Mechanising an algebraic rely-guarantee refinement calculus

Diego Machado Dias

School of Computing Science
Newcastle University

This dissertation is submitted for the degree of
Doctor of Philosophy

October 2017

Abstract

Despite rely-guarantee (RG) being a well-studied program logic established in the 1980s, it was not until recently that researchers realised that rely and guarantee conditions could be treated as independent programming constructs. This recent reformulation of RG paved the way to algebraic characterisations which have helped to better understand the difficulties that arise in the practical application of this development approach.

The primary focus of this thesis is to provide automated tool support for a rely-guarantee refinement calculus proposed by Hayes et. al., where rely and guarantee are defined as independent commands. Our motivation is to investigate the application of an algebraic approach to derive concrete examples using this calculus. In the course of this thesis, we locate and fix a few issues involving the refinement language, its operational semantics and preexisting proofs. Moreover, we extend the refinement calculus of Hayes et. al. to cover indexed parallel composition, non-atomic evaluation of expressions within specifications, and assignment to indexed arrays. These extensions are illustrated via concrete examples.

Special attention is given to design decisions that simplify the application of the mechanised theory. For example, we leave part of the design of the expression language on the hands of the user, at the cost of the requiring the user to define the notion of undefinedness for unary and binary operators; and we also formalise a notion of indexed parallelism that is parametric on the type of the indexes, this is done deliberately to simplify the formalisation of algorithms. Additionally, we use stratification to reduce the number of cases in in simulation proofs involving the operational semantics. Finally, we also use the algebra to discuss the role of types in program derivation.

To my parents, José and Raquel, and my beloved aunt “Gel” (*in memoriam*)

Acknowledgements

First and foremost, I wish to thank to my supervisor, Leo Freitas, for his continued support, encouragement and many constructive comments on the research described in this thesis. I am indebted to him not only for his advices on my research, but for the very opportunity of coming to Newcastle to do a PhD. I want to extend special thanks to my co-supervisor, Prof. Cliff Jones, for his invaluable suggestions in this work, and the opportunity of attending his course on operational semantics.

I thank Prof. Ian Hayes for the numerous discussions about the operational semantics of the refinement calculus mechanised in this thesis, and discussions involving the representation of the refinement language. Additionally, Ian was always keen to discuss failed proof attempts and provide insight in such situations.

My examiners, Brijesh Dongol and Paolo Zuliani, contributed valuable comments and suggestions, which helped to improve this thesis. I also wish to thank people who discussed and challenged my design decisions in the mechanisation: Andrius Velikys, Leo Freitas and Frank Zeyda. Additionally, I wish to thank Ani Bhattacharyya for discussions about simulation, and the users from the Isabelle mailing list for quick and useful responses.

I want to especially thank a couple of friends who helped me during difficult times of the journey. Adolfo Duran put me back on track when I felt desolated with the issues involving the characterisation of the programming language. Rosemeire Fiaccone helped me to deal with the little stones in the journey, and helped me to broaden my horizon with respect to research in other disciplines.

It is my great pleasure to thank my fellow PhD students and the members of the CSR group for the warm working environment provided by them. Special thanks go to David, Sami, Patrick, Maryam, Ehsan, Pablo, Razgar, Yu, Beibei, Xingjie, Andrius, Ilya, and Ayad for saving me from plain boredom and starvation. I also want to thank friends with whom I shared some of my best moments in England: Tim Pinnington, Yana Demyanenko, Paola Fognani, Suzi Ribeiro and Gilberto Pires, and also those from overseas who have never failed to provide support and encouragement: Wilson Pires, Arytan Lemos, Vinicius Pinto and Robson Silva.

A special thanks goes to my parents, José and Raquel, for their love, encouragement, patience and continued support which can never be fully acknowledged.

This work was funded by a scholarship provided by the School of Computing Science.

Contents

List of Figures **xv**

List of Tables **xvii**

1 Introduction **1**
1.1 Motivation . . . 1
1.2 Formal verification and refinement . . . 3
1.3 Need for tool support . . . 4
1.4 Thesis proposition . . . 5
1.5 Contributions . . . 6
1.6 Literature review . . . 6
1.6.1 Program logics . . . 6
1.6.2 Refinement-based approaches . . . 11
1.6.3 Data refinement . . . 12
1.6.4 Isabelle/HOL . . . 13
1.7 Structure of the thesis . . . 16

2 Foundation **19**
2.1 Programming language . . . 21
2.2 Formalisation of the state (σ) . . . 23
2.3 Encoding the programming language . . . 25
2.3.1 Shallow and deep-embedding . . . 25
2.3.2 Expression language . . . 26
2.3.3 Definedness . . . 27
2.3.4 Encoding RG-WSL . . . 28
2.4 Relations . . . 33
2.4.1 Predicates . . . 34

2.4.2 Satisfiability . . . 35
2.4.3 Post state notation . . . 35
2.4.4 Wellfounded relations . . . 35
2.5 Relational interpretation of expressions . . . 36
2.6 Logical interpretation of relations . . . 37
2.6.1 Example: reasoning compositionally about logical interpretation . . . 38
2.7 Operational semantics . . . 39
2.7.1 Expression evaluation . . . 41
2.7.2 Small-step semantics . . . 45
2.7.3 Big-step semantics . . . 51
2.8 Denotational semantics . . . 53
2.9 Refinement . . . 54
2.10 Forward simulation . . . 55
2.11 Unrestricted variables . . . 58
2.12 Discussion and summary of contributions . . . 62
2.12.1 Alternative approaches to formalise states . . . 62
2.12.2 Semantics . . . 64
2.12.3 Well-founded relations . . . 65
2.12.4 Unfair parallelism . . . 65

3 Rely-guarantee refinement calculus 67
3.1 Derived commands . . . 68
3.1.1 Precedence and associativity . . . 70
3.2 Stability . . . 71
3.2.1 Single reference property . . . 72
3.3 Basic refinement laws . . . 73
3.3.1 Associativity, commutativity and distributivity . . . 76
3.3.2 Monotonicity . . . 77
3.3.3 Zeros and units . . . 78
3.3.4 Pre and post-conditioned assumptions . . . 79
3.3.5 Iteration . . . 79
3.3.6 Termination . . . 81
3.4 The guarantee command . . . 85
3.4.1 Properties of strict conjunction . . . 86
3.4.2 Refining the guarantee command . . . 88

3.4.3 Guarantee invariant and frames . . . 90
3.5 The rely command . . . 92
3.5.1 Properties of interference . . . 95
3.5.2 Fundamental properties of rely . . . 95
3.5.3 Refining the rely command . . . 99
3.6 Arranging rely and guarantee commands . . . 101
3.7 Trading postconditions with rely and guarantee . . . 101
3.8 Introducing parallelism . . . 102
3.8.1 Example: nested parallelism . . . 104
3.9 Expressions and tests . . . 106
3.10 Local variables . . . 107
3.10.1 Example: shadowing . . . 108
3.11 Restricting access to variables . . . 109
3.12 Control structures and assignment . . . 112
3.13 Discussion and summary of contributions . . . 115
3.13.1 Intricate aspects of using the R/G refinement calculus . . . 117
3.13.2 Stronger definition of rely command . . . 118

4 Rely-guarantee in Isabelle/HOL 121
4.1 Methodology . . . 122
4.1.1 Naming conventions . . . 122
4.1.2 Encoding lemmas . . . 124
4.1.3 Proof style . . . 126
4.2 Proof engineering . . . 128
4.2.1 Relational operators . . . 128
4.2.2 Single reference property . . . 130
4.2.3 Monotonicity and substitution . . . 132
4.2.4 Shortening proofs . . . 133
4.2.5 Representation issues . . . 134
4.3 Conflicts between semantics and local assumptions . . . 134
4.3.1 Semantic encoding . . . 135
4.3.2 Stratification . . . 136
4.3.3 Justifying design decision . . . 137
4.4 Discussion and summary of contributions . . . 138

5 Extensions to rely-guarantee algebra **141**
5.1 Indexed parallelism . . . 141
5.1.1 Monotonicity and substitution . . . 143
5.1.2 Introducing indexed parallelism . . . 144
5.2 Eguard . . . 147
5.3 Revised abortive conditions . . . 152
5.4 Assignment to indexed arrays . . . 154
5.5 Reachable evaluations . . . 157
5.5.1 Example: parallel assignments . . . 161
5.5.2 Discussion . . . 162
5.6 Sequential laws . . . 164
5.7 Type system . . . 164
5.8 Discussion and summary of contributions . . . 166
5.8.1 Contributions . . . 166
5.8.2 Further extensions . . . 167

6 Applying the refinement calculus **169**
6.1 Typographic conventions . . . 170
6.2 Reading advice . . . 172
6.3 Findp: Sequential . . . 172
6.3.1 Abbreviations . . . 172
6.3.2 Derivation . . . 174
6.3.3 Discussion . . . 180
6.4 Findp: Concurrent . . . 182
6.4.1 Abbreviations . . . 183
6.4.2 Derivation . . . 185
6.4.3 Discussion . . . 195
6.5 Sieve . . . 196
6.5.1 Abbreviations . . . 196
6.5.2 Derivation . . . 196
6.5.3 Discussion . . . 198
6.6 Floyd-Warshall algorithm . . . 199
6.6.1 Abbreviations . . . 201
6.6.2 Derivation . . . 202
6.6.3 Discussion . . . 212

6.7 Discussion . . . 212
6.7.1 Proof metrics . . . 213
6.7.2 Bottlenecks . . . 214

7 Evaluation 217
7.1 Quantitative summary . . . 217
7.1.1 Local assumptions . . . 220
7.1.2 Threats to validity . . . 221
7.2 Lessons learned . . . 223
7.2.1 Isolate concepts . . . 223
7.2.2 Favour usability . . . 224
7.2.3 Benefit from integrated proof tools . . . 225
7.3 Related work . . . 230
7.3.1 Systematic parallel programming . . . 230
7.3.2 Formal analysis of concurrent programs . . . 231
7.3.3 The rely-guarantee method in Isabelle/HOL . . . 232
7.3.4 On the Mechanisation of Rely-Guarantee in Coq . . . 232
7.3.5 Generalised rely-guarantee concurrency . . . 233
7.3.6 An algebra of synchronous atomic steps . . . 234
7.3.7 Algebraic Principles for Program Correctness Tools in Isabelle/HOL . . . 235

8 Conclusion 237
8.1 Summary and contributions . . . 237
8.2 Takeaway message . . . 240
8.3 Limitations . . . 241
8.4 Future work . . . 243

Bibliography 247

A Rely-guarantee algebra in Isabelle/HOL 255
A.1 Mechanisation . . . 255
A.2 Precedence and associativity . . . 256
A.3 Revised paper proofs . . . 257
A.4 Additional material . . . 263
A.4.1 Compiled PDF of the theory . . . 263

A.4.2 Uncountability of RG-WSL 264
A.4.3 Uncountability of the set used to define the rely command 264
A.4.4 Font extension 265

B Summary of laws and definitions used in Chapter 6 **267**

C Applying the refinement algebra (sources) **275**
C.1 Findp Sequential 275
C.1.1 Derivation 275
C.2 Proof obligations 277
C.3 Findp Concurrent 278
C.3.1 Derivation 278
C.3.2 Abbreviations for Findp 283

List of Figures

1.1 Literature review for rely-guarantee refinement calculus 8
2.1 Wide spectrum language 21
2.2 Grammar of expressions 22
2.3 Encoding RG-WSL 29
2.4 Logical connectives and noteworthy relations 34
2.5 Semantic hierarchy 39
2.6 Forward simulation vs. refinement 57
2.7 Unrestricted variables 60
3.1 Derived commands 68
3.2 Precedence and associativity 71
4.1 Trace Semantics 125
4.2 Algebra-Core 125
4.3 Procedural proof example. 127
4.4 Structured proof example. 127
4.5 Semantic encoding for RG-WSL 136
4.6 Nesting control for RG-WSL 136
5.1 Reachable evaluations *versus* history variables. 159
5.2 Graphical representation of relation RelyEF. 163
6.1 Index partitioning for concurrent Findp 182
6.2 Simulating one-based indexing from zero-based indexing. 203
6.3 Stability of S_k 209
7.1 Hasse diagram of theories. 218
8.1 Dekker's algorithm for critical region 242

List of Tables

1.2 Appendices 18
2.1 Precedence and associativity for RG-WSL 33
3.1 Precedence and associativity for derived commands 71
3.2 Summary of novel laws and definitions 116
4.2 Naming convention 123
6.1 Abbreviations for sequential Findp (Part I) 173
6.2 Abbreviations for sequential Findp (Part II) 174
6.3 Abbreviations for concurrent Findp (Part I) 183
6.4 Abbreviations for concurrent Findp (Part II) 184
6.5 Abbreviations for Sieve. 197
6.6 Abbreviations for Floyd-Warshall. 202
6.7 Calibrating weight for measuring total of proof obligations 213
6.8 Proof metrics for mechanised examples 214
7.1 Quantitative summary per theory file 219
7.2 Classification of local assumptions 220
7.3 Local assumptions 220
7.4 Unproved local assumptions 221
A.1 Precedence and associativity for commands 256
A.2 Precedence and associativity for remaining operators 256
C.1 Abbreviations for sequential and concurrent Findp (side by side) 283

Chapter 1

Introduction

> Our ability to control and predict motion changes from an art to a science when we learn a mathematical theory. Similarly programming changes from an art to a science when we learn to understand programs in the same way we understand mathematical theorems. With a scientific outlook, we change our view of how the world works and what is possible. It is a valuable part of education for everyone.
>
> Eric C. R. Hehner, *A Practical Theory of Programming*, 1993

1.1 Motivation

The popularisation of multi-core and many-core technologies has transformed software landscape into a shared-memory parallel platform [28]. Additionally, many computers are now shipped with graphical boards that provide a highly parallel architecture that can be explored both by graphical and non-graphical applications. Consequently, the parallel programming paradigm is becoming mainstream among programmers, and has been employed to maximise the usage of computing resources available in a platform [28].

Although parallel architectures are mainstream, writing correct parallel algorithms is still a challenging task. The difficulty comes from the inherent non-determinism in applications, which hinders the use of tests to validate these programs. In parallel programs, the number of execution paths grows exponentially with the number of threads and the number of instructions in the threads. Even for a fixed input, every time a parallel program is run, it may exercise an execution path that may not have been considered by the programmer. In this context, runtime errors are more often the rule rather than the exception.

Most programming languages evaluate expressions non-atomically, thus commands such as assignment and the test of boolean expressions in conditionals and loops are not atomic as

they appear to be in single-threaded programs. In general, expression evaluation is translated into a sequence of low-level memory fetches that can be interrupted by the scheduler. For example, a compiler enforcing a left-to-right evaluation might translate $y := (x + x) * z$ as:

```
1: LOAD  AX,  &x          4: LOAD   BX,  &z
2: LOAD  BX, &x           5: MUL    AX, BX
3: ADD   AX, BX           6: STORE  &y,  AX
```

These low-level details can be ignored when developing sequential programs, but their effects are perceived in the development of parallel programs because of the non-atomic evaluation of expressions. In this scenario, expression evaluation does not observe mathematical laws, e.g. if a and b are boolean variables, then $a \wedge b$ may be true even if a and b are never true simultaneously. Other example is the evaluation of arithmetic expressions: if x is a shared variable, the evaluation of $x + x$ may result in an odd number! From the perspective of a developer, this means that replacing $x + x$ by $2{*}x$ in a parallel program will make it more deterministic, while replacing $2{*}x$ by $x + x$ will increase the non-determinism [44]. Even experienced programmers are likely to miss certain interactions between threads that can result of non-atomic evaluation of expressions. Eliminating runtime errors can be quite challenging, especially because it is difficult to reproduce the exact conditions that lead to the manifestation of an error.

For non-critical applications, one may be able to afford trial-and-error to get certain degree of confidence about the behaviour of parallel programs; however, for critical applications where failures can endanger human lives, trial and error is not an option. For these applications, certification guidelines such as DO-178C [36] demand the use of formal methodologies to verify the code or guide the development process.

Parallel programs communicate either via shared variables or via message passing through buffered channels. In this thesis, we depart from Hayes et. al. [48], which delimits the context to shared variable parallelism[1]. This thesis, as most of the formal treatments of concurrency, abstracts away from hardware details, such as the actual capacity of executing programs simultaneously, and consider interleaving as the underlying execution model for parallel programs. In the rest of this thesis we use the term *concurrent* to refer to programs that are composed in parallel, but whose execution is formally modelled through the interleaving of the actions of the individual programs.

[1]Note that it is possible use one model of communication to simulate the other. Departing from shared variable, message passing can be simulated as suggested in [32]; the inverse simulation can by achieved by modelling variables as processes [98].

1.2 Formal verification and refinement

Two major techniques can be used to ensure the correctness of programs: posit-and-prove and program refinement. For the first technique, the implementation of a program has to be available in order to the user to be able to verify if it satisfies a given property; whereas for the second technique, only the specification of a program is required, and the technique allows the user to derive an implementation which is correct-by-construction.

- **Posit-and-prove verification** can be achieved using two major approaches: model checking [56] or theorem proving [91, 60, 88]. Model checkers translate the source code provided by the user into a state-transition automaton, and exhaustively check the transitions to decide if the property given by the user holds. Model checking has the advantage of being automatic once the user has formalised the desired property; moreover, they provide a counter-example if there is an execution path for which the specified property is violated. The main disadvantage of model checkers is that they cannot handle infinite types, thus usually the user has to limit the range of values that variables can take in order to obtain a model that can be verified using this technique. An open-source model checker for analysing concurrent programs is SPIN [56]. It accepts programs written in Promela, a modelling language with a C-like syntax, and accepts properties written in LTL [58]. The infrastructure of SPIN is reused by Java Path Finder, which is a model checker that supports a subset of Java [42].

  The second approach to carry out posit-and-prove verification is to use a theorem prover to assist the application of a *program logic*. By a program logic, we refer to a deductive system that takes programs encoded using a predefined syntax, and offers syntax-directed proof rules to show that a program satisfies a *specification*, usually a pre and a postcondition. Hoare logic [51] is perhaps the most well-known example of a program logic. In this approach, the user starts with a complete program and a specification, and applies syntax-directed proof rules backwards to decompose the verification of the complete program to the verification of subprograms. Correctness proofs are commonly recorded using *proof outlines*, that is, a sequence of assertions interspersed with the commands that compose the implementation. Examples of program logics that handle verification of concurrent programs are the *Owicki-Gries* method [91], *rely-guarantee* [60] and *concurrent separation logic* [88]. These three are briefly discussed in in Section 1.6.1.

- **Program refinement** is used to derive programs that are correct-by-construction. The fundamental insight behind refinement is the extension of the programming language with specification commands allowing programs and specifications to be formalised within a uniform semantic framework. The extended language is generally referred to as a wide-spectrum language. In this approach, a partial relation is used to compare programs[2]. Generally, the relation conveys the intuition of reduction of non-determinism between programs. The user of a refinement theory applies semantic-preserving transformations, known as *refinement laws*, to refine an abstract specification into an implementation. Refinement approaches are discussed in more details in Section 1.6.2.

The work in **this thesis fits into the category of refinement-based approaches**, and is based on the rely-guarantee refinement calculus proposed in [48], where rely and guarantee are introduced as independent commands. The algebraic approach [54, 99] is applied in this work to formalise a rely-guarantee refinement algebra in Isabelle/HOL. In this approach a refinement relation is defined through a set of laws relating programming constructs. The laws that form the basis of the refinement algebra are a subset of those provided in [48, 65], and include a few additional laws to reason about iteration from [5, 49]. These laws characterise refinement as a partial order, and provide a foundation for the refinement algebra, from which more complex laws are derived by the application of simpler ones.

1.3 Need for tool support

The practical application of refinement theories involves discharging proof obligations which are the provisos of the refinement laws. For most cases proof obligations are just tedious, but in some cases these can be complex and intellectually challenging. In theory, a user can carry out refinement of programs in a pen-and-paper style, but since this process is quite error prone, it brings the credibility of using the formal approach into question, especially for large-scale developments [89].

From the perspective of a user of a refinement theory, an interactive theorem prover (ITP) records and maintains proofs. The ITP can be used to suggest the refinement laws available for application at different stages of the development, ensure that proof obligations are properly discharged, and identify derivation steps which are affected when the user adjusts

[2]That is slightly different from verification, where we compare a final implementation directly against a specification. In a refinement language, programs can combine both specification and implementation constructors.

specifications, say as result of the introduction of a new requirement or fixing a specification that was incorrect.

From the perspective of the developer of the refinement theory, the ITP contributes to the confidence that the refinement laws are indeed sound. The theorem prover does not allow the user to get away with a proof without discharging all of its sub-goals, or clarify all the assumptions made for the proof to be completed[3].

For the work carried out in this thesis, we choose Isabelle/HOL as our ITP. The main reasons for this choice are: (i) published evidence of its adequacy to represent a rely-guarantee refinement algebra [18, 3]; (ii) up-to-date documentation [109]; and (iii) friendly support offered by the user list. More on this will be discussed in the literature review on Section 1.6.4.

1.4 Thesis proposition

The primary aim of this thesis is to provide automated theorem proving support for proofs based on the rely-guarantee refinement calculus presented in [48] and study the application of the calculus to derive concrete programs. For that, we formulate and investigate the following research questions.

Q1 Can the calculus in [48] can be given automated theorem proving support via the algebraic approach discussed in [54]?

Q2 Is it possible to extend [48] to mechanically derive concurrent programs involving indexed parallelism and arrays?

Q3 Is it possible to prevent undefined expressions of being introduced in programs while retaining the ability of extending the grammar of expressions on-the-fly?

To investigate these questions we propose the use of a state-of-the-art verification system [109]. While answering these questions we locate errors and omissions in [48], which are discussed and corrected along this thesis. We discuss the design decisions behind our encoding, its advantages and weakness, and investigate the practical aspects involved with the application of the mechanisation.

[3] In practice, Isabelle offers the command *sorry* to allow a user to introduce a theorem without a proof. While this command may be useful during proof exploration, the preparation of formal documents from within Isabelle does not allow the use of *sorried* theorems unless the user explicitly sets a flag to enable the cheating mode. The theories contained in this thesis contain no *sorried* theorems.

1.5 Contributions

The overall contribution of this thesis is the provision of automated theorem proving support for an algebraic approach to concurrent programming based on rely-guarantee. However, the work in this thesis does not limit to the mechanisation of theories found elsewhere. We extend the theory proposed in [48] with indexed parallelism, assignment to indexed arrays, and a value model sufficiently expressive to derive programs that manipulate natural numbers, integers, booleans, sets and arrays. Moreover, we design an extensible expression language that provides a mechanism to decide if expressions are undefined. This mechanism is specially useful in the context of concurrent programs, where undefinedness can arise from interference on shared variables.

Additionally, the mechanised theory from this thesis presents some corrections with respect to that published in [48]; in particular, we identified and fixed issues in the syntax and semantics of the wide-spectrum language, as well as minor slips in a few paper proofs. We also provide an implementation of forward simulation that is more efficient in proofs than that suggested in [48], in the sense that it can be used to reduce the number of cases that has to be considered in inductive proofs using the operational semantics. Finally, we discuss a number of proof engineering decisions that were applied to enhance proof automation.

These contributions are illustrated by mechanically deriving a set of classical examples found in the literature (Findp [48], Sieve [65] and Floyd-Warshall [32]). In order to mechanise these examples, we investigate practical aspects that are not discussed in [48], such as the role of types in derivations, the notion of code to allow syntactic restrictions in the development of programs to be removed at the end of development, and the use of syntactic abstractions to represent non-atomic evaluation via binary relations.

1.6 Literature review

1.6.1 Program logics

Program logics for concurrent programs can be classified in non-compositional or compositional depending if proof of correctness of a program abstracts away from the implementation of its environment or not. Non-compositional methods require the code of the program and its environment prior the verification; thus, they do not serve to assist top-down development. On the other hand, they are very successful to verify legacy code; the reason being that the code of the environment is the most concrete piece of information that the user can aim for in

proofs of correctness. The most well-known of such methods is Owicki-Gries [92], proposed in 1975.

One of the first compositional methods for proving correctness of concurrent programs was rely-guarantee [59, 60], proposed in 1981. It conquers compositionality by abstracting the code of the environment using a binary relation, the *rely* condition. This abstraction allows the developer to assume that the interference caused by the environment will be bounded by the rely relation, even if the code of the environment is yet to be developed. The counterpart of the rely condition is the *guarantee* condition, a relation to abstract the atomic actions of a program, which in turn can be taken as a rely condition by the environment. In rely-guarantee, the general argument for correctness is based in showing that a group of threads can collaborate because they tolerate the interference imposed by each other. The abstraction from actual code of the environment allows this method to be used as a foundation for a refinement calculus for parallel programs.

Notable advance in the research on program logics for concurrent programs was triggered by the development *concurrent separation logic* (CSL) in the early 2000s [88]. CSL is a resource logic that provides the notion of *ownership* to reason in terms of transference of resources between threads and resource-holders. In CSL, the state is composed of a heap and a stack; resources are modelled as data structures on the heap. Supporting CSL is a semantics carefully designed to guarantee that programs verified using this logic are free from data-races and memory faults, such as the notorious *segmentation fault* that occurs when programs try to access an illegal memory location.

Independent of the program logic, the verification of non-trivial examples generally requires the usage of a theorem proof assistant (e.g. Isabelle/HOL [109], PVS [93], etc.) to handle the amount of details involved in the application of proof rules. For certain formalisms, like Hoare logic and a subset of CSL, there are semi-automatic tools that accept proof outlines and use decision procedures to validate them. Examples of such a tool for Hoare logic is Dafny [72], and for concurrent separation logic are Smallfoot [8] and Verifast [95]. To the best of our knowledge, no such a tool is available for rely-guarantee.

Figure 1.1 shows the dependencies between some of the works cited in the literature review and related work (Section 7.3). Next we discuss key aspects of Owicki-Gries, rely-guarantee and separation logic before move the discussion towards refinement-based approaches.

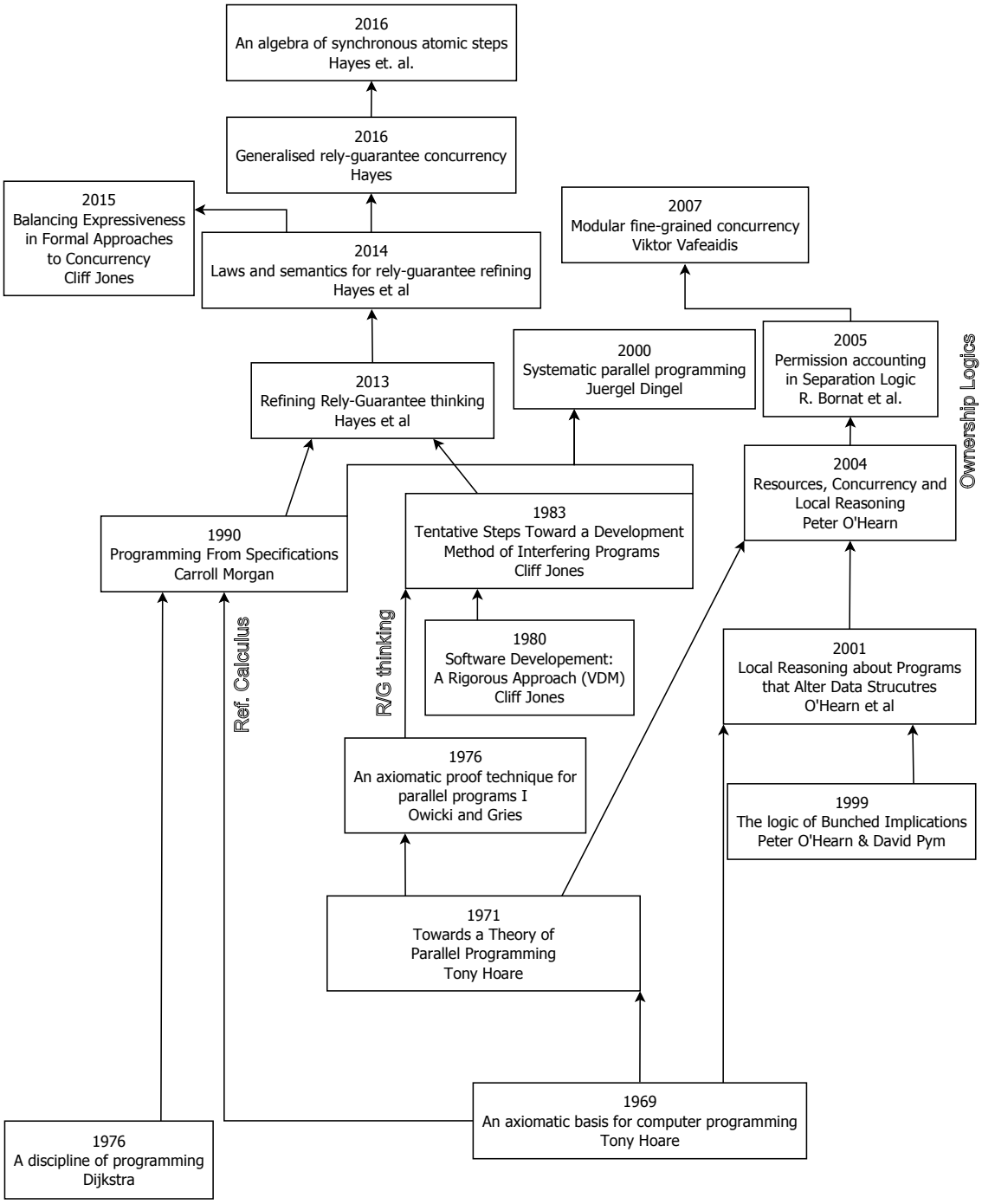


Figure 1.1 Literature review for rely-guarantee refinement calculus

Owicki-Gries

The Owicki-Gries method [91] extends Hoare logic [51, 52] with a parallel composition rule and an **await** statement rule. The parallel composition rule of this method is shown next, where P_i and Q_i are predicates and S_i is a subprogram representing a thread.

$$\frac{\{P_1\}\ S_1\ \{Q_1\} \qquad \{P_2\}\ S_2\ \{Q_2\}}{\{P_1 \wedge P_2\}\ S_1 \parallel S_2\ \{Q_1 \wedge Q_2\}}\ \textsc{Parallel}^*$$

**provided* $\{P_1\}S_1\{Q_1\}$ *and* $\{P_2\}S_2\{Q_2\}$ *are interference-free*

This proof rule requires each thread to be verified in isolation, resulting in a proof outline per thread. At the end of the verification, proof outlines have to be shown to be *interference-free*, i.e. each assertion in each proof outline has to be proved stable (i.e. preserved) under the atomic actions of sibling threads. If after the verification of individual threads the designer discovers that the interference-free test fails, then the intermediate assertions in the proof outline have to be fixed and the verification has to be restarted. In the worst case, the implementation has to be redone and the verification task restarted from scratch.

