid$ means the writers lock is held by the current thread; $w = 0$ means the writers lock is not held by any thread. The third and fourth are less precise: the third one states that the lock is held by some unspecified thread; the fourth one states that the current thread is not holding the writers lock. We have found that these predicates are strong enough to correctly annotate the program code in order to prove the desired non-interference targeted by the r/w lock idiom.

Recalling our running example, each of procedures **AcqRead**, **RelRead**, **AcqWrite** and **RelWrite** performs an operation that corresponds to a transition specified in one of the lines in $\mathbb{A}$. As a result of the annotation, the predicates **w==tid** and **w==0** are inserted to the code blocks in which the write lock is held and the write lock is not held (in fact, the readers lock is held), respectively.

18.2.3 Annotating Programs - Template for Manual Insertion

An idiom template $\langle \mathbb{V}, \mathbb{I}, \mathbb{A}, \mathbb{P} \rangle$ generated as explained in the previous section can be used as the basis of a QED proof script. In this section, we present a script (Figure 18.7) in which the user guesses the program annotations. We do this for ease of exposition. In the next section, we explain how we automatically generate annotations.

The first phase, line (1), temporarily removes existing assertions in the program. These assertions will be restored at the end of the script in line (8). They are temporarily removed so that the proof effort in the script concentrates solely on assertions relating to the synchronization idiom under study. The added annotations should be only about the idiom;

(1)	pushasserts	
(2)	auxannotate	$\mathbb{V}, \mathbb{A}$
(3)	invariant	$\mathbb{I}$
(4)	specify	$\mathbf{p}_1 label_1, pre_1, post_1$
	...	...
	specify	$\mathbf{p}_n label_n, pre_n, post_n$
(5)	assert	$label_1, p_1$
	...	...
	assert	$label_m, p_m$
(6)	reduce	$\mathbf{p}_1, \dots, \mathbf{p}_n$
(7)	check	$\mathbf{p}_1, \dots, \mathbf{p}_n$
(8)	popasserts	

Figure 18.7: The proof script for adding annotations

they will be checked in phase (7).

Line (2) introduces the auxiliary variables $\mathbb{V}$ and their transition relation $\mathbb{A}$. This transition relation is embedded into the program by replacing every gated action $\mathbf{true} \triangleright \tau$ in $Atoms(P)$ with $\mathbf{true} \triangleright (\tau \wedge \mathbb{A})$. This step also annotates the action with proper assignments to the auxiliary variable, as implied by the transition predicate of the action and $\mathbb{A}$. For example, the atomic body of `AcqWrite` together with $\mathbb{A}$ to the r/w lock implies that `w==tid` at the end, so the assignment `w := tid` is appended to the actions that call `AcqWrite`.

Line (3) introduces the invariant $\mathbb{I}$. If $\mathbb{I}$ is not found to be an invariant of the program, this indicates either the idiom is not correctly implemented or it is too early in the proof to apply the proof script: the program invariant is not strong enough, or the atomic blocks are not large enough.

In line (4), the user provides specifications for the procedures and the loops of the program as they concern this synchronization idiom. As such, each specification is a pair of formulas taken from the set $\mathbb{P}$: the pre-condition pre_i , which should hold when the procedure or the loop starts execution and the post-condition $post_i$ which should hold when the execution of

the said part completes. These formulas are what the user thinks will hold about the idiom under consideration. Recall that each element of $\mathbb{P}$ is only about the synchronization idiom. This might be better pictured as a projection of the specifications for procedures and loops onto idiom relevant information. If no specification is given, both formulas are assumed to be equal to the invariant.

Line (5) is the main annotation step. The user annotates the code by specifying a gated action and how its gate should be strengthened. That is, once the user guesses p_i among those contained in $\mathbb{P}$ as an annotation for a particular action $\phi \triangleright \tau$ with label $label_i$, `assert labeli, pi` adds the predicate $p_i \in \mathbb{P}$ to the gated action $\phi \triangleright \tau$, i.e. replaces the action with $(\phi \wedge p_i) \triangleright \tau$.

Line (6) applies Lipton's reduction iteratively. At the end of this phase, certain code blocks are combined into compound atomic actions. Line (7) checks whether the assertions inserted in line (5) are indeed correct. For each gated action in $\mathbf{p}_i$, `check` performs a sequential analysis that verifies the gate of the action by considering the program invariant as the precondition. Disposing assertions by sequential reasoning in this way is made possible by the reductions performed in line (6) over the newly annotated program. If no assertion fails, the atomic blocks computed in line (6) are indeed atomic and the guessed annotations are correct. The assertions might fail either because the idiom is not correctly implemented or because the assertions are too weak, too strong or simply wrong. This failure is likely to provide valuable insight to the user about the program and the implementation of the idiom.

18.2.4 Annotating Programs - Algorithm

The successful discharging of the assertions introduced in line 5 of Figure 18.7 depends on the usefulness of these assertions for combining the annotated actions to larger ones. As explained, failure by the user to come up with appropriate annotations during that phase is likely to be the common case as there are too many pitfalls. In this section, we will provide a remedy: we will explain a new approach which makes assertion generation completely automatic.

We modify the proof script of Figure 18.7 by replacing line 5 with the following proof command:

```

Proof command instrument  $\mathbb{P}, \mathcal{R}, \mathcal{G}$  ( $P, \mathcal{I}$ ):
1    // check the program without adding assertions
2     $\Psi := \text{GenerateVC}(P, \mathcal{I}, \mathcal{R}, \mathcal{G})$ 
3    if ( $\Psi$  is not valid) then
4        Fail and return the original program  $P$ 
5    // do instrumentation per predicate in  $\mathbb{P}$ 
6    foreach ( $p \in \mathbb{P}$ ) do
7        // add  $p$  to gated actions as assertion
8        foreach ( $\phi \triangleright \tau \in \text{Atoms}(P)$ ) do
9            if ( $(\phi \wedge p)$  is satisfiable)
10               Replace  $\phi \triangleright \tau$  with  $(\phi \wedge p) \triangleright \tau$  in  $P$ 
11    // remove invalidated assertions
12    while (true) do
13         $\Psi := \text{GenerateVC}(P, \mathcal{I}, \mathcal{R}, \mathcal{G})$ 
14        break if ( $\Psi$  is valid)
15         $\text{Failed} := Z_3(\Psi)$ 
16        foreach ( $(\phi \wedge p) \triangleright \tau \in \text{Failed}$ ) do
17            Replace  $(\phi \wedge p) \triangleright \tau$  with  $\phi \triangleright \tau$  in  $P$ 
18    Return the current program  $P$ 

```

Figure 18.8: The inference algorithm

(5) `instrument  $\mathbb{P}, \mathcal{R}, \mathcal{G}$`

The proof command `instrument` expects as parameters a set of predicates and two transition formulas, $\mathcal{R}$ and $\mathcal{G}$. $\mathbb{P}$ contains the predicates to be used in the assertions generated by `instrument`. Intuitively, the formulas $\mathcal{R}$ and $\mathcal{G}$ specify which state transitions related to the synchronization idiom are allowed. Both are actually automatically generated once the idiom formulas are supplied; their values are given in Figure 18.6 for the r/w lock idiom.

The formula $\mathcal{R}$ gives a global condition that a thread can assume about the actions executed by other threads. In the case of the r/w lock, $\mathcal{R}$ consists of two conditions. The first one states that the status of a held writers lock cannot change as long as only those

threads not holding the lock are executing. The second states that the readers lock cannot be assigned to or taken from a thread by some other thread.

The formula $\mathcal{G}$, on the other hand, gives a local condition that a thread has to satisfy each time it executes an action. Again referring to our example given in Figure 18.6, there are two conditions that have to be satisfied. The first one states that if the action is executed by a thread that does not hold the writers lock which is held by some other thread, the state of the writers lock cannot change. The second one states that the running thread cannot change the readers lock status of any other thread.

The pseudo-code explaining how `instrument` operates is given in Figure 18.8. The extra parameters seen in the pseudo-code are the predicate component $\mathbb{P}$ of the idiom template and the current proof context $(P, \mathcal{I})$. As will be explained below, the proof context can be modified by the proof command.

The proof command `instrument` starts with a sanity check for the idiom formulas (lines 2-4). A verification condition (VC) generator is called for the current proof context along with the formulas $\mathcal{R}$ and $\mathcal{G}$. We will skip the details of how the VC generator, *GenerateVC*, operates. Briefly, it is similar to [41], except for the following: while the weakest-precondition of each gated action $\phi \triangleright \tau$ is computed as $\text{wp}(\phi \triangleright \tau, \psi) = \forall \text{Var}'. \phi \wedge (\tau \Rightarrow \psi')$ in the sequential case, we compute it in the concurrent case as follows:

$$\text{wp}(\phi \triangleright \tau, \psi) = (\forall \text{Var}'. \phi \wedge (\tau \Rightarrow \mathcal{G})) \wedge (\forall \text{Var}'. (\mathcal{R} \circ \tau \circ \mathcal{R}) \Rightarrow \psi')$$

This new form of the weakest-precondition expresses that each gated action assumes $\mathcal{R}$ and asserts $\mathcal{G}$. `instrument` forwards the VC to a theorem prover, which, in our framework, is the Z3 SMT solver [53].

The output of the VC generation routine called at line 2 is the formula Ψ . If the idiom formulas match with the idiom they specify, Ψ comes out as a valid formula. This is why a non-valid Ψ causes an abortion of the overall annotation script much like (3) of Figure 18.7.

In the remaining part of `instrument` (lines 6-17), a suitable annotation for each gated action is sought. First, a predicate p from the set $\mathbb{P}$ is chosen. The only requirement for p is that it should not be weaker than another predicate not yet chosen. This is to ensure that always the strongest possible annotation is used for gated actions. Then, each gated action whose gate does not contradict with p is annotated with p (lines 8-10). Then, until a valid

VC is generated, those gated actions whose modified assertions may be violated (elements of the *Failed* set at line 15, returned by Z3) are restored to the values they had at the start of the outer loop iteration (lines 16-17). This iterative removal of annotations between lines 12-17 is bound to terminate, because after line 13 either (1) Ψ is valid and the iterations terminate or (2) at least one annotated action is restored to its form before being annotated with p , and how many times (2) can happen is bounded by the number of actions. Due to the successful check at line 2-3, the check at line 14 succeeds if all the actions are restored to their form before being annotated with p .

The assertions added at the end of `instrument` as annotations to the gated actions of the program carry information about the idiom. The candidate annotation predicates given by $\mathbb{P}$ appeal to the understanding of these idioms. Furthermore, in the light of the experimental evidence (Section 18.4), we conjecture that the absence of interference targeted by the synchronization idiom is correctly captured by the assertions added by `instrument`.

18.2.5 Mutual Exclusion

$$\begin{array}{lcl}
 \boxed{\mathcal{S}_{\text{mutex}}(\phi)} & & \\
 \mathbb{V} & = & \{m\} \quad \text{dom}(m) = \text{tid} \cup \{0\} \\
 \mathbb{I} & = & (\neg\phi) \Leftrightarrow (m = 0) \\
 \mathbb{A} & = & \bigwedge \left(\begin{array}{lcl} (\neg\phi \wedge \phi') & \Rightarrow & (m' = \text{tid}) \\ (\phi \wedge \neg\phi') & \Rightarrow & (m' = 0) \\ (\phi \Leftrightarrow \phi') & \Rightarrow & (m' = m) \end{array} \right) \\
 \mathbb{P} & = & \{ m = \text{tid}, m = 0, m \neq 0, m \neq \text{tid} \} \\
 \\
 \mathcal{R} & = & (m = \text{tid}) \Rightarrow (m' = m) \\
 \mathcal{G} & = & (m \neq 0 \vee m \neq \text{tid}) \Rightarrow m' = m
 \end{array}$$

Figure 18.9: The idiom template for the mutual exclusion idiom

The idiom template for the mutual exclusion idiom is given in Figure 18.9. In the mutual exclusion idiom, threads acquire locks to get exclusive access to blocks of code. In the template, we represent the lock conceptually with the predicate ϕ . Similarly to the r/w lock, ϕ is

true whenever the lock is acquired. The variable m stores either 0 or a thread id, indicating whether the lock is free or held by a thread, respectively. $\mathbb{I}$, $\mathcal{R}$, and $\mathcal{G}$ associate ϕ with m and define the semantics of the mutual exclusion idiom. The template $\mathcal{S}_{\text{mutex}}(\phi)$ is used in the proof script given Figure 18.7 to generate annotations for the mutual exclusion idiom. Recalling the example in Figure 18.2, the idiom is implemented by procedures **AcqMutex** and **RelMutex**. Since this implementation guarantees that **held==true** holds within the critical sections, the template $\mathcal{S}_{\text{mutex}}(\text{held}==\text{true})$ is provided to annotate the program.

Variations of mutual exclusion can also be specified in a common form. For example, the template in Figure 18.10 can be used in cases where the conditions that hold in different critical regions are not complements of each other as in the above template.

$$\boxed{\mathcal{S}_{\text{mutex2}}(\phi_1, \phi_2)}$$

$$\begin{aligned}
 \mathbb{V} &= \{m\} \quad \text{dom}(m) = \text{tid} \cup \{0\} \\
 \mathbb{I} &= (\phi_1 \Rightarrow (\neg\phi_2)) \wedge ((\phi_1 \vee \phi_2) \Leftrightarrow (m \neq 0)) \\
 \mathbb{A} &= \bigwedge \left(\begin{array}{l}
 (\neg\phi_1 \wedge \phi'_1) \Rightarrow (m' = \text{tid}) \\
 (\neg\phi_2 \wedge \phi'_2) \Rightarrow (m' = \text{tid}) \\
 (\phi_1 \wedge \neg\phi'_1) \Rightarrow (m' = 0) \\
 (\phi_2 \wedge \neg\phi'_2) \Rightarrow (m' = 0) \\
 (\phi_1 \Leftrightarrow \phi'_1) \Rightarrow (m' = m) \\
 (\phi_2 \Leftrightarrow \phi'_2) \Rightarrow (m' = m)
 \end{array} \right) \\
 \mathbb{P} &= \{m = \text{tid}, m = 0, m \neq 0, m \neq \text{tid}\} \\
 \mathcal{R} &= (m = \text{tid}) \Rightarrow (m' = m) \\
 \mathcal{G} &= (m \neq 0 \vee m \neq \text{tid}) \Rightarrow m' = m
 \end{aligned}$$

Figure 18.10: The idiom template for a variation of the mutual exclusion idiom

18.2.6 Reentrant Locks

The idiom template for the reentrant locks is given in Figure 18.12. A reentrant lock idiom is similar to a mutex lock, except that it can be acquired and released by the same thread multiple times as long as acquires and the releases are properly nested. Reentrant locks are

widely-used to implement monitors in modern languages like Java and C#. We provide a simple reentrant lock implementation in Figure 18.11.

The implementation must keep a counter to remember how many times the lock has been acquired by the same thread, and decrease the counter at each release. In the template we represent the state of the counter with the expression ϕ_c . If $\phi_c > 0$ holds then the lock is acquired, and if $\phi_c = 0$ then the lock is free. In the template the thread id that acquired the lock is represented by the auxiliary variables m . The template $\mathcal{S}_{\text{reentrant}}(\phi_c)$ is used in the proof script given in Figure 18.7 to generate annotations for the reentrant locks idiom. When the implementation in Figure 18.11 is used, then the template is instantiated by providing $\phi_c = \text{count}$.

```

procedure AcqReentrant() {
1: [ if(owner == tid) {
2:   count = count + 1;
3: } else {
4:   assume owner == 0;
5:   count := 1;
6:   owner := tid;
7: } ]
}

procedure RelReentrant() {
1: [ count := count - 1;
2:   if(count == 0) {
3:     owner := 0;
4:   } ]
}

```

Figure 18.11: Example implementation of reentrant locks

18.2.7 Event Synchronization

Different forms of event synchronization appear in concurrent programs. A common form containing one setter and k waiters is the following:

$$(S_1; S; S_2) \parallel (S_1^1; W; S_2^1) \parallel \cdots \parallel (S_1^k; W; S_2^k) \parallel \cdots$$

In this context, an event is represented by a predicate ϕ . At the beginning of the above statement, ϕ evaluates to **false**, meaning that the event has not been set. The event is set by performing an operation after which ϕ evaluates to **true**. The statement $S_1; S; S_2^i$ sets the

$$\begin{array}{lcl}
\boxed{\mathcal{S}_{\text{reentrant}}(\phi_c)} & & \\
\mathbb{V} & = & \{m\} \quad \text{dom}(m) = \text{tid} \cup \{0\} \\
\mathbb{I} & = & ((\phi_c > 0) \Leftrightarrow (m = 0)) \wedge (\phi_c \geq 0) \\
\mathbb{A} & = & \bigwedge \left(\begin{array}{ll} (\phi'_c = \phi_c + 1) & \Rightarrow (m' = \text{tid}) \\ (\phi'_c = 0) & \Rightarrow (m' = 0) \\ (\phi'_c = \phi_c) & \Rightarrow (m' = m) \end{array} \right) \\
\mathbb{P} & = & \{ m = \text{tid}, m = 0, m \neq 0, m \neq \text{tid} \} \\
\\
\mathcal{R} & = & (m = \text{tid}) \Rightarrow (m' = m) \\
\mathcal{G} & = & (m \neq 0 \vee m \neq \text{tid}) \Rightarrow (m' = m)
\end{array}$$

Figure 18.12: The idiom template for the reentrant locks

event (makes ϕ **true**), and the statement $(S_1^i; W; S_2^i)$ waits for the event to be set, i.e., W terminates only when ϕ is true. We will refer to $(S_1; S)$, $S_1^1, \dots, S_1^k$ and $S_2, (W; S_2^1), \dots, (W; S_2^k)$ as *before* and *after* phases of the statements, respectively.

As an example, consider the program in Figure 18.13, which uses event synchronization implemented using the variable **done**. The “before” phase of T2 updates the pair $(\mathbf{x}, \mathbf{v})$ and it sets **done** to **true**. Then the “after” phase of T3 reads the $(\mathbf{x}, \mathbf{v})$ pair. The event prevents T3 from reading the initial values of $\mathbf{x}$ and $\mathbf{v}$.

We handle event synchronization in S through the following steps:

1. We first annotate the gated actions in the parallel statement S with assertions using the technique described in Section 18.2.3. The idiom template for event synchronization is given in Figure 18.14. The predicate ϕ in the template represents the condition over the program variables representing the event. For the program in Figure 18.13, for example, $\mathcal{S}_{\text{event}}(\text{done} == \text{true})$ is used for annotation generation. The variables e and s store the owner of the event and whether the even has been set, respectively.
2. We do reduction using the `reduce all` command, which applies a sequence of reduction rules until no more rule are applicable.
3. We then use the proof rules **HOIST-L-MOVER** and **HOIST-R-MOVER**, given in Fig 18.15, to

```

T1: x := 0;  v := 0;  InitEvent();

-----

T2                                T3
1: Update(1);                      1: WaitEvent();
2: SetEvent();                     2: (x1,v1) := Read();

-----

T1: assert (x1 == 1) && (v1 == 1);

procedure InitEvent(){           procedure SetEvent() {           procedure WaitEvent() {
1: done := false;                1: done := true;                1: assume (done==true);
}                                }                                }

```

Figure 18.13: Synchronizing threads with events

$$\boxed{\mathcal{S}_{\text{event}}(\phi)}$$

$$\begin{aligned}
\mathbb{V} &= \{e, s\} \quad \text{Dom}(e) = \text{tid} \cup \{0\} \quad \text{Dom}(s) = \{\text{true}, \text{false}\} \\
\mathbb{I} &= \phi \Leftrightarrow ((s = \text{true}) \wedge (e = 0)) \\
\mathbb{A} &= \bigwedge \left(\begin{array}{l} (\neg\phi \wedge \phi') \Rightarrow (e' = 0 \wedge s' = \text{true}) \\ (\phi \Leftrightarrow \phi') \Rightarrow (e' = e \wedge s' = s) \end{array} \right) \\
\mathbb{P} &= \{s = \text{true}, s = \text{false}\} \\
\mathcal{R} &= ((e = \text{tid} \wedge s = \text{false}) \vee (e = 0 \wedge s = \text{true})) \Rightarrow \\
&\quad (e' = e \wedge s' = s) \\
\mathcal{G} &= ((e \neq \text{tid} \wedge e \neq 0 \wedge s = \text{false}) \vee (e = 0 \wedge s = \text{true})) \Rightarrow \\
&\quad (e' = e \wedge s' = s)
\end{aligned}$$

Figure 18.14: The idiom template for the event synchronization idiom

hoist before and after phases out of the parallel statement S . If the before phase of a setter statement is a left mover, **HOIST-L-MOVER** hoists this phase before the parallel statement. Similarly, if the after phase of a waiting thread is a right mover, **HOIST-R-MOVER** hoists this phase after the parallel statement. This makes the before and after phases in S sequentially composed with each other, which is in fact enforced by the

event synchronization at runtime.

HOIST-L-MOVER	HOIST-R-MOVER
$(P, \mathcal{I}) \vdash \alpha : \mathbb{L}$	$(P, \mathcal{I}) \vdash \alpha : \mathbb{R}$
$(P, \mathcal{I}) \dashrightarrow (P[(\alpha; S_1) \ S_2] \mapsto (\alpha; (S_1 \ S_2)), \mathcal{I})$	$(P, \mathcal{I}) \dashrightarrow (P[(S_1; \alpha) \ S_2] \mapsto ((S_1 \ S_2); \alpha), \mathcal{I})$

Figure 18.15: Rules for hoisting actions out of a parallel statement

Now recall the example in Figure 18.13. Using the idiom template $\mathcal{S}_{\text{event}}(\text{done} == \text{true})$, we add the assertion $\mathbf{s} == \text{false}$ to lines 1-2 of T2 and the assertion $\mathbf{s} == \text{true}$ to lines 1-2 of T3. The assertions allow us to show that actions of T2 are left movers and actions of T3 are both-right movers. Using HOIST-L-MOVER for T2 and HOIST-R-MOVER for T3, we obtain a sequential composition of the initialization by T1, T2, T3 and the assertion by T1.

18.3 Barrier Synchronization

The proof command $\boxed{\text{barrier } S, \phi, \psi, B_1, \dots, B_n}$ captures the barrier synchronization in the parallel statement S . The statement S is in the form of $(S_1^1; B_1; S_1^2) \| \dots \| (S_n^1; B_n; S_n^2)$. The statements $B_1, \dots, B_n$ implement the barrier operations and divide each branch of the parallel statement into two phases of computation. The command also takes two conditions ϕ and ψ from the user such that both conditions cannot hold at the same state. Intuitively, ϕ holds during the first phase of the computation and ψ holds after threads finish executing their barrier operations. If barrier synchronization is not obeyed, then the following happens: if a thread t goes into the second phase before another thread, say u , finishes its first phase, since ϕ and ψ cannot be true at the same state, either t goes wrong by violating ψ or u goes wrong by violating ϕ .

The command **barrier** uses the proof rule EXTRACT-BARRIER to transform the given parallel statement to a sequential composition of three parallel statements. The new statement is written so that one can prove that the barrier idiom is implemented correctly, that is, all threads pass through **Barrier** together.

$$\begin{array}{c}
\text{EXTRACT-BARRIER} \\
\alpha = \phi \triangleright \langle \text{true} \rangle_\emptyset \quad \beta = \psi \triangleright \langle \text{true} \rangle_\emptyset \quad \models \mathcal{I} \Rightarrow \neg(\phi \wedge \psi) \\
\tilde{S}_1 = (S_1^1; B_1; S_1^2) \parallel \cdots \parallel (S_n^1; B_n; S_n^2) \\
\tilde{S}_2 = ((S_1^1; \alpha) \parallel \cdots \parallel (S_n^1; \alpha)) ; ((\alpha; B_1; \beta) \parallel \cdots \parallel (\alpha; B_n; \beta)) ; (S_1^2 \parallel \cdots \parallel S_n^2) \\
\hline
(P, \mathcal{I}) \dashrightarrow (P[\tilde{S}_1 \mapsto \tilde{S}_2], \mathcal{I})
\end{array}$$

18.4 Experience

In this section we give evidence about the usefulness of our techniques using programs from the literature. We applied our techniques to these programs mechanically or manually; for the former case we give pointers to the description of the full proofs in Chapter 19.

Multiset

In Section 19.2, we give the proof of a concurrent multiset of integers with **InsertPair** and **Delete** operations. Figure 19.4 shows the code for our multiset implementation in QEDPL. The implementation contains an array **M** of cells for storing the multiset elements; the **elt** field of the cell stores the element and the **vld** field indicates whether the value stored in **elt** is valid. Procedures **lock** and **unlock** acquire and release (**M[i].lck**), the lock of cell *i*. In addition to this version of multiset, we have also implemented another version of multiset using the readers/write lock and containing a **LookUp** operation (see Figure 16.1 for sample code for **LookUp**). In the readers/writer version, procedures **FindSlot** and **InsertPair** acquire the write lock, whereas **LookUp** acquires the read lock. For this version, we used the same lock implementation in Figure 18.2 except for using a separate (**readers**, **write**) pair for each cell **M[i]** in the multiset. All the annotations required for the blocks protected by locks were generated mechanically.

For the version using mutex locks, we used the template $\mathcal{S}_{\text{mutex}}(\text{M[i].lck} == \text{true})$ to reduce the code blocks between calls to **acq** and **rel** to atomic actions. As for the version using the readers/writer lock, we used the template $\mathcal{S}_{\text{rwlock}}((\text{M[i].readers} > 0 \ \&\& \ \text{M[i].writer} == \text{false}), (\text{M[i].writer} == \text{true}))$. As a result, 36 assertions were added to the actions between **acq** and **rel** (and their r/w lock versions). The loop in Figure 18.8 took 6 iterations for the mutex and 8 iterations for the r/w lock.

In both versions of multiset, we also used the template $\mathcal{S}_{\text{mutex}}(M[i].\text{elt} \neq \text{nil})$. This allowed us to capture the property that, once `FindSlot` returns an allocated slot, its `elt` and `vld` fields are not modified by other threads. Using this template we annotated the blocks in `InsertPair` containing the first and the second calls to `FindSlot` as acquires of the conceptual lock for `M[i]` and `M[j]` and the block setting the `vld` fields to `True` as the protected blocks. The annotations generated by the `mutex` and `rwlock` commands allowed us to transform `InsertPair` into a single atomic action. For more information about other details of the proof see Section 19.2.

Boxwood

The Boxwood system [116] makes use of both mutex and readers/writer locks. For example, the allocation server uses the `ReadersWriterLock` class for different purposes including preventing contention on a primary mutex lock and accessing its custom cache. The B-link tree uses the same class to prevent concurrent writes to its cache buffer. The implementation of the r/w lock in `ReadersWriterLock` is specified by the template

```
 $\mathcal{S}_{\text{rwlock}}(\text{this.numReaders} > 0 \ \&\& \ \text{this.hasWriter} == \text{false},$   
 $\text{this.numReaders} == 0 \ \&\& \ \text{this.hasWriter} == \text{true})$ 
```

where `this` refers to the lock object. We found that annotating the code using a r/w template with these formulas as parameters will help in showing that the code regions protected by `ReadersWriterLock` atomic.

Readers/writer Lock Implementations

Programs using existing implementations of the readers/writer locks idiom can be handled by our techniques. In addition to mutex locks, the pthreads library also provides a readers/writer lock implementation. The implementation uses the `num_active` and `owner` variables similar to our `readers` and `write` variables in Figure 18.2; a use of the lock in a program can be specified by the template $\mathcal{S}_{\text{rwlock}}((\text{num\_active} == 0 \ \&\& \ \text{owner} == 0), (\text{num\_active} > 0 \ \&\& \ \text{owner} == 1))$. The implementations of the readers/writer lock idiom in the ACE framework [117] and in [118], can be encoded similarly.

Philo

We found that in the philo benchmark from [15] the use of the mutual exclusion template allows us to prove the atomicity of the methods. The benchmark is written in Java and uses Java monitors. Each object has a separate monitor that is locked to synchronize accesses to the fields of the object. We model this using an array `objlock` of booleans indexed by the object reference. The template $\mathcal{S}_{\text{mutex}}((\text{objlock}[\text{o}]==\text{true}))$ specifies that a code block surrounded by `synchronized(o){...}` is protected by the lock of object `o` and can be annotated accordingly.

In the `philo.java` implementation, the synchronized `putForks` procedure is trivially proved to be atomic with the predicate `objlock==tid`. The justification for this predicate is that there is a single shared object from class `Table` and both of its methods are synchronized. The case for the other method, `getForks`, is more involved since the call to `wait()` present in the method is an implicit release of the object lock. Thus, the initial mutex annotation reduces the method to three atomic blocks, each block starts with acquiring the implicit object lock and ends with the release of the lock. This has the side benefit of making explicit the hidden concurrency and can aid the programmer to better understand the trade-offs implied.

Two-lock Queue

The concurrent queue in [119] uses two locks, one for the head and one for the tail of the queue to reduce contention on the lock. The operations (enqueue/dequeue) in this implementation can be proved atomic by specifying the two locks with two separate mutual exclusion policies. In the description of the data structure in [119] locks are modeled conceptually without giving an implementation. In this case, we model the acquire and release operations by associating the auxiliary variable `mutex` and the actions in `AcqMutex` and `RelMutex` in Section 18.2.5 with the corresponding conceptual lock operations. We give the proof of this queue in Section 19.4.

Java Grande Benchmark Suite

The `moldyn` and `raytracer` benchmarks from this suite use barriers to synchronize different phases of the computation. The `TournamentBarrier` implementation (in the `DoBarrier()`

method) in these benchmarks allows the phases of the computation be strictly ordered. Our transformation in Section 18.3 can be used to prove the correct operation of this implementation. During the proof, the predicates `IsDone[0] != donevalue` and `IsDone[0] != donevalue` (where `donevalue` is `true` or `false` depending on the round) specify the conditions that hold before and after threads pass through the barrier, respectively.

Device Drivers

Recall the device driver example in Section 11.4, which copies from the device registers to a cache and from the cache to a user buffer. The example has two uses of the mutual exclusion idiom. First, accesses to driver's fields were protected by a mutex lock. Second, it is ensured that at most one thread transfers data from the device registers to the cache by setting and checking a condition atomically. As a result of using the template that specifies the mutex locks, our annotation engine 15 assertions, after 4 iterations of the loop in Figure 18.8.

Object Initialization

Concurrency patterns in which objects are initialized and used in parallel can be implemented using events. A simple example is given in Section 18.2.7, where the variable `s` indicates whether the object has not been initialized (`NULL`) or being initialized (`INIT`) before the initialization and `FULL` after the initialization. The template $\mathcal{S}_{\text{event}}(s == \text{FULL})$ can be used to capture the synchronization for the object initialization in this case.

18.5 Related Work in Annotation Generation for Static Verification

For sequential programs, several key ideas have been instrumental in making automated reasoning about programs practical [38, 41]. Property checking on industrial-scale designs, for example, is viable if users provide annotations which allow tools to decompose the verification task. In practice, coming up with program annotations for this purpose is error-prone and difficult. This has led to research on automated approaches to generating annotations [120]. There is an analogy between the use of QED for generating annotations and the Houdini tool [120]. Procedure pre- and post-conditions are formulas that make the sequential verification modular, by specifying the effect of the procedure at call points without needing its body. The validity of the specification is separately checked for each procedure using

the body of the procedure. Houdini infers, from a set of candidate conditions, pre- and post-conditions of procedures. The inference algorithm ensures that it outputs the strongest valid formulas as post-conditions for each candidate pre-condition. The inferred pre- and post-conditions confirm the programmer's thinking about the procedure's operation. Similarly, the techniques presented here infer strong and valid annotations for proving atomicity of code blocks. The annotations confirm the programmer's expectations about the atomic computations in the program. For concurrent programs, there is little work along these lines. Roughly stated, one key goal of the work presented here is to provide a similar automated annotation capability for concurrent programs.

Static verification methods for concurrent programs, including the Owicki-Gries [42] method, the rely/guarantee [43] method make use of proof steps very similar to those in our approach while proving assertions and atomicity. These methods require that the programmer come up with the proof steps and provide program annotations in terms of auxiliary variables and synchronization states at which statements are executed. We provide automated assistance for coming up with the proof steps and annotations for programs that use common synchronization idioms. The idea of applying code transformations to facilitate static proofs of atomicity has been explored in the literature (e.g., [51].) Our work is distinguished by the fact that we phrase such transformations as sound rules in a proof calculus.

Our work is built on many of the same ideas reported in [48], [74] and [97]. These studies also automatically apply reduction and take as input hints or specifications of how access to program variables is regulated, e.g., by specifying the locks protecting accesses to particular data variables. Our techniques are more widely applicable, since our modeling of synchronization idioms allows us to uniformly handle a wider variety of idiom implementations. More importantly, our methods are more easily usable because we do not require the user to list the program data variables protected by a particular idiom. Given a particular idiom, the user simply provides a few predicates in terms of program synchronization variables, and we automatically infer which program variables and statements are protected by the synchronization mechanism.

We presented an approach to improving the practical usability of static verification tools

for checking atomicity of code blocks. We formalized various synchronization idioms such as mutual exclusion, readers/writer locks and events. We provided a proof template for each idiom for automating the handling of a use of the idiom in the program. While we concentrated on computing atomicity, the outcomes of this work are significant to support any formal verification method that needs to reason about non-interference and is guided by non-expert users.

Chapter 19

EXPERIENCE WITH QED

In this chapter, we present our experience with the following benchmark programs:

- **Benchmarks using locking protocols**

- Purity benchmarks [50]
- Multiset [121]
- Linked-list with hand-over-hand locking [87]
- Two-lock queue [119]
- Lock-based synchronous queue [122]

- **Benchmarks using non-blocking algorithms**

- Bakery [123]
- Obstruction-free deque [124]
- Treiber’s non-blocking stack [125]
- Non-Blocking stack with hazard pointers [126]
- Non-Blocking mutex-lock implementation adapted from [118]
- Non-Blocking queue [119]

We were able to prove the benchmarks using our tool QED-VERIFIER. All the proof steps driven by high-level tactics were fully mechanized; QED-VERIFIER required only a few minutes to finish each proof. Our experience with the benchmarks is summarized in the following conclusions:

- Our proofs did not require complicated global invariants that capture possible interference at each interleaving point. Instead, we attempted to achieve the correct level

of atomicity by reasoning locally about the effect of synchronization on individual actions. The representation invariants we used (usually at the end of the proof) were isolated from the synchronization facts. The iterative use of reduction and abstraction was crucial in this endeavor.

- The benchmarks, while operating with fine-grained concurrency, ensure a coarse level of atomicity through a variety of sophisticated synchronization protocols. We were able to capture these protocols with few uses of abstraction through the **assert**, **simulate**, **mutex** and **abstract read/write** tactics. For some cases, we needed few auxiliary variables to encode certain noninterference between atomic statements, using a small number of annotations with these variables.
- In many benchmarks, after aggressive use of reduction and abstraction, the atomic blocks obtained were large enough so that global program invariants stated for the transformed program were almost as simple as invariants for sequential programs.
- Our experience with non-blocking algorithms demonstrated that our method can also be used to verify highly-concurrent algorithms from the literature. Except for few benchmarks, simple read/write abstractions were sufficient to transform code with non-blocking operations to atomic blocks. The exceptional cases demonstrated that we precisely reason about nontrivial abstraction relations between statements.

Outline. In the remainder of this chapter, we discuss the proof of each benchmark at a high level. Sections 19.1–19.5 describe the proofs of benchmarks with locking protocols. Section 19.6–19.11 explain the proofs of benchmarks using non-blocking concurrency.

19.1 Purity Benchmarks

We verified the examples in [50], thereby demonstrating that our approach generalizes existing work on enforcing atomicity through abstractions. We were able to handle all the examples by simple applications of **abstract read/write** on variables that are accessed but left unmodified in the pure blocks. Consider the example in Figure 19.1 from Section 5 of [50].

```

var z : int;
var l : Mutex;

function ff(int) returns(int);
procedure {:isatomic} f(i:int) returns(r:int) {
  r := ff(i);
}

procedure apply_f() {
  var x:int;
  var fx:int;
  var guard:bool;

  call lock(l);
  x := z;                // abstract read
  call unlock(l);

  while(true) {
    call fx := f(x);

    call lock(l);
    guard := (x == z); // *
    if(guard) {
      z := fx;
      call unlock(l);
      break;
    }
    x := z;                // abstract read
    call unlock(l);
  }
}

```

Figure 19.1: Optimistic concurrency control using transaction retry

```

procedure apply_f() {
  var x: int;
  var fx: int;
  var guard: bool;

  atomic {
    z := ff(z);
    havoc guard, x, fx;
  }
}

```

Figure 19.2: Translated form of the purity example

```

1 inline all
2 reduce all
3 simulate Call_2 havoc x;
4 peelout Call_5
5 simulate Call_5 havoc guard, x, fx;
6 simulate Call_5 dup havoc guard, x, fx; z := ff(z);
7 reduce loop Call_5 havoc guard, x, fx;
8 reduce all
9 simulate Call_2 z := ff(z); havoc guard, x, fx;

```

Figure 19.3: Proof script of the purity example

This program reads a shared variable **z** in one critical section, performs a long computation on its value (represented by the **f** procedure and the uninterpreted **ff** function), and then attempts to writes back the result into **z** in another critical section. Figure 19.3 shows the proof script we used for this example.

A reduction step allows the code blocks between **lock(1)** and **unlock(1)** (both branches of the **if** statement) to be proved atomic. The reads of **z** at lines marked **// abstract read** are unnecessary for correctness and could therefore be abstracted. The correct operation of the procedure **apply_f** only depends on the read of **z** at line marked **// ***.

Notice that the loop in this example and other examples in this section contain the **break** statement. Instead of rewriting the programs to eliminate **break**, we provide the user the **peelout** tactic, which hoists the final, successful iteration out of the loop and leaves inside the loop only failing iterations. In this particular example, we hoist the successful **if** branch out of the loop. This is a sound operation since every terminating execution of the loop contains these lines once at the end. We apply similar transformations to the loops in other examples given below. The read abstractions are applied to these failing iterations in the resulting body of the loop, and the loop is reduced, using the **reduce loop** command, to an action to one that havocs variables modified within the loop body.

The rest of the loop after the above transformation does not touch global variables, and also implements a simple “skip” operation. Thus this portion of the program is a both-mover. Finally we apply reduction and convert the body of **apply_f** into a single atomic action that assigns **f(z)** to **z**. Figure 19.2 shows the body of **apply_f** after the transformations explained above. The new body exhibits the atomic application of the **f** function on **z**.