Rely-guarantee

The essence of the rely-guarantee approach [26] is summarised by its parallel composition rule, shown next. In this rule S_i is a subprogram representing a thread; P is a predicate that denotes a precondition; and R, G_i and Q_i are relations denoting the rely, guarantee and post conditions, respectively. To prove that a parallel composition of two programs S_1 and S_2 is correct, the designer has to prove that each branch tolerates interference from the external environment as well the interference of its sibling. Moreover, each of the threads has to be shown to be correct with respect to its own specification, and the composed behaviour of the threads has to achieve the overall postcondition Q.

$$\frac{\begin{array}{c}\{P, R \cup G_2\}\ S_1\ \{G_1, Q_1\} \\ \{P, R \cup G_1\}\ S_2\ \{G_2, Q_2\} \\ G_1 \cup G_2 \subseteq G \\ P \cap Q_1 \cap Q_2\ (R \cup G_1 \cup G_2)^* \subseteq Q\end{array}}{\{P, R\}\ S_1 \parallel S_2\ \{G, Q\}}\ \textsc{Parallel}$$

Some of the proof rules in [26] assume that pre and postconditions are stable (invariant) under the rely-condition. This is required for the proof of soundness of the proof rules, but it forces postconditions and preconditions to be relatively weaker than the designer might wish. To compensate for this weakness, auxiliary variables can be used. It is noted in [63], however, that unrestricted use of auxiliary variables can invalidate the inductive proof of soundness for the parallel composition rule, as well as as damage the process of *data refinement*. A strategy adopted in [32] and further discussed in [96] to minimise the need for auxiliary variables is to use programming constructs such as sequential composition and parallel composition in top-level specifications.

Different semantic models have been proposed for rely-guarantee. Most agree that an implementation is only required to satisfy the guarantee condition if the environment actions respects the rely condition and the program starts in a state satisfying its precondition [60, 104, 82, 26]. Some recent formalisations, such as [46] and [2] have proposed the guarantee condition to be enforced even if the environment fails to respects the rely condition. A recent survey of the semantic models for rely-guarantee suggests that only the second interpretation allows for the decomposition of the rely-guarantee conditions into independent commands [108]. We sustain that this decomposition is independent from the interpretation under consideration. That is because, the work in this thesis and those reported in [45] and [43] follow the most common interpretation, and they are able to achieve the decomposition of rely and guarantee as independent commands.

Concurrent separation logic

Concurrent separation logic (CSL) [88] is a relatively recent approach to reason about shared-variable concurrent programs using the idea of *transferable ownership of resources*. In CSL, the state is composed of a heap and a stack; the heap provides a native way of giving semantics to programs that use pointers. Resources (i.e. memory cells on the heap and the data structures that they represent) are owned by resource-holders (e.g. semaphores) or threads. Semaphores can hold and release the ownership of resources during the execution of a thread. Two principles delineate proofs in CSL: (i) modularity: specifications can ignore portions of the heap that are not used; (ii) sharing: programs executing in parallel mutually agree in exchange ownership of memory cells (i.e., a process has to acquire ownership of a resource prior accessing that resource).

In the early days of CSL, only one program could have access to a particular memory cell at a time. This restriction was later lifted by introducing the concept of fractional permissions [20, 17], which allows programs to share the ownership of cells for reading

purposes, but constrain multiple processes to share ownership for writing purposes. For comparison with the other two methods shown in this section, we shown next the rule for parallel composition of threads operating in disjoint parts of the heap using CSL. As before, S_i is a thread; and P_i and Q_i are assertions denoting the precondition and postcondition, respectively. The simplicity of this rule is due to the separating conjunction operator ($P_1 * P_2$), which holds if, and only if, the conjoined assertions P_1 and P_2 hold for disjoint parts of the heap.

$$\frac{\{P_1\}\ S_1\ \{Q_1\} \qquad \{P_2\}\ S_2\ \{Q_2\}}{\{P_1 * P_2\}\ S_1 \parallel S_2\ \{Q_1 * Q_2\}}\ \textsc{Parallel}$$

A method combining the benefits of rely-guarantee and concurrent separation logic (RGSep) has been proposed in [107]. There, rely-guarantee conditions are used to enrich assertions with assumptions and commitments that each process makes about shared resources.

1.6.2 Refinement-based approaches

In this approach the programmer is given a specification to be implemented and a collection of mathematical laws that serve to manipulate specifications and to introduce control structures in a program. Programming is reduced to the task of finding a sequence of refinement laws that can transform a specification into an executable program. Each law that is applied makes a step towards the final implementation, and may generate one or more proof obligations to mathematically justify why that transformation is valid.

What is "valid" depends on how the notion of refinement is defined [24]. Perhaps the most widespread definition is the one based on the predicate transformer semantics, better known by the operator *wp* (weakest precondition), popularised by Dijkstra in [31]. Using this semantics, programs are interpreted as predicate transformers: the execution of a program c from any state satisfying the predicate $wp(c,q)$ terminates in a state that satisfies the predicate q. This foundation is used by Morris and Morgan [79, 78] to define refinement as $(c \sqsubseteq d) \Leftrightarrow (\forall\ q.\ (wp(c,q) \Rightarrow wp(d,q))$, that is, a program c is refined by a program d if, and only if, for every predicate q, d makes fewer assumptions than c to establish such a predicate upon its termination. Back [5] proposes a definition based on contracts (Hoare triples [51]): $(c \sqsubseteq d) \Leftrightarrow (\forall\ p\ q.\ (\{p\}\ c\ \{q\}) \Rightarrow (\{p\}\ d\ \{q\}))$. Other authors, such as Sampaio [99] and Hayes [45], encode programs as sets of traces and use algebraic definitions

to define refinement: $(c \sqsubseteq d) \Leftrightarrow (c \sqcap d = c)$, that is, a program c is refined by a program d if, and only if, the traces of the non-deterministic choice $(c \sqcap d)$ are equivalent to the traces of the program c alone. Even though these definitions vary in shape, they are generally consistent with the intuition of reduction of non-determinism.

When concurrency is taken into account, the definition of refinement becomes relative to the environment. That is because it makes sense to say that a program reduces non-determinism with respect to other in a specific environment, instead of comparing behaviours for an arbitrary environment. In this context, programs are generally interpreted as sets of traces (i.e. sequences of labelled state transitions). The exact notion of what is a valid trace depends on the semantic model chosen; the literature on semantics models is very rich and we do not attempt a comparison here [21, 26, 48, 22]. We note however that trace models can be induced using a modular structural operational semantics (MSOS) [80]. This is the case in [48], which is the main reference for this thesis. More on the definition of refinement adopt in this work is discussed in Chapter 2.

1.6.3 Data refinement

In general, the stepwise derivation of programs may involve changes in the data structures used to specify a program [24]. This process is known as *data refinement* or *data reification* [62]. Algorithmic and data refinement can be study independently or be combined using the notion of blocks, as discussed in [99]. In this work we follow the recent interest in the investigation of refinement algebras [43, 45, 2, 55] and focus our attention in algorithmic refinement. Nevertheless, we discuss in the derivation of Sieve (Section 6.5) a situation where a change in the data representation is imperative to continue the refinement towards code. For this reason, this section briefly discuss the proof obligations involved with data reification.

Reification creates two proof obligations: *adequacy* and *satisfaction* [62]. Adequacy establishes a link between abstract and concrete representation. It requires that all data that can be represented at the abstract level must also be able to be represented at the concrete level, that is, the concrete representation must be at least as expressive as the abstract representation. The usual way of showing adequacy is via a *retrieve function* that maps the concrete representation to the abstract representation [62]. Next we present this proof obligation; a is used to denote an instance of the abstract state (S_A), and c to denote an instance of the concrete state (S_C).

$$\text{Adequacy: } \forall a{\in}S_A.\ \exists c{\in}S_C.\ \mathit{retrieve}(c) = a$$

Satisfaction relates the specifications on the abstract state to the specifications on the concrete state. This generally is split in two proof obligations [96]: *domain* and *result*. The domain proof obligation requires that whenever the precondition of the abstract specification ($pre\text{-}OP_A$) holds for the abstract state ($retrieve(c)$), the precondition of the concrete operation ($pre\text{-}OP_A$) should hold for the concrete state (c). The result proof obligation establishes a relationship between abstract and concrete postconditions (represented by relations $post\text{-}OP_A$ and $post\text{-}OP_C$, respectively) and the retrieve function.

$$\begin{aligned}
&\text{Domain:} \quad \forall\ c \in S_C\ .\ pre\text{-}OP_A\ (retrieve\ (c)) \Rightarrow pre\text{-}OP_C\ (c) \\
&\text{Result:} \quad \forall\ c_1, c_2 \in S_A\ .\ pre\text{-}OP_A\ (retrieve\ (c_1)) \wedge post\text{-}OP_C\ (c_1, c_2) \Rightarrow \\
&\qquad\qquad\qquad\qquad post\text{-}OP_A\ (retrieve\ (c_1), retrieve\ (c_2))
\end{aligned}$$

Sometimes the abstract state contains information that is irrelevant for the implementation. In these occasions it may turn out to be impossible to find a retrieve function for a desired concrete representation. In such cases, either the irrelevant information has to be carried to the implementation or the retrieve function has to be replaced by a retrieve relation [96]. As example, consider the problem of generating primary keys when populating a database. The generator must guarantee that primary keys are uniquely assigned to data sets. At the abstract level, a set can be used to record all keys that have already been assigned, so that the key generator simply returns an element that is not recorded. The implementation can replace the set by the last key that was generated. Thus, the implementation just needs to increment the last key to generate a new, unseen key. In this situation, it is not possible to find a retrieve function from the implementation to the abstraction, but it is possible to establish a relationship between the concrete and abstract states using a relation. As in the derivation of Sieve (Section 6.5), the retrieve relation is also a function, we presented the more specialised laws for data refinement in this section. For the general version of the proof obligations, we refer the reader to [96].

1.6.4 Isabelle/HOL

Isabelle is a generic system for implementing logical formalisms. It is implemented in ML and adopts the LCF approach. Isabelle/HOL is the specialisation of Isabelle for higher order logic (HOL), a predicate calculus with terms from the typed lambda calculus [109]. Formalisations in Isabelle are organised into theories. Each theory is a named collection of types, functions, definitions and theorems. Theories are structured in a similar way that

packages are organised in programming languages: they can be related to other theories forming a hierarchy of theories.

Part of the automation of Isabelle is supplemented by its interaction with powerful external SMT solvers, such as Z3 [29] and CVC3 [7], which are invoked using the proof finder *sledgehammer* [94]. Third-party applications, such as SMT solvers, are used to assist Isabelle in finding the lemmas which are necessary to prove theorems using proof methods such as *metis* and *auto*, but their result is generally only taken to provide guidance. That is, if a SMT solver succeeds to find a proof, Isabelle first attempts to reconstruct the proof, and if it does not succeed it may offer the possibility of trusting the result of the SMT solver (depending on the SMT solver) via the *smt* proof method [13].

An important aspect of the use of Isabelle is that it distinguishes between those proof rules that preserve provability from those that may turn a provable goal into an unprovable one, by throwing away information from the assumptions or trading weaker goals by stronger ones. Isabelle uses the terminology *safe* to refer to proof rules that preserve provability, and *unsafe* to refer to those that can turn a predicate unprovable. Isabelle has specific commands that automatically apply safe rules that are setup for automatic application. Depending on the type of proof rule that a user is using, different proof methods should be adopted to instruct Isabelle to apply the rule. Next we discuss two features of Isabelle that have been intensively used in this thesis.

Locales

Apart from theories, Isabelle provides the structuring notion of locales. These can be understood as detached proof contexts declared via an interface specifying a set of parameters (constants and definitions) and assumptions (the fundamental properties of these parameters). Proof developments in the scope of a locale are not visible in its outer theory unless the locale is interpreted. The process of interpretation requires a user to provide a model for the locale, that is, to show the existence of a set of objects in its outer scope (corresponding to the parameters of the locale) that satisfies all the properties specified in the declaration of the locale. Theorems can be proved within the scope of the locale and then exported to the outer scope at a later stage by means of interpretation. Moreover, the independence with respect to an underlying model makes this method of proof development geared towards reuse, that is, the same locale can be interpreted for different models [6].

It is also possible to develop the theory fully within the scope of the locale without going through the interpretation process. In this case, the interface of the locale makes explicit what are the definitions and properties taken as assumption for that proof development. Although

inconsistencies within a locale cannot contaminate the outermost context, care should be taken when introducing assumptions in the interface of a locale to prevent wasting time on the development of an infeasible theory.

In this work, a proof development is produced within the scope of a hierarchy of locales, but we have not interpreted the locales used. Instead, we use an operational semantics defined in the outer scope of the locales to prove properties that otherwise would need to be taken as additional assumptions in the proof development. We explain the approach taken and its motivation in Chapter 4, and we assess the inherent risks of this approach in Chapter 7. In short, the properties we take to define locales in this thesis com from pen-and-paper proof developments. Our motivation for using this approach is because we are mainly interested in the applying the methodology discussed in [54] to investigate the practical aspects involving the use of theory proposed in [48], instead of directing the effort to formally validate the calculus discussed in [48, 65].

To prevent confusion we use the term *local assumptions* in Chapter 4 to refer to the properties specified in the interface of the locale. Several terms are adopted in the literature [6, 109], such as *local assumptions*, *local axioms*, *module axioms*, etc. Our preference by the term *local assumption* is to emphasize that we do not make use of the mechanism to define axioms (i.e. we have not used the command **axiomatization**) in the scope of the outer theory in this thesis.

Assisted document preparation

The mathematical formulae in the statement of definitions and theorems in this thesis are extracted from the encoding of the respective concepts in Isabelle using *anti-quotations*. From the user perspective, an anti-quotation is a markup command offered by Isabelle that allows informal text within an Isabelle theory to quote formal entities like definitions, theorems, expressions, *etc*. The Isabelle document preparation system performs many technical consistency checks over anti-quoted entities. Anti-quotations provide a systematic translation of formal entities to Latex. Isabelle comes with predefined translation rules, but allows the user customise these translation rules and assign them to print modes (e.g. Latex, HTML, *etc*.). The greatest benefit we take from anti-quotations is the prevention of mismatch between the definitions and theorems shown in this thesis and those encoded in Isabelle.

1.7 Structure of the thesis

As this thesis has been typeset using Isabelle/HOL, the reader might find useful to access the Isabelle theories used to typeset the thesis. These theories are distributed with the Appendix A.1 (CD-ROM) and they facilitate the inspection of proofs or details which are most relevant for the reader. For that, the reader can use Isabelle to open the theory containing the desired chapter, and click over the definitions and laws to be redirected to the relevant theories (some minimal knowledge of Isabelle may be required to understand the proofs). Table 1.2 (located at the end of this section) briefly shows the content of each of the appendices.

Chapter 2 introduces the foundation for the refinement calculus discussed in Chapter 3. That includes: the wide-spectrum programming language (RG-WSL); its formal semantics; and the definition of the refinement relation. Additionally, Chapter 2 discusses the design decisions behind the representation of these concepts in Isabelle/HOL, and provides the earliest evidence of the benefit of using Isabelle in this work.

Chapter 3 introduces a rely-guarantee refinement calculus. The material in that chapter is mainly taken from [48], but is adapted to fit the particular encoding of RG-WSL discussed in the previous chapter. Standard programming constructs are defined in terms of RG-WSL, and rely and guarantee constructors are introduced as independent commands. Refinement laws are organised per subject, which roughly resembles their organisation in the mechanised theory. While mechanising the laws from Chapter 3 we discovered the need for extending RG-WSL and adding extra laws to reproduce the proofs suggested in [48]. Contributions in the form of definitions and laws presented in this chapter are marked using the keyword **contrib.**, and are visually highlighted in gray boxes in this chapter.

Chapter 4 discusses the infrastructure developed to encode the refinement calculus from Chapter 3 as a refinement algebra in Isabelle/HOL. It explains conventions and decisions which were taken to enhance the automation of the mechanisation, as well as make it easy to identify laws by subject. These decisions are practical contributions of our encoding: without careful design and introduction of redundancy in specific points of the mechanisation, the theory would have become quite clumsy to use in practice. This chapter also concludes a discussion started in Chapter 2 about the encoding of RG-WSL in Isabelle, in particular it revisits the encoding of non-deterministic choice and analyses the theoretical consequences of that choice. Finally, Chapter 4 presents a revised version of paper proofs from [48] where we found mistakes.

Chapter 5 presents the main contributions of this thesis in the form of extensions to the algebra discussed in Chapter 4. Support to indexed parallelism and assignment to indexed arrays is introduced, and an additional abstraction is provided to reason about programs that constrain environment steps. This chapter also revisits the operational semantics of RG-WSL, and solves a problem involving the characterisation of parallel composition. Finally, this chapter introduces an abstraction to encode non-atomic evaluation of expressions using relations (reachable expressions) and a minimalist type system to compensate for the lack of knowledge about the type of program variables in proofs.

Chapter 6 presents the derivation of examples to illustrate (i) on-the-fly extension of the expression language and treatment of undefinedness; (ii) compositional reasoning about logical interpretation of relations; (iii) different laws for introducing indexed parallelism; (iv) assignment to indexed arrays; (v) nested parallelism; (vi) reachable expressions; and (vii) shadowing of variables. At the end of the chapter we present a comparative discussion of the examples, and highlight the bottlenecks of the theory.

Chapter 7 discusses learned lessons and the use of proof integrated tools such as *sledgehammer* and *nitpick* to assist the mechanisation process. It also presents a quantitative analysis of the mechanisation, that serves to guide the exploration of the mechanised theories. The chapter ends with a comparison to related works.

Chapter 8 presents a summary of the contributions, limitations, and discusses future work. Further material, such as glossary of precedence, additional proofs and complete source of the derivation of concurrent Findp are provided in the appendix. Due to the size of the mechanised theories, part of the appendix is provided as a companion CD-ROM rather than in printed format. The digital content includes the mechanised theories, the thesis itself, and instructions for extending Isabelle fonts to render the symbol used to represent strict conjunction and compiled PDF of the theories. Table 1.2 shows the content of the appendices.

Appendix	Content
A.1 (Digital)	Mechanisation in directory: /RG-laws
A.2	Table of precedence and associativity
A.3	Revised paper proofs
A.4	Extra material and tutorial to extend Isabelle fonts
B	Summary of laws and definitions used in Chapter 6
C	Complete derivation of concurrent version of Findp

Table 1.2 Appendices

Chapter 2

Foundation

> The scientist is a practical man and his are practical aims. He does not seek the ultimate but the proximate. He does not speak of the last analysis but rather of the next approximation. His are not those beautiful structures so delicately designed that a single flaw may cause the collapse of the whole. The scientist builds slowly and with a gross but solid kind of masonry. If dissatisfied with any of his work, even if it be near the very foundations, he can replace that part without damage to the remainder. On the whole, he is satisfied with his work, for while science may never be wholly right it certainly is never wholly wrong; and it seems to be improving from decade to decade.
>
> Gilbert Newton Lewis, *The Anatomy of Science, 1926*

This chapter discusses the foundation of the refinement calculus presented in Chapter 3. It covers the programming language, its formal semantics, and the definition of the refinement relation. The main reference for this chapter is [48], a technical report that introduces the laws and semantics for a rely-guarantee refinement calculus supporting shared-variable parallelism.

Except for the cases where the concepts given in [48] and the ones encoded in the Isabelle/HOL differ, we simply present the corresponding encoding in Isabelle, the theorem proof assistant (TPA) adopt in this work. For most cases, the encoding is more inclusive of details than the description provided in [48]. It is no surprise that the representation of a theory in a theorem prover forces the user to think about details that are sometimes overlooked in the literature. The additional details in the encoding are thus, a key contribution, instead of being a reproduction of concepts defined elsewhere. Contributions are summarised at the end of the chapter.

Section 2.1 presents the programming language as given in [48]. That description leaves unspecified the range of values that variables can take on and the unary and binary operators supported by the expression language. To set the context for the mechanisation, we delimit the range of values that variables can take on (*vvalue*[1]) and specify the type used to represent states (*state*) and relations (*relation*) in Section 2.2. In our encoding, boolean expressions which are evaluated with respect to a single state or a pair of states are shallow-embedded by relations, and expressions whose evaluation has to take into account the interference in intermediate states are deep-embedded. Shallow and deep-embedding are the subject of Section 2.3, which discusses the encoding of the programming language, and provides the earliest evidence of the benefit of using Isabelle in this work: the identification of an issue in the programming language that went unnoticed in the literature [47–49, 65]. Section 2.3 also supplements the notions provided in [48] with the types used to encode unary and binary operators (*unaryop* and *binop*, respectively), and discusses the treatment of undefinedness in the mechanisation.

Since relations are a recurrent abstraction in this thesis, Section 2.4 presents a collection of relational operators and properties of relations used in this work. Section 2.5 establishes a crucial link between the shallow and deep-embeddings of boolean expressions ($[\![\_]\!]_r$). This link is applied to formalise laws that introduce control structures and assignment from specifications in Section 3.12. Section 2.6 presents the concept of logical evaluation ($\vdash \_$). This concept captures the fact that a relation holds independent of the pair of states taken as parameters. The concept of logical interpretation is used in a number of laws in Chapter 3 to encode proof obligations that involve weakening and strengthening relations.

Section 2.7 introduces an operational semantics for the programming language. This semantics is taken from [48], but is slightly adapted to fit the findings from Section 2.3. The operational semantics subdivides into two semantics: *small-step* and *big-step*. Since small-step semantics are more capable of capturing the high-degree of non-determinism in concurrent computations, it is used to describe most of the commands. The big-step semantics is used to describe commands that are better suited for an end-to-end semantics, such as the specification command. Ultimately, the small-step semantics is used to induce a corresponding big-step semantics for commands originally described using the small-step relation. The big-step semantics is the basis for the definition of refinement used in this work.

Before introducing the definition of refinement, we present a denotational semantics for the programming language in Section 2.8. The denotational semantics provides an

[1]The extra *v* in the word *vvalue* has no special meaning. It has been included because the word *value* is already reserved in Isabelle, and cannot be used to name a new type.

intermediate layer between the definition of refinement and the big-step semantics. This intermediate layer allows the refinement relation to take the environment of a program into account. Environments are abstracted as relations that characterise state updates that can be observed as effect of the interference. The refinement relation is introduced in Section 2.9 as ternary relation: it establishes a relationship between two programs considering a specific environment.

Although the definition of a denotational semantics from the big-step semantics serves the purpose of defining the refinement relation, the lack of properties about the denotational semantics makes it unsuitable for mechanical proofs. To remedy this situation, Section 2.10 introduces *forward simulation*, an elaborated proof technique to prove refinement laws from the small-step semantics. Forward simulation is not necessary condition to establish refinement, but it is a sufficient condition and serves the purpose of proving refinement laws in Chapter 5. Section 2.11 discusses a mechanism to decide if a variable is *unrestricted* in a program. The notion of unrestricted variable approximates that of fresh variable (which is not compatible with shallow-embedded relations), and is used in Chapter 3 to formalise the introduction of local variables in a program. Section 2.12 closes the chapter with a discussion and a summary of contributions.

2.1 Programming language

Refinement languages consist of specifications (non-executable) and implementation (executable) constructs, and for this reason are referred to as *wide spectrum languages*. We use the following convention to describe the refinement language of rely-guarantee refinement calculus (RG-WSL): a *state* is a total map from variable names to values, q and r are relations over a pair of states (the first state in the pair is referred to as the *before* state, and the second, the *after* state), p is a predicate (i.e. a relation which cannot refer to the after-state), b is a boolean expression, c (c_1, c_2, ...) ranges over commands, C ranges over sets of commands, X ranges over sets of variables, x ranges over variables and v ranges over values that variables can take on.

$$\begin{aligned} \mathit{Command} = \langle p, q\rangle \mid [q] \mid \{p\} \mid \bigsqcap C \mid c_1 \Cap c_2 \mid c_1 \mathbin{;_c} c_2 \mid c_1 \parallel c_2 \mid [[b]] \mid \\ \textbf{uses}\ X \cdot c \mid \textbf{state}\ x \mapsto v \cdot c \end{aligned}$$

Figure 2.1 Wide spectrum language: primitive commands

The primitive commands of RG-WSL are shown in Figure 2.1. Atomic specification ($\langle p, q\rangle$) requires q to be established between the initial and final states via a single program step if started in a state that satisfies p; if p is not satisfied at the starting state, then $\langle p, q\rangle$ aborts. The postcondition command ($[q]$) allows any finite number of steps to establish q between its initial and final state; it ensures termination if the environment only does stuttering steps, and aborts otherwise[2]. The precondition command ($\{p\}$) immediately terminates if p holds on the state it is executed, otherwise it aborts; unbounded non-deterministic choice $\bigsqcap C$ can behave as any element of C; strict conjunction ($c_1 \Cap c_2$) synchronises atomic steps of its operands to give an atomic step of their composition; sequential composition ($c_1 \,;\, c_2$) executes c_1 and, if c_1 terminates, it executes c_2. Parallel composition ($c_1 \parallel c_2$) represents interleaving of c_1 and c_2 Tests ($[[b]]$) perform a non-atomic evaluation of its argument; it can succeed by evaluating b to *true*, fail by evaluating b to *false*, or abort by attempting to evaluate an undefined boolean expression. The test command is central to the definition of conditional, assignment and while loop, which are formalised as derived commands in Chapter 3. The uses command (**uses** $X \cdot c$) restricts the program c to only use free variables which are in the set X, and the state command (**state** $x \mapsto v \cdot c$) declares the local variable x and initialises it to v.

The programming language is untyped and global variables are undeclared, thus the type of variables is inferred from the expressions and assignments where they appear in a program. Boolean expressions are formalised in Figure 2.2, which introduces the general type of expressions. Expressions can be a constant (N), a variable (V), or be formed from the application of an unary (UOp) or binary (BOp) operator over existing expressions.

$$Exp = N\ vvalue \mid V\ vname \mid BOp\ binop\ Exp\ Exp \mid UOp\ unaryop\ Exp$$

Figure 2.2 Grammar of expressions

Unary and binary operators are not further specified in [48]. Preceding the encoding of the programming language, the next section defines the range of values that variables can take on and the types used to represent variables, states, relations and unary and binary operators.

[2]Stuttering steps (computations) do not change the observable state of a program.

2.2 Formalisation of the state (σ)

Variable names are represented using the type *string* (i.e. sequence of characters), and the content of a variable is allowed to range over the datatype *vvalue*, which wraps several types under a single type. States are encoded as total functions from variable names to *vvalue*, and binary (*state*) relations are represented in curried form using the type $state \Rightarrow state \Rightarrow \mathbb{B}$.

Definition 2.1 (Basic types)**.** *Variables names (vname), values (vvalue), state and binary state relations are defined next.*

datatype *vvalue* = *VNat* $\mathbb{N}$
| *VInt* $\mathbb{Z}$
| *VBool* $\mathbb{B}$
| *VArray vvalue list*
| *VNatSet* $\mathbb{N}$ *set*
| *VNone*

type-synonym *vname* = *string*
type-synonym *state* = *vname* $\Rightarrow$ *vvalue*
type-synonym *relation* = *state* $\Rightarrow$ *state* $\Rightarrow \mathbb{B}$

Aside. *The characterisation of vvalue is tightened to the collection of examples presented in Chapter 6, but the theory developed in this thesis is not dependent on this characterisation. In fact, the value model can be easily extended to suit more examples and such extension has no major impact on the development of the theory.*

To introduce the type *vvalue* we use the command **datatype** of Isabelle, which formalises a new type from a set of distinct constructors (e.g. *VInt*, *VBool*, etc.). Datatypes are commonly used to encode BNF grammars and are explained in details in the next section. Arrays are modelled as lists of *vvalue* and can be of arbitrary dimension. Multidimensional arrays can be encoded by nesting the constructor *VArray*, e.g. the matrix

$$\begin{pmatrix} p_{11} & p_{12} & p_{13} \\ p_{21} & p_{22} & p_{23} \\ p_{31} & p_{32} & p_{33} \end{pmatrix}$$

is encoded as

$$VArray\ [VArray\ [p_{11}, p_{12}, p_{13}], VArray\ [p_{21}, p_{22}, p_{23}], VArray\ [p_{31}, p_{32}, p_{33}]]$$

The type *vvalue* includes also the value *VNone*, which can be used to model *option* types for the remaining types offered by *vvalue*. Option types are used to lift a type α to a new type α *option* containing all values of α wrapped by the constructor *Some* plus a special value

called *None*. The usage of *VNone* to model option types is more efficient than including a constructor for option types in *vvalue*. Such efficiency can be observed in the amount of time which is necessary for Isabelle to automatically prove termination of partial functions defined over *vvalue* when options types are being represented. The modelling of option types is illustrated in the derivation of the Floyd-Warshall algorithm [28] in Section 6.6. The datatype *vvalue* does not include an additional constructor for representing undefined values, instead, we use the native constant (*undefined*) from Isabelle to represent undefined when formalising expression evaluation.

Consider now the task of writing a relation to establish the value of a variable x with the after state value as *VNat 5*. Using the types seen so far, this can be captured by the relation:

$$\lambda s\ s'.\ s'\ x = VNat\ 5$$

That is, the function that takes the before and after state and evaluates to *True* if the value of variable x in the after state corresponds to *VNat 5*. Alternatively, we can use functions to project the argument encapsulated in a *vvalue*, so that we can operate directly using the wrapped value. Next, we define such projection functions and use them to construct an example[3].

Definition 2.2 (Project-Argument)**.** *The following functions project their arguments.*

fun *StripVNat* :: *vvalue* $\Rightarrow$ $\mathbb{N}$ ($[\![\text{-}]\!]_n$) **where** $[\![VNat\ v]\!]_n = v$
fun *StripVInt* :: *vvalue* $\Rightarrow$ $\mathbb{Z}$ ($[\![\text{-}]\!]_i$) **where** $[\![VInt\ v]\!]_i = v$
fun *StripVBool* :: *vvalue* $\Rightarrow$ $\mathbb{B}$ ($[\![\text{-}]\!]_b$) **where** $[\![VBool\ v]\!]_b = v$
fun *StripVNatSet* :: *vvalue* $\Rightarrow$ $\mathbb{N}$ *set* ($[\![\text{-}]\!]_s$) **where** $[\![VNatSet\ v]\!]_s = v$
fun *StripVArray* :: *vvalue* $\Rightarrow$ (*vvalue list*) ($[\![\text{-}]\!]_a$) **where** $[\![VArray\ v]\!]_a = v$

Using projection $[\![\_]\!]_n$, the previous relation can be written as:

$$\lambda s\ s'.\ [\![s'\ x]\!]_n = 5$$

Moreover, now we can write the following relation to specify that the value of x in the final state is increased by one:

$$\lambda s\ s'.\ [\![s'\ x]\!]_n = [\![s\ x]\!]_n + 1$$

That is, the function that takes before and after state and evaluates to *True* if the value of variable x in the after state corresponds to the value of x in the before state increased by one.

[3]The command **fun**, used in the Definition 2.2, allows a user to introduce partial functions within Isabelle.

Without using projection functions, we would need to define addition over the type *vvalue* in order to represent the relation above.

Projection functions are partially defined (e.g. there is no rule to compute $[\![VBool\ True]\!]_i$). As consequence, the relations ($\lambda s\ s'.\ [\![s'\ x]\!]_n = 5$) and ($\lambda s\ s'.\ s'\ x = VNat\ 5$) are not equivalent. The version written using projection functions is logically weaker than the one without projection functions. That means that projection functions should be used in a consistent manner, i.e. either one must use them in all definitions of a particular example, or not use them at all. Examples of the use of projections to write complex relations can be seen in Appendix A.1, where we provide the source of the mechanisation of the examples discussed in Chapter 6.

The relations given as example in this section leave unspecified the value of variables different from x in the final state. If we wish to use a relation to specify the update of a single variable, we can use Isabelle's built-in notation of function update, since states are total functions. Given a state s, the state $s(x{:=}v)$ is exactly the state s, except that x maps to value v. Updates can be nested, producing the effect of sequential updates, e.g. $(s(x{:=}v))(y{:=}w)$ is exactly as state s, except that it maps variable x to v, and variable y to w.

2.3 Encoding the programming language

2.3.1 Shallow and deep-embedding

When facing the task of encoding a programming language in a theorem prover, a decision between a *deep-embedding* and a *shallow-embedding* is required. In the former approach, syntax and semantics are encoded separately, while in the latter approach, both are encoded as a single structure. Deep-embedding is most suitable when we are interested in studying meta-properties of a theory, because it allows proofs by structural induction on the language; whereas shallow-embedding is most suited to the application of a theory, but not to proving meta theorems about the programming language.

The approach we take here is quite well-established in the literature [83, 84]: it combines the benefits of both shallow and deep-embeddings. We use deep-embedding to encode the general structure of the programming language shown in Figure 2.1 and also to encode boolean expressions taken as parameter by the test command ($[\![\_]\!]$); this decision is prompted by the need of a syntax to formalise the operational semantics for command and expression evaluation in a concurrent environment. We use shallow-embedding to encode expressions taken as parameters by specification commands, i.e. precondition, postcondition and atomic

commands; this is possible because these expressions are evaluated as relations. With respect to expressions taken as parameter by the test command (Figure 2.2), we shallow-embed unary and binary operators. In this case, the motivation is to provide more flexibility to deal with concrete examples than we would have by defining a grammar of unary and binary operators which can appear in programs.

2.3.2 Expression language

We use pairs of functions to encode unary and binary operators that can occur in expressions. Our choice of representation combines the underlying mathematical operation and the precondition that their operands must satisfy to succeed. The precondition is used to decide if expressions are *undefined*. The next definition provides a concrete representation for the types of unary and binary operators shown in Figure 2.2.

Definition 2.3 (Unary/Binary Operators)**.** *Unary and binary operators are defined as follows. The first component of the cartesian product is the underlying mathematical operation and the second component is the precondition of the respective operation.*

type-synonym $unaryop = (vvalue \Rightarrow vvalue) \times (vvalue \Rightarrow \mathbb{B})$
type-synonym $binop = (vvalue \Rightarrow vvalue \Rightarrow vvalue) \times (vvalue \Rightarrow vvalue \Rightarrow \mathbb{B})$

For example, the binary operator "*mod*" (remainder after integer division) is encoded as:

$$(\underbrace{\lambda va\ vb.\ VNat\ (\llbracket va \rrbracket_n\ mod\ \llbracket vb \rrbracket_n)}_{Underlying\ mathematical\ operation}, \underbrace{\lambda va\ vb.\ \llbracket vb \rrbracket_n \neq 0}_{Precondition})$$

The underlying mathematical operation specifies the usual meaning for "*mod*" and interprets the arguments as natural numbers. The precondition states that the result of this operation is defined if the second argument is not zero.

Thus, the grammar for expressions given in Figure 2.2 can be extended on demand with unary and binary operators which are relevant to algorithms being derived by the user. The expressiveness of operators in the expression language is bounded by the expressiveness of Isabelle, instead of being set in stone by a predefined grammar[4]. The user is free to provide operators on-demand, but has the obligation of providing the precondition for mathematical operators at the time of the definition.

[4]Note however that if the user needs an operator that take more than two operands, the grammar of expressions would restrict the ability of encoding such an operator. In this case the user would need to extend the expression language (Figure 2.2) with a general n-ary operator.

For the sake of readability of expressions in concrete programs, we provide definitions to abbreviate expressions which are built from unary and binary operators. For example, for the *mod* operation discussed in this section, we provide the next definition:

$$e_1\ mod_n\ e_2 \equiv BOp\ \underbrace{(\lambda va\ vb.\ VNat\ (\llbracket va \rrbracket_n\ mod\ \llbracket vb \rrbracket_n),\ \lambda va\ vb.\ \llbracket vb \rrbracket_n \neq 0)}_{Binary\ operator}\ e_1\ e_2 \quad (2.4)$$

The *len* operator, that returns the length of the first dimension of an array, can be defined as[5]

$$len\ e \equiv UOp\ \underbrace{(\lambda va.\ VNat\ (length\ \llbracket va \rrbracket_a),\ \lambda va.\ True)}_{Unary\ operator}\ e \quad (2.5)$$

These definitions smooth the representation of concrete programs in Isabelle, by hiding details such as the actual definition of unary and binary operators. For a complete list of unary and binary operators provided with the mechanisation see Appendix A.

2.3.3 Definedness

The precondition of unary and binary operators is used to decide if expressions are well-defined. Undefinedness can arise in programs by applying unary and binary operators outside of their precondition, leading to situations such as division by zero, access out of bounds for arrays, etc. In [48], the signature *defined* is meant to denote a predicate that takes an expression and a state and decides if the expression is well-defined in that state. We contribute with a recursive implementation for this signature. Our definition unpicks the precondition of operators occurring in an expression and tests if the operands satisfy it. Next, is a generic mechanism we defined to determine if an expression is well-defined. It is defined in terms of an auxiliary function, $\llbracket\text{-}\rrbracket_v$, which takes an expression and a state and evaluates the expression in that state.

Definition 2.6 (Expression-Evaluation and Definedness)**.** *Given an expression e and a state s, we denote by* $\llbracket e \rrbracket_v\ s$ *the evaluation of e in the state s. We say that e is well-defined in state s if, and only if, (defExp e s) holds. If an expression is not well-defined, its evaluation results in the native constant undefined.*

fun *SEvalExp* :: *Exp* ⇒ *state* ⇒ *vvalue* ($\llbracket\text{-}\rrbracket_v$) **and**
  defExp :: *Exp* ⇒ *state* ⇒ 𝔹
where

[5] Arrays in RG-WSL are not pointer-based, thus we assume that *len* can always be applied to an array.

$[\![N\ c]\!]_v\ s = c$
$[\![V\ v]\!]_v\ s = s\ v$
$[\![UOp\ u\ e]\!]_v\ s = (if\ (defExp\ (UOp\ u\ e)\ s)\ then\ (fst\ u)([\![e]\!]_v\ s)\ else\ undefined)$
$[\![BOp\ b\ e_1\ e_2]\!]_v\ s = (if\ (defExp\ (BOp\ b\ e_1\ e_2)\ s)\ then\ (fst\ b)([\![e_1]\!]_v\ s)\ ([\![e_2]\!]_v\ s)\ else\ undefined)$
$defExp\ (N\ c)\ s = True$
$defExp\ (V\ v)\ s = True$
$defExp\ (UOp\ u\ e)\ s = ((defExp\ e\ s) \wedge (snd\ u)\ ([\![e]\!]_v\ s))$
$defExp\ (BOp\ b\ e_1\ e_2)\ s = ((defExp\ e_1\ s) \wedge (defExp\ e_2\ s) \wedge ((snd\ b)\ ([\![e_1]\!]_v\ s)\ ([\![e_2]\!]_v\ s)))$

The next abbreviation lifts defExp to the status of relation.

abbreviation *defined* :: *Exp* ⇒ *relation* **where** $defined\ e \equiv (\lambda s\ s'.\ defExp\ e\ s)$

Using the definition of *defined* we can infer, for example, that $(V\ x)\ mod_n\ (N\ (VNat\ 0))$ is not well defined independently of the state in which it is evaluated. We return to discuss undefinedness in the description of the operational semantics in Section 2.7, when discussing non-atomic expression evaluation and semantics of tests.

2.3.4 Encoding RG-WSL

The grammar presented in Figure 2.3 is extracted from our encoding and shows the representation of RG-WSL in Isabelle/HOL. The encoded version of RG-WSL includes the additional command *eguard*, that is subject of discussion in Section 5.2.

In the literature, preconditions and postconditions are described as unary and binary relations, respectively. Without loss of generality, we use a single type, *relation*, to represent these entities. The uniform treatment prevents the duplication of relational operators and laws to reason about the parameters taken by the atomic, precondition and postcondition commands. To stay consistent with [48], the encoded version of the semantics and refinement laws include additional provisos which require that relations used to denote preconditions are *predicates*, i.e. they must not constrain the after state. The formal definition of predicates is introduced in Section 2.4.1.

Note in Figure 2.3 that the test command ([[_]]) takes a deep-embedded expression (*Exp*) as parameter. The deep-embedding is relevant because the test command has to discriminate between expressions such as $x+x$ and $2*x$, for example, since these expressions can be evaluated to distinct values in an interfering environment. The formalisation of the expression language is given as already shown in Figure 2.2. The uniform treatment

datatype
Command = *Atomic relation relation* (⟨-, -⟩)
| *Post relation* ([-])
| *Pre relation* ({-})
| *DemNonDetChoice Command cset* ((⨅-) *75*)
| *StrictConj Command Command* (**infix** ⩓ *70*)
| *SeqComp Command Command* (**infixl** ;$_c$ *80*)
| *ParComp Command Command* (**infix** ‖ *70*)
| *Conditional Exp* ([[-]])
| *Uses vname set Command* ((**uses** - ·/ -) [*71*,*70*] *70*)
| *State vname vvalue Command* ((**state** - ↦ - ·/ -) [*71*,*71*,*70*] *70*)
| *EGard relation* ((**eguard** -) [*71*] *70*)

Figure 2.3 Encoding RG-WSL. In this thesis, the syntax for sequential composition (;$_c$) is written without the subscript *c*. In Isabelle, the subscript is used to prevent ambiguity between sequential composition and the symbol used to separate assumptions in theorem statements. The notation **infix** is used to define an infix operator. An extra **l** at the end of this notation is used to enforce left-associativity. Precedence and associativity are discussed at the end of this section.

of expressions and boolean expressions allows the same syntactic entity to be used in the condition of an *if-then-else* and at the right-hand side of assignments, for example.

The crucial difference between the version of RG-WSL published in [48] (Figure 2.1 on page 21) and its mechanised version shown in Figure 2.3 (page 29) is that in the mechanised version the parameter taken by unbounded choice ⨅ _ is constrained to be a countable set of commands, i.e. the cardinality of that set cannot exceed the cardinality of the set of natural numbers (**ℕ**). In the description of the language gave in [48], no restriction is imposed over the cardinality of the set of commands taken by the unbounded choice. To understand better why this restriction matters, and how Isabelle helped to reveal the issue in the original design of RG-WSL, it is worth to take a step back and analyse how the syntax of programming languages are encoded in proof assistants.

Representing programming languages as datatypes

In general, the syntax of programming languages is encoded in proof assistants, such as Isabelle/HOL, using the concept of **datatype**. This specification constructor is used to formalise a type (D) from a collection of *constructors* (C_i), where each constructor can have zero or more parameters of distinct types (${\tau_i}^j$). The simplified syntax for introducing a datatype is shown next

$$\textbf{datatype}\ D = C_1\ {\tau_1}^1\ {\tau_1}^2\ \ldots\ {\tau_1}^n \mid C_2\ {\tau_2}^1\ {\tau_2}^2\ \ldots\ {\tau_2}^m \mid \ldots \mid C_i\ {\tau_i}^1\ {\tau_i}^2\ \ldots\ {\tau_i}^s$$

From a typed set-theoretical perspective, datatypes define a set of mathematical objects that share the same type. To ensure that the collection of mathematical objects defined by a datatype can be represented as a typed set, in addition to other properties which are characteristic of datatypes in HOL (e.g. non-emptiness[6], distinction between the datatype constructors C_i, and injectivity of the datatype constructors), Isabelle performs a series of checks and proofs at the time of the definition of a new datatype. Injectivity ensures the existence of a typed set which contains all the elements generated by the constructors of the datatype [9].

In principle, Isabelle/HOL proves at the time of the definition of a new datatype D, that there exists an injection from the domain of each constructor C_i to the type D. The domain of a constructor C_i is the type $(\tau_i{}^1 \Rightarrow (\tau_i{}^2 \Rightarrow (... \Rightarrow \tau_i{}^s)))$, thus the task for Isabelle is to prove the existence of an injection of the type $((\tau_i{}^1 \Rightarrow (\tau_i{}^2 \Rightarrow (... \Rightarrow \tau_i{}^s))) \Rightarrow D)$. However, it is well-known from Cantor's diagonalisation theorem that some injective functions cannot exist, e.g. there is no injective function of the type $'a\ set \Rightarrow 'a$, because the domain of this function ($'a\ set$) is larger than its codomain ($'a$). If the user attempts to define a datatype where the domain of at least one of its constructors would not allow such an injective function to be constructed, Isabelle displays an error message. For the general scenario, however, Isabelle is unable to automatically decide if such an injective function exists. This is the case of some situations where one of the constructors (C_i) is recursive, i.e. when one of the types ($\tau_i{}^s$) occurring in the parameters of that constructor has a nested instance of the type of the datatype (D) under definition. In Isabelle, the recursive occurrence of a datatype in the type of its constructors is referred to as *(co)recursion*. It can involve *type constructors* (e.g. *set*, *list*, etc.) which nest the recursive datatype, as in

datatype *DTA* = *Base* | *Idx DTA list*

For a limited range of type constructors (*list* included), Isabelle is able to automatically decide if its occurrence in nested (co)recursion preserves injectivity. To allow the user to extend this range, the **datatype** package provides an interface based on the concept of *bounded natural functor* (BNF), a semantic criterion that determines if (co)recursion can appear on the right-hand side of an equation [14]. This criterion can be interpreted as a cardinality constraint on type constructors. Isabelle only accepts type constructors in nested (co)recursion if the type constructor is registered as a BNF.

[6] In Isabelle/HOL, types are not allowed to be empty. This decision simplifies the logic, ruling out corner cases which otherwise would need to be taken care properly. For example, $(\forall x.\ P\ x) \Longrightarrow (\exists x.\ P\ x)$ is a theorem in first-order logic and it should still hold even if types are empty.

The attempt of encoding RG-WSL as a datatype in Isabelle/HOL results in the next message being displayed to the user. The reason being the fact that the power set type constructor (*set*) in the definition of the parameter of unbounded choice $\bigsqcap$ _ is not registered as a BNF.

> Unsupported recursive occurrence of type "*Command*" via type constructor "*Set.set*" in type expression "*Command set*".
>
> Use the "**bnf**" command to register "*Set.set*" as a bounded natural functor to allow nested (co)recursion through it.

This error message lead us to formulate three questions:

i Is there any theoretical limitation to the encoding of RG-WSL as a datatype?

ii If there is a limitation, what does it reflect about the design of RG-WSL?

iii Can we constrain RG-WSL such that it can be encoded in Isabelle/HOL?

Our previous discussion sets the context to answer question (i). There exists a theoretical limitation to the representation of RG-WSL as a datatype. Cantor's theorem says that there is no injective function of the type *Command set* $\Rightarrow$ *Command*, and thus, RG-WSL as published in [48] cannot pass the injectivity check.

Question (ii) is more open-ended. Any proof assistant which gives a typed set theoretical foundation for datatypes is unable to represent the syntax of RG-WSL as proposed in [48]. This language can be formalised using other logics supported by Isabelle, such as Isabelle/ZF [71] or HOLZF [87], which are untyped, but in these cases we would lose certain features of HOL such as decidable type checking. To the best of our knowledge, the relative consistency of these logics have not been established and in both cases we would not be able to build recursive set constructions in datatypes and would have to develop our own model for the syntax of RG-WSL. That is, we would need to introduce the constructors of the language using the specification command **consts**, and we would need to introduce the relevant axioms about the constructors of RG-WSL, to ensure properties such as distinctiveness between syntactic terms of the language. An example of the construction of a toy datatype using Isabelle/HOLZF is described in [87].

The answer to question (iii) is positive. We can constrain the parameter taken by unbounded choice to be a countable set in order to work around the injectivity problem. To the best of our knowledge, this is the best approximation to the definition provided in [48]. Our solution solves the syntactic problem at the cost of creating another one. The language of

commands itself is not countable[7], but the definition of rely, provided in Chapter 3 (Definition 3.72 on page 94), is structured using unbounded choice and set comprehension over the language of commands. Unbounded choice is also used to define other derived commands, such as assignment and local variables, but these commands could, in theory, be introduced as primitive commands in RG-WSL with little impact in the proof of the refinement laws introduced in Chapter 3. On the other hand, the definition of the rely command cannot be restructured, at least without impacting in the ability of benefiting from the theory published in [48].

A definitive solution to the problem of mechanising the rely command requires RG-WSL to be stratified to syntactically determine a maximum level of nesting of unbounded choice. That is, by splitting the language into multiple syntactic layers, we can precisely control the number of times that unbounded choice can be nested[8]. Although this approach works in theory, its implementation would require the replication of laws and operators for each layer of RG-WSL. The approach taken in this work is to recognise that some laws taken as basis for the algebraic mechanisation cannot be derived from the semantics given to RG-WSL. To compensate for this weakness, Section 4.3 explores the relationship between our refinement algebra based on RG-WSL and, and a hypothetical refinement algebra based on a stratified version of RG-WSL, where laws can be proved sound with respect to the semantics.

Precedence and associativity

To reduce the need for parentheses, we assign precedences and associativity to the commands of RG-WSL. Our encoding extends the description of RG-WSL given in [48], where the only information given about precedence is the fact that sequential composition should have the highest precedence among the primitive commands of RG-WSL. The numbers within parenthesis at the right-hand side of the language constructors in Figure 2.3 represent the precedence of the constructor. The higher the number, the higher its precedence. For constructors that have the same precedence, such as $\Cap$ and $\|$, parentheses are required to disambiguate expressions.

Associativity is encoded using *mixfix* annotations. In the definition of RG-WSL, **infixl** is used to define associativity to the left. Table 2.1 illustrates the use of associativity and precedence to omit parentheses. Note that in the some cases, parentheses cannot be omitted. A glossary of precedences is available in Appendix A.2.

[7] Its cardinality is, at least, the same cardinality of the space of functions over states. See a proof of this fact on Appendix A.4.2 on page 264.

[8] Stratification is discussed and exemplified in Section 4.3.2 on page 136.

Fully parenthesised	Using precedence and associativity
$(c_0 \,;\, c_1) \Cap (d_0 \,;\, d_1)$	$c_0 \,;\, c_1 \Cap d_0 \,;\, d_1$
$(a \,;_c\, b) \,;_c\, c$	$a \,;\, b \,;\, c$
$(c_0 \parallel c_1) \Cap (d_0 \parallel d_1)$	$(c_0 \parallel c_1) \Cap (d_0 \parallel d_1)$

Table 2.1 Precedence and associativity for RG-WSL

2.4 Relations

Most of the refinement laws contained in the chapters to follow deal with relations; thus, it is convenient to define a collection of relational operators used to compose relations. Figure 2.4 presents relational operators and also noteworthy relations used in this work. The relational connectives follow the usual precedence order from predicate calculus.

Since relations are defined over states, and these are modelled as total functions, relational composition (Eq. 2.11) is pretty straightforward, because it does not need to check if the domain of the states in the relations under composition match. The relation *idset X* (Eq. 2.15) is the identity relation over a set of variables X. The special case when X is the universe set (*UNIV*) is noted as *idrel*. The relation *depends-only* (g, X) (Eq. 2.16) is true if the relation g only depends on variables within X. Isabelle already has a built-in operator to denote the reflexive-transitive closure of a relation (r^{**}) and its transitive closure (r^{++}). For dealing with control structures (Section 3.12), we also need the reflexive-transitive closure on a closed subset A (Eq. 2.17), which behaves as the transitive closure, but allows reflexive transitions on variables within A.

Typographic conventions

To make specifications look more natural, we omit the subscript r in the relational operators defined in Table 2.4 when it is clear from the context that the arguments are relations. For example, if it is clear from the context that q and s are both relations, we write $q \wedge s$ instead of $q \wedge_r s$.

Operator		Definition	Associativity	
$\neg_r\, p$	$\equiv$	$\lambda s\ s'.\ \neg\, p\ s\ s'$		(2.7)
$p_0 \wedge_r p_1$	$\equiv$	$\lambda s\ s'.\ p_0\ s\ s' \wedge p_1\ s\ s'$	Left	(2.8)
$p_0 \vee_r p_1$	$\equiv$	$\lambda s\ s'.\ p_0\ s\ s' \vee p_1\ s\ s'$	Left	(2.9)
$p_0 \Rightarrow_r p_1$	$\equiv$	$\lambda s\ s'.\ p_0\ s\ s' \longrightarrow p_1\ s\ s'$	Right	(2.10)
$q_0\ ;_r\ q_1$	$\equiv$	$\lambda s\ s'.\ \exists s''.\ q_0\ s\ s'' \wedge q_1\ s''\ s'$	Right	(2.11)
true	$\equiv$	$\lambda s\ s'.\ True$		(2.12)
false	$\equiv$	$\lambda s\ s'.\ False$		(2.13)
idrel	$\equiv$	$\lambda s\ s'.\ s = s'$		(2.14)
idset X	$\equiv$	$\lambda s\ s'.\ \forall v.\ v \in X \longrightarrow s'\ v = s\ v$		(2.15)
depends-only (g, X)	$\equiv$	$(idset\ X\ ;\ g)\ ;\ idset\ X \Rightarrow g$		(2.16)
$r^{**}{}_X$	$\equiv$	$r^{++} \vee idset\ X$		(2.17)

Figure 2.4 Logical connectives and noteworthy relations

2.4.1 Predicates

To provide a uniform treatment of pre, post, guar and rely conditions, we use a single type to represent these entities, namely binary state relations (*relation*). The uniform treatment unburdens us from defining multiple operators for dealing with predicates (i.e. single-state relations), binary relations and their combination. For example, if p_1 and p_2 are predicates and q_1 and q_2 are relations, the conjunction of any two of these terms requires a single operator over relations: $\wedge_\lambda$, already defined in Figure 2.4. If we were using different types for representing these entities, say a type for relations and other for predicates, we would need to define four operators for conjunction (e.g. $p_1 \wedge_p p_2$, $p_1 \wedge_{pr} q_1$, $q_1 \wedge_{rp} p_1$ and $q_1 \wedge_r q_2$), which would lead to further redundancy in theorems.

Relations that do not restrict the post state are called *predicates*. Only predicates should be used to instantiate preconditions[9]. Example of predicates are the relations $\lambda s\ s'.\ [\![s\ x]\!]_n \geq 0$ and $\lambda s\ s'.\ [\![s\ x]\!]_s = \emptyset$. The relation $\lambda s\ s'.\ [\![s'\ x]\!]_n \geq 0$ is not a predicate because it constrains the after state. Formally, a relation p is a predicate if the satisfies the next definition.

Definition 2.18 (Predicate)**.** *For any relation p, we say that p is a predicate if pred p holds.*

$$pred\ p \equiv \forall s\ s_1\ s_2.\ p\ s\ s_1 = p\ s\ s_2$$

[9] If this condition is not satisfied, the user will find herself in a position that proof obligations cannot be discharged when applying refinement laws which involve preconditions.

2.4.2 Satisfiability

A relation r is satisfied for a pair of states σ and σ' if $r\ \sigma\ \sigma'$ holds. In such case we write $\sigma, \sigma' \models r$; if r is also a predicate, we shorthen the notation to $\sigma \models r$.

2.4.3 Post state notation

We sometimes need to state that a predicate holds in the after state. To formalise this notion we define a post-state notation, p‘, which only makes sense for relations that are also predicates. This notation is formalised as follows.

Definition 2.19 (Post-state notation)**.** *For a predicate p,*

$$p\text{‘} \equiv \lambda s\ s'.\ p\ s'\ s$$

2.4.4 Wellfounded relations

To ensure termination of loops, the rely-guarantee refinement calculus from Chapter 3 requires any relation used to introduce a loop to be *well-founded*. Here we adopt the formulation proposed in [49], which is close to the one proposed by the mathematician *Pierre de Fermat*: a relation r is well-founded provided there are no infinite sequences of states s_0, s_1, s_2,... such that all pairs of successive states are related by r. Logically, this can be formalised in terms of iteration of relations as follows.