19.2 Multiset

```

type ref;

const unique null : ref;

record Cell {
  elt: ref;
  vld: boolean;
  lck: Mutex;
}

var M: [int]Cell;
invariant (forall j:int ::
  0 <= j && j <= N ==> IsAlloc(M[j].alloc));
const N : int;
axiom 0 <= N;

procedure FindSlot(x : ref)
returns (r : int) {
  var i : int;

  i := 0;

  while(i < N) {
    call lock(M[i].lck);
    if(M[i].elt == null) {
      M[i].elt := x;
      call unlock(M[i].lck);
      r := i;
      return;
    }
    call unlock(M[i].lck);
    i := i + 1;
  }
  r := -1;
  return;
}

procedure Insert(x : ref)
returns (r : bool) {
  var i : int;
  call i := FindSlot(x);

  if(i == -1)
  {
    r := false;
    return;
  }

  call lock(M[i].lck);
  assert M[i].elt == x;
  assert M[i].vld == False;
  M[i].vld := True;
  call unlock(M[i].lck);
  r := true;
}

procedure InsertPair(x : ref, y : ref)
returns (r : bool) {
  var i : int, j : int;
  call i := FindSlot(x);

  if(i == -1) {
    r := false;
    return;
  }

  call j := FindSlot(y);

  call lock(M[i].lck);
  call lock(M[j].lck);
  assert M[i].elt == x;
  assert M[j].elt == y;
  assert M[i].vld == False;
  assert M[j].vld == False;
  M[i].vld := True;
  M[j].vld := True;
  call unlock(M[i].lck);
  call unlock(M[j].lck);
  r := true;
}

procedure Delete(x : ref)
returns (r : bool) {
  var i : int;
  i := 0;

  while(i < N) {
    call lock(M[i].lck);
    if(M[i].elt == x && M[i].vld == True) {
      M[i].elt := null;
      M[i].vld := False;
      call unlock(M[i].lck);
      r := true;
      return;
    }
    call unlock(M[i].lck);
    i := i + 1;
  }
  r := false;
}

```

Figure 19.4: Implementation of the multiset example in QEDPL

```

Insert(x:int)
returns r:bool {
  var i : int;
  atomic {
    havoc i;
    if (*) {
      r := false;
    } else {
      assume 0 <= i && i < N;
      assume M[i].elt == null;
      assume M[i].vld == False;
      M[i].elt := x;
      M[i].vld := True;
      r := true;
    }
  }
}

Delete(x:int)
returns r:bool {
  var i : int;
  atomic {
    havoc i;
    if (*) {
      r := false;
    } else {
      assume 0 <= i && i < N;
      assume M[i].elt == x;
      assume M[i].vld == True;
      M[i].elt := null;
      M[i].vld := False;
      r := true;
    }
  }
}

InsertPair(x:int, y:int)
returns r:bool {
  var i,j : int;
  atomic {
    havoc i, j;
    if (*) {
      r := false;
    } else {
      assume 0 <= i && i < N;
      assume 0 <= j && j < N;
      assume M[i].elt == null;
      assume M[i].vld == False;
      M[i].elt := x;
      M[i].vld := True;
      assume M[j].elt == null;
      assume M[j].vld == False;
      M[j].elt := y;
      M[j].vld := True;
      r := true;
    }
  }
}

```

Figure 19.5: The atomic multiset after the QED transformations.

```

1  inline all
2  reduce all
3  peelout Call_2
4  simulate Call_2 havoc guard_0; assume i < N; i := i + 1;
5  simulate Atomic_1 havoc i; assume 0 <= i && i <= N; havoc guard_0;
6  reduce loop2 Atomic_1
7  reduce proc FindSlot
8  simulate Atomic_1 havoc i,guard_0; havoc r; if(*) { r := -1; } else { assume (0 <= r
    && r < N && M[r].elt == null); M[r].elt := x; }
9  inline FindSlot
10 assert Call_12 x != null
11 assert Atomic_56 x != null
12 assert Call_21 x != null
13 assert Call_21 y != null
14 assert Call_24 x != null
15 assert Call_24 y != null
16 assert Atomic_58 x != null
17 assert Atomic_58 y != null
18 assert Atomic_58 M[i].elt == x && M[i].vld == False
19 invariant (forall j: int :: 0 <= j && j < N && M[j].elt == null ==> M[j].vld == False
    )
20 reduce all
21 check all block

```



```

22 peelout Call_42
23 simulate Call_42 assume i < N; i := i + 1; havoc guard_1;
24 simulate Atomic_41 havoc i; assume 0 <= i && i <= N; havoc guard_1;
25 reduce loop2 Atomic_41
26 reduce proc Delete
27 simulate Call_12 assert x != null; havoc i, fresh_0, fresh_1; if (*) { r := false; }
    else { assume 0 <= i && i < N; assume M[i].elt == null; assume M[i].vld == False;
    M[i].elt := x; M[i].vld := True; r := true; }
28 simulate Call_21 assert x != null && y != null; havoc i, j, fresh_0, fresh_1, fresh_2,
    fresh_3; if (*) { r := false; } else { assume 0 <= i && i < N; assume 0 <= j &&
    j < N; assume M[i].elt == null; assume M[i].vld == False; M[i].elt := x; M[i].
    vld := True; assume M[j].elt == null; assume M[j].vld == False; M[j].elt := y; M[
    j].vld := True; r := true; }
29 simulate Atomic_41 havoc i, guard_1; if(*) { r := false; } else { assume 0 <= i && i
    < N; assume M[i].elt == x; assume M[i].vld == True; M[i].elt:=null; M[i].vld:=
    False; r := true; }

```

Figure 19.6: The proof script used for the multiset example in QEDPL

Figure 19.4 shows a concurrent multiset of integers with **Insert**, **InsertPair** and **Delete** operations. The implementation contains an array **M** of cells for storing the multiset elements; the **elt** field of the cell stores the element and the **vld** field indicates whether the value stored in **elt** is valid. Procedures **lock** and **unlock** acquire and release (**M[i].lck**), the lock of cell **i**. In [121], we proved that **InsertPair** and **Delete** are atomic using an abstraction map from **M** to an abstract specification variable **S** representing the multiset contents. The variable **S** was required to abstract away from the concrete values of the indices of **M** into which multiset elements are stored. Using our method, we transformed the bodies of the procedures in the implementation to atomic actions given in Figure 19.5. Thus, we replaced a proof based on abstraction mappings with a simpler, layered correctness proof that ends with sequential reasoning. Figure 19.6 shows the proof script we used in this proof.

We first proved the following specification for **FindSlot** and inlined the specification at call points in **InsertPair**. This proof required reasoning similar to the example in Figure 19.1: using the **peelout** command, we hoisted the last iteration of the loop in **FindSlot**, which allocates an empty slot, outside and performed abstractions on the other iterations. In the non-final iterations of the loop, the abstractions were done on the **elt** and **vld** fields of **M[i]**. As a result of the transformations, we obtained the following atomic body of **FindSlot**.

The above body of **FindSlot** allowed us to capture the property that, once **FindSlot** returns an allocated slot, its **elt** and **vld** fields are not modified by other threads. This

```

procedure FindSlot(x:int)
returns (r:int) {
  atomic {
    if(*) {
      r := -1;
    } else {
      havoc r;
      assume (0 <= r < N && M[r].elt == nil && !(M[i].vld));
      M[i].elt := x;
    }
  }
}

```

fact is essential for proving that the atomic actions spanning the code blocks in `Insert` and `InsertPair` that use `FindSlot` to allocate cells are right movers. We also introduced the invariant (`forall j: int :: 0 <= j && j < N && M[j].elt == null ==> M[j].vld == False`) at the very end of the proof. Then we were able to merge, using the `reduce` command, all the three atomic blocks in `Insert` and `InsertPair` into a single atomic action. It was crucial to introduce the invariant above before the reduction step for the success of this step. Proving this invariant enforced us to add few assertions to the code (as the individual actions in `InsertPair` did not preserve this invariant); these assertions were discharged right after the reduction step. The correctness proof for a sequential multiset implementation would have required the same simple invariant.

19.3 *Linked-list with Hand-over-hand Locking*

Figure 19.7 shows a linked-list based set implementation (in QEDPL) using a hand-over-hand locking protocol [87]. The procedures `Insert` and `Delete` are used to add and remove an element to/from the set. The linked list is kept sorted to make the look-up operations faster on average. Both procedures use the helper procedure `Find` to find the right place in the linked-list to add or remove the element. `Find` traverses the linked-list using the hand-over-hand locking protocol. In this protocol, every node has a separate lock, and the lock of the current (visited) node is not released before the lock of the next node is acquired.

This ensures that threads process the linked-list in the order they acquire the lock of the head of the list. In other words, a thread cannot take over another thread that has locked a node before itself.

```

type Data = int;
record Node {
  next: Node;
  val: Data;
  lock: boolean;
}

const unique null : Node;
invariant IsNull(null.alloc);

record List { head: Node; }
invariant
  (forall list:List :: {List_head[list]}
    list.head != null);

procedure Find(list : List, x : Data)
returns (node: Node) {
  var p:Node;
  var n:Node;
  var guard: bool;

  p := list.head;
  call Acq(p);

  n := p.next;

  guard := (n != null && n.val < x);
  while(guard) {
    call Acq(n);
    call Rel(p);
    p := n;
    n := p.next;
    guard := (n != null && n.val < x);
  }

  node := p;
}

procedure Insert(list : List, x : Data)
returns (wasPresent: bool) {
  var p:Node;
  var n:Node;
  var t:Node;

  call p := Find(list, x);

  n := p.next;
  if(n == null || n.val > x) {
    t := New Node;
    t.val := x;
    t.next := n;
    p.next := t;
    wasPresent := false;
  } else {
    wasPresent := true;
  }

  call Rel(p);
}

procedure Delete(list : List, x : Data)
returns (wasPresent: bool) {
  var p:Node;
  var n:Node;

  call p := Find(list, x);

  n := p.next;
  if(n != null || n.val == x) {
    call Acq(n);
    p.next := n.next;
    call Rel(n);
    wasPresent := true;
  } else {
    wasPresent := false;
  }

  call Rel(p);
  Free n;
}

```

Figure 19.7: Linked-list based set implementation with hand-over-hand locking.

```

1 inline all
2 mutexptr aux (forall x: Node :: x.lock == True)
3 assert Atomic_5@Find IsAlloc(p.alloc) && aux[p] == tid
4 assert Atomic_6@Find n == null || IsAlloc(n.alloc)
5 assert Atomic_10@Find IsAlloc(p.alloc) && aux[p] == tid
6 assert Atomic_11@Find n == null || IsAlloc(n.alloc)
7 assert Atomic_14@Insert IsAlloc(p.alloc) && aux[p] == tid
8 assert Atomic_15@Insert n == null || IsAlloc(n.alloc)

```

```

9  assert Atomic_19@Insert IsAlloc(p.alloc) && aux[p] == tid
10 reduce all
11 invariant (forall list: List, node: Node :: ReachBetween(Node_next, list.head, node,
    null) && node != null ==> IsAlloc(node.alloc)) && (forall list: List ::
    ReachBetween(Node_next, list.head, null, null))
12 simulate Atomic_3@Find havoc p; assume ReachBetween(Node_next, list.head, p, null) &&
    p != null; assume (forall k: Node :: (ReachBetween(Node_next, list.head.next, k,
    null) && (!ReachBetween(Node_next, p.next, k, null)) && (k != null)) ==> (k.val <
    x)); assume p.lock == False; p.lock := True; aux[p] := tid; n := p.next; assert
    n == null || IsAlloc(n.alloc); guard := n != null && n.val < x;
13 reduce loop2 Atomic_3@Find
14 reduce all
15 simulate Atomic_3@Find havoc p; assume ReachBetween(Node_next, list.head, p, null) &&
    p != null; assume (forall k: Node :: ReachBetween(Node_next, list.head.next, k,
    null) && !ReachBetween(Node_next, p.next, k, null) && k != null ==> k.val < x);
    assume p.lock == False; p.lock := True; aux[p] := tid; n := p.next; assert n ==
    null || IsAlloc(n.alloc); guard := n != null && n.val < x; assume !guard; node :=
    p;
16 inline all
17 mover proc Insert
18 merge proc Insert
19 check Insert block
20 invariant (forall list: List, e: Node, f: Node :: ReachBetween(Node_next, list.head.
    next, e, null) && ReachBetween(Node_next, e, f, null) && e != null && f != null
    && e != f ==> e.val < f.val)
21 simulate Call_13@Insert if ((forall e: Node :: ReachBetween(Node_next, list.head.next,
    e, null) && e != null ==> e.val != x)) { havoc p; assume ReachBetween(Node_next,
    list.head, p, null) && p != null; assume (p == list.head || p.val < x) && (p.
    next == null || x < p.next.val); t := New Node; t.val := x; t.next := p.next; p.
    next := t; wasPresent := false; } else { wasPresent := true; } havoc fresh_0,
    fresh_1, fresh_2, guard_0;
22 simulate Call_13@Insert havoc fresh_0; assume ReachBetween(Node_next, list.head,
    fresh_0, null) && fresh_0 != null; assume (forall k: Node :: ReachBetween(
    Node_next, list.head.next, k, null) && !ReachBetween(Node_next, fresh_0.next, k,
    null) && k != null ==> k.val < x); fresh_1 := fresh_0.next; fresh_2 := fresh_1 !=
    null && fresh_1.val < x; assume !fresh_2; p := fresh_0; n := p.next; guard_0 :=
    n == null || n.val > x; if (guard_0) { t := New Node; t.val := x; t.next := n; p.
    next := t; wasPresent := false; } else { wasPresent := true; }
23 simulate Call_13@Insert havoc p; assume ReachBetween(Node_next, list.head, p, null)
    && p != null; assume (forall k: Node :: ReachBetween(Node_next, list.head.next, k,
    null) && !ReachBetween(Node_next, p.next, k, null) && k != null ==> k.val < x);
    n := p.next; assume !(n != null && n.val < x); if (n == null || n.val > x) { t :=
    New Node; t.val := x; t.next := n; p.next := t; wasPresent := false; } else {
    wasPresent := true; } havoc fresh_0, fresh_1, fresh_2, guard_0;
24 inline all
25 mutexptr aux (forall x: Node :: x.lock == True)
26 assert Atomic_5@Find IsAlloc(p.alloc) && aux[p] == tid
27 assert Atomic_6@Find n == null || IsAlloc(n.alloc)
28 assert Atomic_10@Find IsAlloc(p.alloc) && aux[p] == tid
29 assert Atomic_11@Find n == null || IsAlloc(n.alloc)
30 assert Atomic_14@Insert IsAlloc(p.alloc) && aux[p] == tid
31 assert Atomic_15@Insert n == null || IsAlloc(n.alloc)
32 assert Atomic_19@Insert IsAlloc(p.alloc) && aux[p] == tid
33 reduce all
34 invariant (forall list: List, node: Node :: ReachBetween(Node_next, list.head, node,
    null) && node != null ==> IsAlloc(node.alloc)) && (forall list: List ::
    ReachBetween(Node_next, list.head, null, null))
35 simulate Atomic_3@Find havoc p; assume ReachBetween(Node_next, list.head, p, null) &&
    p != null; assume (forall k: Node :: (ReachBetween(Node_next, list.head.next, k,
    null) && (!ReachBetween(Node_next, p.next, k, null)) && (k != null)) ==> (k.val <
    x)); assume p.lock == False; p.lock := True; aux[p] := tid; n := p.next; assert
    n == null || IsAlloc(n.alloc); guard := n != null && n.val < x;
36 reduce loop2 Atomic_3@Find
37 reduce all
38 simulate Atomic_3@Find havoc p; assume ReachBetween(Node_next, list.head, p, null) &&
    p != null; assume (forall k: Node :: ReachBetween(Node_next, list.head.next, k,

```

```

    null) && !ReachBetween(Node_next, p.next, k, null) && k != null ==> k.val < x);
    assume p.lock == False; p.lock := True; aux[p] := tid; n := p.next; assert n ==
    null || IsAlloc(n.alloc); guard := n != null && n.val < x; assume !guard; node :=
    p;
39 inline all
40 mover proc Insert
41 merge proc Insert
42 check Insert block
43 simulate Call_13 havoc fresh_0; assume ReachBetween(Node_next, list.head, fresh_0,
    null) && fresh_0 != null; assume (forall k: Node :: ReachBetween(Node_next, list.
    head.next, k, null) && !ReachBetween(Node_next, fresh_0.next, k, null) && k !=
    null ==> k.val < x); fresh_1 := fresh_0.next; fresh_2 := fresh_1 != null &&
    fresh_1.val < x; assume !fresh_2; p := fresh_0; n := p.next; guard_0 := n == null
    || n.val > x; if (guard_0) { t := New Node; t.val := x; t.next := n; p.next := t
    ; wasPresent := false; } else { wasPresent := true; }
44 simulate Call_13 havoc p; assume ReachBetween(Node_next, list.head, p, null) && p !=
    null; assume (forall k: Node :: ReachBetween(Node_next, list.head.next, k, null)
    && !ReachBetween(Node_next, p.next, k, null) && k != null ==> k.val < x); n := p.
    next; assume !(n != null && n.val < x); if (n == null || n.val > x) { t := New
    Node; t.val := x; t.next := n; p.next := t; wasPresent := false; } else {
    wasPresent := true; } havoc fresh_0, fresh_1, fresh_2, guard_0;
45 invariant (forall list: List, e: Node, f: Node :: ReachBetween(Node_next, list.head.
    next, e, null) && ReachBetween(Node_next, e, f, null) && e != null && f != null
    && e != f ==> e.val < f.val)

```

Figure 19.8: The proof script used for the linked-list based set implementation.

We proved the linearizability of this implementation by transforming this program to a QEDPL program in which the bodies of **Insert** and **Delete** are atomic and perform set addition and removal operations. Calcagno et al. provided a proof of this benchmark using a combination of separation logic and rely/guarantee approach [88]. Although the proof is automated, the proof requires a symbolic execution producing a complicated proof outline propagating facts about the data and locking state throughout the entire code of the operations; in addition, the general applicability of their automation technique to other data structures is an open question. Our technique also proves the atomicity of lock-protected code blocks automatically, without needing the use of separation logic, and using a simple abstraction pattern we were able to show the atomicity of **Insert** and **Delete** operations.

Figure 19.8 shows the proof script used for this benchmark. We were able to capture the atomic blocks protected by the locks using the `mutexptr` command, which is an extended form of `mutex` for cases where each record use a separate lock.

The challenge in this proof is not only prove that the lock protected blocks are atomic, but also to show that (1) the entire body of **Find** is atomic and (2) the sequential composition of **Find** and the final operations of **Insert** and **Delete** are atomic. By using abstractions

shown in the proof script and the `reduce loop2` command, we have transformed the body of `Find` to the following:

```
havoc p;
assume ReachBetween(Node_next, list.head, p, null) && p != null;
assume (forall k: Node ::
    ReachBetween(Node_next, list.head.next, k, null) &&
    !ReachBetween(Node_next, p.next, k, null) && k != null ==> k.val < x);
assume p.lock == False;
p.lock := True;
aux[p] := tid;
n := p.next;
assert n == null || IsAlloc(n.alloc);
guard := n != null && n.val < x;
assume !guard;
node := p;
```

The above code clearly encodes that `Find` returns the first node in the list (`node`) whose value (`val`) is greater than or equal to a given key `k`. In addition, `node` is locked so that no other thread can take over the operation of the current thread. When `Find` returns, the procedures `Insert` and `Delete` also acquire the lock of the next node (`n`), if exists, and perform their operation. In addition, the code above is proved to be a right mover. Thus, it can be combined with the atomic blocks in `Insert` and `Delete` after the call to `Find` to form the atomic bodies of these procedures.