Definition 2.20 (Iterated-Relation)**.** *For any relation r and natural number n,*

$$(iter\ r\ n) = \begin{cases} idrel, & if\ n = 0 \\ r\ ;\ iter\ r\ k, & n = Suc\ k. \end{cases}$$

Definition 2.21 (Wellfounded)**.** *For any relation r,*

$$wellfounded\ r \equiv \exists\ k.\ \forall\ s'.\ (iter\ r\ k)\ s\ s' = false\ s\ s'$$

A relation is well-founded if, for every state s, there is a natural number k, such that after k iterations, the iterated relation *iter r k* refuses to provide any matching state s' that relates to the initial state s, i.e., the iterated relation eventually disables itself after k iterations. Note that k can have different values based on the choice of the s.

In the context of proving termination of loops, we are interested in initial states s that satisfy a given precondition p. Thus, the definition above can be relaxed to require the relation r to be well-founded on a subset of its domain. To this purpose, a predicate p is used to filter out initial states which are not relevant.

Definition 2.22 (Wellfounded-Precondition)**.** *For any relation r and predicate p,*

$$\mathit{wellfounded}\ r\ p \equiv \forall s.\ p\ s\ s \longrightarrow (\exists k.\ \forall s'.\ \mathit{iter}\ r\ k\ s\ s' = \mathit{false}\ s\ s')$$

The next lemma introduces a well known result about well-founded relations: the transitive closure of a well-founded relation is also well-founded [15, 48]. This result is necessary to prove the introduction of loops in the refinement calculus in Chapter 3 (see law 3.112 on page 113).

Lemma 2.23 (Wellfounded-Transitive-Closure)**.** *For a predicate p and a relation w which is well-founded on p,*

$$\mathit{wellfounded}\ w^{++}\ p$$

2.5 Relational interpretation of expressions

The introduction of assignments, conditionals and loops requires a mechanism to interpret boolean expressions as relations. Boolean expressions are just expressions whose evaluation results in a *vvalue* denoting a boolean. This section introduces a denotational semantics for boolean expressions in terms of relations, which is represented by the function $[\![\text{-}]\!]_r$. This denotational semantics illustrates the interplay between our deep and shallow-embedding design decision. The application of relational interpretation of expressions is seen in laws which introduce control structures from specifications (see Section 3.12). The consequence of having separated embeddings for expressions is that program development starts from specifications formulated using shallow-embedded expressions (i.e. relations) which are replaced by deep-embedded expressions in the places where control structure and assignment are introduced.

Abbreviation 2.24 (Relational-Interpretation)**.** *Let bexp be a boolean expression,*

$$[\![bexp]\!]_r \equiv \lambda s\ s'.\ [\![[\![bexp]\!]_v\ s]\!]_b$$

Remark. *In this definition, the function $[\![\_]\!]_b$ strips the constructor VBool from its argument, resulting in a boolean, and function $[\![\_]\!]_v$ atomically evaluates its argument, resulting in a vvalue. Function $[\![\_]\!]_b$ was introduced in Definition 2.2, whereas function $[\![\_]\!]_v$ was introduced in Definition 2.6. Note that the grammar of expressions (Fig. 2.2) does not include primed variables, thus the relational denotation of a boolean expression is always a predicate.*

2.6 Logical interpretation of relations

In the refinement calculus from Chapter 3, a number of laws have proof obligations involving weakening and strengthening of relations, e.g. law 3.19 on page 74. Weakening or strengthening a relation generally involves proving that a relation in the form of implication is a tautology, e.g. $p \wedge q \Rightarrow q$. In [48], to denote that a relation whose main connective is an implication is a tautology, the main implication is replaced by $\Rrightarrow$, as in $p \Rightarrow q \Rrightarrow q$. This notation has the disadvantage that it can only capture tautologies whose main connective is an implication. To handle a broader variety of tautologies (e.g. $true \vee false$), we introduce the operator $(\vdash \_)$ to represent logical interpretation, i.e. that a relation evaluates to *true* independent of the pair of states taken as argument. Thus, instead of writing $(p \wedge q) \Rrightarrow q$ we write $\vdash p \wedge q \Rightarrow q$.

Definition 2.25 (Logical-Interpretation)**.** *Let r be a relation, then*

$$(\vdash r) \ \Leftrightarrow\ (r = true)$$

The next law provides an alternative definition of logical interpretation that is more suitable for practical application within theorem proving, because it applies a relation to a pair of states and get rid of the lambda operator in the definition of the relation. Thus, built-in laws of Isabelle can be directly applied to complete proofs involving logical interpretation.

Law 2.26 (Logical-Interpretation-Expanded)**.** *For any relation r,*

$$(\vdash q) \ \Leftrightarrow\ (\forall s\ s'.\ q\ s\ s')$$

Since proof obligations involving logical interpretation are recurrent in the application of laws from Chapter 3, law 2.26 is used quite frequently in our mechanisation. For non-composite relations, it suffices to apply law 2.26 and call the simplifier. For composite relations, this approach also requires expanding the definition of the connectives involved ($\wedge_r$, $\vee_r$, $\Rightarrow_r$, etc.). The drawback of expanding the definition of connectives is that it skips the abstraction level of relations, and when a proof involving logical interpretation fails, it complicates the identification of the precise reason why the proof failed. To remedy this situation, we provide a small collection of laws to allow compositional reasoning over logical interpretation in Section 4.2.1. These laws not only make the identification of problems in failed proofs easier, but they also enhance the automation of the mechanisation.

2.6.1 Example: reasoning compositionally about logical interpretation

To clarify the value of compositional reasoning over logical interpretation, we anticipate one of the laws presented in Section 4.2.1 and discuss how we used it to investigate a failed proof obligation in the derivation of the sequential version of Findp presented in Section 6.3. The next law splits the proof of $\vdash r \Rightarrow q \wedge s$ into two smaller subproofs. The symbol $\Longrightarrow$ is called meta-implication, and is used within Isabelle to separate the assumptions from the conclusion of a law.

$$\vdash r \Rightarrow q \wedge \vdash r \Rightarrow s \implies \vdash r \Rightarrow q \wedge s \tag{2.27}$$

Since a contextualised discussion of the failed proof obligation involves derived commands and laws which have not yet been discussed, we do not provide a detailed explanation of the proof obligation, and only care about its shape. Assume that s, and g_1,...,g_4 are relations; and that that the next proof obligation has to be discharged in order to complete the derivation of an algorithm.

$$\vdash s \Rightarrow g_1 \wedge g_2 \wedge g_3 \wedge g_4 \tag{2.28}$$

Without laws to decompose this proof, one has to expand all the definitions before invoking the simplifier within Isabelle. Performing this step generally leads to a large proof goal, where it may not be entirely clear the relationship between the original sub-goals (i.e. g_1,...,g_4) and those in the expanded goal. We met one of such situations in the derivation of Findp. There, the proof obligation had the shape described in Equation 2.28, and the premises were

insufficient to derive the conclusion. To help tracing the offending conjunct we used law 2.27 to split the original proof obligation into four independent sub proofs. This helped to reveal the offending conjunct and solve the problem.

2.7 Operational semantics

The main objective of this chapter is to discuss the notion of correctness for the refinement relation used in the refinement calculus of Chapter 3. To give further background to those unfamiliar, a refinement calculus establishes an ordering relation between programs, and requires three elements for its definition: a programming language, a formal semantics and a refinement relation. The first of these elements, a programming language, has already been discussed in Sections 2.1 and 2.3. This and the next section discuss the formal semantics.

Two semantics are provided for RG-WSL: an operational and a denotational semantics; these are taken from [48]. The operational semantics describes how a program executes, whereas the denotational semantics maps programs into sets of *traces* (i.e. sequences of labelled state transitions). The denotational semantics is used to define the refinement relation and its definition reuses the operational semantics. The dependencies between the semantic models are explained in Figure 2.5, which also shows the relationship between the semantic models and forward simulation (defined in Section 2.10).

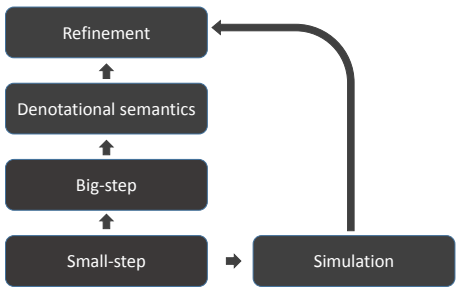


Figure 2.5 Semantic hierarchy. A dark arrow indicates that the target depends on the source. Small-step and big-step are subdivisions of the operational semantics. Labelled state transitions defined at the small-step semantics are used to induce a big-step semantics, and used to do proofs by forward simulation [75, 76, 73]. The denotational semantics builds on top of the big-step semantics and is used to formalise the notion of refinement. The longest arrow connects forward simulation to refinement and represents that forward simulation is a sufficient (but not necessary) condition to establish refinement.

The operational semantics subdivides into two semantics: *small-step* and *big-step*. The small-step semantics is the most powerful between the two. It allows a precise description of

the non-determinism and interleaving involved in concurrent executions, and allows us to reason about the intermediate states that programs go through during their execution. For RG-WSL, the small-step semantics is formalised as a ternary relation: it takes a command, a labelled state transition, and another command. The labelled state transition contains three components: the state preceding the execution of an atomic action, the state succeeding the execution, and a label to identify if the state transition is result of a program step or an environment step, or if it is a termination step (in this case the preceding and succeeding states are the same). Big-step semantics (also known as *natural semantics* [81, 85]) describe the execution of a program as an end-to-end transformation. In RG-WSL, the big-step semantics is formalised by a ternary relation: it takes a command, a sequence of labelled state transitions, and a final command. The sequence of labelled state transitions is called a *trace*, and can be finite or infinite (this is the case of non-terminating programs). Although commands generally admit both styles of semantics, each of these styles has particular advantages over the other. For example, the big-step semantics cannot distinguish situations where a program does not terminate and aborted from situations where a program does not terminate but did not abort. On the other hand, the small-step semantics can differentiate between abortive and non-abortive behaviours. In the case of concurrent programming, it is well-known that certain features of programming languages are more easily described using a small-step semantics. We do not attempt a comparison of the power of these semantics in this thesis; instead, we simply discuss how the operational semantics of RG-WSL can be formulated as a combination of small-step and big-step semantics. Our mechanisation includes the small-step semantics presented in [48], adapted to fit the findings from Section 2.3 (i.e., the problem involving the encoding of unbounded non-determinism). The big-step semantics and denotational semantics are only partially mechanised, and do not play a key role in proofs in our mechanisation.

The use of a big-step semantics to define a denotational semantics is an arbitrary decision taken in [48]. The most common approach in the literature is to define the denotational semantics without resorting to an operational semantics; this is done, for example, by Dingel in [32], who also develops a refinement calculus for rely-guarantee. The preference for an operational stylem in [48] is justified on the grounds of it being more readable than a denotational alternative.

Before moving to the next subsection, we discuss the type of labelled state transitions. These are encoded using the datatype *LSTransition* whose definition is given next.

datatype *LSTransition* = π *state state* | ϵ *state state* | υ *state*

For program (π) and environment (ϵ) steps (i.e. transitions), both preceding and succeeding states are recorded. For termination (υ) steps, only the final state is recorded. This datatype is complemented with two additional functions: *pre* and *post*, which extract, respectively, the before and after state of a state transition. For clarity, we also introduce a projection to extract the label of a transition. Labels are defined as program step (π), environment step (ϵ) or termination step (υ).

Definition 2.29 (LSTransition-Projections)**.** *For any states* σ *and* σ'*,*

$$\begin{array}{lll} \mathit{pre}\ (\pi\ \sigma\ \sigma') = \sigma & \mathit{post}\ (\pi\ \sigma\ \sigma') = \sigma' & \mathit{label}\ (\pi\ \sigma\ \sigma') = \pi \\ \mathit{pre}\ (\epsilon\ \sigma\ \sigma') = \sigma & \mathit{post}\ (\epsilon\ \sigma\ \sigma') = \sigma' & \mathit{label}\ (\epsilon\ \sigma\ \sigma') = \epsilon \\ \mathit{pre}\ (\upsilon\ \sigma) = \sigma & & \mathit{label}\ (\upsilon\ \sigma) = \upsilon \end{array}$$

The next subsection discusses evaluation of expressions considering interference. This evaluation is used later to define the semantics for the test command.

2.7.1 Expression evaluation

The evaluation of expressions (*Exp*) assumes that each occurrence of a variable can be fetched atomically from the state. In standard programming languages such as C and Java, this sampling policy can be enforced by declaring variables as *volatile* [44]. The fetching order is arbitrary and the evaluation can be interrupted by environment actions. Such interruptions do not change the expression under evaluation, but can change the state, possibly affecting the subsequent evaluation of variables.

The assumption that variables are read atomically ensures that programs cannot perform inconsistent reads in the sense that update and fetching operations never overlap[10]. Variable reads are formalised by the next law. A program step without state changes (i.e. atomic fetch) inspects the before state (σ) to look up the value (v) of a variable (x).

$$\frac{\sigma\ x = v}{V\ x \xrightarrow{\pi\ \sigma\ \sigma}_e N\ v} \tag{2.30}$$

To evaluate a binary expression, the operands are evaluated to values in a non-deterministic order. In an operational semantics, non-determinism is encoded by having multiple rules

[10]This is a realistic assumption for variables of scalar types, but it does not realistically model the reading of arrays. Arrays are object of discussion in Section 5.4.

which can be chosen at a certain point of the execution. Two rules are provided for evaluating operands of a binary expression in 2.31 and 2.32. In these rules the term α subsumes state transitions of the form ($\pi\ \sigma\ \sigma$) and ($\epsilon\ \sigma\ \sigma'$), the term e is an expression (*Exp*) and the term b is a binary operator (i.e. *binop*).

$$\frac{exp_1 \xrightarrow{\alpha}_e exp_1'}{BOp\ b\ exp_1\ exp_2 \xrightarrow{\alpha}_e BOp\ b\ exp_1'\ exp_2} \tag{2.31}$$

$$\frac{exp_2 \xrightarrow{\alpha}_e exp_2'}{BOp\ b\ exp_1\ exp_2 \xrightarrow{\alpha}_e BOp\ b\ exp_1\ exp_2'} \tag{2.32}$$

The final value of a binary expression is computed by applying the corresponding underlying mathematical operator to the values resulting from the evaluation of the operands. Two cases are considered in the evaluation of binary operators: in the first case (2.34) the expression is well-defined and in the second case (2.35) the expression is not well-defined. Our encoding of undefined using the constant *undefined* can be seen as simplistic, in the sense that it does not introduce enough information to prove that *undefined* is different from any ordinary value (e.g. *VBool True*). Nevertheless, it suffices the purpose of discussing the effect of interference in expression evaluation. The function *eval-binary* extracts the underlying mathematical operation from the definition of a binary operator.

$$eval\text{-}binary\ b \equiv fst\ b \tag{2.33}$$

$$\frac{\sigma \models defined\ (BOp\ b\ (N\ v_1)\ (N\ v_2)) \qquad v' = eval\text{-}binary\ b\ v_1\ v_2}{BOp\ b\ (N\ v_1)\ (N\ v_2) \xrightarrow{\pi\ \sigma\ \sigma}_e N\ v'} \tag{2.34}$$

$$\frac{\sigma \models (\neg\ defined\ (BOp\ b\ (N\ v_1)\ (N\ v_2)))}{BOp\ b\ (N\ v_1)\ (N\ v_2) \xrightarrow{\pi\ \sigma\ \sigma}_e N\ undefined} \tag{2.35}$$

Unary operators (represented by u in the next rules) are evaluated similarly to binary operators (cf. rules 2.31 and 2.37, 2.34 and 2.38, and 2.35 and 2.39). Similarly, the function *eval-unary* extracts from the definition of an unary operator the underlying mathematical operator

represented by it.

$$\mathit{eval\text{-}unary}\ u \equiv \mathit{fst}\ u \tag{2.36}$$

$$\frac{\mathit{exp} \xrightarrow{\alpha}_e \mathit{exp}'}{\mathit{UOp}\ u\ \mathit{exp} \xrightarrow{\alpha}_e \mathit{UOp}\ u\ \mathit{exp}'} \tag{2.37}$$

$$\frac{\sigma \models \mathit{defined}\ (\mathit{UOp}\ u\ (N\ v)) \qquad v' = \mathit{eval\text{-}unary}\ u\ v}{\mathit{UOp}\ u\ (N\ v) \xrightarrow{\pi\ \sigma\ \sigma}_e N\ v'} \tag{2.38}$$

$$\frac{\sigma \models (\neg\ \mathit{defined}\ (\mathit{UOp}\ u\ (N\ v)))}{\mathit{UOp}\ u\ (N\ v) \xrightarrow{\pi\ \sigma\ \sigma}_e N\ \mathit{undefined}} \tag{2.39}$$

The environment can interrupt the evaluation of an expression at any time. The next rule formalises such an interruption, and allows for traces which are finitely and infinitely interrupted by the environment.

$$\frac{}{\mathit{exp} \xrightarrow{\epsilon\ \sigma\ \sigma'}_e \mathit{exp}} \tag{2.40}$$

To better explain expression evaluation, we return to an example given in the introduction chapter: the expression $(x{+}x)$. We stated previously that in the presence of interference, expression evaluation does not observe mathematical laws, e.g. if x is a shared variable, the evaluation of $(x{+}x)$ can result in an odd value. Now we illustrate how the semantics explains this non-observance. Assume that the evaluation of $(x{+}x)$ starts in a state σ such that $(\sigma\ x = \mathit{VNat\ 1})$, and assume that $(\sigma'\ x = \mathit{VNat\ 4})$. Then, formally

$$\begin{aligned}
V\ x +_n V\ x &\xrightarrow{\pi\ \sigma\ \sigma}_e N\ (\mathit{VNat\ 1}) +_n V\ x && \text{by 2.31, 2.30}\\
N\ (\mathit{VNat\ 1}) +_n V\ x &\xrightarrow{\epsilon\ \sigma\ \sigma'}_e N\ (\mathit{VNat\ 1}) +_n V\ x && \text{by 2.40}\\
N\ (\mathit{VNat\ 1}) +_n V\ x &\xrightarrow{\pi\ \sigma'\ \sigma'}_e N\ (\mathit{VNat\ 1}) +_n N\ (\mathit{VNat\ 4}) && \text{by 2.32, 2.30}\\
N\ (\mathit{VNat\ 1}) +_n N\ (\mathit{VNat\ 4}) &\xrightarrow{\pi\ \sigma'\ \sigma'}_e N\ (\mathit{VNat\ 5}) && \text{by 2.34}
\end{aligned}$$

Thus, the evaluation of $x + x$ in an interfering environment can result in an odd number. In the expression above $+_n$ corresponds to addition over naturals (cf. mod_n on page 27). Addition

is defined for any pair of naturals, and thus the expression $V\,x +_n V\,x$ is well-defined. To illustrate undefinedness we consider the expression $(N\ (VNat\ 5)\ mod_n\ V\,y) +_n V\,x$. Assume that the evaluation starts in a state σ such that ($\sigma\ x = VNat\ 4$) and ($\sigma\ y = VNat\ 0$). To fit the evaluation into a single line, we omit the references to the rules applied at each step.

$$
\begin{aligned}
(N\ (VNat\ 5)\ mod_n\ y) +_n V\,x &\xrightarrow{\pi\ \sigma\ \sigma}_e (N\ (VNat\ 5)\ mod_n\ N\ (VNat\ 0)) +_n V\,x \\
(N\ (VNat\ 5)\ mod_n\ N\ (VNat\ 0)) +_n V\,x &\xrightarrow{\pi\ \sigma\ \sigma}_e N\ undefined +_n V\,x \\
N\ undefined +_n V\,x &\xrightarrow{\pi\ \sigma\ \sigma}_e N\ undefined +_n N\ (VNat\ 4)
\end{aligned}
$$

At this point, the execution cannot proceed unless we define the result of addition ($+_n$) for the case where the left-hand side operand is *undefined*. Since in the context of a refinement calculus we are generally interested in preventing computing undefined, it does not make sense to define special cases to handle undefinedness. Instead, a generic approach to handle undefinedness is to use the precondition of $+_n$ to specify that this operation can only be applied if the operands are well-defined. This prevents the introduction of undefined expressions in derived programs. Taking this approach corresponds to propagating *undefined* whenever a sub-term of an expression evaluates to *undefined*, because it forces the evaluation to follow rule 2.35. Using this approach the evaluation would proceeds as:

$$N\ undefined +_n N\ (VNat\ 4) \quad \rightarrow_e[\pi\ \sigma\ \sigma] \quad N\ undefined.$$

However, to detect *undefined* as a value in the definition of the precondition of $+_n$ we would need to be able to distinguish *undefined* from any other ordinary value, e.g. *VNat 1*. Since our representation of *undefined* is simplistic, it does not suffice this purpose (e.g. we cannot prove that $1 \neq undefined$). To enable the theory to tell apart a value from *undefined*, the user would need to extend the *vvalue* (Definition 2.1) with a new constructor, specifically designed to represent *undefined* as a distinct value. We do not need this extension for deriving programs, because the refinement laws in Chapter 3 prevent the introduction of undefined expressions in programs.

Multi-step expression evaluation

The single-step transition $e \xrightarrow{\alpha}_e e'$ induces a multi-step transition, $e \xrightarrow{l_\alpha}_{e*} e'$, where l_α is a sequence of labelled state transitions. The transition $e \xrightarrow{l_\alpha}_{e*} e'$ allows for sequences of labelled state transitions which have arbitrary length. In [48], the description of the semantics suggests that sequences of transitions can have infinite length. Such infinite sequence of

transitions correspond to evaluations which fail to terminate. As we use lists to represent sequences of transitions, our mechanisation only models finite traces. As our refinement proofs involving expressions do not descent to the level of the semantics, this simplification does not affect our algebraic characterisation. Next, *hd* is the conventional head operator for lists (e.g. $hd\ [a, b, c] = a$) and *last* returns the last element of a list (e.g. $last\ [a, b, c] = c$).

$$\frac{exp \xrightarrow{\alpha}_{e} exp'}{exp \xrightarrow{[\alpha]}_{e*} exp'} \tag{2.41}$$

$$\frac{\begin{array}{ccc} l_1 \neq [] & l_2 \neq [] & l = l_1 \frown l_2 \\ exp \xrightarrow{l_1}_{e*} exp'' & exp'' \xrightarrow{l_2}_{e*} exp' & pre\ (hd\ l_2) = post\ (last\ l_1) \end{array}}{exp \xrightarrow{l}_{e*} exp'} \tag{2.42}$$

For example, using $\rightarrow_{e*}$ the evaluation of $x{+}x$ discussed before is synthesised next. Recall from our previous discussion that we assume that $\sigma\ x = VNat\ 1$ and $\sigma'\ x = VNat\ 4$.

$$(V\ x +_n V\ x) \xrightarrow{[\pi\ \sigma\ \sigma,\ \epsilon\ \sigma\ \sigma',\ \pi\ \sigma'\ \sigma',\ \pi\ \sigma'\ \sigma']}_{e*} N\ (VNat\ 5).$$

The multi-step evaluation is used to define an operational semantics for tests ($[[\_]]$) in Section 2.7.3, and the notion of reachable expressions in Section 5.5.

2.7.2 Small-step semantics

To describe the operational semantics, the programming language is extended with the command *nil*, which represents the terminated command, i.e. a command that cannot do any steps whatsoever. The small-step semantics for commands is able to tell apart normal and abortive behaviour. Such expressiveness allows one to differentiate programs that do not terminate because they engaged into an abortive behaviour from programs that do not terminate because they fail to emit a termination step (υ).

The fact that a command meets the criteria for aborting from a given state is denoted by its *abortive condition*. The abortive conditions ($c_{\sigma\times}$) are defined via an *inductive predicate*[11] in Isabelle/HOL, and its definition takes two parameters: a command (c) and a state (σ) – the

[11] Inductive predicates are used to specify relations inductively from a set of rules. The relation generated by an inductive predicate corresponds to the smallest relation induced by the rules provided by the user. For example, the relations $\_ \xrightarrow{\alpha}_{e} \_$ and $\_ \xrightarrow{[\alpha]}_{e*} \_$ are typical examples of relations that are encoded in Isabelle/HOL using inductive predicates.

subscript "$\times$" in the definition is just decorative. Whenever the *abortive condition* holds for a given command c and state σ, the execution of c from state σ can take a non-deterministic route, described by a set of three rules in the small-step semantics that are characteristic of programs that have aborted. Following the presentation style from [48], we introduce the abortive conditions together with the small-step semantics.

To give the reader a better understanding of our contributions in the formalisation of the small-step semantics, this section reproduces the abortive conditions as they are defined in [48]. In the original definition, if a branch of a parallel composition aborts, the parallel composition itself aborts. As we will see in Section 5.3 (Revised abortive conditions), the abortive conditions for parallel composition reproduced in this section can be inadvertently used to equate the commands *magic*, defined as the non-deterministic choice over an empty set, and *abort*, defined as $\{false\}$, which have very distinct roles in a refinement calculus. To prevent this situation from happening, we discuss the problem and amend the abortive conditions in Chapter 5. The current chapter is not the appropriate place to discuss this problem into further details, because to understand how the theory can be exploited, we need to be familiarised with the refinement laws introduced in Chapter 3.

A precondition $\{p\}$ terminates immediately in any state σ in which p holds, and aborts in any state p does not hold. The command *skip*, defined as $skip \equiv \{true\}$, terminates immediately, and *abort*, defined as $abort \equiv \{false\}$, aborts immediately.

$$\frac{\sigma \models p}{\{p\} \xrightarrow{\upsilon\ \sigma} nil} \qquad \frac{\neg\ \sigma \models p}{\{p\}_{\sigma\times}} \tag{2.43}$$

The next three rules characterise a program that has aborted. These rules define that a program that aborted in a state σ may take any further finite or infinite behaviour, or terminate immediately. Finite behaviour corresponds to any sequence of steps that ends in a termination step (υ), while infinite behaviour corresponds to a sequence of steps which includes no termination step.

$$\frac{c_{\sigma\times}}{c \xrightarrow{\pi\ \sigma\ \sigma'} abort} \qquad \frac{c_{\sigma\times}}{c \xrightarrow{\epsilon\ \sigma\ \sigma'} abort} \qquad \frac{c_{\sigma\times}}{c \xrightarrow{\upsilon\ \sigma} nil} \tag{2.44}$$

The atomic command $\langle p,\, q\rangle$ can do a program step if p holds or abort if p does not hold. The execution of the program step can be preceded by any number of environment steps ($\epsilon\ \sigma\ \sigma'$).

$$\frac{\sigma \models p \qquad \sigma,\sigma' \models q}{\langle p,\, q\rangle \xrightarrow{\pi\ \sigma\ \sigma'} \mathit{skip}} \qquad \frac{}{\langle p,\, q\rangle \xrightarrow{\epsilon\ \sigma\ \sigma'} \langle p,\, q\rangle} \qquad \frac{\neg\ \sigma \models p}{\langle p,\, q\rangle_{\sigma\times}} \tag{2.45}$$

Non-deterministic choice between a countable set of commands C can behave as any command within C. If any command $c \in C$ can terminate, then the choice can terminate; whereas if any $c \in C$ can abort, then the choice can abort [48]. In the next definition, *acset* is a function that takes a set of commands C, and returns an object of type *Command cset*. Recall from Figure 2.3 (page 29) that the type expected by non-deterministic choice is *Command cset*. The success of the application of *acset* depends on C to be a countable set. Thus, this requirement is introduced in the set of premises of the next rules.

$$\frac{c \in C \qquad \mathit{countable}\ C \qquad c \xrightarrow{\alpha} c'}{\prod \mathit{acset}\ C \xrightarrow{\alpha} c'} \qquad \frac{c \in C \qquad \mathit{countable}\ C \qquad c_{\sigma\times}}{(\prod \mathit{acset}\ C)_{\sigma\times}} \tag{2.46}$$

Note that it is possible for both rules to be active if a C contains a program that aborts. This means that non-deterministic choice does not force the execution of a program that can abort to necessarily follow using rule 2.44.

A sequential composition, $c_1\ ;\ c_2$, executes c_1 until it terminates, after which c_2 may begin, provided c_2 begin execution in the state in which c_1 terminated. In general, the small-step semantics formalised in [48] does not force abortive and non-abortive rules to be mutually exclusive. Such flexibility allows programs to recover from abortion. For example, if the first component of a sequential composition is a precondition that aborts (e.g. $\{\mathit{false}\}\ ;\ c$), then the execution of the second component (c) can either abort or proceed. In the first case the abortion is propagated to the whole sequential composition by rule 2.47, whereas in the second case, the effect of the abortion of the precondition is kept local, by using rule 2.44 to describe the abortive behaviour, and then using rule 2.48 to allow the sequential composition to continue its execution upon termination of the command that aborted.

$$\frac{\mathit{label}\ \alpha \neq \upsilon \qquad c_1 \xrightarrow{\alpha} {c_1}' \qquad c_{1\sigma\times}}{c_1\ ;\ c_2 \xrightarrow{\alpha} {c_1}'\ ;\ c_2 \qquad (c_1;c_2)_{\sigma\times}} \tag{2.47}$$

$$\frac{\mathit{pre}\ \alpha = \sigma \qquad c_1 \xrightarrow{\upsilon\ \sigma} \mathit{nil} \qquad c_2 \xrightarrow{\alpha} {c_2}'}{c_1\ ;\ c_2 \xrightarrow{\alpha} {c_2}'} \tag{2.48}$$

Sequential composition $c_1\ ;\ c_2$ fails to terminate if c_1 fails to terminate. Additionally, a sequential composition does not anticipate abortion, e.g. the composition c ; *abort* first executes c, and only upon termination of c, aborts.

A strict conjunction of two commands $c \Cap d$ behaves in a manner consistent with both c and d, terminating when both terminate. It only requires one branch to abort for the whole composition to abort.

$$\frac{\mathit{label}\ \alpha \neq \upsilon \qquad c_1 \xrightarrow{\alpha} {c_1}' \qquad c_2 \xrightarrow{\alpha} {c_2}'}{c_1 \Cap c_2 \xrightarrow{\alpha} {c_1}' \Cap {c_2}'} \tag{2.49}$$

$$\frac{c_1 \xrightarrow{\upsilon\ \sigma} \mathit{nil} \qquad c_2 \xrightarrow{\upsilon\ \sigma} \mathit{nil}}{c_1 \Cap c_2 \xrightarrow{\upsilon\ \sigma} \mathit{nil}} \tag{2.50}$$

$$\frac{c_{1\sigma\times}}{(c_1 \Cap c_2)_{\sigma\times}} \qquad \frac{c_{2\sigma\times}}{(c_1 \Cap c_2)_{\sigma\times}} \tag{2.51}$$

The characterisation of parallel composition, $c \parallel d$, requires a matching relation to compose a program step of c with an environment step of d (and vice-versa) to give a program step of the composition. This relation also matches an environment of both commands to produce an environment step of their composition.

Definition 2.52 (Match)**.** *Matching between environment and program steps is defined inductively as*

$$(\pi\ \sigma\ \sigma', \epsilon\ \sigma\ \sigma')\ \mathit{match}\ \pi\ \sigma\ \sigma'$$
$$(\epsilon\ \sigma\ \sigma', \pi\ \sigma\ \sigma')\ \mathit{match}\ \pi\ \sigma\ \sigma'$$

$$(\epsilon\ \sigma\ \sigma', \epsilon\ \sigma\ \sigma')\ \mathit{match}\ \epsilon\ \sigma\ \sigma'$$

The next rule models unfair parallel composition, *i.e.* it does not prevent one branch from infinitely overtaking the chance of the other branch perform program steps. A consequence of the fact that parallel composition used in this work is unfair is that the refinement theory in this thesis cannot be used to prove properties over programs that depend on the environment to terminate, such as algorithms involving busy waiting and critical regions.

$$\frac{c_1 \xrightarrow{\alpha_1} c_1{}' \qquad c_2 \xrightarrow{\alpha_2} c_2{}' \qquad (\alpha_1, \alpha_2)\ \mathit{match}\ \alpha}{c_1 \parallel c_2 \xrightarrow{\alpha} c_1{}' \parallel c_2{}'} \tag{2.53}$$

If one of the branches terminates earlier than the other, then the remaining branch subsumes the parallel composition. To ensure that the remaining branch continues its execution from the state where its sibling terminates, the condition $\mathit{pre}\ \alpha = \sigma$ is required to hold.

$$\frac{c_2 \xrightarrow{v\ \sigma} \mathit{nil} \qquad c_1 \xrightarrow{\alpha} c_1{}' \qquad \mathit{pre}\ \alpha = \sigma}{c_1 \parallel c_2 \xrightarrow{\alpha} c_1{}'} \tag{2.54}$$

$$\frac{c_1 \xrightarrow{v\ \sigma} \mathit{nil} \qquad c_2 \xrightarrow{\alpha} c_2{}' \qquad \mathit{pre}\ \alpha = \sigma}{c_1 \parallel c_2 \xrightarrow{\alpha} c_2{}'} \tag{2.55}$$

The whole composition aborts if any of the branches aborts.

$$\frac{c_1 \sigma\times}{(c_1 \| c_2)\sigma\times} \qquad \frac{c_2 \sigma\times}{(c_1 \| c_2)\sigma\times} \tag{2.56}$$

A local state command (**state** $y \mapsto v \cdot c$) introduces and initialises a local variable y to value v. It also limits the scope of the local variable (y) to the program in its body (i.e., c), and shadows the global variable y within (**state** $y \mapsto v \cdot c$)[12]. Environment steps of (**state** $y \mapsto v \cdot c$) are explicitly prevented from reading and modifying the local variable y,

[12]Recall that variables are not declared in RG-WSL. Variables are meant to be *global* per definition, with local variables being introduced using the (**state** $y \mapsto v \cdot c$) command. Shadowing of variables is illustrated in Section refsec:shadowing.

but can access the global variable y. The notation $\sigma(y := v)$ denotes the state σ with the value of y updated to v.

$$\frac{c \xrightarrow{\pi\ (\sigma(y := v))\ (\sigma'(y := v'))} c' \qquad \sigma'\, y = \sigma\, y}{\textbf{state}\ y \mapsto v \cdot c \xrightarrow{\pi\ \sigma\ \sigma'} \textbf{state}\ y \mapsto v' \cdot c'} \tag{2.57}$$

$$\frac{c \xrightarrow{\epsilon\ \sigma(y := v)\ \sigma'(y := v)} c'}{\textbf{state}\ y \mapsto v \cdot c \xrightarrow{\epsilon\ \sigma\ \sigma'} \textbf{state}\ y \mapsto v \cdot c'} \tag{2.58}$$

$$\frac{c \xrightarrow{\upsilon\ (\sigma(y := v))} nil}{\textbf{state}\ y \mapsto v \cdot c \xrightarrow{\upsilon\ \sigma} nil} \tag{2.59}$$

$$\frac{c_{\sigma(y := v)\times}}{\textbf{state}\ y \mapsto v \cdot c_{\sigma\times}} \tag{2.60}$$

Rule 2.57 states that program steps do not affect the value of the global variable y (*i.e.* $\sigma'\, y = \sigma\, y$), but can update the value hold by the local variable y from v to v'. Updates to the local variable y are recorded using the binder $y \mapsto v$ of the **state** command. Rule 2.58 states that the local variable y is private, thus it cannot be read or modified by the environment of c. The environment can however to read and modify the global variable y. Rule 2.59 promotes the termination step (υ) in a state which is consistent with the local variable to a termination step of the state block. When this rule is applied, the local value of y is forgotten by the operational semantics.

The **uses** command enforces a syntactic restriction on the set of global variables that a program can use. A program step $\pi\ \sigma\ \sigma'$ of **uses** $X \cdot c$ is permitted if, and only if, c may take essentially the same program step for every pair of states that is equal to (σ, σ') in X. The term $\overline{X}$ denotes the complement of the set X, and the notation $\sigma \overset{X}{=} \sigma'$ denotes that states σ and σ' are identical with respect to the value of variables in X.

$$\frac{\sigma \overset{\overline{X}}{=} \sigma' \qquad \forall \sigma_1\ \sigma_1'.\ \sigma \overset{X}{=} \sigma_1 \wedge \sigma' \overset{X}{=} \sigma_1' \wedge \sigma_1 \overset{\overline{X}}{=} \sigma_1' \longrightarrow c \xrightarrow{\pi\ \sigma_1\ \sigma_1'} c'}{\textbf{uses}\ X \cdot c \xrightarrow{\pi\ \sigma\ \sigma'} \textbf{uses}\ X \cdot c'} \tag{2.61}$$

Next rules state that a **uses** command does not constrain environment and termination steps. Thus, it only imposes restrictions on the implementation of a program, not on the actions of its environment.

$$\frac{c \xrightarrow{\epsilon\ \sigma\ \sigma'} c'}{\textbf{uses}\ X \cdot c \xrightarrow{\epsilon\ \sigma\ \sigma'} \textbf{uses}\ X \cdot c'} \tag{2.62}$$

$$\frac{c \xrightarrow{\upsilon\ \sigma} nil}{\textbf{uses}\ X \cdot c \xrightarrow{\upsilon\ \sigma} nil} \tag{2.63}$$

$$\frac{c\sigma\times}{\textbf{uses}\ X \cdot c\sigma\times} \tag{2.64}$$

2.7.3 Big-step semantics

This section discusses two commands which are described using a big-step semantics in [48]: test and postcondition command. Although we provide an overview of the big-step semantics, we did not mechanise it. The reason for not encoding the full-model for the semantics in this work is because our characterisation of rely guarantee is based on the algebraic approach, and for that we only need to assume the validity of a set of refinement laws from which we can derive more laws. The laws taken as basis for the algebraic characterisation corresponds to the *lemmas* introduced in the next chapter. The encoding of the semantics is needed to prove the soundness of these lemmas, but this is not included in the scope of our investigation.

The mechanisation of the big-step semantics provided in [48] would require the extension of the programming language with commands to represent the result of infinite and partial computations. These additional complications are not necessary for providing an algebraic mechanisation of the rely-guarantee refinement calculus discussed in the next chapter. Next we discuss parts of the big-step semantics of the test command provided in [48]. To preventing introducing a number of definitions that are not relevant in the scope of this work, we provide only comments on the big-step semantics for the postcondition command ($[\_]$). For more details we refer the reader to the original presentation of the semantics in [48].

Semantics of tests

Tests ([[_]]) are one of the two commands of RG-WSL which are described using a big-step semantics ($\_ \overset{t}{\Rightarrow} \_$), which gives an end-to-end view of its execution; the other command is the specification command ([_]). The transition relation $c \overset{t}{\Rightarrow} c'$ takes three arguments: a command c, a trace t (i.e. a sequence of labelled state transitions), and another command c'. It denotes that c evolves to c' after the sequence of labelled state transitions has been executed. For $c' = nil$, it represents the complete execution of c.

Only tests that succeed contribute towards the trace of a program, thus the semantics for tests does not provide a rule to promote the traces of tests which evaluate to *false*, that is, failed tests are trace equivalent to $magic \equiv \bigsqcap acset\ \emptyset$. This lack of trace for tests which evaluate to *false* is necessary to model commands such as *if-then-else* (Definition 3.13 on page 69), where the decision of which branch to execute is modelled using non-deterministic choice. This formulation is viable because the branch whose conditional evaluates to *false* is eliminated from the set of choices, since *magic* is absorbed by non-deterministic choice. Next we use the notation $t_0 \overset{st}{=} t_1$ to state that traces t_0 and t_1 are identical modulo finite stuttering of programs.

$$\frac{e \xrightarrow{t_0}_{e*} N\ (VBool\ True) \qquad t_0 \overset{st}{=} t_1}{[[e]] \overset{t_1}{\Rightarrow} skip} \tag{2.65}$$

The next rule states that a test aborts if the evaluation of an expression results in undefined.

$$\frac{e \xrightarrow{t_0}_{e*} undefined \qquad t_0 \overset{st}{=} t_1}{[[e]] \overset{t_1}{\Rightarrow} abort} \tag{2.66}$$

The next two rules induce a big-step semantics for primitive commands which have a defined small-step semantics. Taking c' to be *nil* corresponds to the termination of command c.

$$\frac{c \xrightarrow{\alpha} c'}{c \overset{[\alpha]}{\Longrightarrow} c'} \tag{2.67}$$

$$\frac{l = l_1 \frown l_2 \qquad c \overset{l_1}{\Rightarrow} c'' \qquad l_1 \neq [] \qquad l_2 \neq [] \qquad c'' \overset{l_2}{\Rightarrow} c' \qquad pre\ (hd\ l_2) = post\ (last\ l_1)}{c \overset{l}{\Rightarrow} c'} \tag{2.68}$$

Semantics for postcondition command

The semantics of $[q]$ requires the relation q to be established between the initial (σ) and final (σ') states. This requirement is suppressed in two situations: (i) if the environment does any step that changes the state or, if the environment unfairly interrupts the execution of $[q]$. In the first situation $[q]$ aborts, while in the second situation its execution is described by traces that contain an infinite and continuous sequence of environment steps. The execution of $[q]$ allows for immediate termination if $\sigma, \sigma \models q$, thus, a specification such as $[idrel]$ can terminate immediately.

2.8 Denotational semantics

This section uses the big-step semantics from Section 2.7.3 to define a denotational semantics, which is the one used to define the refinement relation in the next section. For a finite trace of steps, t, the relation is written $c \overset{t}{\Rightarrow} nil$, and if t is infinite, it is written $c \overset{t}{\Rightarrow} \infty$. The meaning of a command c is the collection of all (finite and infinite) complete traces it may generate that either terminate or are non-terminating [48].

$$[\![c]\!] \equiv \{t \mid c \overset{t}{\Rightarrow} nil \vee c \overset{t}{\Rightarrow} \infty\} \tag{2.69}$$

Recall that we have not mechanised the big-step semantics. In the mechanisation (Appendix A) we introduce the denotational semantics ($[\![\_]\!]$) as an uninterpreted constant. This type of specification is useful when we want to define the type of a constant without providing further details about its definition.

In general, the semantics of a command is given for arbitrary environment transitions, as defined above. However, to be able to discuss refinement in a concurrent setting, as it will be the case for rely-guarantee, it is often necessary to restrict consideration to a particular environment. Given a trace t, the environment relation *env t* can be constructed by collecting the pairs of states in each of the environment transitions in t. Next, the operator *set* returns

the set of elements of a list (e.g. $set\ [a, b, a] = \{a, b\}$).

$$env\ t \equiv \lambda s\ s'.\ \epsilon\ s\ s' \in set\ t \tag{2.70}$$

For example, consider the finite trace t_0 given next

$$t_0 = [\epsilon\ \sigma_0\ \sigma_1, \pi\ \sigma_1\ \sigma_2, \epsilon\ \sigma_2\ \sigma_3, \epsilon\ \sigma_3\ \sigma_4, \pi\ \sigma_4\ \sigma_5, \epsilon\ \sigma_5\ \sigma_6, \epsilon\ \sigma_6\ \sigma_7, \pi\ \sigma_7\ \sigma_8, \upsilon\ \sigma_8]$$

The relation $(env\ t_0)$ holds only for pairs of states that are related by environment steps (ϵ) in t_0. Thus, assuming the states to be distinct, we have $(\sigma_0, \sigma_1 \models env\ t_0)$ and $(\neg\ (\sigma_1, \sigma_2 \models env\ t_0))$. The traces of c for which the environment steps are defined to satisfy r or stutter are defined as follows.

$$[\![c]\!][r] \equiv \{t \in [\![c]\!] \mid \vdash ((env\ t) \Rightarrow (r \vee idrel))\} \tag{2.71}$$

The next definition is provided in [48] and serves to structure proofs involving unbounded choice using the denotational semantics.

Definition 2.72 (Trace countable-choice)**.** *Let C be a countable set of commands, and r a relation,*

$$[\![\bigsqcap\ acset\ C]\!][r] = \bigcup \{[\![c]\!][r] \mid c \in C\}$$

2.9 Refinement

We now introduce a refinement relation ($a \sqsubseteq[r]\ c$) takes three parameters: the syntactic encoding of the abstract program (a), a relation representing the context in which the refinement holds (r), and the syntactic encoding of the concrete program (c). The notion of refinement that we will introduce corresponds to the intuition of reducing non-determinism. This means that we consider a program d to be a refinement of a program c if all the traces d are contained in the traces of c. The relation r abstracts the state transitions of the environment, and is used to delimit the set of traces that are relevant for establishing the comparison.

Formally, we say that a program d refines a program c in an environment r if the traces of d are contained in the traces of c when only environment transitions that respect r are considered.

Definition 2.73 (Refinement-in-Context)**.** *Let c and d be commands, and r a relation,*

$$c \sqsubseteq[r]\ d \ \equiv\ [\![d]\!][r] \subseteq [\![c]\!][true]$$

An immediate consequence of the definition of refinement is that this is a transitive and reflexive relation. This formulation is not anti-symmetric because it allows for an abstract and concrete programs to refine each other even if they are syntactically different, e.g. $c \sqsubseteq[r]\ c \Cap c$ and $c \Cap c \sqsubseteq[r]\ c$.

Definition 2.74 (Trace-Equality)**.** *Let c and d be commands, and r a relation,*

$$c \sim[r]\ d \ \equiv\ c \sqsubseteq[r]\ d \wedge d \sqsubseteq[r]\ c$$

If two programs are trace-equivalent in a context r, then it is safe to replace one by the other in a refinement chain. The laws that govern the substitution of programs in a refinement chain are discussed in Chapter 4. The next abbreviation is used throughout this thesis to omit the context relation r in a refinement or trace equality if its value is *true*[13].

Abbreviation 2.75 (Context-Independent)**.** *Let c and d be commands,*

$$c \sqsubseteq d \equiv c \sqsubseteq[true]\ d$$
$$c \sim d \equiv c \sim[true]\ d$$

2.10 Forward simulation

Forward simulation [75, 76, 73] is a relation between state transition systems that establishes that a system can reproduce the behaviour of other system. An abstract system *a simulates* a concrete system c if it a can *match* all transitions offered by c. In the rely-guarantee refinement calculus [48], forward simulation is applied to provide a proof technique that allows a user to directly employ the small-step operational semantics to carry out refinement

[13]The commands **definition** and **abbreviation** serve different purposes in Isabelle. The former introduces a concept and provides resources to expand and contract the concept on-demand, thus introducing a new abstraction for the user. Abbreviations introduce a syntactic sugar without adding resources for expanding and contracting the abbreviation.

proofs. The proof technique can be easily summarised as a consequence rule: if a program a simulates a program c ($a \preceq[r]\ c$), then a is refined by c ($a \sqsubseteq[r]\ c$). The small-step semantics is used as the rule book from which the allowed transitions are taken. Next we formalise forward simulation using two proof rules.

Definition 2.76 (Forward simulation)**.** *Let c and d be commands, and r a relation. We write $c \preceq[r]\ d$ to denote that program c simulates d in an environment r.*

$$\frac{}{c \preceq[r]\ c}$$

$$\frac{\forall \alpha\ d'.\ ((\alpha = \epsilon\ \sigma\ \sigma') \longrightarrow (\sigma, \sigma' \models r \vee \mathit{idrel})) \wedge d \xrightarrow{\alpha} d' \longrightarrow (\exists c'.\ c \xrightarrow{\alpha} c' \wedge c' \preceq[r]\ d')}{c \preceq[r]\ d}$$

To use definition 2.76 to prove refinements, one needs to enumerate all the transitions $d \xrightarrow{\alpha} d'$ offered by the concrete program d and check that there is a corresponding transition $c \xrightarrow{\alpha} c'$ offered by the abstract program c, requiring furthermore that d' is a refinement of c' under the environment r. To account for refinements for a particular environment r, only pairs of state (σ, σ') such $\sigma, \sigma' \models r$ are considered in the case-analysis for the case when $\alpha = \epsilon\ \sigma\ \sigma'$. We write $a \preceq[r]\ c$ to denote that a simulates c. The consequence rule connecting forward simulation and refinement is given by the next lemma.

Lemma 2.77 (Refinement-Forward-Simulation)**.** *For any commands a and c, and relation r,*

$$a \preceq[r]\ c \Longrightarrow a \sqsubseteq[r]\ c$$

Although forward simulation is a *sufficient* condition to prove refinement, it is not necessary to establish refinement. In fact, forward simulation is a stronger property than refinement, i.e., it can distinguish two programs when the refinement relation does not. To illustrate the distinction between forward simulation and refinement we consider two programs, $\mathit{Abs} \equiv (a\ ;\ b)\ \sqcap\ (a\ ;\ c)$ and $\mathit{Conc} \equiv a\ ;\ (b \sqcap c)$, whose program transitions are abstracted in Figure 2.6. We can show $\mathit{Abs} \sqsubseteq \mathit{Conc}$ using the definition of refinement presented in Section 2.9, but we cannot show this refinement via forward simulation. The reason for forward simulation to fail as a proof strategy for this refinement is that it exhausts the possibility of choices at every transition, and it distinguishes two programs if the concrete

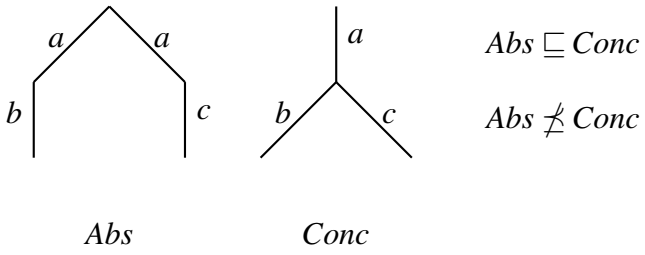


Figure 2.6 Forward simulation vs. refinement

program offers more choices of transitions than the abstract program. In this example, *Conc* postpones the non-deterministic choice until the execution of a terminates, while in the abstract program *Abs*, this choice precedes the execution of a. This difference is indistinguishable from the perspective of a set of traces for these programs, and therefore the definition of refinement based on reverse trace inclusion ignores this difference.

Forward simulation also admits a game-theoretic interpretation [105]. A forward simulation proof can be viewed as an interactive game between two players, *Attacker* and *Defender*. To explain the game, we refer to c as the abstract program, and d as the concrete program. For a pair of abstract and concrete programs, and a relation r, the game consists of a series of rounds. For each round, Attacker chooses a labelled state transition using the concrete program. Next, Defender must match this transition using an equally labelled state transition and the abstract program. At the end of each round, both players update their programs to reflect the transitions taken (that is, the new state and the new shape of the concrete and abstract programs). If one of the players is not able to perform his next move then his opponent wins. Infinite plays are won by Defender; this allows a decision to be taken when both the concrete and abstract programs are non-terminating, and the behaviour of the concrete is a subset of the behaviour of the abstract. Attacker can only choose environment transitions that satisfy r, that is, if $\alpha = \epsilon\ \sigma\ \sigma'$, then it has to be the case that $\sigma, \sigma' \models r$.

Refinement proofs constructed using definition 2.76 can be quite inefficient. Consider, for example, an abstract program c, and a concrete program **uses** $X \cdot c$. To prove that the abstract program simulates the concrete, one has to apply induction on the structure of c. This proof strategy generates one case per each primitive constructor of RG-WSL (Figure 2.3), totaling 11 cases. For some situations like this, involving an infinite play between Attacker and Defender, a more compact proof can be produced using an alternative formulation of forward simulation based on *observational equivalence* (also known as *stratified* or *approximants to bisimilarity* [100, 50]), which is presented next. This concept extends the notion of forward simulation with an extra parameter of type $\mathbb{N}$ ($\_ \preceq[n,\_]\ \_$). The additional parameter n

imposes an upper limit on the number of rounds that Defender has to defeat Attacker. A proof by forward simulation corresponds to showing that, for any number of rounds n, Defender wins. Going back to our example, to prove that c simulates **uses** $X \cdot c$ we can universally quantify c and use induction over naturals on the new parameter n. Two cases have to be considered: zero and successor. In this example, the successor case requires case split on the type of transition performed by the concrete program, but the case analysis is done for a single quantified program (c), rather than for each of the 11 language constructors. Next we present the concept of stratified forward simulation, and a law representing its relationship with refinement.

Definition 2.78 (Stratified-Forward-Simulation)**.** *Let c and d be commands, n be a natural number and r a relation. We write $c \preceq[n,r] d$ to denote that program c simulates d in an environment r up to the first n transitions.*

$$\frac{}{c \preceq[0,r]\ d}$$

$$\frac{\forall\, \alpha\ d'.\ (\alpha = \epsilon\ \sigma\ \sigma' \longrightarrow \sigma, \sigma' \models r \vee \mathit{idrel}) \wedge d \xrightarrow{\alpha} d' \longrightarrow (\exists c'.\ c \xrightarrow{\alpha} c' \wedge c' \preceq[k,r]\ d')}{c \preceq[\mathit{Suc}\ k,r]\ d}$$

Lemma 2.79 (Refinement-Stratified-Forward-Simulation)**.** *For any commands a and c, and relation r,*

$$\forall n.\ a \preceq[n,r]\ c \Longrightarrow a \sqsubseteq[r]\ c$$

Note that forward simulation cannot be used to prove laws that have the refinement symbol among the premises, such as transitivity and monotonicity laws. This limitation is connected with the fact that refinement and forward simulation are not equivalent concepts. Thus, it is not possible to extract information from the premises about the internal transitions, which are generally needed to complete proofs by forward simulation.

2.11 Unrestricted variables

Refinement laws that introduce local variables and restrict access to variables require the concept of free variables of a program for their formalisation. The characterisation of the

set of free variables for deep-embedded languages is straightforward: it just requires pattern matching on the constructors of the language. On the other hand, the definition of free variables for shallow-embedded languages is a tricky task. The problem in this case is that it is not possible to pattern match against a relation as it does not have a predefined syntax.

The difficulty in defining the set of free variables for RG-WSL is determining the set of free variables for shallow-embedded relations. The closest to free-variables for relations we can provide is the minimal set of variables a relation depends on, which we call the *alphabet* of the relation. It is defined as[14]

$$\alpha\ g \equiv \mathit{SOME}\ X.\ \vdash \mathit{depends\text{-}only}\ (g, X) \wedge (\forall Y.\ \vdash \mathit{depends\text{-}only}\ (g, Y) \longrightarrow X \subseteq Y)$$

This formalisation of alphabets for relations, however, lacks a much needed decomposition rule to infer the alphabet of composite relations (e.g. $g \wedge h$, $g \vee h$, etc.) from the alphabet of their parts. The problem is better discussed by considering the next two relations which are defined using conjunction of simpler relations.

$$(3 < x) \vee (x < 3)$$
$$(x = 3) \vee (x \neq 3)$$

For the first example, the alphabet coincides with the union of the alphabet of the parts, but in the second example the alphabet is the empty set instead of the union of the alphabet of the parts. In practice, there is no general method to decompose the alphabet of a relation in terms of the alphabet of the parts.

Investigating the use of free variables in the RG refinement calculus of [48] we identify two situations that summarise the application scenarios:

i to determine if an implementation uses variables outside a set specified by a user;

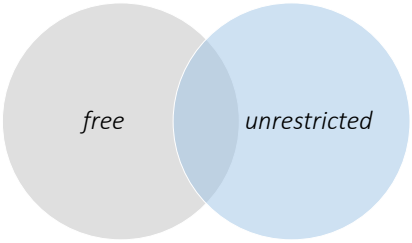
ii to determine conditions under which a local variable can be introduced.

For (i), only *code* needs to be considered and, for this subset of RG-WSL, a definition of free variables is formalised on page 111 (Def. 3.107). For (ii), we need a mechanism to decide if a program c can be nested in a local variable block of x without capturing any free occurrences of x. The usual condition for (ii) is to require that x is not free in c, i.e. x is *fresh* in c. The

[14] The notation *SOME X*. *P X* refers to Hilbert's ϵ-operator.

usage of shallow-embedding, however, restricts our ability to test if a variable is *fresh* in RG-WSL. Nonetheless, we can still decide if a variable is *unrestricted* [37], i.e. its value is not restricted by a program, and use this condition to ensure no capturing of restricted variables.