19.4 Two-lock Queue

This data structure implements a queue whose head and tail are protected by different locks [119]. Therefore the enqueue and dequeue operations acquire distinct locks. We proved the linearizability of the implementation given in Figure 19.9 to the atomic specification given in Figure 19.11 by transforming the former to the latter using the proof script given in Figure 19.10.

```

type Object;

record QueueItem
{
  next: QueueItem;
  value: Object;
}

const unique NULL_OBJECT: Object;
const unique NULL_QUEUE_ITEM: QueueItem;

var head: QueueItem;
var tail: QueueItem;
var headLock: Mutex;
var tailLock: Mutex;

procedure enqueue(value: Object) {
  var node: QueueItem;
  var t: QueueItem;

  call node := NewQueueItem(value);

  call lock(tailLock);
  t := tail;
  t.next := node;
  tail := node;
  call unlock(tailLock);
}

procedure {:ispublic false} NewQueueItem(val: Object)
returns (this: QueueItem)
{
  this := New QueueItem;
  this.value := val;
  this.next := NULL_QUEUE_ITEM;
}

procedure dequeue() returns (result: Object) {
  var x: Object;
  var newHead: QueueItem;
  var node: QueueItem;

  call lock(headLock);
  node := head;
  newHead := node.next;
  if (newHead != NULL_QUEUE_ITEM)
  {
    x := newHead.value;
    head := newHead;
  } else {
    x := NULL_OBJECT;
  }
  call unlock(headLock);

  result := x;
}

```

Figure 19.9: The two-lock queue implementation in QEDPL

```

1  inline all
2  nullcheck
3  reduce all
4  inline NewQueueItem
5  reduce all
6  invariant IsNull(NULL_QUEUE_ITEM.alloc) && NULL_QUEUE_ITEM.next == NULL_QUEUE_ITEM
7  invariant ReachBetween(QueueItem_next, head, tail, NULL_QUEUE_ITEM) && tail.next ==
   NULL_QUEUE_ITEM && IsAlloc(head.alloc) && IsAlloc(tail.alloc) && IsAlloc(headLock
   .alloc) && IsAlloc(tailLock.alloc)
8  invariant (forall node: QueueItem :: ReachBetween(QueueItem_next, head, node, tail)
   ==> IsAlloc(node.alloc))
9  check all block
10 simulate Call_4 node := New QueueItem; node.value := value; node.next :=
   NULL_QUEUE_ITEM; t := tail; t.next := node; tail := node;
11 simulate Call_10 node := head; newHead := node.next; if (newHead != NULL_QUEUE_ITEM)
   { x := newHead.value; head := newHead; } else { x := NULL_OBJECT; } result := x;

```

Figure 19.10: The proof script for the two-lock queue

```

type Object;

record QueueItem {
  next: QueueItem;
  value: Object;
}

const unique NULL_OBJECT: Object;
const unique NULL_QUEUE_ITEM: QueueItem;
invariant IsNull(NULL_QUEUE_ITEM.alloc);

var head: QueueItem;
var tail: QueueItem;
var headLock: Mutex;
var tailLock: Mutex;

invariant
  IsNull(NULL_QUEUE_ITEM.alloc)
  && NULL_QUEUE_ITEM.next == NULL_QUEUE_ITEM;

invariant
  ReachBetween(QueueItem.next, head, tail, NULL_QUEUE_ITEM)
  && tail.next == NULL_QUEUE_ITEM
  && IsAlloc(head.alloc) && IsAlloc(tail.alloc)
  && IsAlloc(headLock.alloc) && IsAlloc(tailLock.alloc);

invariant
  (forall node: QueueItem ::
    ReachBetween(QueueItem.next, head, node, tail)
    ==> IsAlloc(node.alloc));

procedure enqueue(value: Object) {
  var node: QueueItem;
  var t: QueueItem;

  atomic {
    node := New QueueItem;
    node.value := value;
    node.next := NULL_QUEUE_ITEM;
    t := tail;
    t.next := node;
    tail := node;
  }
}

procedure dequeue()
returns (result: Object) {
  var x: Object;
  var newHead: QueueItem;
  var node: QueueItem;

  atomic {
    node := head;
    newHead := node.next;
    if (newHead != NULL_QUEUE_ITEM) {
      x := newHead.value;
      head := newHead;
    } else {
      x := NULL_OBJECT;
    }

    result := x;
  }
}

```

Figure 19.11: The final version of the two-lock queue after the QED transformations.

For this example, few proof commands were sufficient to prove the linearizability of the implementation. We first applied a reduction step to make the lock-protected blocks atomic. Then, the representation invariants were introduced. Note that the prior reduction steps enabled us to use a clean invariant that does not refer to locks protecting to the head and tail (except for the invariants declaring that the locks are allocated). Without the reduction step, the invariants would need to use the locking information to model the program states reached through fine-grained actions of the original implementation. After the reduction step, the invariant only needs to describe the program states at the beginning and the end of the enqueue and dequeue operations. The last two `simulate` commands were used to simplify the code to obtain the program in Figure 19.11. The proof of this example demonstrates how reduction helps us to simplify the proof of programs using simple locking disciplines to few number of commands.

19.5 Lock-based Synchronous Queue

```

type ref;
const unique null : ref;

record Monitor{
  held : boolean;
}

record NaiveSQ {
  putting : boolean;
  item : ref;
  monitor : Monitor;
}

procedure {:isatomic} acquire(monitor:Monitor) {
  assume monitor.held == False;
  monitor.held := True;
}

procedure {:isatomic} release(monitor:Monitor) {
  assert monitor.held == True;
  monitor.held := False;
}

procedure take(q:NaiveSQ) returns (e:ref) {
  var itm:ref;

  while(*) {
    call acquire(q.monitor);
    itm := q.item;
    assume (itm == null);
    call release(q.monitor);
  }

  call acquire(q.monitor);

  itm := q.item;
  assume (itm != null);

  e := itm;
  q.item := null;

  call release(q.monitor);
}

procedure put(q:NaiveSQ, e:ref) {
  var itm:ref;
  var ptng:boolean;

  if(e == null) {
    return;
  }

  while(*) {
    call acquire(q.monitor);
    ptng := q.putting;
    assume (ptng == True);
    call release(q.monitor);
  }

  call acquire(q.monitor);

  ptng := q.putting;
  assume (ptng == False);

  q.putting := True;
  q.item := e;

  call release(q.monitor);

  while(*) {
    call acquire(q.monitor);
    itm := q.item;
    assume (itm != null);
    call release(q.monitor);
  }

  call acquire(q.monitor);

  itm := q.item;
  assume (itm == null);

  q.putting := False;

  call release(q.monitor);
}

```

Figure 19.12: The lock-based synchronous queue implementation in QEDPL

```

procedure take(q: NaiveSQ) returns (e: ref) {
  var itm: ref;

  atomic {
    itm := q.item;
    assume itm != null;
    e := itm;
    q.item := null;
  }
}

procedure put(q: NaiveSQ, e: ref) {
  var itm: ref;
  var ptng: boolean;

  atomic {
    if (e == null) {
      return;
    }
    ptng := q.putting;
    assume ptng == False;
    q.putting := True;
    q.item := e;
  }

  atomic {
    itm := q.item;
    assume itm == null;
    q.putting := False;
  }
}

```

Figure 19.13: The final version of the lock-based synchronous queue.

```

1  inline all
2  reduce all
3  abstract read NaiveSQ_item Call_3@take
4  abstract read NaiveSQ_item Call_25@put
5  abstract read NaiveSQ_putting Call_15@put
6  reduce all
7  simulate Call_3@take havoc abst_NaiveSQ_item_0; itm := abst_NaiveSQ_item_0[q];
8  simulate Call_25@put havoc abst_NaiveSQ_item_1; itm := abst_NaiveSQ_item_1[q];
9  simulate Call_15@put havoc abst_NaiveSQ_putting_2; ptng := abst_NaiveSQ_putting_2[q];
10 reduce all
11 reduce loop Call_3@take havoc abst_NaiveSQ_item_0; havoc itm;
12 reduce loop Call_25@put havoc abst_NaiveSQ_item_1; havoc itm;
13 reduce loop Call_15@put havoc abst_NaiveSQ_putting_2; havoc ptng;
14 reduce all
15 simulate Atomic_42@take havoc abst_NaiveSQ_item_0; itm := q.item; assume itm != null;
    e := itm; q.item := null; return;
16 simulate Atomic_38@put if (e == null) { return; } havoc abst_NaiveSQ_putting_2,
    abst_NaiveSQ_item_1; ptng := q.putting; assume ptng == False; q.putting := True;
    q.item := e; havoc itm;
17 simulate Call_29@put itm := q.item; assume itm == null; q.putting := False; return;

```

Figure 19.14: The proof script for the lock-based synchronous queue

Our implementation of the lock-based synchronous queue [122] in QEDPL is given in Figure 19.12. The synchronous queue models a buffer-less queue providing the `put` and `take` operations. Each thread that performs a `put` on the queue waits until another thread performs a `take` on the same queue. Then the threads exchange their elements (a non-null

element by **put** and a null element by **take**). This models a rendezvous communication between two threads.

The proof script for this benchmark is shown in Figure 19.14. The nondeterministic while loops in **take** and **put** represent the waiting of the thread for another thread to set the **item** or **ptng** variables. These loops have no impact on the correctness of the program, so we eliminated the loops using simple read abstractions within their bodies and replaced with **havoc** statements.

Figure 19.13 shows the final version of the program. Notice that while the procedure **take** is reduced to a single atomic statement, the procedure **put** consists of two atomic blocks. This is because, between the atomic blocks of **put** another thread has to execute **take** and receive the item given by **put**. We could do one more abstraction in order to obtain an atomic body of **put**. However, in this case, the final version of the program would not implement rendezvous communication correctly.

19.6 The Bakery Algorithm

```

var choosing : [int] bool;
var number : [int] int;
var counter: int;

procedure bakery() {
  var j,k : int;
  var max : int;

  choosing[tid] := true;

  atomic{
    havoc max;
    assume (forall t:TID :: 1 <= t && t <= Tid ==> max > number[t]);
    number[tid] := 1 + max;
  }

  choosing[tid] := false;

  j := 1;
  while(j <= Tid) {
    if(j != tid) {
      assume choosing[j] == false;
      assume number[j] == 0 ||
        (number[j] > number[tid]) ||
        ((number[j] == number[tid]) && (j > tid));
    }
    j := j + 1;
  }

  // begin critical region
  counter := counter + 1;
  assert counter == 1;
  counter := counter - 1;
  // end critical region
}
```

```

// release the lock
number[tid] := 0;
}

```

Figure 19.15: The implementation of the bakery algorithm in QEDPL

```

1  reduce all
2  invariant (forall t: TID :: 1 <= t && t <= Tid ==> number[t] >= 0)
3  invariant (forall t: TID :: 1 <= t && t <= Tid ==> choosing[t] == false)
4  simulate Atomic_16 havoc j; assume 1 <= j && j <= Tid+1; assume (forall t:TID :: 1<=
    t && t < j && t != tid ==> number[t] == 0 || number[t] > number[tid] || (number[t
    ] == number[tid] && t > tid));
5  peelout Atomic_15
6  reduce all
7  simulate Atomic_15 assume j <= Tid; if (j != tid) { assume choosing[j] <==> false;
    assume number[j] == 0 || number[j] > number[tid] || (number[j] == number[tid] &&
    j > tid); } j := j + 1;
8  simulate Atomic_21 assume j > Tid; counter := counter + 1;
9  reduce loop2 Atomic_16
10 reduce all
11 assert Atomic_10 counter == 1
12 simulate Atomic_16 assert number[tid] > 0; havoc j; assume (forall t: TID :: 1 <= t
    && t < Tid && t != tid ==> number[t] == 0 || number[t] > number[tid] || (number[t
    ] == number[tid] && t > tid)); assert counter == 0; counter := counter + 1;
13 aux begin owner int
14 simulate Atomic_10 assert counter == 1; counter := counter - 1; number[tid] := 0;
    owner := 0;
15 aux end owner
16 assert Atomic_9 owner == tid
17 assert Atomic_10 owner == tid
18 assert Atomic_1 owner != tid
19 assert Atomic_1 number[tid] == 0
20 invariant 0 <= owner && owner <= Tid
21 invariant owner != 0 <==> number[owner] > 0 && (forall t: TID :: { number[t] } 1 <= t
    && t < Tid && t != owner ==> number[t] == 0 || number[t] > number[owner] || (
    number[t] == number[owner] && t > owner))
22 invariant owner != 0 ==> number[owner] > 0 && (forall t: TID :: { number[t] } 1 <= t
    && t < Tid && t != owner ==> number[t] == 0 || number[t] > number[owner] || (
    number[t] == number[owner] && t > owner))
23 reduce all
24 invariant counter == 0
25 check all block

```

Figure 19.16: The proof script for the bakery algorithm

The bakery algorithm (Figure 19.15) provides non-blocking critical sections [123]. We encoded the mutual exclusion property by adding a global variable `counter` that counts the number of threads in the critical section and adding the assertion `counter == 1` in the critical section. We proved the property by obtaining, using the proof script given in

Figure 19.16, an action that spans the code including and after the loop. This action is enabled only when `(number[i],i)` is less than `(number[k],k)` for all `k` different from `i`. Surprisingly, the operation that checks this condition was shown to be a right mover. We reduced the loop using the `reduce loop2` command. When reducing the loop, the body of the loop was the following:

```
assume j <= Tid;
if (j != tid) {
  assume choosing[j] <=> false;
  assume number[j] == 0
    || number[j] > number[tid] || (number[j] == number[tid] && j > tid);
}
j := j + 1;
```

and the action given as the specification of the loop was the following:

```
havoc j;
assume 1 <= j && j <= Tid+1;
assume (forall t:TID :: 1<= t && t < j && t != tid ==> number[t] == 0
    || number[t] > number[tid] || (number[t] == number[tid] && t > tid));
```

The specification given above clearly abstracts zero or more iterations of the loop whose body is also given above. After eliminating the loop, we added an auxiliary variable that encodes the owner of the conceptual lock modeled by the Bakery algorithm. This conceptual lock protects `c` ounter. The action replacing the loop models the acquire operation of this conceptual lock, and the line `number[tid] := 0` models the release operation of the lock. Note that these steps could also be automated using a `mutex` command.

19.7 Obstruction-free Deque

The double-ended queue (deque) from [124] provides four operations—`rightpop`, `rightpush`, `leftpop`, and `leftpush`—with similar designs, to insert/remove elements to/from both ends of the queue. Figure 19.17 shows the `rightpush` operation, which inserts an element into the right end of the deque. After reading two consecutive array elements in lines 3-4 to local variables `prev` and `cur`, `rightpush` inserts the given value `v` if CAS operations at lines 7-8

```

procedure rightpush(v) returns r:bool
1 while(true) {
2   k := oracle(right);
3   prev := A[k-1];
4   cur := A[k];
5   if(prev.val != RN && cur.val = RN) {
6     if (k = MAX + 1) { r := false; return; }
7     if (CAS(&A[k-1], prev, <prev.val,prev.ctr+1>))
8       if (CAS(&A[k], cur, <v,cur.ctr+1>))
9         { r := true; return; }
10  }
11 }

```

Figure 19.17: Example operation of obstruction-free deque

both succeed. Because of the reads at lines 3-4, the **CAS** operations cannot be proved to be movers in the original program. We abstracted these reads using the **abstract read** tactic, which made **prev** and **cur** point to arbitrary elements. In addition, we abstracted the write to **k** at line 2, using the **abstract write** tactic. This abstraction does not affect the correctness of the operation, since the function **oracle** can return any index from the deque; its purpose is to return the optimum index for having fewer failing attempts.

The above proof steps allowed us to reason about the operation of two consecutive **CAS** operations. We then hoisted the part of the loop body with the successful **CAS** operations out of the loop. Then it was easy, by using the specification of **CAS**, to prove that the remaining part of the loop was a skip operation. By adding the deque invariants indicated in the algorithm description in [124], we were able to prove that the **CAS** operations, which were hoisted out of the loop, are left-movers in the abstracted program. This gave us an atomic block that contains two **CAS** operations with the desired behavior. We proved the other operations using a similar approach.

19.8 Treiber's Non-Blocking Stack

In our experimental study, we have proved the linearizability of two non-blocking stacks: Treiber's [125] and Michael's [126]. The code of our implementation of Treiber's stack in QEDPL is shown in Figure 19.18. By applying the proof script in Figure 19.19, we transformed the implementation to the atomic procedures given in Figure 19.20. Our proof strategy consists of peeling out the final iterations of the while loops in `pop` and `push` and performing a read abstraction on the `top` variable in the non-final iterations and the extracted final iteration. In particular, we replace the statement `t := top` with `havoc t;`.

In addition, we use the auxiliary variable `owner` to indicate that there is no aliasing between the new node `x` created by `pop` is not aliased to any node (`t` and `x`) referenced by `push`. This demonstrated the use of assertions to eliminate apparent conflicts between actions that actually do not share any variables.

```

record Node {
  data: int;
  next: Node;
}

const unique null: Node;
invariant IsNull(null.alloc);

var TOP: Node;

const unique EMPTY: int;

procedure push(v: int) {
  var t, x: Node;
  var g: bool;

  x := New Node;
  x.data := v;

  while(true) {
    t := TOP;
    x.next := t;
    call g := CAS(t, x);
    if(g) { break; }
  }
}

procedure {isatomic}
CAS(oldVal: Node, newVal: Node)
returns (r: bool) {
  r := (TOP == oldVal);
  if(r) {
    TOP := newVal;
  }
}

procedure pop()
returns (v: int) {
  var t, x: Node;
  var g: bool;

  while(true) {
    t := TOP;
    if(t == null) {
      v := EMPTY;
      return;
    }
    x := t.next;
    call g := CAS(t, x);
    if(g) { break; }
  }
  v := t.data;
}

```

Figure 19.18: The implementation of Treiber's non-blocking stack in QEDPL

```

1 inline all
2 nullcheck
3 reduce all
4 assert Atomic_5 t == null || IsAlloc(t.alloc)

```

```

5  invariant (forall n1: Node, n2: Node :: IsDealloc(n1.alloc) ==> n2.next != n1) && (
    forall n: Node :: !IsAlloc(n.alloc) ==> n.next = null)
6  aux begin owner [Node] int
7  annotate Atomic_2 owner owner[x] := tid;
8  assert Atomic_5 owner[x] == tid
9  assert Call_6 owner[x] == tid
10 simulate Call_6 assert owner[x] == tid; havoc g; if (TOP == t) { TOP := x; owner[x]
    := 0; break; }
11 assert Atomic_11 owner[t] == 0
12 assert Call_12 owner[x] == 0
13 aux end owner
14 invariant (forall n: Node :: !IsAlloc(n.alloc) ==> owner[n] == 0)
15 assert Call_12 x == null || IsAlloc(x.alloc)
16 invariant owner[TOP] == 0 && (TOP == null || IsAlloc(TOP.alloc))
17 simulate Atomic_4 havoc t; assume t == null || IsAlloc(t.alloc);
18 reduce all
19 check all block
20 simulate Atomic_8 t := TOP; if (t == null) { v := EMPTY; return; } havoc t; assume
    IsAlloc(t.alloc) && owner[t] == 0; x := t.next;
21 reduce all
22 check all block
23 peelout Atomic_4
24 simulate Atomic_4 havoc t; assume t == null || IsAlloc(t.alloc); assert owner[x] ==
    tid; assert x.alloc == Alloc; x.next := t; havoc g;
25 simulate Atomic_4_dup havoc t; assume t == null || IsAlloc(t.alloc); assert owner[x]
    == tid; assert x.alloc == Alloc; x.next := t; havoc g; assume (TOP == t); TOP :=
    x; owner[x] := 0;
26 mover Atomic_4
27 reduce loop Atomic_4 if(*) { havoc t; assume t == null || IsAlloc(t.alloc); assert
    owner[x] == tid; assert x.alloc == Alloc; x.next := t; havoc g; }
28 reduce all
29 simulate Atomic_2 x := New Node; x.data := v; owner[x] := tid; havoc t; x.next := t;
    havoc g; assume TOP == t; TOP := x; owner[x] := 0;
30 simulate Atomic_8 if (*) { assume TOP == null; havoc t; v := EMPTY; return; } havoc t
    ; assume IsAlloc(t.alloc) && owner[t] == 0; x := t.next;
31 mover Atomic_8
32 merge all
33 peelout Atomic_8
34 invariant (forall n: Node :: owner[n] == 0 && IsAlloc(n.next.alloc) ==> owner[n.next]
    == 0)
35 invariant (forall n: Node :: owner[n] == 0 ==> IsAlloc(n.next.alloc) || n.next ==
    null)
36 check all block
37 simulate Atomic_8 havoc x,g,t;
38 reduce loop Atomic_8 havoc x,g,t;
39 simulate Atomic_8_dup if (*) { assume TOP == null; havoc t; v := EMPTY; return; }
    havoc t; assume IsAlloc(t.alloc) && owner[t] == 0; x := t.next; assume TOP == t;
    TOP := x; havoc g,ex;
40 reduce all
41 check all block
42 simulate Atomic_43 havoc x, g, t; if (TOP == null) { v := EMPTY; return; } t := TOP;
    x := t.next; TOP := x; v := t.data;
43 simulate Atomic_2: x := New Node; x.data := v; havoc t, g; x.next := TOP; TOP := x;
    owner[x] := 0;

```

Figure 19.19: The proof script for Treiber's non-blocking stack.


```

procedure push(v: int) {
  var t: Node;
  var x: Node;
  var g: bool;

  atomic {
    havoc t, g;
    x := New Node;
    x.data := v;
    x.next := TOP;
    TOP := x;
    owner[x] := 0;
  }
}

procedure pop()
returns (v: int) {
  var t: Node;
  var x: Node;
  var g: bool;

  atomic {
    havoc x, g, t;
    if (TOP == null) {
      v := EMPTY;
      return;
    }
    t := TOP;
    x := t.next;
    TOP := x;
    v := t.data;
  }
}

```

Figure 19.20: The final version of Treiber’s non-blocking stack after the QED transformations.