To illustrate the differences among free, fresh and unrestricted variables we consider a relation q defined as $x' = y + w - y$. The set of free-variables of q consists of those variables which are syntactically referred by q, i.e. $\mathit{free}\ (q) = \{x, y, w\}$; whereas the set of fresh variables is the complement of the set of free variables (i.e. $\mathit{fresh}\ q = \overline{\{x, y, z\}}$). Finally, the set of unrestricted variables is formed by all variables that are not restricted by q, i.e. $\mathit{unrestricted}\ q = \overline{\{x, w\}}$. This example is summarised in Figure 2.7. The fact that y is unrestricted in q means that a program such as $[q]$, i.e. a postcondition command supplied with a relation q, can be nested in a local variable block of y without affecting restrictions already imposed over the global variable y. The next two definitions formalise the notion of unrestricted variables of a program. The first denotes a set of free variables of an expression (Figure 2.2 on page 22), whereas the second inductively defines the set of unrestricted variables for each command of RG-WSL.



$$q \equiv (x' = y + w - y)$$

$$\begin{aligned} \mathit{free} &= \{x, w, y\} \\ \mathit{fresh} &= \overline{\{x, w, y\}} \\ \mathit{unrestricted} &= \overline{\{x, w\}} \\ \mathit{free} \cap \mathit{unrestricted} &= \{y\} \end{aligned}$$

Figure 2.7 Unrestricted variables

Definition 2.80 (Free-Exp)**.** *Expressions are deeply-embedded, and thus enjoy a notion of free variables. Let c be a constant, v a variable, and e, e_1 and e_2 be expressions. The set of free variables of an expression is defined on the structure of expressions as follows:*

$$\mathit{free\text{-}exp}\ (N\ c) \ = \ \emptyset$$

$$\begin{aligned} \mathit{free\text{-}exp}\ (V\ v) &= \{v\} \\ \mathit{free\text{-}exp}\ (\mathit{UOp}\ \mathit{uop}\ e) &= \mathit{free\text{-}exp}\ e \\ \mathit{free\text{-}exp}\ (\mathit{BOp}\ \mathit{bop}\ e_1\ e_2) &= \mathit{free\text{-}exp}\ e_1 \cup \mathit{free\text{-}exp}\ e_2 \end{aligned}$$

Definition 2.81 (Unrestricted-Variable)**.** *Let c and d be commands, x be a variable, e be an expression, and b be a boolean expression, and remember that depends-only is defined in Figure 2.4. The set of unrestricted variables of a program is inductively defined as follows,*

$$\frac{\vdash \mathit{depends\text{-}only}\ (p, \overline{\{x\}})}{\mathit{unrest}(x, \{p\})} \qquad \frac{\vdash \mathit{depends\text{-}only}\ (q, \overline{\{x\}})}{\mathit{unrest}(x, [q])}$$

$$\frac{\begin{array}{l}\vdash \mathit{depends\text{-}only}\ (p, \overline{\{x\}}) \\ \vdash \mathit{depends\text{-}only}\ (q, \overline{\{x\}})\end{array}}{\mathit{unrest}(x, \langle p, q\rangle)} \qquad \frac{\mathit{unrest}(x, c) \qquad \mathit{unrest}(x, d)}{\mathit{unrest}(x, c \Cap d)}$$

$$\frac{\mathit{unrest}(x, c) \qquad \mathit{unrest}(x, d)}{\mathit{unrest}(x, c\ ;\ d)} \qquad \frac{\mathit{unrest}(x, c) \qquad \mathit{unrest}(x, d)}{\mathit{unrest}(x, c \parallel d)}$$

$$\frac{\forall c{\in}C.\ \mathit{unrest}(x, c)}{\mathit{unrest}(x, \bigsqcap \mathit{acset}\ C)} \qquad \frac{x \notin \mathit{free\text{-}exp}\ b}{\mathit{unrest}(x, [[b]])}$$

$$\frac{y = x \vee \mathit{unrest}(y, c)}{\mathit{unrest}(y, \textbf{state}\ x \mapsto v \cdot c)} \qquad \frac{\mathit{unrest}(x, c)}{\mathit{unrest}(x, \textbf{uses}\ X \cdot c)}$$

A variable x is unrestricted in a postcondition $[q]$, precondition command $\{p\}$ or atomic command $\langle p, q\rangle$, if the relations used to instantiate any of these commands do not depend on x. A variable x is unrestricted in a test command $[[b]]$ if x does not occur free on b, that is, $x \notin \mathit{free\text{-}exp}\ b$. Any variable that is unrestricted in c is also unrestricted in **state** $x \mapsto v \cdot c$. Moreover, the variable x is by definition unrestricted in **state** $x \mapsto v \cdot c$; this is to allow the nesting of local variable blocks of the same variable. We omit the description of remaining cases as these are straightforward.

2.12 Discussion and summary of contributions

This chapter introduced concepts that are required to understand the remaining chapters and our characterisation of [48] in Isabelle/HOL. A key contribution with respect to [48] is that all details were teased out and iron out with necessary adjustments and corrections as discussed in following chapters. In particular, we use Isabelle to assist the preparation of this manuscript [40]. This means that parts of this thesis are automatically extracted from our Isabelle theories, and are inspected and validated by Isabelle prior to the generation of this document. Our contributions in this chapter are:

1. A characterisation of a value model for variables that suits the derivation of examples involving natural numbers, integers, booleans, sets, arrays and option types (Section 2.2);

2. An extensible expression language (*Exp*) that incorporates a mechanism to check undefidness of expressions (Sections 2.3.2 and 2.3.3);

3. Identification and discussion of a serious issue involving the syntactic characterisation of RG-WSL (Section 2.3.4);

4. A compositional treatment of logical interpretation of relations (Section 2.6);

5. Mechanisation of the small-step operational semantics of RG-WSL (Section 2.7.2);

6. An efficient characterisation of forward simulation, which provides an interface for using the small-step semantics for performing refinement proofs in Chapter 5 (Section 2.10);

7. A characterisation of unrestricted variables to approximate the notion of free-variables for shallow-embedded programs (Section 2.11).

Most of these contributions extrapolate the scope of this work and can be reused in similar situations involving an algebraic characterisation of a refinement calculus in a proof assistant. Next, we reflect upon aspects that could be improved in our encoding.

2.12.1 Alternative approaches to formalise states

In this work we represent states as total functions, but there are alternative approaches to encoding the state of a program in a proof assistant. In the *state as record* approach [2, 82], each field of a record represents a variable of a program. Comparatively to the *state as total*

function approach, the state as records has the advantage of enforcing that relations are type consistent, but it has the disadvantage of not allowing the user to quantify over program variables. That is because it is not possible to quantify over the fields of a record in Isabelle, while it is possible to quantify elements in the domain of a function.

The state as total function can be described as *weakly typed*, in the sense that the datatype *vvalue* masks the real type of program variables, i.e. variables assigned to distinct values such as *VInt 0* and *VBool False* have the same *explicit* type, namely *vvalue*, instead of being assigned to the *implicit* types $\mathbb{Z}$ and $\mathbb{B}$, respectively. Thus, in our representation, one can write a relation that holds for pairs of states that assign variables to distinct constructors, e.g. *idset* $\{x\}$ holds for σ and σ', where

$$\sigma = (\lambda s.\ \mathit{VBool\ True})(x := \mathit{VInt\ 0})$$
$$\sigma' = \lambda s.\ \mathit{VInt\ 0}$$

Note that state σ assigns *VBool True* to any variables except x, while σ' assigns *VInt 0* to all variables. Since the value of x is the same for both states, we have $\sigma, \sigma' \models \mathit{idset}\ \{x\}$. This means that the implicit type of a variable can change in a specification. In the state as record approach the type of each variable is declared when defining the record that represents the state.

To effectively combine the state as records approach and the shallow-embedding of programming languages, the language has to be parametric on the state. This parameterisation allows the theory development to be split into state-free and state-dependent laws and definitions. On an abstract level the access to the state is defined using a lookup and update functions, which provide an uniform interface for reading/writing on the state, respectively. The specification of lookup and update functions are complemented at the moment of application of the theory. While such division looks attractive, we struggled to reach a satisfactory level of automation while trying to setup the abstraction of states in this manner. Alternative ways of encoding states include tuples and locales. An excellent survey of the existing approaches to characterise states is presented in [101], which also discusses the use of locales to represent the state.

It is worth noting that the design of the expression language (Figure 2.2) is independent from the choice of representation for the state. The expression language adopted in this thesis is inherently weakly typed, because it unifies expressions of different types using a single grammar. To ensure that refinement language that is type consistent by construction, the expression language would need to be adapted. This would require boolean expressions,

arithmetic expressions, list expressions, etc. to be placed into separate syntactic categories, and the type of variables to be explicitly declared before their first use.

2.12.2 Semantics

Forward simulation Our experience using the small-step semantics to do forward simulation proofs is quite narrow. We used it to prove just about a dozen properties over primitive commands of RG-WSL. We observed that for some situations, it would be easier to prove properties if the small-step semantics were adapted to eliminate the overlapping between abortive and non-abortive execution scenarios. To make this point clearer, consider for example the abstract program *abort* and the concrete program $(abort\ ;\ c)$. Law 2.79 (Refinement-Stratified-Forward-Simulation) allows us to use stratified forward simulation to prove that the concrete program refines the abstract one. For that, we have to show

$$\forall n.\ abort \preceq [n,r]\ abort\ ;\ c \tag{2.82}$$

This is a type of proof where we can reduce structural induction over RG-WSL (c) to induction over natural numbers (n). In practice, we first have to prove

$$\forall c.\ abort \preceq [n,r]\ abort\ ;\ c \tag{2.83}$$

by induction on n and use this result to show that (2.82) holds. In this example the extra work in the proof of (2.83) comes from the fact that rules (2.47) and (2.48) (page 48) allow a program that has aborted to recover from abortion. This is because these rules do not force a sequential composition whose leftmost program has aborted to take the execution path formalised by rule (2.44) (page 46). Consequently, a sequential composition $c_1\ ;\ c_2$ where c_1 aborts can recover from abortion, i.e. the execution of c_2 can proceed from the state where c_1 terminates. This flexibility causes a mismatch between the induction hypothesis and the conclusion in the proof of the inductive case of (2.83) when rule (2.48) is used in the case of a program step (π). To solve the problem, we have two options: either we adapt rules (2.47) and (2.48) to make abortive and non-abortive execution scenarios mutually exclusive, or we reformulate the conjecture (2.83) to prevent the mismatch between the induction hypothesis and the conclusion. In our experiments, we have always taken the second path, that is, to adjust conjectures making them more generic and suitable for inductive proofs. In this case,

we would prove

$$\forall c.\ abort \preceq [n,r]\ c \tag{2.84}$$

using induction on n and use this result to prove 2.83. Nevertheless, it seems a reasonable choice to adapt the small-step semantics to make abortive and normal execution cases mutually exclusive. This would make the semantics more deterministic, therefore easier to do proofs with. We believe that this modification would not invalidate the proof of lemmas assumed from [48], but the full understanding of the impact of this modification would demand more experimentation with the semantics.

Expression evaluation Our encoding of the rules for expression evaluation introduced in Section 2.7.1 use separate cases for *defined* and *undefined* expressions. In [48], the rules for expression evaluation are condensed, and the signatures *eval-binary* and *eval-unary* are expected to handle undefinedness. The advantage of our presentation is that it introduces a novel and concrete method to handle undefinedness using the definition of binary and unary operators.

2.12.3 Well-founded relations

Isabelle/HOL already comes with the theory *Wellfounded* containing a formalisation of well-foundedness for curried relations (*wfP*) and proof rules to show that a specific relation is well-founded. We experimented with applying this theory to show that the relations used to introduce loops in the examples of Chapter 6 are well-founded, but we did not succeed. To remedy this situation, we introduced the alternative definition of well-foundedness described in Section 2.4.4. This definition also showed it be difficult to apply in practice. Currently, we do not know how to prove well-foundedness for the relevant relations in Chapter 6. The derivations presented in Chapter 6 use locales to assume that the relations used to introduce loops are well-founded. Chapter 8 highlights the need of further experimentation involving the concept of well-foundedness as future work.

2.12.4 Unfair parallelism

The semantics for parallel composition in [48] and in this work does not impose fairness. This can be seen in rule (2.53) on page 49. This formulation allows a non-terminating branch of parallel composition to infinitely overtake other branches. Unfair parallelism can only

be used to prove a limited class of properties about algorithms involving busy waiting and critical regions. For example, consider the program

$$x{:=}1 \;;\; ((\textbf{while}\; x = 0\; \textbf{do}\;\; skip) \parallel x{:=}0)$$

Unless we assume parallel composition to be fair, we cannot prove that this program terminates. This is because, if the assignment of zero to x is never executed in the parallel composition, then the while loop does not terminate. In general, to prove properties about programs that depend on its environment to terminate, one has to rely on each branch of parallel composition to be given opportunity to execute infinitely often, and unfair parallelism breaks this assumption.

To impose fairness, it is necessary to eliminate traces containing an infinite sequence of program steps of one process while other process can make a program step [43]. Modelling of fair parallelism is well understood for denotational semantics [21, 32], but it is not clear to us how this can be formalised using an operational semantics.

Chapter 3

Rely-guarantee refinement calculus

> The main characteristic of intelligent thinking is that one is willing and able to study in depth an aspect of one's subject matter in isolation [...]. The crucial choice is, of course, what aspects to study 'in isolation', how to disentangle the original amorphous knot of obligations, constraints and goals into a set of 'concerns' that admit a reasonably effective separation.
>
> Edsger W. Dijkstra, *A discipline of programming, 1976*

This chapter introduces rely-guarantee refinement calculus, a programming methodology for designing shared-variable concurrent programs. The material introduced here is taken from [48], but is adapted to fit the particular encoding of RG-WSL discussed in Chapter 2. Except for the laws where we found inconsistencies in the proofs, we omit proofs already available in [48]. We call *lemmas* the theorems whose proof directly refers to the semantics of the RG-WSL, and we call *laws* those whose proof can be done in terms of previously defined lemmas.

Standard programming constructs, such as assignment, conditionals and loops are defined in terms of RG-WSL in Section 3.1. Rely and guarantee constructors are introduced later as independent commands. The separation between primitive commands and derived commands is reflected in proofs: properties about derived commands are proved by expanding their definitions and then applying the properties of the primitive commands that are part of the definition. An exception to this strategy are the iterated commands, which are defined using fix-point operators. Their properties are derived from lemmas that accompany their definition.

Section 3.2 discuss the concept of stability and single reference property, which play a key role in the formalisation of laws involving rely conditions. Basic laws of refinement are introduced in Section 3.3, and rely and guarantee commands are the subject of discussion in

$$skip \equiv \{true\} \tag{3.1}$$
$$abort \equiv \{false\} \tag{3.2}$$
$$\langle q \rangle \equiv \langle true, q \rangle \tag{3.3}$$
$$[p, q] \equiv \{p\} \,;\, [q] \tag{3.4}$$
$$magic \equiv \bigsqcap acset\ \emptyset \tag{3.5}$$
$$c_1 \sqcap c_2 \equiv \bigsqcap acset\ \{c_1, c_2\} \tag{3.6}$$
$$c^\star \equiv \nu x \cdot skip \sqcap c; x \tag{3.7}$$
$$c^\infty \equiv \mu x \cdot c; x \tag{3.8}$$
$$c^\omega \equiv \mu x \cdot skip \sqcap c; x \tag{3.9}$$
$$c^{\star+} \equiv c^\star \,;\, c \tag{3.10}$$
$$c^{\omega+} \equiv c^\omega \,;\, c \tag{3.11}$$

Figure 3.1 Derived commands

Sections 3.4-3.7. Introduction of parallelism is discussed in Section 3.8. The final sections treat expression evaluation, local variables, syntactic control of interference, control structures and assignment. This chapter ends with discussion and summary of contributions.

While mechanising the theory, we discovered the need for extending RG-WSL and also adding extra laws to reproduce the proofs suggested in [48]. Some of these extensions are simple lemmas and laws, which are assumed to hold, but were not spelled out in [48]; some are minor contributions that on their own do not justify a section in Chapter 5 (Extensions to rely-guarantee algebra), where more solid contributions appear. To distinguish the extra laws and definitions from those already given in the literature we extend their label with the keyword **contrib**.

3.1 Derived commands

The following conventions are used to present the derived commands in this section: q is a relation, p is a predicate, b is a boolean expression, v ranges over values that variables can take on, and c, c_1 and c_2 range over commands. Figure 3.1 formalises derived commands in terms of the primitive ones from Section 2.3.

Skip is the command that does a termination step immediately (3.1), and *abort* is the most non-deterministic program (3.2). An atomic command with no assumptions on the before state can be abbreviated by omitting its precondition *true* (3.3). The specification command

is the sequential composition of the precondition and postcondition commands (3.4). Magic is defined as an empty choice (3.5); from an operational perspective, it offers no transition whatsoever. A non-deterministic choice over two commands is abbreviated using the infix operator $\sqcap$ (3.6). The notation $\mu x \cdot F(x)$ denotes that x is the least fix point to satisfy F with respect to the refinement ordering ($\sqsubseteq$). Similarly, $\nu x \cdot \mathit{skip}$ denotes the greatest fix point. Finite ($c^\star$), infinite (c^∞), and potentially infinite (c^ω) iteration are defined via fixed points (3.7-3.9). These definitions are then used to define non-empty iteration ($c^{\star+}$, $c^{\omega+}$). Next we discuss derived commands whose definition is more elaborated. These are also taken from [48].

Assignment

An assignment consists of two stages: multi-step expression evaluation and variable update. The evaluation stage non-deterministically chooses a value v, such that the test $[[N\ v = e]]$ succeeds, and the update stage sets x to v via a single atomic step, leaving variables other than x unchanged ($\mathit{idset}\ \overline{\{x\}}$).

$$x{:=}e \equiv \bigsqcap \mathit{acset}\ \{[[N\ v = e]]\ ;\ \langle(\lambda s\ s'.\ s'\ x = v) \wedge \mathit{idset}\ \overline{\{x\}}\rangle \mid \mathit{True}\} \tag{3.12}$$

The logical transfer of the value v between the commands is possible by enclosing both commands into a non-deterministic choice and universally quantifying v by setting it as a logical variable of the set comprehension. This formulation assures that we are considering all possible evaluations of e instead of a particular evaluation. This is different from previous works [59–61, 104], which treated assignment as atomic. This definition enables derivation of programs considering fine-grain concurrency, where only the reading and writing operations over variables are assumed to be atomic. In this direction, it differs from definitions of assignment which assume all occurrences of the same variable to be fetched simultaneously, such as the one proposed in [34].

Conditionals

Conditionals are defined via the non-deterministic choice between two branches. As the semantics of tests does not provide evaluations to *false*, any test that would evaluate to false is eliminated from the non-deterministic composition (i.e. the corresponding branch becomes equivalent to *magic*, which is absorbed by the choice).

$$\textbf{if}\ b\ \ \textbf{then}\ c_0\ \ \textbf{else}\ c_1 \equiv ([[b]]\ ;\ c_0) \sqcap ([[\neg b]]\ ;\ c_1) \tag{3.13}$$

Loops

To formalise while loops, a sequential composition of a test ($[[b]]$) and a program (c) is iterated using the omega iterator ($([[b]] \,;\, c)^{\omega}$). The omega iterator denotes a potentially infinite loop, and its use with the test command suffice to model the behaviour of terminating and non-terminating while loops.

$$\textbf{while } b \textbf{ do } \; c \equiv ([[b]] \,;\, c)^{\omega} \,;\, [[\neg b]] \tag{3.14}$$

Loops that terminate must have their guard (b) falsified by the body of the loop (c), or the environment of the loop. A test is used to model that the loop guard is falsified upon termination of the loop.

Local variables

Local variables are defined using the **state** command, which provides the abstraction of hiding variables at the level of traces. Local variables are shielded from external interference and have no predefined initial value.

$$\textbf{var } x \cdot c \equiv \bigsqcap \mathit{acset}\ \{\textbf{state } x \mapsto v \cdot c \mid \mathit{True}\} \tag{3.15}$$

Note that local variables as well as global variables are untyped. Thus, a variable can be assigned to values of different types during the lifetime of a program (e.g. booleans, integers, etc.), without affecting the validity of the program. Type consistence could be enforced, for example, by using conditions to check that once a variable is assigned to a *vvalue* with a certain constructor (e.g. *VBool*), it is never assigned to a *vvalue* with a different constructor (e.g. *VInt*).

3.1.1 Precedence and associativity

Figure 3.2 describes the precedence and associativity for binary operators of RG-WSL. This convention is illustrated in Table 3.1 and is adopted in rest of this thesis.

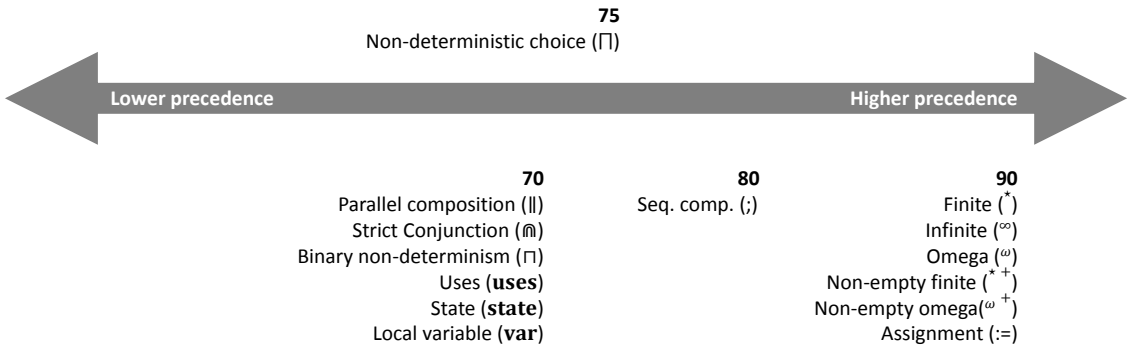


Figure 3.2 Precedence and associativity. Commands are aligned according to their precedence order. The higher the number at the top of the column, the higher the precedence of the operators at that column. Sequential composition is left-associative and all other binary commands in this figure are right-associative.

Fully parenthesised	**Using precedence and associativity**
$d \sqcap (c \,;\, x)$	$d \sqcap c \,;\, b$
$skip \sqcap (c \,;\, (c^\star))$	$skip \sqcap c \,;\, c^\star$
$((c; d) \parallel (\langle true \rangle^\omega)) \Cap f$	$(c \,;\, d \parallel \langle true \rangle^\omega) \Cap f$
$(a \parallel b) \parallel c$	$(a \parallel b) \parallel c$
$(a; b); c$	$a \,;\, b \,;\, c$

Table 3.1 Precedence and associativity for derived commands

3.2 Stability

A specification $[p, q]$ is *stable* under a relation r if, and only if, three conditions hold[1]:

- $\vdash r \Rightarrow (p \Rightarrow p\text{`})$, i.e. p is preserved by interference steps satisfying r;

[1]This chapter uses both precedence order and type distinction to omit parenthesis around logical evaluation ($\vdash$). For example, type distinction combined with precedence order allows us to omit all but the innermost parenthesis in $\vdash ((p \wedge (r \,;\, q)) \Rightarrow q)$, and omit all parenthesis in $(\vdash (p_0 \Rightarrow p_1)) \wedge (\vdash ((p_0 \wedge q_1) \Rightarrow q_0))$. Recall from Section 2.4 that $\wedge_r$, $\Rightarrow_r$, and $;_r$ are relational operators representing conjunction (left-associative), implication (right-associative) and relational composition (right-associative). Grave accent (`) is used to encode the concept of post-state for predicates. Moreover, we omit the subscript r in the relational operators if its clear from the context that we are referring to binary relations.

- $\vdash p \wedge (r\ ;\ q) \Rightarrow q$, i.e. q is preserved by preceding interference steps satisfying r;
- $\vdash p \wedge (q\ ;\ r) \Rightarrow q$, i.e. q is preserved by following interference steps satisfying r.

Intuitively this means that r preserves the precondition p and the postcondition q. This concept is applied to determine the feasibility of implementing a specification in the context of a rely condition. Formally we write *tol-interf*(p, q, r) to represent that the conditions above hold for a predicate p and relations q and r.

Definition 3.16 (Tolerate-Interference)**.** *Let p a predicate, and q and r relations. The specification $[p, q]$ tolerates interference r if,*

$$\begin{aligned} \textit{tol-interf}(p, q, r) \ \equiv\ & \vdash r \Rightarrow (p \Rightarrow p\text{`}) \wedge \\ & \vdash p \wedge (r\ ;\ q) \Rightarrow q \wedge \\ & \vdash p \wedge (q\ ;\ r) \Rightarrow q \end{aligned}$$

Law 3.17 (Closure-Interference)**.** *For any predicate p and relations q and r, such that* *tol-interf*(p, q, r)*, the following hold.*

$$\vdash r^{**} \Rightarrow p \Rightarrow p\text{`} \tag{3.17a}$$

$$\vdash p \wedge (r^{**}\ ;\ q\ ;\ r^{**}) \Rightarrow q \tag{3.17b}$$

3.2.1 Single reference property

An expression e satisfies the single reference property with respect to a binary relation r, if e contains at most one variable that is modified by r; additionally, e must contain no more than one reference to such a variable. Intuitively, this property characterises that an expression has at most a single reference to an *unstable variable* with respect to r. To illustrate this property, consider the situation where y is a local variable (and hence, cannot be modified by the environment) and x and z are shared variables that are monotonically increased by the environment. The boolean expression $x \leq y$ respects the single reference property, because it only contains no more than a reference to an unstable variable, which in this situation is x. On the other hand, the boolean expressions $x + x \leq y$ and $x \leq y + z$ do not satisfy the single

reference property, because they either have more than one unstable variable, or have more than a reference to an unstable variable.

Programs derived using rely-guarantee refinement calculus observe the single reference property in the guards of conditionals and loops. This means that these expressions have no more than one reference to unstable variables. The single reference property allows the multi-step evaluation of an expression (performed by the test command) to be abstracted using a relation and the postcondition command.

3.3 Basic refinement laws

In general, the application of refinement laws require the user to perform preparatory steps in between the application of relevant laws[2]. These steps generally correspond to weakening a precondition, strengthening of a postcondition, splitting a specification into a sequential composition, weakening the context of the refinement relation, etc. This section presents laws which are relevant on their own, but that are generally applied as preparatory steps. It follows directly from the definition of refinement (Def. 2.73) that it is reflexive, transitive and monotonic with respect to the context (r).

Law 3.18 (Refinement-Preorder)**.** *For any commands c, d and f, and relations r, r_0 and r_1,*

$$c \sqsubseteq c \tag{3.18a}$$

$$c \sqsubseteq[r]\ d \wedge d \sqsubseteq[r]\ f \implies c \sqsubseteq[r]\ f \tag{3.18b}$$

$$\vdash r_0 \Rightarrow r_1 \vee \mathit{idrel} \wedge c \sqsubseteq[r_1]\ d \implies c \sqsubseteq[r_0]\ d \tag{3.18c}$$

Remark. *Most laws in this chapter hold independently of the refinement context, that is, $r = true$. Thus, to enable their application one first needs to apply monotonicity (law 3.18c) to weaken the context if it is different from true. However, if such weakening is done carelessly it blocks the application of subsequent context-specific laws. To avoid losing the refinement context one needs to apply monotonicity point-wisely. This is done by splitting the refinement chain into two sub-proofs via transitivity (law 3.18b), and by applying monotonicity only to the relevant branch. Since the use of the laws above is quite frequent, we omit the reference to them in paper proofs. For full details on proofs, we refer the reader to the Isabelle theories in Appendix A.*

[2] A recurrent example of this is the introduction of loops, that requires the specification command to be in a predefined shape for its introduction. See, for example, discussion about step $\mathscr{R}_{19}$ on page 192.

A precondition declares assumptions on the before state that a program expects to hold, and a postcondition declares commitments a program must satisfy under its termination. Weakening a precondition corresponds to removing assumptions on the before state, and strengthening a postcondition corresponds to reducing non-determinism on the after state.

Lemma 3.19 (Consequence)**.** *For any predicates p_0 and p_1, and relations q_0 and q_1,*

$$\vdash p_0 \Rightarrow p_1 \wedge \vdash p_0 \wedge q_1 \Rightarrow q_0 \implies [p_0, q_0] \sqsubseteq [p_1, q_1] \tag{3.19a}$$

$$\vdash p_0 \Rightarrow p_1 \wedge \vdash p_0 \wedge q_1 \Rightarrow q_0 \implies \langle p_0, q_0 \rangle \sqsubseteq \langle p_1, q_1 \rangle \tag{3.19b}$$

$$\vdash p_0 \Rightarrow p_1 \implies \{p_0\} \sqsubseteq \{p_1\} \tag{3.19c}$$

The precondition command sequentially distributes over parallel composition, strict conjunction and itself. In the last case, a sequential composition of preconditions is semantically equivalent ($\sim$) to a single precondition that conjoins the predicates enclosed in the sequential composition.

Lemma 3.20 (Dist-Precondition)**.** *For any predicates p, p_0 and p_1, and commands c, d,*

$$\{p\} \,;\, (c \parallel d) \;\sim\; (\{p\} \,;\, c) \parallel (\{p\} \,;\, d) \tag{3.20a}$$

$$\{p\} \,;\, (c \Cap d) \;\sim\; (\{p\} \,;\, c) \Cap (\{p\} \,;\, d) \tag{3.20b}$$

$$\{p_0\} \,;\, \{p_1\} \;\sim\; \{p_0 \wedge p_1\} \tag{3.20c}$$

A specification ($[p, q]$) allows any number of atomic steps to be used to implement a desired postcondition, and an atomic specification ($\langle p, q \rangle$) requires the implementation to be achieved in a single atomic step. Additionally, a specification aborts in any environment different from *idrel*, while an atomic command cannot abort unless its precondition does not hold.

Lemma 3.21 (Make-Atomic)**.** *For any predicate p and relation r,*

$$[p, q] \sqsubseteq \langle p, q \rangle$$

The specification command aborts in any context different from *idrel*. Thus, to refine the specification command one only needs to concern about the refinement in an *idrel* context.

Lemma 3.22 (Refine-Specification)**.** *For any predicate p and relation q,*

$$[p, q] \sqsubseteq [idrel]\ c \ \Leftrightarrow\ [p, q] \sqsubseteq c$$

A postcondition that already holds in the before state can be implemented by a program that performs no program steps and terminates immediately (*skip*).