19.9 Michael’s Non-Blocking Stack

Figure 19.21 shows the `pop` and `push` operations from Michael’s non-blocking stack algorithm [126]. The operations make use of a hazard pointer per thread in order to solve the well-known ABA problem. [127] gave a proof of this algorithm using concurrent separation logic. Since the algorithm uses fine-grained concurrency, the proof in [127] required reasoning about invariants at each interleaving point throughout the code. Using our method, we performed a simpler proof that transforms the bodies of `pop` and `push` to the atomic actions given in Figure 19.22.

Figure 19.23 shows the proof script we used for the non-blocking stack. Our proof strategy was aimed at making the (final) successful iterations of the while loops in `pop` and `push` atomic. We transformed the (non-final) iterations of the while loops by performing read abstractions on `top` since these reads do not affect correctness. For each loop, we hoisted the loop iteration containing a successful `CAS` operation outside the loop. There is an ownership protocol implicit in the use of the stack. A thread should only push into the stack an element that it owns. The push operation transfers the ownership of the element to the stack. Ownership is transferred back to the thread that manages to pop that element from the stack. We used the auxiliary variable *owner*, a map from stack elements to thread

ids, to capture this ownership transfer protocol. We also introduced assertions into the code to check that the ownership protocol is not being violated.

In addition, we relaxed the use of H by introducing a new auxiliary variable AH . The invariant $(\text{forall } t: \text{ int} :: AH[t] == \text{null} \mid\mid AH[t] == H[t])$ expresses that AH can have `null` entries. Replacing H with AH allowed us to eliminate conflicts on H .

As with other examples, we delayed the introduction of invariants until the atomic actions in the program had become coarse enough; all invariants were consequently fairly simple.

```

var H: [int]Node;

record Node {
  tl: Node;
}

var top: Node;

const unique null: Node;
invariant (IsNull(null.alloc));

procedure pop()
returns (t: Node) {
  var n: Node;
  var guard: bool;

  while(true) {
    t := top;
    if(t == null) {
      break;
    }

    H[tid] := t;

    if(t == top) {
      n := t.tl;

      call guard := CAS(t, n);
      if(guard) {
        break;
      }
    }
  } // end while

  H[tid] := null;
}

procedure {::isatomic} CAS(chk: Node, set: Node)
returns (r: bool) {
  r := (top == chk);
  if(r) {
    top := set;
  }
}

procedure push(b: Node)
returns (r: bool) {
  var t: Node;
  var n: int;
  var h: Node;
  var guard: bool;

  n := 1;
  while(n <= Tid) {
    h := H[n];
    if(h == b) {
      r := false;
      return;
    }
    n := n + 1;
  } // end while

  while(true) {
    t := top;

    b.tl := t;

    call guard := CAS(t, b);
    if(guard) {
      break;
    }
  } // end while

  r := true;
}

```

Figure 19.21: Michael's algorithm with hazard pointers

```

pop()
  returns r:ref {
    var m, n : ref;
    [havoc m,n;
    if(*) {
      assume (TOP == nil);
      r := nil;
    } else {
      assume (m != nil && m == TOP);
      n := tl[m]; TOP := n; r := m;
    }
  }
}

push(b:ref) {
  var m : ref, i : int;
  [assert b != nil;
  if(*) {
    r := false;
  } else {
    tl[b] = TOP;
    TOP := b; r := true;
  }
}]
}

```

Figure 19.22: The final version of Michael's stack after the QED transformations.

```

1  inline CAS
2  reduce all
3  aux begin AH [int]Node
4  simulate Atomic_4 H[tid]:=t; AH[tid]:= t;
5  simulate Atomic_9 H[tid]:=null; AH[tid]:= null;
6  skipafter Atomic_5 1
7  simulate Atomic_31 AH[tid]:=null;
8  aux end AH
9  invariant (forall t: int :: AH[t] == null || AH[t] == H[t])
10 peelout2 Atomic_2
11 peelout2 Atomic_13
12 simulate Atomic_2 havoc t; assume t != null;
13 simulate Atomic_5 havoc guard_0;
14 simulate Atomic_6 havoc n;
15 simulate Call_7 havoc guard;
16 reduce proc pop
17 simulate Atomic_13 assert b != null; assume guard_1; havoc h; assume AH[n] != b; n :=
    n + 1; guard_1 := n <= Tid;
18 simulate Atomic_11 assert b != null; havoc h,n; assume 1 <= n && n <= Tid; assume (
    forall m:TID :: 1 <= m && m < n ==> AH[m] != b); guard_1 := n <= Tid;
19 simulate Atomic_13_dup havoc h; r := false; return;
20 reduce loop Atomic_33 assert AH[tid]==null; if(*){ havoc t, n, guard_0, guard; H[tid]
    := t; }
21 assert Atomic_4_dup AH[tid] == null && t != null
22 simulate Atomic_11 assert b != null; havoc h, n; assume 1 <= n && n <= Tid; guard_1
    := n <= Tid; if(*){ assume (forall m: TID :: 1 <= m && m < n ==> AH[m] != b); }
    else { r := false; return; }
23 reduce loop2 Atomic_11
24 reduce all
25 aux begin owner [Node]int
26 simulate Atomic_11 assert b != null; havoc h, n; assume 1 <= n && n <= Tid; guard_1
    := n <= Tid; if (*) { assume (forall m: TID :: 1 <= m && m < n ==> AH[m] != b); }
    else { r := false; return; } if (guard_1) { havoc h; r := false; return; } owner
    [b]:= tid;
27 simulate Call_19 assert true; guard := top == t; if (guard) { top := b; owner[b]:= 0;
    } if (guard) { break; }
28 aux end owner
29 assert Atomic_18 b != null
30 invariant null.tl == null
31 simulate Call_7_dup assert true; /* call guard := CAS(t, n); */ guard := top == t; if
    (guard) { assert top.tl == n; top := n; } if (guard) { throw ExBreak; }
32 assert Atomic_18 !ReachBetween(Node_tl, top, b, b)
33 simulate Call_19 assert true; guard := top == t; if (guard) { assert b.tl == top; top
    := b; owner[b] := 0; } if (guard) { break; }
34 invariant ReachBetween(Node_tl, top, null, null)

```

```

35 simulate Atomic_11 assert b != null; havoc h, n; assume 1 <= n && n <= Tid; guard_1
    := n <= Tid; if (*) { assume (forall m: TID :: 1 <= m && m < n ==> AH[m] != b); }
    else { r := false; return; } if (guard_1) { havoc h; r := false; return; }
    assert !ReachBetween(Node_tl, top, b, b); owner[b] := tid;
36 invariant (forall t: Node :: ReachBetween(Node_tl, top, t, null) ==> owner[t] == 0)
37 assert Atomic_18 owner[b] == tid
38 assert Call_19 owner[b] == tid
39 simulate Atomic_2_dup havoc t; if (t == null) { throw ExBreak; } assume owner[t] ==
    0;
40 reduce all
41 simulate Atomic_2_dup havoc t; if (t == null) { throw ExBreak; } assume owner[t] ==
    0; assert AH[tid] == null && t != null; H[tid] := t; AH[tid] := t; havoc guard_0;
42 assert Atomic_6_dup owner[t] == 0
43 reduce all
44 simulate Atomic_17 havoc t; assert owner[b] == tid; assert !ReachBetween(Node_tl, top
    , b, b); assert b != null; b.tl := t;
45 reduce proc push
46 peelout2 Atomic_17
47 simulate Atomic_17 havoc t; assert owner[b] == tid; assert !ReachBetween(Node_tl, top
    , b, b); assert b != null; b.tl := t; havoc guard;
48 reduce loop Atomic_17 if(*) {havoc t; assert owner[b] == tid; assert !ReachBetween(
    Node_tl, top, b, b); assert b != null; b.tl := t; havoc guard; }
49 reduce all
50 simulate Atomic_46
51 simulate Atomic_46 assert H[tid] == null; if (*) { havoc t, n, guard_0, guard; H[tid]
    := t; }
52 fixmover 1 Atomic_31_dup
53 reduce proc pop
54 assert Atomic_31_dup AH[tid] != null && H[tid] != null
55 reduce proc pop
56 invariant (forall t: int :: H[t] == null)
57 check all block

```

Figure 19.23: The proof script for Michael's algorithm with hazard pointers

19.10 Non-Blocking Queue

Figure 19.24 shows the non-blocking queue introduced by Michael and Scott [119]. The enqueue and dequeue operations are designed to be highly-concurrent using the CAS operations. We proved that the implementation is linearizable to a simpler specification with atomic bodies of **Enqueue** and **Dequeue** procedures. The proof script for this example is shown in Figure 19.25. In the following we will elaborate on using variable introduction and hiding in the proof in order to eliminate a superficial conflict.

```

type Data;

record Object {
  data: Data;
}

record Node {
  next: Node;
  value: Object;
}

record NBQueue {
  Head: Node;
  Tail: Node;
}

const unique NULL_OBJECT: Object;
const unique NULL_QUEUE_ITEM: Node;
const unique NULL_NB_QUEUE: NBQueue;

invariant IsNull(NULL_OBJECT.alloc);
invariant IsNull(NULL_QUEUE_ITEM.alloc);
invariant IsNull(NULL_NB_QUEUE.alloc);

procedure {:isatomic} CAS_tail(queue:NBQueue,
  expected:Node, newValue:Node)
returns (result: bool) {
  result := (queue.Tail == expected);
  if (result) {
    queue.Tail := newValue;
  }
}

procedure {:isatomic} CAS_head(queue:NBQueue,
  expected:Node, newValue:Node)
returns (result: bool) {
  result := (queue.Head == expected);
  if (result) {
    queue.Head := newValue;
  }
}

procedure {:isatomic} CAS_tail_next(queue:NBQueue,
  expected:Node, newValue:Node)
returns (result: bool) {
  result := (queue.Tail.next == expected);
  if (result) {
    queue.Tail.next := newValue;
  }
}

procedure CreateNode(val:Object)
returns (this: Node) {
  this := New Node;
  assume this != NULL_QUEUE_ITEM;
  this.next := NULL_QUEUE_ITEM;
  this.value := val;
}

procedure Enqueue(this:NBQueue, value:Object) {
  var node: Node;
  var tail: Node;
  var n: Node;
  var guard: bool;

  call node := CreateNode(value);

  while (true) {
    tail := this.Tail;
    n := tail.next;
    if (tail == this.Tail) {
      if (n == NULL_QUEUE_ITEM) {
        call guard := CAS_tail_next(this, n, node);
        if (guard) {
          break;
        }
      } else {
        call guard := CAS_tail(this, tail, n);
      }
    }
    call guard := CAS_tail(this, tail, node);
  }

  procedure Dequeue(this:NBQueue)
returns (result: Object) {
  var head: Node;
  var tail: Node;
  var n: Node;
  var guard: bool;

  result := NULL_OBJECT;
  while (true) {
    head := this.Head;
    tail := this.Tail;
    n := head.next;

    if (head == this.Head) {
      if (head == tail) {
        if (n == NULL_QUEUE_ITEM) {
          return;
        } else {
          call guard := CAS_tail(this, tail, n);
        }
      } else {
        result := n.value;
        call guard := CAS_head(this, head, n);
        if(guard) {
          return;
        }
      }
    }
  }
}

```

Figure 19.24: The non-blocking queue implementation in QEDPL

```

1  inline all
2  nullcheck
3  reduce proc CreateNode
4  inline all
5  assert Call_12 n == NULL_QUEUE_ITEM && (forall q: NBQueue :: q.Head != node && q.Tail
    != node) && IsAlloc(node.alloc) && node.next == NULL_QUEUE_ITEM && (forall item:
    Node :: { Node_next[item] } item.next != node)
6  simulate Call_14 assert this.alloc == Alloc; guard := this.Tail == tail; if (guard) {
    assert this.alloc == Alloc; assert n != NULL_QUEUE_ITEM && ReachBetween(
    Node_next, this.Tail.next, n, NULL_QUEUE_ITEM); this.Tail := n; }
7  simulate Call_15 assert node != NULL_QUEUE_ITEM && ReachBetween(Node_next, tail.next,
    node, NULL_QUEUE_ITEM); assert this.alloc == Alloc; guard := this.Tail == tail;
    if (guard) { this.Tail := node; }
8  simulate Call_22 assert this.alloc == Alloc; guard := this.Tail == tail; if (guard) {
    assert this.alloc == Alloc; assert n != NULL_QUEUE_ITEM && ReachBetween(
    Node_next, this.Tail.next, n, NULL_QUEUE_ITEM); this.Tail := n; }
9  simulate Call_24 assert this.alloc == Alloc; guard := this.Head == head; if (guard) {
    assert this.alloc == Alloc; assert n != NULL_QUEUE_ITEM && ReachBetween(
    Node_next, this.Head.next, n, this.Tail); this.Head := n; }
10 invariant (forall item: Node :: !IsAlloc(item.alloc) ==> item.next == NULL_QUEUE_ITEM
    )
11 invariant (forall q: NBQueue, item: Node :: ReachBetween(Node_next, q.Head, item, q.
    Tail) ==> IsAlloc(item.alloc)) && (forall q: NBQueue :: ReachBetween(Node_next, q
    .Head, q.Tail, NULL_QUEUE_ITEM)) && (forall q: NBQueue, item: Node ::
    ReachBetween(Node_next, q.Head, item, NULL_QUEUE_ITEM) && item != NULL_QUEUE_ITEM
    ==> IsAlloc(item.alloc))
12 invariant (forall q1: NBQueue, q2: NBQueue, item: Node :: IsAlloc(item.alloc) && q1
    != q2 && (ReachBetween(Node_next, q1.Head, item, item) || ReachBetween(Node_next,
    item, q1.Head, q1.Head)) ==> !ReachBetween(Node_next, q2.Head, item, item) && !
    ReachBetween(Node_next, item, q2.Head, q2.Head))
13 simulate Atomic_9 assert this.alloc == Alloc; havoc tail; assume ReachBetween(
    Node_next, tail, this.Tail, this.Tail) && (forall item: Node :: ReachBetween(
    Node_next, tail, item, this.Tail) ==> IsAlloc(item.alloc));
14 simulate Atomic_10 assert tail.alloc == Alloc; havoc n; assume ReachBetween(Node_next
    , tail.next, n, n);
15 simulate Atomic_11 assert this.alloc == Alloc; havoc guard_0;
16 reduce proc Enqueue
17 peelout Atomic_9
18 simulate Atomic_9 var k: Node; assert this.alloc == Alloc; assert this.Tail.alloc ==
    Alloc; assert (forall q: NBQueue :: q.Head != node && q.Tail != node) && IsAlloc(
    node.alloc) && node.next == NULL_QUEUE_ITEM && (forall item: Node :: { Node_next[
    item] } item.next != node); havoc guard_0, guard, tail, n, k; if (*) { if(
    ReachBetween(Node_next, this.Tail.next, k, NULL_QUEUE_ITEM) && k !=
    NULL_QUEUE_ITEM){ this.Tail := k; } }
19 simulate Call_15 assert node != NULL_QUEUE_ITEM && ReachBetween(Node_next, tail.next,
    node, NULL_QUEUE_ITEM); assert this.alloc == Alloc; if (this.Tail == tail) {
    this.Tail := node; } havoc guard;
20 merge all
21 simulate Atomic_46 assert this.alloc == Alloc; havoc tail; assume ReachBetween(
    Node_next, tail, this.Tail, this.Tail) && (forall item: Node :: ReachBetween(
    Node_next, tail, item, this.Tail) ==> IsAlloc(item.alloc)); havoc guard_0, guard,
    n, ex; assert (forall q: NBQueue :: q.Head != node && q.Tail != node) && IsAlloc
    (node.alloc) && node.next == NULL_QUEUE_ITEM && (forall item: Node :: { Node_next
    [item] } item.next != node); assume this.Tail.next == NULL_QUEUE_ITEM; this.Tail.
    next := node; assume ReachBetween(Node_next, tail.next, node, NULL_QUEUE_ITEM);
22 simulate Atomic_17 assert this.alloc == Alloc; havoc head; assume ReachBetween(
    Node_next, head, this.Head, this.Head) && (forall item: Node :: ReachBetween(
    Node_next, head, item, this.Head) ==> IsAlloc(item.alloc));
23 mover Atomic_17
24 merge all
25 simulate Atomic_17 assert this.alloc == Alloc; havoc head; assume ReachBetween(
    Node_next, head, this.Head, this.Head) && (forall item: Node :: ReachBetween(
    Node_next, head, item, this.Head) ==> IsAlloc(item.alloc)); havoc tail; assume
    ReachBetween(Node_next, head, tail, this.Tail) && IsAlloc(tail.alloc);
26 mover Atomic_17
27 merge all

```

```

28 simulate Atomic_20 assert this.alloc == Alloc; havoc guard_1;
29 mover Atomic_20
30 merge all
31 hoist Atomic_17 after
32 hoist Atomic_17_then after
33 check all block
34 simulate Atomic_17_else assert this.alloc == Alloc; havoc head, tail, n, guard_1;
35 simulate Atomic_17_then_then assert this.alloc == Alloc; havoc head; assume
    ReachBetween(Node_next, head, this.Head, this.Head) && (forall item: Node ::
    ReachBetween(Node_next, head, item, this.Head) ==> IsAlloc(item.alloc)); havoc
    tail; assume ReachBetween(Node_next, head, tail, this.Tail) && IsAlloc(tail.alloc)
    ); if(head.next == NULL_QUEUE_ITEM) { n := NULL_QUEUE_ITEM; } else { havoc n;
    assume ReachBetween(Node_next, head.next, n, NULL_QUEUE_ITEM); } havoc guard_1;
    assume head == tail;
36 mover Atomic_17_then_then
37 merge all
38 mover Atomic_17_then_else
39 merge all
40 mover Atomic_17_then_else
41 merge all
42 mover Atomic_48
43 merge all
44 peelout Atomic_56
45 merge all
46 check all block
47 simulate Atomic_56 var k: Node; assert this.alloc == Alloc; havoc guard_1, guard,
    head, tail, n, result, k; if (*) { if(ReachBetween(Node_next, this.Tail.next, k,
    NULL_QUEUE_ITEM) && k != NULL_QUEUE_ITEM) { this.Tail := k; } }
48 simulate Atomic_60 assert this.alloc == Alloc; havoc guard_1, guard, head, tail, n,
    ex; if(*) { assume this.Head.next != NULL_QUEUE_ITEM; assume this.Head != this.
    Tail; result := this.Head.next.value; this.Head := this.Head.next; } return;
49 invariant (forall node1: Node, node2: Node :: IsDealloc(node1.alloc) ==> node2.next
    != node1)
50 aux begin owner [Node]int
51 annotate Call_8 owner owner[node] := tid;
52 annotate Atomic_46 owner owner[node] := 0;
53 aux end owner
54 invariant (forall n: Node :: !IsAlloc(n.alloc) ==> owner[n] == 0)
55 assert Atomic_9 owner[node] == tid
56 assert Atomic_46 owner[node] == tid
57 invariant (forall q: NBQueue, n: Node :: ReachBetween(Node_next, q.Head, n,
    NULL_QUEUE_ITEM) ==> owner[n] == 0)
58 invariant (forall node: Node :: owner[node] != 0 ==> (forall q: NBQueue :: q.Head !=
    node && q.Tail != node) && IsAlloc(node.alloc) && node.next == NULL_QUEUE_ITEM &&
    (forall item: Node :: { Node_next[item] } item.next != node))
59 simulate Atomic_46 var m: Node; if(*) { havoc m; assume ReachBetween(Node_next, this.
    Tail, m, NULL_QUEUE_ITEM) && m != NULL_QUEUE_ITEM && m.next == NULL_QUEUE_ITEM;
    this.Tail := m; } else { m := this.Tail; } assert owner[node] == tid; assert this
    .alloc == Alloc; havoc guard_0, guard, n, tail, ex; assert (forall q: NBQueue ::
    q.Head != node && q.Tail != node) && IsAlloc(node.alloc) && node.next ==
    NULL_QUEUE_ITEM && (forall item: Node :: { Node_next[item] } item.next != node);
    assume this.Tail.next == NULL_QUEUE_ITEM; this.Tail.next := node; assume
    ReachBetween(Node_next, tail.next, node, NULL_QUEUE_ITEM); owner[node] := 0;
60 simulate Call_15 var z: Node; assert node != NULL_QUEUE_ITEM && ReachBetween(Node_next
    , tail.next, node, NULL_QUEUE_ITEM); assert this.alloc == Alloc; havoc z; if (*)
    { if (ReachBetween(Node_next, this.Tail.next, z, NULL_QUEUE_ITEM) && z !=
    NULL_QUEUE_ITEM) { this.Tail := z; } } havoc guard;
61 simulate Atomic_60 var m: Node; havoc m; if (*) { assume ReachBetween(Node_next, this.
    Tail, m, NULL_QUEUE_ITEM) && m != NULL_QUEUE_ITEM && m != this.Head; this.Tail :=
    m; } else { m := this.Tail; } assert this.alloc == Alloc; havoc guard_1, guard,
    head, tail, n, ex; if (*) { assume this.Head.next != NULL_QUEUE_ITEM; assume this
    .Head != this.Tail; result := this.Head.next.value; this.Head := this.Head.next;
    } return;
62 check all block

```

Figure 19.25: The proof script for the non-blocking queue

19.10.1 Eliminating the Superficial Conflict

Implementation (Impl)

```

procedure Enqueue(x: int) {
  var node, tail: Node;

  atomic {
    node := new Node(x);
    node.next := null;
  }

  while(true) {
    Move_Tail: atomic {
      havoc tail;
      assume Reach(next, Tail, tail, null)
        && tail != null;
      Tail := tail;
    }
  } // end while

  Do_Enqueue: atomic {
    assume (Tail.next == null);
    Tail.next := node; tail := Tail;
  }

  Update_Tail: atomic {
    if (Tail == tail) Tail := node;
  }
}

```

Specification (Spec)

```

procedure Enqueue(x: int) {
  atomic {
    node := new Node(x);
    node.next := null;
    _Tail.next := node;
    _Tail := _Tail.next;
  }
}

```

```

procedure Dequeue() returns (x: int) {
  var tail: Node;

  while(true) {
    Move_Tail: atomic {
      havoc tail;
      assume Reach(next, Tail, tail, null)
        && tail != null;
      Tail := tail;
    }
  } // end while

  Do_Dequeue: atomic {
    if (Head.next == null) {
      x := null;
    } else {
      assume (Head != Tail);
      Head := Head.next; x := Head.data;
    }
  }
}

```

```

procedure Dequeue() returns (x: int) {
  atomic {
    if (Head.next == null) {
      return null;
    } else {
      Head := Head.next;
      x := Head.data;
    }
  }
}

```

Figure 19.26: The reduced implementation of the non-blocking queue and its specification

Figure 19.26 shows the version of the non-blocking queue after applying a reduction phase on the original implementation. We will now explain the key steps of the proof, which allow us to transform that version to the specification given in the same figure. We mark each atomic action with a label. Atomic action `Do_Dequeue` removes an element from


```

procedure Enqueue(x: int) {
  var node, tail: Node;
  atomic {
    node := new Node(x);
    node.next := null;
  }

  while(true) {
Move_Tail: atomic { havoc tail; }
  } // end while

Do_Enqueue: atomic {
  _Tail.next := node;
  _Tail := _Tail.next;
  havoc tail;
}
Update_Tail: atomic { assume true; }
}

procedure Dequeue() returns (x: int) {
  var tail: Node;

  while(true) {
Move_Tail: atomic { havoc tail; }
  } // end while

Do_Dequeue: atomic {
  if (Head.next == null) {
    x := null;
  } else {
    Head := Head.next; x := Head.data;
  }
}
}

```

Figure 19.27: The version of the non-blocking queue after hiding Tail

the queue, and `Do_Enqueue` appends a new element to the queue. The implementation is lazy in that `Do_Enqueue` does not update the `Tail` variable after adding the new node. As a result, at any time `Tail` may point to any node between `Head` and `null`. The actions labeled `Move_Tail` and `Update_Tail` try to move the `Tail` towards the end of the list. This resembles relaxed balancing [128] in concurrent implementation of tree-like data structures, in which restructuring the data structure is separated from actual operations, and delayed.

The predicate `Reach(next,k,l,m)` expresses that, from node `k`, following zero or more `next` pointers, we first reach `l` and then `m` [129]. The predicate `Reach(next,k,m,m)` (sometimes shortened to `Reach(next,k,m)`) expresses that we can reach `m` starting from `k` and following the `next` pointers. The `Reach` predicate gives us the ability to do simple abstractions on actions accessing the list nodes. For example, a former abstraction step in the reduction phase replaces the action `n := tail.next` with the action `havoc n; assume Reach(next,tail,n,n)`; while the former is not mover, the latter is.

In order to apply reduction, the only option is to show that `Move_Tail` is a right-mover, since `Do_Enqueue` and `Do_Dequeue` perform the actual operations, thus are not movers. `Move_Tail` conflicts with `Do_Enqueue` and `Do_Dequeue` on `Tail`. Notice that `Move_Tail` performs an internal operation that does not affect the semantics of the queue. Thus, these conflicts are superficial. Havocing `Tail` in the conflicting actions, or hiding `Tail` are a valid proof steps, and would make reduction pass. However, the resulting code would perform

incorrect operation.

We eliminate the conflict by coupling the hiding of `Tail` with introducing the history variable `_Tail` of the same type. Differently from `Tail`, `_Tail` always points to the end of the queue. We then associate the existing variables with the new variable `_Tail` by the following invariant.

```
Reach(next, Head, Tail, Tail) && Reach(next, Tail, _Tail, null)
&& (_Tail != null) && (_Tail.next == null)
```

In order to satisfy the invariant, we add to the end of `Do_Enqueue` the assignment `_Tail := _Tail.next`. Once there is `_Tail` to keep track of the end of the list, we are ready to hide `Tail`. This is done by replacing the actions in the program with actions that do not refer to `Tail`, but now uses `_Tail` to access the end of the linked list. Figure 19.27 shows the version of the program after hiding `Tail`. Notice that the new form of `Move_Tail` does not perform any semantic operation in the new program, and does not conflict with other actions. In addition, the actions `Do_Enqueue` and `Do_Dequeue` now use `_Tail` to correctly perform their operations.

The hiding step also includes existentially quantifying `Tail` in the invariant given above. This produces the following invariant for the new program.

```
Reach(next, Head, _Tail, null) && (_Tail != null) && (_Tail.next == null)
```

We proceed with a reduction phase that combines the blocks into a single action for each operation. The combined operations, together with the above invariant (for simplicity, we omit parts of the representation invariant), give the correct behavior of a sequential queue implementation. Corollary 11 ensures that the original implementation in [119] is linearizable to this final program from the invariant. Note that it is also possible to continue the proof with an extra refinement phase to prove the linearizability to a more generic specification of the queue.

19.11 Non-Blocking Mutex-lock

```

type State;
const unique READER: State;
const unique WRITER: State;
const unique ACTIVE_READER: State;
axiom (forall s:State:: (s == READER)
      || (s == WRITER) || (s == ACTIVE_READER));

const unique null : Lelem;
invariant IsNull(null.alloc)
  && (forall l:Lelem ::
      IsNull(l.alloc) <=> l == null);

var L: Lelem;

record Lelem {
  state: State;
  spin: int;
  next: Lelem;
  prev: Lelem;
}

procedure {:isatomic} CAS(newVal:Lelem, oldVal:Lelem)
returns (result: Lelem) {
  result := L;
  if (L == oldVal) {
    L := newVal;
  }
}

procedure {:isatomic} fetch_and_store (I:Lelem)
returns (result: Lelem) {
  result := L;
  L := I;
}

procedure writerLock ()
returns (I: Lelem) {
  var pred: Lelem;

  I := New Lelem;
  I.state := WRITER;
  I.spin := 1;
  I.next := null;
  call pred := fetch_and_store(I);
  if (pred != null) {
    pred.next := I;
    assume (I.spin != 1);
  }
}

procedure writerUnlock (I: Lelem) {
  var pred: Lelem;
  var tmp : Lelem;

  if (I.next == null) {
    call tmp := CAS(null, I);
    if(tmp == I) {
      return;
    }
  }

  assume (I.next != null);
  I.next.prev := null;
  I.next.spin := 0;
}

```

Figure 19.28: The non-blocking mutex-lock implementation in QEDPL

```

1 inline all
2 reduce all
3 assert proc writerUnlock IsAlloc(I.alloc)
4 assert Atomic_16@writerUnlock IsAlloc(I.next.alloc)
5 assert Atomic_17@writerUnlock IsAlloc(I.next.alloc)
6 assert Atomic_10@writerLock IsAlloc(pred.alloc)
7 assert Atomic_10@writerLock pred.next == null
8 assert Atomic_10@writerLock I != null
9 abstract read Lelem_next Atomic_12@writerUnlock
10 reduce all
11 hoist Atomic_5 after
12 reduce all
13 simulate Atomic_12@writerUnlock havoc abst_Lelem_next_0, tmp, guard_0; assert IsAlloc
    (I.alloc); if (*) { if (L == I) { L := null; return; } }
14 simulate Atomic_15@writerUnlock assert IsAlloc(I.alloc); assume I.next != null;
    assert IsAlloc(I.next.alloc); I.next.prev := null; I.next.spin := 0;
15 aux begin owner [Lelem]int
16 annotate Atomic_5_then@writerLock owner owner[I] := tid;
17 annotate Atomic_5_else@writerLock owner owner[I] := tid;
18 assert Atomic_11@writerLock owner[I] == tid

```

```

19 assert Atomic_12@writerUnlock owner[I] == tid
20 assert Atomic_15@writerUnlock owner[I] == tid
21 aux end owner
22 invariant (forall L: Lelem :: !IsAlloc(L.alloc) ==> owner[L] == 0)
23 invariant IsAlloc(L.alloc) || IsNull(L.alloc)
24 invariant null.next == null && null.prev == null
25 invariant L.next == null
26 aux begin wlock Lelem
27 simulate Atomic_11 assert owner[I] == tid; assume I.spin != 1; assert wlock == I;
28 simulate Atomic_5_else I := New Lelem; I.state := WRITER; I.spin := 1; I.next := null
    ; pred := L; L := I; assume pred == null; owner[I] := tid; wlock := I;
29 simulate Atomic_12 assert owner[I] == tid; havoc abst_Lelem_next_0, tmp, guard_0;
    assert IsAlloc(I.alloc); if (*) { if (L == I) { L := null; wlock := null; return;
    } }
30 simulate Atomic_15 assert owner[I] == tid; assert IsAlloc(I.alloc); assume I.next !=
    null; assert IsAlloc(I.next.alloc); I.next.prev := null; I.next.spin := 0; wlock
    := I.next;
31 aux end wlock
32 assert Atomic_12 L != null
33 assert Atomic_15 L != null
34 assert Atomic_11 L != null
35 invariant L == null <==> wlock == null
36 aux begin acq [Lelem]boolean
37 annotate Atomic_5_else acq acq[I] := False;
38 annotate Atomic_11 acq acq[I] := False;
39 aux end acq
40 assert Atomic_11 acq[I] == True
41 assert Atomic_12 acq[I] == False
42 assert Atomic_15 acq[I] == False
43 invariant (forall l: Lelem :: l.spin == 0 || l.spin == 1)
44 invariant (forall l: Lelem :: !IsAlloc(l.alloc) ==> acq[l] == True)
45 assert Atomic_12 wlock == I
46 assert Atomic_15 wlock == I
47 assert Atomic_11 IsAlloc(I.alloc)
48 invariant (forall l: Lelem :: !IsAlloc(l.alloc) ==> l.next == null)
49 invariant (forall k: Lelem, l: Lelem :: IsAlloc(k.alloc) && IsDealloc(l.alloc) ==> !
    ReachBetween(Lelem_next, k, l, l))
50 invariant (forall l: Lelem :: IsAlloc(l.alloc) ==> ReachBetween(Lelem_next, l, null,
    null))
51 invariant ReachBetween(Lelem_next, wlock, L, null)
52 invariant (forall k: Lelem, l: Lelem, m: Lelem :: IsAlloc(l.alloc) && k != m && l !=
    m && k != l && ReachBetween(Lelem_next, k, l, null) && ReachBetween(Lelem_next, m
    , l, null) ==> ReachBetween(Lelem_next, k, m, m) || ReachBetween(Lelem_next, m, k
    , k))
53 invariant (forall l: Lelem :: IsAlloc(l.alloc) && ReachBetween(Lelem_next, wlock, l,
    L) && l != wlock ==> acq[l] == True)
54 invariant (forall l: Lelem :: IsAlloc(l.alloc) && ReachBetween(Lelem_next, l, wlock,
    wlock) && l != wlock ==> acq[l] == False) && (forall l: Lelem :: L != null &&
    IsAlloc(l.alloc) && !ReachBetween(Lelem_next, l, L, null) ==> acq[l] == False)
55 invariant (forall l: Lelem :: IsAlloc(l.alloc) && ReachBetween(Lelem_next, wlock, l,
    null) && l != null && l != wlock ==> l.spin == 1)
56 invariant (forall l: Lelem :: IsAlloc(l.alloc) && acq[l] == True && l.spin == 0 ==> l
    == wlock)
57 reduce all
58 check all

```

Figure 19.29: The proof script for the non-blocking mutex-lock

Figure 19.28 shows a non-blocking mutex-lock implementation adapted from the reader-s/writer lock in [118]. The proof script for this example is shown in Figure 19.29. The proof

of this example was one of the most sophisticated proofs we performed. The reasoning about the mover type of fine-grained actions in the initial program relies on complex representation invariants. Instead of introducing these invariants, we added assertions to individual actions that allowed us to show the mover types of these actions. We added the representation invariants after the program got into a coarser atomicity level by the initial reduction steps. In the following we will elaborate on using variable introduction and hiding in the proof in order to eliminate a superficial conflict that we faced near the end of the proof.

19.11.1 Eliminating the Superficial Conflict

Implementation (*Impl*)

```

record Lelem { spin: int; next: Lelem; }
var L: Lelem; // tail of list

Unlock(I: Lelem)
Check_Empty: atomic {
  if (L == I) {
    L := null;
    return;
  }
}
Signal_Next: atomic {
  I.next.spin := 0;
}

Lock()
Create_Node: atomic {
  I := New Lelem;
  I.spin := 1;
  I.next := null;
  pred := L;
  L := I;
  if (pred != null)
    pred.next := I;
}
Wait_Prev: atomic {
  if (pred != null)
    assume I.spin != 1;
}
return I;

```

Specification (*Spec*)

```

atomic Unlock(I: Lelem)
  assert I != null && owner == I;
  owner := null;

atomic Lock()
  I := New Lelem;
  assume owner == null;
  owner := I;
  return I;

```

Figure 19.30: The reduced implementation and the specification of the lock

Figure 19.30 shows the version of the mutex-lock implementation after a reduction phase. Similarly to the queue example, we will explain the application of the refinement phase to transform that version to the specification given in the same figure.

In the example, locking requests are kept as nodes in a linked list. At the head of the list is the node holding the lock, and the rest belong to the threads waiting for the lock to become free. When the head node releases the lock, it signals its next node to allow it to

```

writerUnlock(I: Lelem)
Check_Empty: atomic {
  if (L == I) {
    * assert I != null && owner == I;
    L := null;
    * owner := H := null;
    return;
  }
}
Signal_Next: atomic {
  * assert I != null && owner == I;
  I.next.spin := 0;
  * H := I.next;
  * owner := null;
}

writerLock()
Create_Node: atomic {
  I := New Lelem;
  I.spin := 1;
  I.next := null;
  * thread[I] := tid;
  pred := L;
  L := I;
  if (pred != null)
    pred.next := I;
  else
    * owner := H := I;
}
Wait_Prev: atomic {
  if (pred != null) {
    assert thread[I] == tid && Reach(next, H, I, L);
    assume I.spin != 1;
    * owner := I;
    * thread[I] := 0;
  }
}
return I;

```

Figure 19.31: The lock implementation after introducing **thread**, **H** and **owner**

grab the lock. The **spin** field is used for this wait/signal protocol.

When the list is empty, the variable **L** is **null**, otherwise **L** points to the tail of the list. The **writerLock** procedure creates a node named **I** and appends it to the tail of the list. If the list is empty, the action **Create_Node** completes the acquisition of the lock. Otherwise, the action **Wait_Prev** waits, on its for the previous node to signal it.

The created node **I** is returned to the caller, which later passes it to the **writerUnlock** procedure. Action **Check_Empty** of **writerUnlock** completes the release if **I=L**, i.e. **I** is the only node in the list. Otherwise, **Signal_Next** signals the next node by setting its **spin** field.

In Figure 19.30, the superficial conflict happens between **writerLock** and **writerUnlock** on the **spin** field. This conflict is more subtle than the one in the queue example, since **spin** is essential for correctness of the operations. Abstracting the actions accessing **spin** or hiding **spin** violates the mutual-exclusion property that the program must provide. The conflict prevents us from combining the pairs of actions in either procedure.

We resolve the superficial conflict by keeping enough information, in new variables, about when the **spin** is signaled. Figure 19.31 shows the annotated code with the new variables. We introduce a new variable **thread** that maps every **Node** to the id of the thread that

created the node. In addition `H` points to the head of the list, and `owner` points to the node holding the lock. Usually `H==owner`; however, when `Signal_Next` signals the next node, `H` is set to the signalled node, and `owner` is set to `null`. The following invariant allows to infer that when `Wait_Prev` in `writerLock` sees `spin!=1`, `owner==null` and the current node `I` is the head, so it can acquire the node safely.

```
(L == null || L.next == null) && (L == null <==> H == null)
&& Reach(next, H, L, null) && (owner == null || owner == H)
&& (forall u: Node :: Reach(next, H, u, L) && n != null
    ==> u.alloc && (u.spin == 1 || u == H))
&& ((H != null && H.spin != 1 && thread[H] != 0) ==> owner == null)
```

At this point, we can hide the `spin` field, but now replace the instruction `assume (spin != 1)` in `Wait_Prev` with `assume (owner==null && H==I)`, which will make `writerLock` wait until the node becomes the head of the list. Figure 19.30 shows the final version of the code after another reduction phase that combines the new actions, followed by a refinement phase that hides the unnecessary variables. Notice that, an assertion remains in `writerUnlock`, which states that the given node must be node that is holding the lock, created by a previous call to `writerLock`. This assertion is in fact the pre-condition of `writerUnlock`; the linearizability proof will hold as long as the clients satisfy the precondition.

In this chapter, we elaborated on proofs we performed following the approach introduced in this thesis and using the QED-VERIFIER software. The proofs indicate the simplicity and effectiveness of atomicity-based verification of concurrent programs. While our experience with lock-based programs showed how our proof commands concisely express the use of common synchronization mechanisms and automatically compute atomic statements, our experience with non-blocking algorithms demonstrated that our method can also be used to verify highly-concurrent algorithms from the literature.

Chapter 20

CONCLUSION

In the second part of this thesis, we introduced QED, our proof system for statically verifying concurrent programs. QED provides a novel proof strategy in which atomicity is used as a proof tool: a program with fine-grained concurrency is transformed iteratively to make it consist of larger atomic actions, and the correctness conditions are checked after the program reaches an atomicity level appropriate for local analyses. This gradually reduces the influence of thread interleavings on the complexity of the reasoning, and therefore permits significantly more tractable proofs than those provided by existing methods. For example, we prove assertions by performing sequential (local) checks within atomic blocks of the final program and prove linearizability by computing the abstraction map iteratively during the proof.