Lemma 3.23 (Specification-Idrel)**.** *For any predicate p*

$$[p, idrel \wedge p`] \sqsubseteq skip$$

A specification $[p, q_0 ; q_1]$ can be implemented via a sequential composition of specifications that progressively establish the desired postcondition. This is quite similar to the standard refinement calculus [78].

Lemma (contrib.) 3.24 (Relational-Composition-Split)**.** *For any predicates p and mid, and relations q_0 and q_1,*

$$[p, q_0 ; q_1] \sqsubseteq [p, q_0 \wedge mid`] ; [mid, q_1]$$

In contrast to [48], the next law is derived from predefined lemmas instead of being proved directly using the semantics of RG-WSL.

Law 3.25 (Sequential)**.** *For any predicates p and mid, and relations q, q_0 and q_1, such that $\vdash p \wedge (q_0 \wedge mid` ; q_1) \Rightarrow q$,*

$$[p, q] \sqsubseteq [p, q_0 \wedge mid`] ; [mid, q_1]$$

Proof.

$[p, q]$

$\sqsubseteq$ by lemma 3.19a (Consequence) and assumption

$[p, q_0 \wedge mid` ; q_1]$

$\sqsubseteq$ by law 3.24 (Relational-Composition-Split)

$[p, q_0 \wedge mid`] ; [mid, q_1]$

$\sqsubseteq$ by logical simplification

$[p, q_0 \wedge mid`] \,;\, [mid, q_1]$ $\square$

Remark. *The choice of mid impacts on the feasibility of the refined program. The choice of a too strong predicate for mid, e.g. false, introduces a specification that cannot be implemented by code. The choice of a too weak predicate for mid, e.g. true, may fail to provide relevant information to discharge proof obligations in later stages of a derivation, such as implementation of assignments. In general, a good choice for mid is one that can maximise the transference of knowledge about the state of a program.*

3.3.1 Associativity, commutativity and distributivity

All binary primitive commands of RG-WSL are associative; additionally, strict conjunction and parallel composition are commutative. Sequential composition distributes over binary non-deterministic choice on the left, and unbounded non-deterministic choice on the right. Strict conjunction and parallel composition distribute on the right over unbounded non-deterministic choice. The **uses** block distributes over several commands.

Lemma 3.26 (Assoc-Comm-Dist)**.** *For any commands a, b and c, and set of variables X, and set commands C and D, such that $D \neq \emptyset$,*

$$a \,;\, (b \,;\, c) \sim (a \,;\, b) \,;\, c \tag{3.26a}$$

$$a \parallel (b \parallel c) \sim (a \parallel b) \parallel c \tag{3.26b}$$

$$c_0 \Cap (c_1 \Cap c_2) \sim (a \Cap b) \Cap c \tag{3.26c}$$

$$a \Cap b \sim b \Cap a \tag{3.26d}$$

$$c \parallel d \sim d \parallel c \tag{3.26e}$$

$$a \,;\, (c \sqcap d) \sim (a \,;\, c) \sqcap (a \,;\, d) \tag{3.26f}$$

$$\bigsqcap acset\ D \Cap c \sim \bigsqcap acset\ \{c \Cap d \mid d \in D\} \tag{3.26g}$$

$$\bigsqcap acset\ C \,;\, d \sim \bigsqcap acset\ \{c \,;\, d \mid c \in C\} \tag{3.26h}$$

$$\bigsqcap acset\ C \parallel d \sim \bigsqcap acset\ \{c \parallel d \mid c \in C\} \tag{3.26i}$$

$$\mathbf{uses}\ X \cdot \bigsqcap acset\ C \sim \bigsqcap acset\ \{\mathbf{uses}\ X \cdot c \mid c \in C\} \tag{3.26j}$$

$$\mathbf{uses}\ X \cdot c \Cap d \sim (\mathbf{uses}\ X \cdot c) \Cap (\mathbf{uses}\ X \cdot d) \tag{3.26k}$$

$$\mathbf{uses}\ X \cdot c \,;\, d \sim (\mathbf{uses}\ X \cdot c) \,;\, (\mathbf{uses}\ X \cdot d) \tag{3.26l}$$

$$\mathbf{uses}\ X \cdot c \parallel d \sim (\mathbf{uses}\ X \cdot c) \parallel (\mathbf{uses}\ X \cdot d) \tag{3.26m}$$

$$\mathbf{uses}\ X \cdot c^{\star} \ \sim\ (\mathbf{uses}\ X \cdot c)^{\star} \tag{3.26n}$$

$$\mathbf{uses}\ X \cdot c^{\infty} \ \sim\ (\mathbf{uses}\ X \cdot c)^{\infty} \tag{3.26o}$$

All the lemmas introduced in this chapter are used in proofs of derived laws or examples (in Chapter 6) at least once. This means that this chapter contains no *orphan* lemmas, in the sense of properties which are introduced but never used.

3.3.2 Monotonicity

A key feature of a refinement calculus is the ability to refine parts of a program independently. Compositionality is only enjoyed for programming constructs that are monotonic with respect to the refinement order. The following lemma covers monotonicity properties of RG-WSL and derived commands. For a gentle revision of precedence (Section 3.1.1), law 3.27a is presented fully parenthetised. All parentheses in this law can be omitted.

Lemma 3.27 (Monotonic-WSL)**.** *For any commands a, b, c and d, relation r, and variable x and set of variables X,*

$$((a \sqsubseteq[r]\ d_0) \wedge (c_1 \sqsubseteq[r]\ d_1)) \implies ((a\ ;\ c) \sqsubseteq[r]\ (b\ ;\ d)) \tag{3.27a}$$

$$a \sqsubseteq[r]\ b \wedge c \sqsubseteq[r]\ d \implies a \Cap c \sqsubseteq[r]\ b \Cap d \tag{3.27b}$$

$$c \sqsubseteq[\mathit{idset}\ \{x\}]\ d \implies \mathbf{var}\ x \cdot c \sqsubseteq \mathbf{var}\ x \cdot d \tag{3.27c}$$

$$c \sqsubseteq[r]\ d \implies \mathbf{uses}\ X \cdot c \sqsubseteq[r]\ \mathbf{uses}\ X \cdot d \tag{3.27d}$$

$$c \sqsubseteq[r]\ d \implies c^{\star} \sqsubseteq[r]\ d^{\star} \tag{3.27e}$$

$$c \sqsubseteq[r]\ d \implies c^{\omega} \sqsubseteq[r]\ d^{\omega} \tag{3.27f}$$

$$a \sqsubseteq b \wedge\ c \sqsubseteq d \implies a \parallel c \sqsubseteq b \parallel d \tag{3.27g}$$

Remark. *The branches of parallel composition can be refined independently if the refinements are carried out in an environment independent of context (i.e. $r = true$,law 3.27g). Refinement in a particular context requires one to know the interference that each branch imposes and can tolerate from its environment.*

The next law is derived using lemma 2.72 (Trace countable-choice) and definition 2.73 (Refinement-in-Context).

Law 3.28 (Monotonic-WSL)**.** *For any sets of commands C and D, commands c and d, and relation r,*

$$(\forall d.\ d \in D \longrightarrow (\exists c.\ c \in C \wedge c \sqsubseteq[r]\ d)) \implies \bigsqcap \mathit{acset}\ C \sqsubseteq[r] \bigsqcap \mathit{acset}\ D \tag{3.28a}$$

$$d \in D \implies \bigsqcap \mathit{acset}\ D \sqsubseteq[r]\ d \tag{3.28b}$$

$$(\forall d.\ d \in D \longrightarrow c \sqsubseteq[r]\ d) \implies c \sqsubseteq[r] \bigsqcap \mathit{acset}\ D \tag{3.28c}$$

Proofs in Chapter 6 deliberately omit the application of properties involving monotonicity. These laws are applied in almost every proof of refinement to select the specific component to be refined. We omit the reference to these laws to prevent obfuscating derivations of examples with minor details. Such omission does not imply that these laws are "less important" than others: in fact, if they were not available in our mechanisation, there would be no way around to perform refinement proofs of nested subprograms.

3.3.3 Zeros and units

Skip is the identity for sequential and parallel compositions, *magic* is the unit for non-deterministic choice, and $\langle \mathit{true} \rangle^\omega$ (the program that can do any step whatsoever but is not allowed to abort) is the unit for strict conjunction. *Magic* is the absorbing element for parallel composition and a left-absorbing element for sequential composition; *abort* is the absorbing element for strict conjunction and non-deterministic choice, and is also a left-absorbing element for sequential composition.

Lemma 3.29 (Zeros-and-units)**.** *For any command c,*

$$\mathit{skip}\ ;\ c \sim c \tag{3.29a}$$

$$c\ ;\ \mathit{skip} \sim c \tag{3.29b}$$

$$\mathit{skip} \parallel c \sim c \tag{3.29c}$$

$$\mathit{magic} \sqcap c \sim c \tag{3.29d}$$

$$\langle \mathit{true} \rangle^\omega \Cap c \sim c \tag{3.29e}$$

$$\mathit{magic} \parallel c \sim \mathit{magic} \tag{3.29f}$$

$$\mathit{magic}\ ;\ c \sim \mathit{magic} \tag{3.29g}$$

$$\mathit{abort} \Cap c \sim \mathit{abort} \tag{3.29h}$$

$$\mathit{abort} \sqcap p \sim \mathit{abort} \tag{3.29i}$$

$$\mathit{abort}\ ;\ c \sim \mathit{abort} \tag{3.29j}$$

Remark. *Lemma 3.26 (Assoc-Comm-Dist) states that sequential composition does not distribute through unbounded choice on the left, i.e.* $c \mathbin{;} \bigsqcap \mathit{acset}\ C \sim \bigsqcap \mathit{acset}\ \{c \mathbin{;} d \mid d \in C\}$ *is not a law. The problem with this equivalence is the case* $C = \emptyset$*. It leads to the equivalence* $c \mathbin{;} \mathit{magic} \sim \mathit{magic}$*, which can be combined with law 3.29j to show* $\mathit{magic} \sim \mathit{abort}$*, which is false and not what we want.*

Most of these equivalences were proved using stratified forward simulation (Definition 2.79 on page 58).

3.3.4 Pre and post-conditioned assumptions

A redundant precondition can be introduced/eliminated in the postcondition.

Law (contrib.) 3.30 (Redundant-Pre-Post)**.** *For any predicate p and relation q,*

$$[p, p \wedge q] \sim [p, q]$$

Remark. *Shifting an assumption from the postcondition to the precondition is not possible, i.e.* $[p \wedge q] \not\sqsubseteq [p, q]$*. Preconditions give permission to programs to abort if the assumptions do not hold; assumptions enclosed in a postcondition do not give permission to the specification to abort, instead they force the postcondition command to behave as magic if they do not hold on the before state.*

3.3.5 Iteration

Finite $(c^\star)$, infinite (c^∞) and potentially infinite (c^ω) iteration are defined via fixed-point operators as shown in Figure 3.1. For proof purposes, their definitions can be abstracted using a set of laws for folding and unfolding iteration, and induction over iteration. Recall from Section 3.1.1 that the iteration operators have the highest priority among the commands of the language, thus an expression such as $(\mathit{skip} \sqcap (c \mathbin{;} (c^\star)))$ can have all parenthesis elided.

Lemma 3.31 (Fold/Unfold-Iteration)**.** *For any command c,*

$$c^\star \sim \mathit{skip} \sqcap c \mathbin{;} c^\star \tag{3.31a}$$

$$c^\omega \sim \mathit{skip} \sqcap c \mathbin{;} c^\omega \tag{3.31b}$$

Lemma 3.32 (Iteration-Induction)**.** *For any relation r and commands b, c and d,*

$$b \sqsubseteq[r] d \sqcap c \,;\, b \implies b \sqsubseteq[r] c^\star \,;\, d \tag{3.32a}$$
$$d \sqcap c \,;\, b \sqsubseteq[r] b \implies c^\omega \,;\, d \sqsubseteq[r] b \tag{3.32b}$$

Lemma 3.33 (Isolation)**.** *For any command c,*

$$c^\omega \sim c^\star \sqcap c^\infty$$

Lemma (contrib.) 3.34 (Iteration-Commute)**.** *For any command c,*

$$c \,;\, c^\star \sim c^\star \,;\, c \tag{3.34a}$$
$$c \,;\, c^\infty \sim c^\infty \,;\, c \tag{3.34b}$$

Law 3.35 (Iteration-Properties)**.** *For any relation g and command c,*

$$[g^{**}] \sqsubseteq \langle g \rangle^\star$$

The infinite iteration of a specification $[p, p' \wedge r]$, where p is a predicate and r is a well-founded relation, generates no possible behaviours from states satisfying p, i.e. it is infeasible. The next lemma is taken from [49] and is used in the proof of law 3.112 (Rely-Loop).

Lemma 3.36 (Wellfounded-Infinite-Iteration)**.** *If a relation r is well-founded on a predicate p,*

$$[p, p' \wedge r]^\infty \sim [p, \mathit{false}]$$

Next, we extend Definition 2.81 (page 61) to cover iterated commands. Unrestricted variables are used in Section 3.10 to formalise the nesting of a command inside of a local variable block.

Definition (contrib.) 3.37 (Unrestricted-Iteration)**.** *Let x be a variable and c a command,*

$$\frac{\mathit{unrest}(x, c)}{\mathit{unrest}(x, c^\star)} \quad \frac{\mathit{unrest}(x, c)}{\mathit{unrest}(x, c^\infty)} \quad \frac{\mathit{unrest}(x, c)}{\mathit{unrest}(x, c^\omega)}$$

3.3.6 Termination

The next concept defines from which states a command is required to terminate when running in a specific environment. This notion is particularly useful for the definition of the rely command in Section 3.5, because that definition pulls apart the concerns of behavioural preservation and termination.

The *weakest precondition for termination* of program c in an environment r is formalised as a predicate, $stops(c, r)$. The concept is akin to Dijkstra's weakest precondition [31], but it does not restrict the final state in which the program must terminate.

Definition 3.38 (Stops)**.** *For any command c and relation r, $stops(c, r)$ is the weakest predicate such that from states satisfying $stops(c, r)$, the command c is guaranteed to stop in an environment r.*

$$(\vdash p \Rightarrow stops(c, r)) \;\Leftrightarrow\; (\{p\} \,;\, \langle true\rangle^\star \sqsubseteq [r]\; c)$$

Remark. *The precondition $\{stops(c, r)\}$ is interpreted as the weakest precondition for ensuring that c only performs a finite number of program steps when started in an environment r. That is, if the environment does not unfairly interrupt c, and c does not abort, then its execution ends in a termination step (υ). This interpretation follows from the use of atomic steps in the definition of stops. The program $\langle true\rangle$ can only perform one program step followed by a termination step, but it admits any number of environment steps before the program step takes place. The program $\langle true\rangle^\star$ can perform a finite number of program steps (zero or more), admitting any number of environment steps in between program steps.*

The operator *stops* is monotonic on both arguments. Thus, if we know that a program c stops in an environment r, we also know that it stops in any environment that is more restrictive than r. Similarly, if we know that c stops in an environment r, and we know that d refines c in the environment r, we also know that d stops in the environment r.

Law 3.39 (Term-Monotonic)**.** *For any relations r, r_0 and r_1, and commands c and d,*

$$\vdash r_0 \Rightarrow r_1 \vee idrel \implies \vdash stops(c, r_1) \Rightarrow stops(c, r_0) \tag{3.39a}$$

$$c \sqsubseteq [r]\; d \implies \vdash stops(c, r) \Rightarrow stops(d, r) \tag{3.39b}$$

The next two equivalences allow one to absorb the precondition command from the argument taken by *stops*.

Lemma 3.40 (Term-Equivalences)**.** *For any predicate p, relation r and commands c, c_0 and c_1,*

$$stops(c_0 \,;\, c_1, r) \;=\; stops(c_0 \,;\, \{stops(c_1, r)\}, r) \tag{3.40a}$$

$$stops(\{p\} \,;\, c, r) \;=\; p \wedge stops(c, r) \tag{3.40b}$$

A postcondition ($[q]$) terminates in a stuttering environment r, or aborts if the environment r does any step that changes the state.

Lemma 3.41 (Term-Postcondition)**.** *For any relations q and r,*

$$stops([q], r) = \begin{cases} true, & \text{if} \vdash r \Rightarrow idrel \\ false, & \text{otherwise.} \end{cases} \tag{3.42}$$

An atomic command $\langle p, q \rangle$ stops in an environment r if its precondition p holds in the state where it is executed.

Lemma 3.43 (Term-Atomic)**.** *For any predicate p, and relations q and r, such that p is preserved by relation r,*

$$stops(\langle p, q \rangle, r) = p$$

A test $[[e]]$ only fails to terminate in an environment r if the evaluation of e results in undefined. If a test $[[e]]$ starts its execution from a state that satisfies a predicate p, such that p is both preserved by r and strong enough to ensure that e is well-defined, then $[[e]]$ terminates.

Lemma (contrib.) 3.44 (Term-Test)**.** *Let p be a predicate p, b a boolean expression, and r a relation r, and remember that notion of defined expressions is introduced in Definition 2.6 on page 28. If p is preserved by r and $\vdash p \Rightarrow$ defined b, then*

$$\vdash p \Rightarrow stops([[b]], r)$$

Law (contrib.) 3.45 (Term-Precondition)**.** *For any predicate p and relation r,*

$$stops(\{p\}, r) = p$$

If a precondition p is strong enough to ensure the termination of $c_0 \,;\, c_1$ in an environment z, then it necessarily ensures that the execution of c_0 will terminate in a state from where the execution of c_1 is guaranteed to terminate (considering the environment z).

Lemma (contrib.) 3.46 (Distribute-Stops-Sequential)**.** *For any predicate p and commands c_0 and c_1, such that $\vdash p \Rightarrow stops(c_0 \,;\, c_1, z)$,*

$$\{p\} \,;\, c_0 \sim[z]\, \{p\} \,;\, c_0 \,;\, \{stops(c_1, z)\}$$

The next lemma states that if the parallel composition of finite interference $r \vee idrel$ with a program guarded by a precondition p terminates in an environment z when iterated for a potentially infinite number of times, then the potentially infinite iteration of the guarded program also terminates in an environment $z \vee r$. The insight behind this lemma is that, potentially infinite iteration distributes to the branches of parallel composition, and the potentially infinite iteration of finite interference results in potentially infinite interference. This lemma is used to prove that the rely command distributes over iteration (law 3.85 on page 99).

Lemma 3.47 (Term-Iteration)**.** *For any predicate p, relations r and z, and command c, such that $\vdash p \Rightarrow stops(c, z \vee r)$,*

$$\vdash stops((\{p\} \,;\, c \parallel \langle r \vee idrel\rangle^\star)^{\omega+}, z) \Rightarrow stops((\{p\} \,;\, c)^{\omega+}, z \vee r)$$

If the sequential composition $c \,;\, d$ stops in an environment z, then the first component of the sequential composition also does. Note that we cannot conclude that the second component of the sequential composition always stops in an environment z if $c \,;\, d$ does it; the reason is that the termination of d may be conditional on the state in which c terminates.

Law (contrib.) 3.48 (Term-Sequential)**.** *For any commands c and d and relation z,*

$$\vdash stops(c \,;\, d, z) \Rightarrow stops(c, z)$$

A specification **uses** $X \cdot [p, q]$ stops when executed from a state that satisfies p in an environment that protects all variables in X.

Lemma 3.49 (Term-Uses)**.** *For any command predicate p,*

$$\vdash p \Rightarrow stops(\mathbf{uses}\ X \cdot [p, q], idset\ X)$$

The addition of finite interference that respects r_0 or r_1 to a program c that already terminates in an environment $(r_0 \vee r_1)$ does not affect the termination of c.

Law 3.50 (Term-In-Context)**.** *For any relation r and command c,*

$$stops(c \parallel \langle r \vee idrel \rangle^\star, r) \;=\; stops(c, r)$$

For any predicate p, the execution of the postcondition command $[p]$ in an environment *idrel* terminates in a state where p holds.

Law (contrib.) 3.51 (Term-Post-Precondition)**.** *For any predicate p,*

$$stops([p\text{`}] \,;\, \{p\}, idrel) = true$$

Law (contrib.) 3.52 (Term-Sequential-Special-Case)**.** *For any predicate p, relations r, r_0, r_1, q and z,*

$$\vdash p \Rightarrow stops(c, r) \wedge \vdash stops(d, r) \implies \vdash p \Rightarrow stops(c \,;\, d, r)$$

Law 3.52 is useful to reason about termination of sequential compositions where the rightmost program always terminate, and one has only to care about the termination of the leftmost program to ensure the termination of the composition, e.g. $c \,;\, \langle q \rangle$.

The laws in this section allow one to algebraically reason about the termination of commands, and are particularly useful to prove laws involving the rely command in Section 3.5. They can also be used to compute the weakest precondition for termination of composed specifications, for example, consider the concrete program:

$$[x = 0, (x = 1)\text{`}] \,;\, [x = 1, x' = x + 1] \parallel \langle idrel \rangle^\star$$

The weakest precondition for its termination in a non-interfering environment is $\{x = 0\}$. The key laws involved in the computation of the weakest precondition are laws 3.50 (Term-In-Context), 3.40 (Term-Equivalences), 3.41 (Term-Postcondition), 3.45 (Term-Precondition) and 3.51 (Term-Post-Precondition).

3.4 The guarantee command

The guarantee command restricts the program steps ($\pi\ \sigma\ \sigma'$) that a program can perform. It is designed to handle terminating and non-terminating programs via the strict conjunction of a program and the potentially infinite iteration of atomic steps that respect the guarantee condition or do not modify the state (i.e. stutter).

Definition 3.53 (Guarantee)**.** *Let g be a relation and c a command,*

$$\mathbf{guar}\ g \cdot c \equiv \langle g \vee \mathit{idrel} \rangle^{\omega} \Cap c$$

Remark. *The relation g does not need to be reflexive or transitive; instead, its reflexive-transitive closure is used in the relevant proof obligations of laws that demand g to be reflexive-transitive.*

To understand the intuition behind the guarantee command a few refinement examples are discussed before presenting its formal definition. The next examples are taken from [48], and the laws referred on them are given later in this chapter. To allow the representation of relations in a single line, we represent relations by their characteristic expression (i.e. omitting lambda expressions).

1. Use two assignments that both satisfy the guarantee.

   $\mathbf{guar}\ (x < x') \cdot [\mathit{true}, x' = x + 2]$
   $\sqsubseteq$ by law 3.25 (Sequential) and 3.60b (Guarantee-Monotonic)
   $\mathbf{guar}\ (x < x') \cdot [x' = x + 1]\ ;\ [x' = x + 1]$
   $\sqsubseteq$ by law 3.62a (Distribute-Guarantee)
   $(\mathbf{guar}\ (x < x') \cdot [x' = x + 1])\ ;\ (\mathbf{guar}\ (x < x') \cdot [x' = x + 1])$
   $\sqsubseteq$ by law 3.113 (Assignment-Guarantee, twice)
   $x{:=}x + 1\ ;\ x{:=}x + 1$

2. Using a guarantee to restrict choices: the nested specification is non-deterministic, but in the context of the guarantee surrounding the specification it becomes deterministic.

   $$\mathbf{guar}\ (x < x') \cdot [x' = x + 1 \vee x' = x - 1]$$

$\sim$ by law 3.62g (Distribute-Guarantee) and $(x < x')^{**} = (x \leq x')$

guar $(x < x') \cdot [(x' = x + 1 \vee x' = x - 1) \wedge (x \leq x')]$

$\sim$ by logical simplification

guar $(x < x') \cdot [x' = x + 1]$

3. A specification constrained by a guarantee cannot be implemented if there is no sequence of atomic steps satisfying g that satisfy the postcondition.

guar $(x' < x) \cdot [x' = x + 1]$

$\sim$ by law 3.62g (Distribute-Guarantee) and $(x < x')^{**} = (x \leq x')$

guar $(x' < x) \cdot [(x' = x + 1) \wedge (x \leq x')]$

$\sim$ by law 3.24 (Relational-Composition-Split)

guar $(x' < x) \cdot [\mathit{false}]$

The next section presents a collection of properties of strict conjunction. These properties are used to prove laws involving the guarantee command which are given in Section 3.4.2 (Refining the guarantee command).

3.4.1 Properties of strict conjunction

Strict conjunction ($\Cap$) is at the very core of the definition of the guarantee command. This is an associative, commutative and idempotent operator with identity $\langle \mathit{true} \rangle^{\omega}$ and zero *abort*. It is monotonic with respect to refinement in each of its arguments.

For the next lemma, recall from Figure 2.4 on page 34 that *depends-only* $(g_0, \overline{\{y\}})$ means that the relation g_0 does not restrict the (initial or final) value of y. For example, the relation denoted by $(x \leq x') \wedge (x' \leq z)$ does not constrains the value of y, but constrains the values of x and z.

Lemma 3.54 (Conjunction-Properties)**.** *For any predicates p, p_0 and p_1, relations g, g_0, q, q_0 and q_1, and variable y, such that* $\vdash$ *depends-only* $(g_0, \overline{\{y\}})$*, and commands c, c_0, c_1 and d,*

$$\{p\} \,;\, (c \Cap d) \;\sim\; (\{p\} \,;\, c) \Cap d \tag{3.54a}$$

$$\langle p_0, q_0\rangle \Cap \langle p_1, q_1\rangle \sim \langle p_0 \wedge p_1, q_0 \wedge q_1\rangle \tag{3.54b}$$

$$(\langle p_0, q_0\rangle ; c_0) \Cap (\langle p_1, q_1\rangle ; c_1) \sim (\langle p_0, q_0\rangle \Cap \langle p_1, q_1\rangle) ; (c_0 \Cap c_1) \tag{3.54c}$$

$$(\langle p, q\rangle ; c) \Cap \mathit{skip} \sim \{p\} ; \mathit{magic} \tag{3.54d}$$

$$\mathit{magic} \Cap \langle q\rangle^\omega \sim \mathit{magic} \tag{3.54e}$$

$$\langle g\rangle^\omega \Cap (c ; d) \sim (\langle g\rangle^\omega \Cap c) ; (\langle g\rangle^\omega \Cap d) \tag{3.54f}$$

$$\langle g\rangle^\omega \Cap (c \parallel d) \sim (\langle g\rangle^\omega \Cap c) \parallel (\langle g\rangle^\omega \Cap d) \tag{3.54g}$$

$$\langle \mathit{idrel}\rangle^\omega \Cap [[b]] \sim [[b]] \tag{3.54h}$$

$$\langle g\rangle^\omega \Cap c^\omega \sim (\langle g\rangle^\omega \Cap c)^\omega \tag{3.54i}$$

$$[q_0] \Cap [q_1] \sim [q_0 \wedge q_1] \tag{3.54j}$$

$$\langle g_0\rangle^\omega \Cap (\mathbf{var}\ y \cdot c) \sim \mathbf{var}\ y \cdot \langle g_0\rangle^\omega \Cap c \tag{3.54k}$$

$$\langle \mathit{idset}\ \{y\}\rangle^\omega \Cap (\mathbf{var}\ y \cdot c) \sim \mathbf{var}\ y \cdot c \tag{3.54l}$$

In a non-stuttering environment, the specification $[q]$ aborts, and so does the strict conjunction of $[q]$ and $\langle \mathit{true}\rangle^\star$.

Lemma 3.55 (Conjunction-Spec-Finite)**.** *For any relations q and r, such that $r \neq \mathit{idrel}$,*

$$\langle \mathit{true}\rangle^\star \Cap [q] \sqsubseteq [r] \ [q]$$

In [48], a property known as fusion [5] is introduced to reason about fixed-points in a complete lattice. In this mechanisation, the iterated commands $c^\star$, c^∞ and c^ω are not represented as fixed points, but uninterpreted functions due to technical difficulties that arose in their fixed-point characterisation. In particular, we also did not prove that RG-WSL and the refinement relation form a complete lattice. Therefore, we are unable to apply the property of fusion. To compensate for our abstract characterisation of iterated commands, we have to introduce the next bridging lemma to encode one of the consequences of fusion. It is used to prove law 3.57d. It is worth emphasizing that law 3.57d is an exceptional case: all remaining laws over iterated commands do not require the application of fusion and follow from the application of laws given in Section 3.3.5 (Iteration).