Proofs in QED replace complex annotations and abstraction maps with code transformations, which are based on novel applications of reduction and abstraction techniques. In contrast to the standard uses of these techniques, reduction and abstraction in QED are symbiotic, which allows us to reach the desired level of atomicity even when there is apparent interference between threads. Moreover, the transformations provide strong guarantees so that one can simplify a program with larger atomic blocks using QED and continue the proof with another method, such as separation logic.

We presented the contributions of this part in several layers. In the first layer we described the formal aspects of the QED proof system: we introduced a formal programming language, ACTIONPL, then described our notion of proving assertions in an ACTIONPL program applying a sequence of QED transformations leading to a program with all assertions validated. In the second level, we explained how to apply the conceptual ideas in QED to programming language used in practice. For this, we introduced the QEDPL language and defined the application of each proof rule on a QEDPL program as a proof tactic.

Given the mechanisms in the second level, we also showed how to reason about the

linearizability of a concurrent data structure implementation with respect to its sequential specification by transforming the former to the latter using the QED proof system. We demonstrated that QED provides a rich set of techniques to eliminate various difficulties in proving linearizability, ranging from fine-grained concurrency to differences in representation between the implementation and the specification.

In the third level, we presented our implementation of QED in the interactive proof tool QED-VERIFIER. QED-VERIFIER is open source and can be downloaded from <http://qed.codeplex.com>. QED-VERIFIER accepts as input a QEDPL program and carries out the entire proof on QEDPL. The user guides a proof in QED-VERIFIER by giving concise and intuitive proof commands, each of which automates the application of one or multiple proof tactics to the current program. By using the Z3 theorem prover, QED-VERIFIER performs all the complicated low-level logical reasoning automatically, leaving to the user only high-level reasoning at the programming language level. In addition to the base set of proof commands, we provided proof templates that are used to express the use of common synchronization idioms and coarse-level proof rules to automatically generate annotations required for these templates to succeed.

Using our tool QED-VERIFIER, we evaluated our technique by proving assertions in and the linearizability of a wide-range of programs from the literature. Our experience has produced strong evidence for two claims. First, QED is effective: the notions of reduction and abstraction in QED are flexible enough to construct proofs of a wide spectrum of programs using fine-grained locking, non-blocking operations, and optimistic concurrency. Second, QED is usable: QED-style proofs captures the design intent of the programmer by separating concerns about concurrency and data-flow during the proof, and thus helps him to express and check each correctness argument on a program in a clear way. As the proof script for a program serves as a reproducible documentation of its correctness, one of the outcomes of this work is a proof repository for a collection of concurrent software.

20.1 Future Work

20.1.1 Applying QED to Other Languages

There are two approaches to applying the QED proof system to a higher-level programming language such as Java. The first approach is to develop a translator from Java to QEDPL

and to perform the proofs in QEDPL. This may limit the practicality of the proof technique. For example, many object-oriented programming constructs of Java should either be omitted in the analysis (without loss of soundness) or encoded in terms of QEDPL constructs, both of which reduce the readability of the program and rule out various abstractions made possible by the object-orientation.

The second approach is by lifting the proof steps originally defined over ACTIONPL to the Java level. This enables the user to perform the proof using the language of the original program. We demonstrated this approach for QEDPL in Chapter 14 and 15. A possible direction to continue this effort is to lifting the proof concepts in the QEDPL level to more realistic languages such as C, Java, and C#. For example, a similar link between ACTIONPL and QEDPL can be established between QEDPL and Java, and proof tactics over Java programs can be formulated using the tactics over QEDPL programs.

Programming platforms (languages, runtimes, and disciplines) will also considerably influence the effectiveness and success of the above formulation. For example, Cyclone [130], a restricted version of the C language, provides strong safety guarantees, such as no dangling pointers, and is therefore easier to analyze statically. Additionally, the TBB, Cilk, and Open MP platforms offer carefully selected programming idioms, such as structured fork/join parallelism, that implicitly restrict the use of concurrency and prevent the mistakes that can occur in a more flexible platform. An open question is how specializing the techniques in QED to similar programming environments can enable a more effective and automated way of verifying programs.

20.1.2 Improving the Mover Checks

The mover (commutativity) check is a fundamental operation in atomicity-based proofs. Increasing the scalability and efficiency of the mover check is key to improving the effectiveness of static verification using QED. In QED, we perform the check by generating verification conditions (VC) and calling a theorem prover to decide the validity of the VCs. The memory and time cost of performing a mover check becomes considerably high (quadratic on the number of actions in the program) while verifying large programs. In this thesis, we have investigated how to reduce this cost by 1) formulating the mover checks to exploit the modular design of the program and 2) making use of the type and effect systems for

atomicity, for example that of Atomizer [131], to implement short-circuit mover checks. We believe that there is still much room to make use of additional techniques to (1) improve the efficiency of answering the decision query in the mover check ($\mathcal{MC}$) or (2) to make use of other low-cost commutativity checks. An example to (1) is to preprocess the VC for the mover check to make it remain in a decidable fragment of logic, for example, by bounding infinite types to a finite size. For (2), in order to reason about recursive data structures (e.g., trees), one can use the type and effect system of Deterministic Parallel Java (DPJ) [132] to compute atomicity of code blocks with recursive operations.

20.1.3 Using Mover Checks for Precise Commutativity Analysis

While building the QED proof system we developed novel, more semantic, and therefore more widely applicable notions of reduction and abstraction. We foresee that these techniques will improve the work in related areas on concurrent software. For instance, existing approaches to proving determinism, parallelizing sequential programs [133, 134, 135], and reasoning about atomicity of computations, for example, in software transactional memory [136] and data structure implementations [137], are based on showing that different threads access distinct memory locations. These approaches become ineffective when parallel tasks interact with each other on the shared memory and the program uses synchronization primitives. In these cases, a commutativity analysis can further the determinism proof and the code parallelization by showing that executions of conflicting code blocks in different orders will all result in semantically equivalent states. The mover check in QED is well-suited to perform such analyses precisely and automatically using a theorem prover. A topic to investigate is how the mover check can be used as a general purpose reasoning tool to complement existing commutativity analyses.

20.1.4 Concise Proof Commands To Simplify User-Driven Proofs

We believe that programmer's help is necessary for software validation tools to support complex and large programs. Programming platforms for sequential programs have been successful in providing developers with a carefully designed set of programming constructs and code annotations to add specifications about the program. For example, Spec# allows the programmer to give loop invariants and procedure contracts as annotations and define

variables of non-null types, which enables precise and modular (thus efficient) verification of the program. In this thesis we investigated a natural way of allowing programmers to provide similar annotations for concurrent programs. For example in QED, the user's help through program annotations is essential to support, for example, recursion (Section 13.6.1), common synchronization idioms such as mutual exclusion and events, and thread-local variables (Chapter 18). Through proof commands, we provided to the user a convenient interface to the verification tool. We foresee that additional methodologies are worth investigating to allow the programmer to express his reasoning about the functionality of the software by, for example, indicating an execution ordering between parallel statements, and letting the verification tool capture this reasoning to apply a more tractable (e.g., local) analysis than would be possible in a standard approach.

20.1.5 Automated Derivation of Proof Command Sequences

An orthogonal direction to simplifying proofs using concise proof commands is to automatically infer the sequence of commands that will validate an indicated assertion in a proof or transform an indicated code block to a single atomic statement. While the reduction commands are fully automated, abstraction commands require hints from the user between reduction steps. We have found a number of abstraction patterns helpful to remove certain types of conflicts. For example, read abstraction on a single variable or abstracting the next pointer of a node to reading an arbitrary node from a linked list are common abstraction patterns. By restricting the set of possible abstraction patterns and applying them heuristically, guided by hints derived from failing mover checks, one can develop a technique to iteratively try possible command sequences and help the derivation of the proof script. We believe that the idea of concise proof commands in Section 20.1.4 is essential for circumventing the difficulties in automating proof generation. For example, when the hint required for the abstraction step is too complicated to infer or the noninterference between two actions depends on a complex representation invariant for a data structure.

20.1.6 Using Runtime Analysis to Help Static Verification

In GOLDLOCKS, we used static analysis tools to soundly eliminate runtime race checks for variables that these tools have proved to be race-free and reduced the cost of runtime

checking dramatically. Similarly, the reasoning about the correctness of a complex system can be divided into a set of proof obligations, each separately verified by a QED-based static verification technique or a runtime monitoring or model checking technique. For instance, a strict runtime execution ordering between the program statements can eliminate many commutativity checks in reduction. While such an ordering is difficult to express and check using a theorem prover, a model checker can easily verify the ordering. The challenges in developing such hybrid methods are: 1) how to divide the whole verification task into pieces manageable by each technique, and 2) how to ensure the soundness of the combined analysis, i.e., the static and runtime (or the deductive and algorithmic) checks together imply the correctness of the software.

20.1.7 Proof System for Parallelizing Programs

In this thesis, we proposed a proof system to transform a concurrent program to a sequential program and showed that every transformation preserves the soundness of assertion checking. An open question is how to establish a similar framework to transform a sequential program to a concurrent program, so that certain aspects of the program, such as correctness as defined by a property, are preserved by every transformation. Such a proof system can be used specify the operation of a parallelizing compiler by expressing the transformations performed in the compiler in terms of the rules of the proof system. In this way, the parallelization scheme can be described and its correctness is argued in a more organized and well-defined way.

BIBLIOGRAPHY

- [1] Robert H. B. Netzer and Barton P. Miller, “What are race conditions?: Some issues and formalizations,” *ACM Lett. Program. Lang. Syst.*, vol. 1, no. 1, pp. 74–88, 1992.
- [2] Jeremy Manson, William Pugh, and Sarita V. Adve, “The java memory model,” in *POPL ’05: Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, New York, NY, USA, 2005, pp. 378–391, ACM.
- [3] Hans-J. Boehm and Sarita V. Adve, “Foundations of the c++ concurrency memory model,” in *PLDI ’08: Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, New York, NY, USA, 2008, pp. 68–78, ACM.
- [4] Tayfun Elmas, Shaz Qadeer, and Serdar Tasiran, “Goldilocks: a race and transaction-aware java runtime,” in *PLDI ’07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, New York, NY, USA, 2007, pp. 245–255, ACM.
- [5] Dan Marino, Abhayendra Singh, Todd Millstein, Madan Musuvathi, and Satish Narayanasamy, “Drfx: A simple and efficient memory model for concurrent programming languages,” in *PLDI ’10: Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation*, New York, NY, USA, 2010, ACM.
- [6] Brandon Lucia, Luis Ceze, Karin Strauss, Shaz Qadeer, and Hans-J. Boehm, “Conflict exceptions: Simplifying concurrent language semantics with precise hardware exceptions for data-races,” in *ISCA ’10: Proceedings of the 37th International Symposium on Computer Architecture*, New York, NY, USA, 2010, ACM.
- [7] T. Wilkinson., “Kaffe: A JIT and interpreting virtual machine to run Java code. <http://www.transvirtual.com/>,” 1998.

- [8] Eli Pozniansky and Assaf Schuster, “Multirace: efficient on-the-fly data race detection in multithreaded c++ programs: Research articles,” *Concurr. Comput. : Pract. Exper.*, vol. 19, no. 3, pp. 327–340, 2007.
- [9] Cormac Flanagan and Stephen N. Freund, “Fasttrack: efficient and precise dynamic race detection,” in *PLDI '09: Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, New York, NY, USA, 2009, pp. 121–133, ACM.
- [10] Nir Shavit and Dan Touitou, “Software transactional memory,” in *Symposium on Principles of Distributed Computing*, 1995, pp. 204–213.
- [11] Mayur Naik, Alex Aiken, and John Whaley, “Effective static race detection for Java,” in *PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, New York, NY, USA, 2006, pp. 308–319, ACM Press.
- [12] Martin Abadi, Cormac Flanagan, and Stephen N. Freund, “Types for safe locking: Static race detection for Java,” *ACM Trans. Program. Lang. Syst.*, vol. 28, no. 2, pp. 207–255, 2006.
- [13] Leslie Lamport, “How to make a multiprocessor computer that correctly executes multiprocess program,” *IEEE Trans. Comput.*, vol. 28, no. 9, pp. 690–691, 1979.
- [14] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson, “Eraser: a dynamic data race detector for multithreaded programs,” *ACM Trans. Comput. Syst.*, vol. 15, no. 4, pp. 391–411, 1997.
- [15] Christoph von Praun and Thomas R. Gross, “Object race detection,” in *OOPSLA '01: Proceedings of the 16th ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*, New York, NY, USA, 2001, pp. 70–82, ACM Press.

- [16] Yuan Yu, Tom Rodeheffer, and Wei Chen, “Racetrack: efficient detection of data race conditions via adaptive tracking,” in *SOSP '05: Proceedings of the twentieth ACM symposium on Operating systems principles*, New York, NY, USA, 2005, pp. 221–234, ACM Press.
- [17] M. Christiaens and K. De Bosschere, “TRaDe: Data race detection for Java,” in *Proceedings of the International Conference on Computational Science - ICCS2001*, V. Alexandrov, J. Dongarra, B. Juliano, R. Renner, and C. Tan, Eds., San Francisco, May 2001, number 2074, pp. 761–770, Springer Verlag.
- [18] Friedemann Mattern, “Virtual time and global states of distributed systems,” in *Parallel and Distributed Algorithms: proceedings of the International Workshop on Parallel and Distributed Algorithms*. 1988.
- [19] Jerry J. Harrow, “Runtime checking of multithreaded applications with visual threads,” in *Proceedings of the 7th International SPIN Workshop on SPIN Model Checking and Software Verification*, London, UK, 2000, pp. 331–342, Springer-Verlag.
- [20] Guang-Ien Cheng, Mingdong Feng, Charles E. Leiserson, Keith H. Randall, and Andrew F. Stark, “Detecting data races in Cilk programs that use locks,” in *SPAA 98: Annual ACM Symposium on Parallel Algorithms and Architectures*, 1998.
- [21] Michiel Ronsse and Koen De Bosschere, “RecPlay: a fully integrated practical record/replay system,” *ACM Trans. Comput. Syst.*, vol. 17, no. 2, pp. 133–152, 1999.
- [22] Edmond Schonberg, “On-the-fly detection of access anomalies,” in *Proceedings of the ACM SIGPLAN '89 Conference on Programming Language Design and Implementation*, Portland, OR, June 1989, vol. 24, pp. 285–297.
- [23] Jong-Deok Choi, Keunwoo Lee, Alexey Loginov, Robert O’Callahan, Vivek Sarkar, and Manu Sridharan, “Efficient and precise datarace detection for multithreaded object-oriented programs,” in *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, New York, NY, USA, 2002, pp. 258–269, ACM Press.

- [24] M. Erkan Keremoglu, Serdar Tasiran, and Tayfun Elmas, “A classification of concurrency bugs in java benchmarks by developer intent,” in *PADTAD '06: Proceedings of the 2006 workshop on Parallel and distributed systems: testing and debugging*, New York, NY, USA, 2006, pp. 23–26, ACM.
- [25] James R. Larus and Ravi Rajwar, *Transactional Memory*, Morgan & Claypool, 2006.
- [26] Tim Harris, Simon Marlow, Simon Peyton-Jones, and Maurice Herlihy, “Composable memory transactions,” in *PPoPP '05: Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, New York, NY, USA, 2005, pp. 48–60, ACM Press.
- [27] Dan Grossman, Jeremy Manson, and William Pugh, “What do high-level memory models mean for transactions?,” in *MSPC '06: Proceedings of the 2006 workshop on Memory system performance and correctness*, New York, NY, USA, 2006, pp. 62–69, ACM Press.
- [28] Tim Harris and Keir Fraser, “Language support for lightweight transactions,” in *OOPSLA '03: Proceedings of the 18th annual ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications*, New York, NY, USA, 2003, pp. 388–402, ACM Press.
- [29] M. Herlihy and J. E. B. Moss, “Transactional memory: Architectural support for lock-free data structures,” in *Proceedings of the Twentieth Annual International Symposium on Computer Architecture*, 1993.
- [30] M. Herlihy, “Herlihy, m. SXM1.1: Software transactional memory package for c#. Tech. rep., Brown University & Microsoft Research, May,” 2005.
- [31] Cormac Flanagan and Stephen N. Freund, “The roadrunner dynamic analysis framework for concurrent programs,” 2010.
- [32] S. Browne, J. Dongarra, N. Garner, G. Ho, and P. Mucci, “A portable programming

- interface for performance evaluation on modern processors,” *The International Journal of High Performance Computing Applications*, vol. 14, no. 3, pp. 189–204, Fall 2000.
- [33] Benjamin Hindman and Dan Grossman, “Atomicity via source-to-source translation,” in *MSPC '06: Proceedings of the 2006 workshop on Memory system performance and correctness*, New York, NY, USA, 2006, pp. 82–91, ACM Press.
- [34] Tayfun Elmas, Serdar Tasiran, and Shaz Qadeer, “VyrD: verifying concurrent programs by runtime refinement-violation detection,” in *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, New York, NY, USA, 2005, pp. 27–37, ACM Press.
- [35] Edward A. Ashcroft, “Proving assertions about parallel programs,” *J. Comput. Syst. Sci.*, vol. 10, no. 1, pp. 110–135, 1975.
- [36] Maurice P. Herlihy and Jeannette M. Wing, “Linearizability: a correctness condition for concurrent objects,” *ACM Trans. Program. Lang. Syst.*, vol. 12, no. 3, pp. 463–492, 1990.
- [37] A. M. Turing, “On computable numbers, with an application to the *Entscheidungsproblem*,” *Proceeding of the London Mathematical Society, series 2*, vol. 42, pp. 230–265, 1936–1937.
- [38] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata, “Extended static checking for java,” in *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, New York, NY, USA, 2002, pp. 234–245, ACM Press.
- [39] Jean-Christophe Filliâtre, “Verification of non-functional programs using interpretations in type theory,” *J. Funct. Program.*, vol. 13, no. 4, pp. 709–745, 2003.
- [40] Jean-Christophe Filliâtre and Claude Marché, “Caduceus: Multi-prover verification of c programs,” in *ICFEM*, 2004, pp. 15–29.

- [41] Mike Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino, “Boogie: A modular reusable verifier for object-oriented programs,” *FMCO*, 2005.
- [42] Susan Owicki and David Gries, “Verifying properties of parallel programs: an axiomatic approach,” *Commun. ACM*, vol. 19, no. 5, pp. 279–285, 1976.
- [43] C. B. Jones, *Development Methods for Computer Programs including a Notion of Interference*, Ph.D. thesis, Oxford University, June 1981, Printed as: Programming Research Group, Technical Monograph 25.
- [44] Peter W. O’Hearn, Hongseok Yang, and John C. Reynolds, “Separation and information hiding,” in *POPL ’04: Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, New York, NY, USA, 2004, pp. 268–280, ACM.
- [45] Peter W. O’Hearn, “Resources, concurrency, and local reasoning,” *Theor. Comput. Sci.*, vol. 375, no. 1-3, pp. 271–307, 2007.
- [46] Stephen Brookes, “A semantics for concurrent separation logic,” *Theor. Comput. Sci.*, vol. 375, no. 1-3, pp. 227–270, 2007.
- [47] Richard J. Lipton, “Reduction: a method of proving properties of parallel programs,” *Commun. ACM*, vol. 18, no. 12, pp. 717–721, 1975.
- [48] Cormac Flanagan and Shaz Qadeer, “Types for atomicity,” in *TLDI ’03: Proceedings of the 2003 ACM SIGPLAN international workshop on Types in languages design and implementation*, New York, NY, USA, 2003, pp. 1–12, ACM.
- [49] Liqiang Wang and Scott D. Stoller, “Runtime analysis of atomicity for multi-threaded programs,” *IEEE Transactions on Software Engineering*, vol. 32, pp. 93–110, Feb. 2006.
- [50] Stephen N. Freund and Shaz Qadeer, “Exploiting purity for atomicity,” *IEEE Trans. Softw. Eng.*, vol. 31, no. 4, pp. 275–291, 2005, Member-Cormac Flanagan.

- [51] Liqiang Wang and Scott D. Stoller, “Static analysis for programs with non-blocking synchronization,” in *ACM SIGPLAN 2005 Symposium on Principles and Practice of Parallel Programming (PPoPP)*. June 2005, ACM Press.
- [52] R. DeLine and K. R. M. Leino., “Boogiepl: A typed procedural language for checking object-oriented programs,” *Technical Report MSR-TR-2005-70, Microsoft Research*, 2005.
- [53] Leonardo de Moura and Nikolaj Bjørner, *Z3: An Efficient SMT Solver*, vol. 4963/2008 of *Lecture Notes in Computer Science*, pp. 337–340, Springer Berlin, April 2008.
- [54] Tony Andrews, Shaz Qadeer, Sriram K. Rajamani, Jakob Rehof, and Yichen Xie, “Zing: A model checker for concurrent software.,” in *CAV*, 2004, pp. 484–487.
- [55] E. Kindler, “Safety and liveness properties: A survey,” *Bulletin of the European Association for Theoretical Computer Science*, vol. 53, pp. 268–272, 1994.
- [56] Bowen Alpern and Fred B. Schneider, “Defining liveness,” Tech. Rep., Ithaca, NY, USA, 1984.
- [57] James H. Andrews, “Executing formal specifications by translation to higher order logic programming,” in *TPHOLs ’97: Proceedings of the 10th International Conference on Theorem Proving in Higher Order Logics*, London, UK, 1997, pp. 17–32, Springer-Verlag.
- [58] Markus Müller-Olm and Helmut Seidl, “A note on karr’s algorithm.,” in *ICALP*, 2004, pp. 1016–1028.
- [59] Markus Müller-Olm, Oliver Rüthing, and Helmut Seidl, “Checking herbrand equalities and beyond.,” in *VMCAI*, 2005, pp. 79–96.
- [60] S. Gulwani and A. Tiwari, “Assertion checking over combined abstraction of linear arithmetic and uninterpreted functions,” in *European Symp. on Programming, ESOP 2006*, P. Sestoft, Ed., 2006, vol. 3924 of *LNCS*, pp. 279–293.

- [61] Sumit Gulwani and Ashish Tiwari, “Assertion checking unified,” in *The 8th International Conference on Verification, Model Checking and Abstract Interpretation*. Jan. 2007, Springer.
- [62] Martin Abadi and Leslie Lamport, “The existence of refinement mappings,” *Theor. Comput. Sci.*, vol. 82, no. 2, pp. 253–284, 1991.
- [63] Martin Abadi and Gordon Plotkin, “A logical view of composition and refinement,” in *POPL ’91: Proceedings of the 18th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, New York, NY, USA, 1991, pp. 323–332, ACM Press.
- [64] L. Lamport, “Refinement in state-based formalisms,” 1996.
- [65] Marco Antonio Barbosa, “A refinement calculus for software components and architectures,” in *ESEC/FSE-13: Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, New York, NY, USA, 2005, pp. 377–380, ACM Press.
- [66] K. Mani Chandy, *Parallel program design: a foundation*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1988.
- [67] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith, “Progress on the state explosion problem in model checking,” in *Informatics - 10 Years Back. 10 Years Ahead.*, London, UK, 2001, pp. 176–194, Springer-Verlag.
- [68] Cormac Flanagan and Patrice Godefroid, “Dynamic partial-order reduction for model checking software,” in *POPL ’05: Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, New York, NY, USA, 2005, pp. 110–121, ACM Press.
- [69] Mayur Naik, Alex Aiken, and John Whaley, “Effective static race detection for java,” *SIGPLAN Not.*, vol. 41, no. 6, pp. 308–319, 2006.

- [70] Cormac Flanagan and Stephen N. Freund, “Type-based race detection for java,” in *PLDI '00: Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, New York, NY, USA, 2000, pp. 219–232, ACM Press.
- [71] Cormac Flanagan and Shaz Qadeer, “A type and effect system for atomicity,” in *PLDI '03: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, New York, NY, USA, 2003, pp. 338–349, ACM Press.
- [72] Willem-Paul de Roever, “The quest for compositionality—a survey of assertion-based proof systems for concurrent programs, part i: Concurrency based on shared variables,” in *Proc. of the IFIP Working Conference “The role of abstract models in computer science”*. 1985, North-Holland.
- [73] S. Chaki, E. Clarke, A. Groce, J. Ouaknine, O. Strichman, and K. Yorav, “Efficient verification of sequential and concurrent c programs,” *Form. Methods Syst. Des.*, vol. 25, no. 2-3, pp. 129–166, 2004.
- [74] Cormac Flanagan, Stephen N. Freund, Shaz Qadeer, and Sanjit A. Seshia, “Modular verification of multithreaded programs,” *Theor. Comput. Sci.*, vol. 338, no. 1-3, pp. 153–183, 2005.
- [75] C. A. R. Hoare, “An axiomatic basis for computer programming,” *Commun. ACM*, vol. 12, no. 10, pp. 576–580, 1969.
- [76] R. W. Floyd, “Assigning meanings to programs,” *Proc. Symp. Appl. Math.*, vol. 19, pp. 19–31, 1967.
- [77] V. R. Pratt, “Semantical considerations on floyd-hoare logic,” in *Proc. 17th Symp. Found. Comput. Sci.*, 1976, IEEE, pp. 109 – 121.
- [78] Qiwen Xu, Willem P. de Roever, and Jifeng He, “The rely-guarantee method for verifying shared variable concurrent programs,” *Formal Aspects of Computing*, vol. 9, no. 2, pp. 149–174, 1997.

-
- [79] Edsger Wybe Dijkstra, *A Discipline of Programming*, Prentice Hall PTR, Upper Saddle River, NJ, USA, 1997.
- [80] F. S. de Boer, U. Hannemann, and W.-P. de Roever, “A compositional proof system for shared variable concurrency,” in *FME’97: Industrial Applications and Strengthened Foundations of Formal Methods*. 1997, vol. 1313, pp. 515–532, Springer-Verlag.
- [81] Jayadev Misra and K. Mani Chandy, “Proofs of networks of processes,” *IEEE Trans. Software Eng.*, vol. 7, no. 4, pp. 417–426, 1981.
- [82] Martin Abadi and Leslie Lamport, “Conjoining specifications,” *ACM Trans. Program. Lang. Syst.*, vol. 17, no. 3, pp. 507–535, 1995.
- [83] John C. Reynolds, “Separation logic: A logic for shared mutable data structures,” in *LICS ’02: Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science*, Washington, DC, USA, 2002, pp. 55–74, IEEE Computer Society.
- [84] Viktor Vafeiadis and Matthew Parkinson, “A marriage of rely/guarantee and separation logic,” in *In 18th CONCUR*. 2007, pp. 256–271, Springer.
- [85] Xinyu Feng, “Local rely-guarantee reasoning,” *SIGPLAN Not.*, vol. 44, no. 1, pp. 315–327, 2009.
- [86] Mike Dodds, Xinyu Feng, Matthew Parkinson, and Viktor Vafeiadis, “Deny-guarantee reasoning,” in *In ESOP’09: European Symposium on Programming, volume 5502 of LNCS*. 2009, pp. 363–377, Springer.
- [87] Viktor Vafeiadis, Maurice Herlihy, Tony Hoare, and Marc Shapiro, “Proving correctness of highly-concurrent linearisable objects,” in *PPoPP ’06 ACM Symposium on Principles and practice of parallel programming*, New York, NY, USA, 2006, pp. 129–136, ACM.
- [88] Cristiano Calcagno, Matthew Parkinson, and Viktor Vafeiadis, “Modular safety checking for fine-grained concurrency,” in *In SAS. LNCS*. 2007, The MIT Press.

- [89] J. W. Coleman and C. B. Jones, “Guaranteeing the soundness of rely/guarantee rules,” *Journal of Logic and Computation*, vol. 17, no. 4, pp. 807–841, 2007.
- [90] Bart Jacobs, Jan Smans, Frank Piessens, and Wolfram Schulte, “A simple sequential reasoning approach for sound modular verification of mainstream multithreaded programs,” *Electron. Notes Theor. Comput. Sci.*, vol. 174, no. 9, pp. 23–47, 2007.
- [91] P. Cousot and R. Cousot, “On abstraction in software verification,” in *Proceedings of the 14th International Conference on Computer Aided Verification, CAV 2002*, E. Brinksma and K.G. Larsen, Eds. 27–31 July 2002, Copenhagen, Denmark, LNCS 2404, pp. 37–56, Springer-Verlag Berlin Heidelberg.
- [92] Edmund M. Clarke, Orna Grumberg, and David E. Long, “Model checking and abstraction,” *ACM Transactions on Programming Languages and Systems*, vol. 16, no. 5, pp. 1512–1542, September 1994.
- [93] Patrick Cousot and Radhia Cousot, “Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints,” in *POPL ’77: Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, New York, NY, USA, 1977, pp. 238–252, ACM.
- [94] Cormac Flanagan and Shaz Qadeer, “Predicate abstraction for software verification,” in *POPL ’02: Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, New York, NY, USA, 2002, pp. 191–202, ACM Press.
- [95] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith, “Counterexample-guided abstraction refinement,” in *Computer Aided Verification*, 2000, pp. 154–169.
- [96] Stephen Freund and Shaz Qadeer, “Checking concise specifications for multithreaded software,” *Journal of Object Technology*, vol. 3, no. 6, pp. 81–101, 2004, Special issue: ECOOP 2003 workshop on FTfJP.

-
- [97] C. Flanagan and S. Qadeer, “Transactions for software model checking,” *Electronic Notes in Theoretical Computer Science*, vol. 89, 2003.
 - [98] Herbert B. Enderton, *A Mathematical Introduction to Logic, Second Edition*, Academic Press, 2 edition, 2001.
 - [99] Tayfun Elmas, Shaz Qadeer, and Serdar Tasiran, “A calculus of atomic actions,” in *POPL ’09: ACM Symposium on Principles of Programming Languages*, New York, NY, USA, 2009, ACM.
 - [100] Byron Cook, Andreas Podelski, and Andrey Rybalchenko, “Termination proofs for systems code,” in *PLDI ’06: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, New York, NY, USA, 2006, pp. 415–426, ACM.
 - [101] K. Rustan M. Leino, “Dafny: An automatic program verifier for functional correctness,” 2010.
 - [102] Aaron R. Bradley, Zohar Manna, and Henny B. Sipma, “What ’s decidable about arrays?,” in *Verification, Model Checking, and Abstract Interpretation: 7th International Conference, (VMCAI)*, Charleston, SC, January 2006, vol. 3855, pp. 427–442, Springer Verlag.
 - [103] Seungjoon Park and David L. Dill, “Protocol verification by aggregation of distributed transactions,” in *CAV ’96: The International Conference on Computer Aided Verification*, London, UK, 1996, pp. 300–310, Springer-Verlag.
 - [104] Viktor Vafeiadis, “Shape-value abstraction for verifying linearizability,” in *VMCAI ’09: The International Conference on Verification, Model Checking, and Abstract Interpretation*, 335–348, Springer-Verlag.
 - [105] Tayfun Elmas, Ali Sezgin, Serdar Tasiran, and Shaz Qadeer, “An annotation assistant for interactive debugging of programs with common synchronization idioms,” in

- Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging*, July 2009.
- [106] Wim H. Hesselink, “Eternity variables to prove simulation of specifications,” *ACM Trans. Comput. Logic*, vol. 6, no. 1, pp. 175–201, 2005.
- [107] Bengt Jonsson, Amir Pnueli, and Camilla Rump, “Proving refinement using transduction,” *Distrib. Comput.*, vol. 12, no. 2-3, pp. 129–149, 1999.
- [108] Yonit Kesten, Amir Pnueli, Elad Shahar, and Lenore D. Zuck, “Network invariants in action,” in *CONCUR ’02: The 13th International Conference on Concurrency Theory*, London, UK, 2002, pp. 101–115, Springer-Verlag.
- [109] Danny Hendler, Nir Shavit, and Lena Yerushalmi, “A scalable lock-free stack algorithm,” in *SPAA ’04: ACM symposium on Parallelism in algorithms and architectures*, New York, NY, USA, 2004, pp. 206–215, ACM.
- [110] H. Gao, J. F. Groote, and W. H. Hesselink, “Lock-free dynamic hash tables with open addressing,” *Distrib. Comput.*, vol. 18, no. 1, pp. 21–42, 2005.
- [111] Robert Colvin, Lindsay Groves, Victor Luchangco, and Mark Moir, “Formal verification of a lazy concurrent list-based set algorithm,” in *CAV*, 2006, pp. 475–488.
- [112] Daphna Amit, Noam Rinetzky, Thomas W. Reps, Mooly Sagiv, and Eran Yahav, “Comparison under abstraction for verifying linearizability,” in *CAV*, 2007, pp. 477–490.
- [113] Lindsay Groves, “Verifying michael and scott’s lock-free queue algorithm using trace reduction,” in *CATS ’08: Symposium on Computing: the Australasian theory*, Darlinghurst, Australia, 2008, pp. 133–142, Australian Computer Society, Inc.
- [114] Cormac Flanagan and James B. Saxe, “Avoiding exponential explosion: generating compact verification conditions,” in *POPL ’01: Proceedings of the 28th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, New York, NY, USA, 2001, pp. 193–205, ACM Press.

- [115] K. Rustan M. Leino and Philipp R  mmer., “The boogie 2 type system: Design and verification condition generation,” *Draft, Microsoft Research*, 2009.
- [116] John MacCormick, Nick Murphy, Marc Najork, Chandramohan A. Thekkath, and Lidong Zhou, “Boxwood: abstractions as the foundation for storage infrastructure,” in *OSDI’04: Proceedings of the 6th conference on Symposium on Operating Systems Design & Implementation*, Berkeley, CA, USA, 2004, pp. 8–8, USENIX Association.
- [117] Douglas C. Schmidt and Stephen D. Huston, *C++ Network Programming: Resolving Complexity Using Ace and Patterns (C++ in-Depth Series)*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.
- [118] Orran Krieger, Michael Stumm, Ron Unrau, and Jonathan Hanna, “A fair fast scalable reader-writer lock,” in *ICPP ’93: The International Conference on Parallel Processing*, Washington, DC, USA, 1993, pp. 201–204, IEEE Computer Society.
- [119] Maged M. Michael and Michael L. Scott, “Simple, fast, and practical non-blocking and blocking concurrent queue algorithms,” in *PODC ’96: ACM symposium on Principles of distributed computing*, New York, NY, USA, 1996, pp. 267–275, ACM.
- [120] Cormac Flanagan and K. Rustan M. Leino, “Houdini, an annotation assistant for esc/java,” in *FME ’01: Proceedings of the International Symposium of Formal Methods Europe on Formal Methods for Increasing Software Productivity*, London, UK, 2001, pp. 500–517, Springer-Verlag.
- [121] Tayfun Elmas, Serdar Tasiran, and Shaz Qadeer, “Vyrd: verifying concurrent programs by runtime refinement-violation detection,” in *PLDI ’05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, New York, NY, USA, 2005, pp. 27–37, ACM Press.
- [122] William N. Scherer, III, Doug Lea, and Michael L. Scott, “Scalable synchronous queues,” in *PPoPP ’06: Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, New York, NY, USA, 2006, pp. 147–156, ACM.

- [123] Leslie Lamport, “A new solution of dijkstra’s concurrent programming problem,” *Commun. ACM*, vol. 17, no. 8, pp. 453–455, 1974.
- [124] Maurice Herlihy, Victor Luchangco, and Mark Moir, “Obstruction-free synchronization: Double-ended queues as an example,” in *ICDCS ’03: Proceedings of the 23rd International Conference on Distributed Computing Systems*, Washington, DC, USA, 2003, p. 522, IEEE Computer Society.
- [125] R.K.Treiber, “Systems programming: Coping with parallelism. rj5118,” April 1986.
- [126] Maged M. Michael, “Hazard pointers: Safe memory reclamation for lock-free objects,” *IEEE Trans. Parallel Distrib. Syst.*, vol. 15, no. 6, pp. 491–504, 2004.
- [127] Matthew Parkinson, Richard Bornat, and Peter O’Hearn, “Modular verification of a non-blocking stack,” *SIGPLAN Not.*, vol. 42, no. 1, pp. 297–302, 2007.
- [128] Kim S. Larsen, Eljas Soisalon-soininen, and Peter Widmayer, “Relaxed balance through standard rotations,” in *In Fifth International Workshop on Algorithms and Data Structures*. 1997, pp. 450–461, Springer-Verlag.
- [129] Shuvendu Lahiri and Shaz Qadeer, “Back to the future: revisiting precise program verification using smt solvers,” in *POPL ’08: ACM symposium on Principles of programming languages*, New York, NY, USA, 2008, pp. 171–182, ACM.
- [130] Trevor Jim, J. Greg Morrisett, Dan Grossman, Michael W. Hicks, James Cheney, and Yanling Wang, “Cyclone: A safe dialect of c,” in *ATEC ’02: Proceedings of the General Track of the annual conference on USENIX Annual Technical Conference*, Berkeley, CA, USA, 2002, pp. 275–288, USENIX Association.
- [131] Cormac Flanagan and Stephen N. Freund, “Atomizer: A dynamic atomicity checker for multithreaded programs,” *Sci. Comput. Program.*, vol. 71, no. 2, pp. 89–109, 2008.
- [132] Robert L. Bocchino Jr., Vikram S. Adve, Danny Dig, Sarita V. Adve, Stephen Heumann, Rakesh Komuravelli, Jeffrey Overbey, Patrick Simmons, Hyojin Sung, and

- Mohsen Vakilian, “A type and effect system for deterministic parallel java,” in *OOP-SLA*, 2009, pp. 97–116.
- [133] Martin C. Rinard and Pedro C. Diniz, “Commutativity analysis: a new analysis framework for parallelizing compilers,” in *PLDI '96: Proceedings of the ACM SIGPLAN 1996 conference on Programming language design and implementation*, New York, NY, USA, 1996, pp. 54–67, ACM.
- [134] William M. Pottenger, “The role of associativity and commutativity in the detection and transformation of loop-level parallelism,” in *ICS '98: Proceedings of the 12th international conference on Supercomputing*, New York, NY, USA, 1998, pp. 188–195, ACM.
- [135] Farhana Aleen and Nathan Clark, “Commutativity analysis for software parallelization: letting program transformations see the big picture,” in *ASPLOS '09: Proceeding of the 14th international conference on Architectural support for programming languages and operating systems*, New York, NY, USA, 2009, pp. 241–252, ACM.
- [136] Eric Koskinen, Matthew Parkinson, and Maurice Herlihy, “Coarse-grained transactions,” in *POPL '10: Proceedings of the 37th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, New York, NY, USA, 2010, pp. 19–30, ACM.
- [137] Milind Kulkarni, Keshav Pingali, Bruce Walter, Ganesh Ramanarayanan, Kavita Bala, and L. Paul Chew, “Optimistic parallelism requires abstractions,” in *PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, New York, NY, USA, 2007, pp. 211–222, ACM.

VITA

TAYFUN ELMAS was born in Fethiye (Muğla), Turkey on November 6, 1981. He received his B.S. in Computer Engineering from Ege University (İzmir, Turkey) in September 2003, and M.S. in Electrical and Computer Engineering from Koç University (İstanbul, Turkey) in December 2005. Since January 2006, Tayfun has been pursuing his Ph.D. study in Computer Engineering, advised by Asst. Prof. Serdar Taşiran (Koç University) and Dr. Shaz Qadeer (Microsoft Research). During his M.S. and Ph.D. studies, he has been a research and teaching assistant at the Center for Advanced Design Technologies (CADT) at Koç University. His research interests include formal methods and tools for design, analysis, and verification of concurrent software. Tayfun's recent work appeared at premier programming languages conferences (PLDI, POPL), workshops (RV, FATES, PADTAD), and national conferences (UYMS, YKGS). His research is supported by the Scientific and Technological Research Council of Turkey (TÜBİTAK), the Turkish Academy of Sciences (TÜBA) and Microsoft Research. The GOLDILOCKS project he primarily involved was selected for publication in the SIGPLAN CACM Research Highlights (nominated in January 2009). Tayfun was an intern at Microsoft Research (Redmond, WA) during the summer of 2005 and 2007.