Lemma 3.56 (Terminating-Iteration-Fusion)**.** *For any relation q and command c such that $(\mathit{skip} \sqcap \langle \mathit{true}\rangle ; c) \Cap \langle q\rangle^\omega \sim \mathit{skip} \sqcap \langle q\rangle ; (c \Cap \langle q\rangle^\omega)$,*

$$\langle \mathit{true}\rangle^\star \Cap \langle q\rangle^\omega \sim \langle q\rangle^\star$$

Law 3.57 (Conjunction-Atomic)**.** *For any relations g_0 and g_1 the following hold.*

$$\langle g_0\rangle^{\omega} \Cap \langle g_1\rangle^{\omega} \sim \langle g_0 \wedge g_1\rangle^{\omega} \tag{3.57a}$$

$$\langle g_0\rangle^{\star} \Cap \langle g_1\rangle^{\star} \sim \langle g_0 \wedge g_1\rangle^{\star} \tag{3.57b}$$

$$\langle g\rangle^{\omega} \Cap \langle p, q\rangle \sim \langle p, g \wedge q\rangle \tag{3.57c}$$

$$\langle \mathit{true}\rangle^{\star} \Cap \langle q\rangle^{\omega} \sim \langle q\rangle^{\star} \tag{3.57d}$$

Lemma 3.58 (Interchange-Conjunction)**.** *For any commands c_0, c_1, d_0 and d_1,*

$$(c_0 \parallel c_1) \Cap (d_0 \parallel d_1) \sqsubseteq (c_0 \Cap d_0) \parallel (c_1 \Cap d_1) \tag{3.58a}$$

$$(c_0 \mathbin{;} c_1) \Cap (d_0 \mathbin{;} d_1) \sqsubseteq (c_0 \Cap d_0) \mathbin{;} (c_1 \Cap d_1) \tag{3.58b}$$

3.4.2 Refining the guarantee command

The laws in this section exempt the user from expanding the definition of guarantee while deriving programs.

Law 3.59 (Introduce-Guarantee)**.** *For any relation g and command c,*

$$c \sqsubseteq \mathbf{guar}\ g \cdot c$$

The guarantee command is monotonic with respect to the refinement of its body and the strengthening of the guarantee condition.

Law 3.60 (Guarantee-Monotonic)**.** *For any relations g, g_0 and g_1, and commands c and d,*

$$\vdash g_0 \Rightarrow g_1 \vee \mathit{idrel} \implies \mathbf{guar}\ g_1 \cdot c \sqsubseteq \mathbf{guar}\ g_0 \cdot c \tag{3.60a}$$

$$c \sqsubseteq[r]\ d \implies \mathbf{guar}\ g \cdot c \sqsubseteq[r]\ \mathbf{guar}\ g \cdot d \tag{3.60b}$$

The next lemma is a fundamental property of guarantees used in parallel composition and is used in the proofs of parallel introduction laws from Section 3.8.

Lemma 3.61 (Refine-In-Guar-Context)**.** *For any relations g and r and commands c_0, c_1 and d, such that $a \sqsubseteq[g \vee r]\ b$,*

$$a \parallel (\textbf{guar}\ g \cdot c) \sqsubseteq[r]\ b \parallel (\textbf{guar}\ g \cdot c)$$

Guarantee is a specification command and must not appear in the final implementation. There are several ways of eliminating guarantee commands in a program, but the most practical is to ensure that a guarantee is met when an assignment is introduced. To prepare a program for this elimination strategy, the guarantee must be distributed to the branches of all composite commands before introducing assignments. The next law shows how the guarantee constructor can be distributed over the main commands of RG-WSL.

Law 3.62 (Distribute-Guarantee)**.** *For any predicate p, relations g, g_0, g_1 and q, boolean expression b, commands c and d, and variable x, the following hold.*

$$\textbf{guar}\ g \cdot c\ ;\ d \ \sim\ (\textbf{guar}\ g \cdot c)\ ;\ (\textbf{guar}\ g \cdot d) \tag{3.62a}$$

$$\textbf{guar}\ g \cdot c \parallel d \ \sim\ (\textbf{guar}\ g \cdot c) \parallel (\textbf{guar}\ g \cdot d) \tag{3.62b}$$

$$\textbf{guar}\ g_0 \cdot (\textbf{guar}\ g_1 \cdot c) \ \sim\ \textbf{guar}\ g_0 \wedge g_1 \cdot c \tag{3.62c}$$

$$\textbf{guar}\ \mathit{idset}\ \{x\} \cdot \textbf{var}\ x \cdot c \ \sim\ \textbf{var}\ x \cdot c \tag{3.62d}$$

$$\textbf{guar}\ g \cdot (\textbf{if}\ b\ \ \textbf{then}\ c\ \ \textbf{else}\ d) \ \sim\ \textbf{if}\ b\ \ \textbf{then guar}\ g \cdot c\ \ \textbf{else guar}\ g \cdot d \tag{3.62e}$$

$$\textbf{guar}\ g \cdot (\textbf{while}\ b\ \textbf{do}\ \ c) \ \sim\ \textbf{while}\ b\ \textbf{do}\ \ \textbf{guar}\ g \cdot c \tag{3.62f}$$

$$\textbf{guar}\ g \cdot [q] \ \sim\ \textbf{guar}\ g \cdot [g^{**} \wedge q] \tag{3.62g}$$

$$\textbf{guar}\ g \cdot \langle p, q\rangle \ \sim\ \langle p, (g \vee \mathit{idrel}) \wedge q\rangle \tag{3.62h}$$

$$\textbf{guar}\ g \cdot [[b]] \ \sim\ [[b]] \tag{3.62i}$$

Note that any restriction imposed by the guarantee condition over the global variable x has no effect on the local variable of same name (law 3.62d).

Law (contrib.) 3.63 (Distribute-Guarantee-Var)**.** *For any variable x and relation g, such that g does not depend on x,*

$$\textbf{guar}\ g \cdot \textbf{var}\ x \cdot c \ \sim\ \textbf{var}\ x \cdot (\textbf{guar}\ g \vee \mathit{idset}\ \overline{\{x\}} \cdot c)$$

Remark. *In [48], a different law for distributing the guarantee over local variables is proposed. It is similar to the one presented here, but replaces $g \vee \mathit{idset}\ \overline{\{x\}}$ by g on the right*

hand side of the equivalence. While attempting to mechanise its proof, we discovered the need for an additional assumption, namely: $\exists S. \vdash$ *idset* $S \Rightarrow g$*. This assumption holds, for example, for any reflexive guarantee condition. Without the additional assumption, only the refinement from the left to the right hand-side can be proved.*

The additional assumption is used to prove that $\vdash$ *depends-only* $(g \vee idrel, \overline{\{x\}})$ *from the fact that* $\vdash$ *depends-only* $(g, \overline{\{x\}})$*. The original proof reduces to this sub-goal after expanding the guarantee command and applying law 3.54k (Conjunction-Properties).*

The next section uses guarantees to adapt the concept of frames and invariants from the sequential refinement calculus to the concurrent context. The notion of frames is strengthened to ensure that the restriction on the set of variables that a program can modify is enforced over all atomic transitions of the program, instead of only being enforced between the initial and final states of the execution of the program. A stronger version of the concept of invariants that requires the preservation of a predicate over all atomic actions of a program is also introduced.

3.4.3 Guarantee invariant and frames

An invariant is a predicate that is preserved by the specification command. A guarantee invariant is an extension of the concept of invariant to any command, but enforcing the preservation of the predicate over all atomic actions that a program can perform. In the next definition, a guarantee invariant is formalised as a guarantee with a fixed format. For this definition to make sense, p must be a predicate for p‘ to represent it holding in the after state.

Definition 3.64 (Guarantee invariant)**.** *Let p be a predicate and c a command,*

$$\mathbf{guar{-}inv}\ p \cdot c \equiv \mathbf{guar}\ p \Rightarrow p\text{‘} \cdot c$$

Frames delimit the set of variables that can be changed by a nested command. The concept is taken from the sequential refinement calculus [78]. There, $\{x\}\colon [q]$ stands for $[q \wedge idset\ \overline{\{x\}}]$. The definition in [78] is too coarse-grained to suit concurrent programs, where the intermediate behaviour is as important as the end-to-end behaviour. The problem in reusing Carroll Morgan’s definition is that it would allow intermediate changes to variables specified in the frame as long as their final value were unchanged with respect to the initial one.

To cope with concurrency, a stronger definition of frame is given in terms of the guarantee command. The definition applies to any command instead of being restricted to the specification command.

Definition 3.65 (Frame)**.** *Let x be a set of variables and c a command,*

$$x\colon c \equiv \mathbf{guar}\ \mathit{idset}\ \overline{x} \cdot c$$

A program without frames can be converted into a framed version by taking the frame to be universe of variables names (*UNIV*). Nested frames are equivalent to a single frame formed from the intersection of the sets of variables.

Law 3.66 (Frames)**.** *For any set of variables X and Y, and command c,*

$$c \ \sim\ \mathit{UNIV}\colon c \tag{3.66a}$$

$$X\colon Y\colon c \ \sim\ (X \cap Y)\colon c \tag{3.66b}$$

As general advice, nesting of frames should be avoided to improve the readability of specifications and to avoid confusions. Assuming $x \neq y$, the equivalence between $\{x\}\colon \{y\}\colon c$ and $\emptyset\colon c$ follows directly from the previous theorem, but it is much easier to understand that the frame is empty in $\emptyset\colon c$. Note that a frame formed by the union of sets of variables cannot be decomposed into nested frames.

Law 3.67 (Distribute-Guarantee-Frame)**.** *For any relation g, set of variables X and command c, the following hold.*

$$\mathbf{guar}\ g \cdot X\colon c \ \sim\ X\colon (\mathbf{guar}\ g \cdot c)$$

The next law is more general than the one offered in [48], where $p_0 = p$. The reason for introducing a weakening step in this law is that more often than not, a guarantee invariant will not enforce the preservation of all predicates in a precondition, but only a subset of them. This can be seen, for example, in the refinement step $\mathscr{R}_4$ in the derivation of Findp at page 175.

Law 3.68 (Trade-Guarantee-Invariant)**.** *For any predicates p and p_0 and relation q, such that* $\vdash p_0 \Rightarrow p$*,*

$$[p_0, p\,` \wedge q] \sqsubseteq \textbf{guar}{-}\textbf{inv}\ p \cdot [p_0, q]$$

Care must be taken when introducing guarantee invariants. Nested guarantee invariants impose a stronger requirement on a program than a single guarantee invariant of conjoined predicates. This can be seen in the next law.

Law (contrib.) 3.69 (Distribute-Guarantee-Invariant)**.** *For any predicates p_1 and p_2 and command c,*

$$\textbf{guar}{-}\textbf{inv}\ p_1 \wedge p_2 \cdot c \sqsubseteq \textbf{guar}{-}\textbf{inv}\ p_1 \cdot (\textbf{guar}{-}\textbf{inv}\ p_2 \cdot c)$$

Remark. *The refinement in the opposite direction does not hold. To see why it, take the predicates p_1 and p_2 to be such that $p_1 \wedge p_2 = false$. In this case, a guarantee invariant of $p_1 \wedge p_2$ does not impose any restriction on its body, while the nested guarantee invariants restrict the body.*

Note that although we alert about the consequences of nesting frames and guarantee invariant, there is no actual problem in nesting multiple instances of the guarantee command. This nesting occurs naturally in the development of programs that involve nested parallelism, as can be seen in Section 3.8.1 on page 104.

3.5 The rely command

Unlike the guarantee command, which constrains the steps of a program, the rely command does not constrain the steps the environment of a program can do. Instead, the command **rely** $r \cdot c$ gives permission for the program c to abort if the environment performs a step that does not satisfy the rely-condition r.

Just like the guarantee command, nesting a program into a rely command can potentially turn the resulting program infeasible. To understand the intuition behind the rely command a few examples are discussed before presenting its definition. Here again we adopt the

convention of representing relations by their characteristic expressions (i.e. omitting lambda expressions).

1. **rely** $x < x' \cdot [x + 1 \leq x']$, requires that, when it is put in an environment that may increase x, the value of x is increased by at least one. This command can be implemented by an assignment $x{:=}x + 1$. Note that the environment may interfere by further increasing x, but the combined effect ensures that x is increased by at least one.

2. **rely** $x < x' \cdot [x' = x]$, requires that, when it is put in an environment that may increase x, the value of x is unchanged. There is no implementation for this specification. Even the likely candidate, *skip*, is not an implementation because the environment may increase x, making the overall effect to be the increment of x;

3. **rely** *idset* $\{x, y\} \cdot [x' = x + 2]$ specifies that variables x and y are not affected by the interference. This specification can be implemented by a direct assignment of $x{+}2$ to x, or a sequence of two unary increments. It also allows y to be used as a temporary location for the assignment, since y is not affected by the interference.

The definition of the rely command pulls apart the concerns of behaviour and termination, and is designed specifically to allow the nesting of rely commands, just like multiple instances of a guarantee command can be nested. To allow nesting, the rely command is formulated as a ternary operator, **rely** $(r, z) \cdot c$, where r and z are relations and c is a command. Relation r represents the environment of program c and relation z the rely context within c. For a rely command that has no other rely command nested in c, the relation z assumes the default value *idrel*, which we omit for the sake of readability. If there are nested instances of the rely command (e.g. **rely** $(r_0, z_0) \cdot$ (**rely** $(r_1, z_1) \cdot c$)), the relation z of the outermost rely command has to match the interference recorded by the nested rely (e.g. $z_0 = z_1 \vee r_1$).

The command **rely** $(r, z) \cdot c$ is the most general command that when run in parallel with finite interference $\langle r \vee \mathit{idrel} \rangle^\star$, the composite behaviour implements c from states in which c terminates with interference z, i.e.

$$\{\mathit{stops}(c, z)\} \,;\, c \sqsubseteq [z]\ (\textbf{rely}\ (r, z) \cdot c) \parallel \langle r \vee \mathit{idrel} \rangle^\star \tag{3.70}$$

Equation 3.70 poses a behavioural requirement on the rely command. The parameter z determines the context in which the refinement is required to hold. Equation 3.70 does not

specify the behaviour of **rely** $(r, z) \cdot c$ when it runs with interference that is neither bounded by r nor finite. As example of program generating interference, consider

$$\textit{loop-incr} \equiv \textbf{while } \textit{true} \textbf{ do } \ x{:=}x + 1.$$

The interference it generates is bounded by the relation $x \leq x'$, but it does not terminate. To account for environments such as *loop-incr*, a second requirement is introduced. It states that the command (**rely** $(r, z) \cdot c$) must terminate in an environment $(z \vee r)$ if c does it in an environment z, even if the interference is infinite. Formally:

$$\vdash \textit{stops}(c, z) \Rightarrow \textit{stops}(\textbf{rely } (r, z) \cdot c, z \vee r) \tag{3.71}$$

The role of requirement 3.71 is to prevent the derivation of programs that may not terminate in presence of infinite, yet bounded interference. This requirement prevents, for example, the derivation of a program such as

$$\textbf{var } lx \cdot lx{:=}x \ ; (\textbf{while } lx < x \textbf{ do } \ lx{:=}x)$$

from a specification enclosed in a rely condition of $x \leq x'$. The insight behind this condition is that the program **rely** $(r, z) \cdot c$ must not condition its termination to the termination of the interference. The definition of the rely command given next satisfies conditions 3.70 and 3.71.

Definition 3.72 (Rely)**.** *Let r and z be relations and c a command,*

$$\textbf{rely } (r, z) \cdot c \ \equiv \bigsqcap \textit{acset} \left\{ d \;\middle|\; \begin{array}{l} (\{\textit{stops}(c, z)\} \ ; c \sqsubseteq [z]\ d \parallel \langle r \vee \textit{idrel} \rangle^\star) \wedge \\ (\vdash \textit{stops}(c, z) \Rightarrow \textit{stops}(d, z \vee r)) \end{array} \right\}$$

Next abbreviation is used to omit the parameter z in the rely command if its value is *idrel*.

Abbreviation 3.73 (Rely-Idrel)**.** *Let r be a relation and c a command,*

$$\textbf{rely } r \cdot c \equiv \textbf{rely } (r, \textit{idrel}) \cdot c$$

To prevent ambiguity about the scope of a rely command, whenever this command is composed with other commands, it appears parenthesised. For example, in $(\textbf{rely } r \cdot c) \ ; d$ the rely command is the left argument of the sequential composition. When parentheses are omitted, as in **rely** $r \cdot c \ ; d$, any command appearing at the right of the bullet symbol $(\cdot)$ is meant to be within the scope of the rely command.

3.5.1 Properties of interference

This section presents some basic properties of iteration of atomic commands which represent *interference* in the definition of the rely command. The motivation of this set of laws is to provide support for the proof of properties involving the rely command.

The parallel composition of different sources of interference is equivalent to a single source that behaves as disjunction of the sources (3.74a). Interference on an atomic command can only precede it or follow it (3.74c).

Lemma 3.74 (Properties-Finite-Interference)**.** *For any predicate p, relations r, r_0, r_1 and q, and commands c_0 and c_1,*

$$\langle r_0 \vee r_1 \rangle^\star \sim \langle r_0 \rangle^\star \parallel \langle r_1 \rangle^\star \tag{3.74a}$$

$$c^\star ; c^\star \sim c^\star \tag{3.74b}$$

$$\langle p, q \rangle \parallel \langle r \rangle^\star \sim \langle r \rangle^\star ; \langle p, q \rangle ; \langle r \rangle^\star \tag{3.74c}$$

The next laws allow interference to be distributed to sub-components during the development.

Lemma 3.75 (Distribute-Interference)**.** *For any commands c_0 and c_1 and relation r,*

$$c_0 ; c_1 \parallel \langle r \rangle^\star \sim (c_0 \parallel \langle r \rangle^\star) ; (c_1 \parallel \langle r \rangle^\star)$$

3.5.2 Fundamental properties of rely

The laws in this section are proved from the definition of rely and previously given laws. The next law trades the refinement of the rely command by a refinement and termination proof. It can be used to prevent the expansion of the rely command when it appears at the left hand-side of the refinement relation.

Law 3.76 (Rely-Refinement)**.** *For any relations z and r, and commands c and d,*

$$\mathbf{rely}\ (r, z) \cdot c \sqsubseteq d \Leftrightarrow \begin{pmatrix} \{stops(c, z)\} ; c \sqsubseteq [z]\ d \parallel \langle r \vee idrel \rangle^\star \wedge \\ \vdash stops(c, z) \Rightarrow stops(d, z \vee r) \end{pmatrix}$$

The next law determines the termination condition for the rely command. It is useful for proofs which follow from the application of law 3.76 where d is a rely command.

Law 3.77 (Rely-Stops)**.** *For any relations z and r and command c,*

$$stops(\textbf{rely}\ (r, z) \cdot c, z \vee r) = stops(c, z)$$

The next law over the rely command is similar to lemma 3.22 (Refine-Specification on page 75) over specifications. To refine **rely** $(r, z) \cdot c$ one only needs to concern with the refinement in the environment $(z \vee r)$. In an environment different from $(z \vee r)$, the command **rely** $(r, z) \cdot c$ aborts.

Law 3.78 (Rely-Environment)**.** *For any relations z and r and commands c and d,*

$$\textbf{rely}\ (r, z) \cdot c \sqsubseteq [z \vee r]\ d \Leftrightarrow\ \textbf{rely}\ (r, z) \cdot c \sqsubseteq d$$

Next law shows how nested relies can be refined into a single rely. This refinement is only possible if the total interference recorded by the inner rely (**rely** $(r_1, z) \cdot c$) is taken into account by the external rely, i.e. the parameter z of the external rely has to match $(z \vee r_1)$.

Law 3.79 (Distribute-Rely)**.** *For any relations z, r_0 and r_1, and command c,*

$$\textbf{rely}\ (r_0, z \vee r_1) \cdot (\textbf{rely}\ (r_1, z) \cdot c)\ \sqsubseteq\ \textbf{rely}\ (r_0 \vee r_1, z) \cdot c$$

Next law can be used to introduce a rely command from a command guarded by a precondition. It is stepping stone to prove the more useful law 3.81 (Rely-Idrel-Specification) that is given afterwards.

Law 3.80 (Introduce-Rely-Precondition)**.** *For any predicate p, relations z and r, and command c, such that $\vdash p \Rightarrow stops(c, z)$,*

$$\{p\}\ ;\ c \sqsubseteq [z]\ (\textbf{rely}\ (r, z) \cdot \{p\}\ ;\ c)\ \|\ \langle r \vee idrel \rangle^{\star}$$

The next law states that a specification command with no explicit rely command wrapping it is equivalent to itself wrapped in the context of a rely command of *idrel*.

Law (contrib.) 3.81 (Rely-Idrel-Specification)**.** *For any predicate p and relation q,*

$$(\textbf{rely}\ idrel \cdot [p, q]) \sim [p, q]$$

Remark. *This equivalece does not hold in the general case, where the rely command wraps a command c instead of a specification. The problem is the refinement from the left to the right. By law 3.76 one needs to show the following refinement, which does not hold in general.*

$$\{stops(c, idrel)\}\ ;\ c \sqsubseteq [idrel]\ c \parallel \langle idrel \rangle^\star$$

For example, for $c = skip$, this would require skip to be able to perform stuttering program steps, but it semantics does not allow such kind of transitions.

The next law pulls out a precondition from within the body of a rely command.

Law 3.82 (Distribute-Rely-Precondition)**.** *For any predicate p, relations z and r, and command c, such that $\vdash p \Rightarrow stops(c, z)$,*

$$\textbf{rely}\ (r, z) \cdot \{p\}\ ;\ c \sim \{p\}\ ;\ (\textbf{rely}\ (r, z) \cdot \{p\}\ ;\ c)$$

Lemma 3.83 (Distribute-Rely-Post-Assertion)**.** *For any predicate p, relations z and r, and command c,*

$$\textbf{rely}\ (r, z) \cdot c\ ;\ \{p\} \sqsubseteq (\textbf{rely}\ (r, z) \cdot c)\ ;\ \{p\}$$

Remark. *This lemma is presented as a derived law in [48]. It plays a key role in establishing the distributivity of rely over sequential composition (next law). While mechanising its proof, we discovered the need for lemmas that are not spelled out in [48], and which are challenging to prove. The description in [48] suggest to begin the proof of this refinement by applying law 3.76 (Rely-Refinement on page 95). After this step, one has to prove the next two sub-goals in order to show the validity of the refinement.*

G1: $\{stops(c\ ;\ \{p\}, z)\}\ ;\ (c\ ;\ \{p\}) \sqsubseteq [z]\ (\textbf{rely}\ (r, z) \cdot c)\ ;\ \{p\} \parallel \langle r \vee idrel \rangle^\star$

G2: $\vdash stops(c\ ;\ \{p\}, z) \Rightarrow stops((\textbf{rely}\ (r, z) \cdot c)\ ;\ \{p\}, z \vee r)$

Note that law 3.76 formalises an equivalence. Thus, it is correct to say that its application preserves provability, in the sense that it does not trade a provable conjecture by an unprovable one. Let us consider the first goal, G1. To complete its proof as suggested in [48], we need the next property to hold.

$$P: \quad ((\textbf{rely}\ (r, z) \cdot c) \parallel \langle r \vee idrel \rangle^\star)\ ;\ \{p\} \sqsubseteq (\textbf{rely}\ (r, z) \cdot c)\ ;\ \{p\} \parallel \langle r \vee idrel \rangle^\star$$

Although property P might hold, we were unable to find a proof or produce a counter-example. Despite our efforts to find an alternative proof for G1 without using P, we did not succeed in that. The proof of G2 also stands as an open challenge. Approximately, one man-month was dedicated to search for a proof of this lemma. During this investigation we attempt to produce an alternative proof for this lemma, as well an alternative proof for law 3.84 without using this lemma.

Currently, the only alternative we are aware of that can be used to get rid of the lemma 3.83 in the proof of law 3.84 is to drop the termination requirement in the definition of the rely command. This requirement corresponds to the right hand-side of the conjunction in Definition 3.72 (page 94). The simplified definition allows the rely command to abort whenever the environment is characterised by potentially infinite iteration. The problem is, even terminating loops (e.g. **while** $i < 10$ **do** $i{:=}i + 1$*) are described using the notion of potentially infinite iteration. Thus, the weaker definition of the rely command leaves too many cases unconstrained, where one would like the behaviour of the rely command to be more well-defined. At the end of this chapter we discuss a direction to tweak the definition of the rely command so that it can be made stronger, and perhaps sufficient for establishing law 3.84 without resorting to lemma 3.83.*

Law 3.84 (Distribute-Rely-Sequential)**.** *For any predicate p, relations z and r, commands* c_0 *and* c_1*, such that* $\vdash p \Rightarrow stops(c_0\ ;\ c_1, z)$*,*

$$\textbf{rely}\ (r, z) \cdot \{p\}\ ;\ (c_0\ ;\ c_1) \sqsubseteq (\textbf{rely}\ (r, z) \cdot \{p\}\ ;\ c_0)\ ;\ (\textbf{rely}\ (r, z) \cdot c_1)$$

Law 3.85 (Distribute-Rely-Iteration)**.** *For any predicate p, relations z and r, and command c, such that* $\vdash p \Rightarrow stops(c, z)$*,*

$$\textbf{rely}\ (r, z) \cdot (\{p\}\ ;\ c)^{\omega+} \sqsubseteq (\textbf{rely}\ (r, z) \cdot \{p\}\ ;\ c)^{\omega+}$$

Next, we extend Definition 2.81 (page 61) to cover the rely command. Recall that unrestricted variables are used in Section 3.10 to formalise the nesting of a command inside of a local variable block.

Definition (contrib.) 3.86 (Unrestricted-Rely)**.** *Let z and r be relations, x a variable, and c a command,*

$$\frac{\vdash \mathit{depends\text{-}only}\ (r, \overline{\{x\}}) \vee \vdash r \Rightarrow \mathit{idrel} \qquad \vdash \mathit{depends\text{-}only}\ (z, \overline{\{x\}}) \vee \vdash z \Rightarrow \mathit{idrel} \qquad \mathit{unrest}(x, c)}{\mathit{unrest}(x, \mathbf{rely}\ (r, z) \cdot c)}$$

Remark. *The extension of unrest to deal with the rely command is done via local assumptions (discussed in Section 1.6.4 on page 14). The reason for this is that we cannot derive this rule from the rule for unbounded non-determinism by expanding the definition of rely. The problem in this derivation is to show ($\mathit{unrest}(x, d)$) from the premises $\{\mathit{stops}(c, z)\}\ ;\ c \sqsubseteq [z]\ d \parallel \langle r \vee \mathit{idrel}\rangle^\star$ and $\vdash \mathit{stops}(c, z) \Rightarrow \mathit{stops}(d, z \vee r)$.*

3.5.3 Refining the rely command

The rely command is monotonic with respect to the refinement of its body and weakening of the rely condition.

Law 3.87 (Rely-Monotonic)**.** *For any relations r, r_0 and r_1, such that $\vdash r_0 \Rightarrow r_1 \vee \mathit{idrel}$, and commands c and d, such that $\{\mathit{stops}(c, z)\}\ ;\ c \sqsubseteq [z]\ d$,*

$$\mathbf{rely}\ (r_0, z) \cdot c \ \sqsubseteq\ \mathbf{rely}\ (r_1, z) \cdot c \tag{3.87a}$$

$$\mathbf{rely}\ (r, z) \cdot c \ \sqsubseteq\ \mathbf{rely}\ (r, z) \cdot d \tag{3.87b}$$

Law 3.88 (Rely-Specification)**.** *For any predicate p, relations r and q and command d, such that $\vdash p \Rightarrow \mathit{stops}(d, r)$,*

$$\mathbf{rely}\ r \cdot [p, q] \sqsubseteq d \ \Leftrightarrow\ [p, q] \sqsubseteq d \parallel \langle r \vee \mathit{idrel}\rangle^\star$$

The following law splits a specification command inside of a rely command and applies law 3.84 to distribute the rely command over the sequentially composed specifications. This

law motivates the view of the rely command as a permission given to the specification command, which is equally transferred to the branches of the sequential composition whenever the original specification is split into a sequential composition of specifications.

Law 3.89 (Rely-Sequential)**.** *For any predicates p and mid, and any relations r, q_0 and q_1, such that $\vdash p \wedge (q_0 \wedge mid\text{`} \; ; q_1) \Rightarrow q$,*

$$\textbf{rely}\ r \cdot [p, q] \sqsubseteq (\textbf{rely}\ r \cdot [p, q_0 \wedge mid\text{`}]) \; ; (\textbf{rely}\ r \cdot [mid, q_1])$$

When refining a rely command in a context *rx*, one can strengthen the rely condition to reflect that the environment steps are bounded by *rx*.

Law 3.90 (Strengthen-Rely-In-Context)**.** *For any relations r, z and rx, and command c,*

$$\textbf{rely}\ (r, z) \cdot c \ \sqsubseteq[rx]\ \textbf{rely}\ (rx \wedge r, z) \cdot c$$

Remark. *The mechanised proof of this law requires the extension of RG-WSL with an additional command to complete the instantiation of an existential quantifier introduced in the proof via application of lemma 3.28a. In the paper proof in [48], programs are treated as simple abbreviation for their semantics (i.e. sets of traces), and thus, set comprehension suffices to constrain the environment steps of a program. In the mechanisation, programs are syntactic entities, and the instantiation requires a command to constrain environment steps. The extension of RG-WSL and the proof of the mechanised version of this law are discussed in Section 5.2.*

Similarly to law 3.68, the next law is presented in a version more flexible than the one presented in [48], where p_0 is replaced by p. The motivation for introducing the weakening step is that more often than not, a guarantee invariant will not enforce the preservation of all predicates in a precondition, but only a subset of them.

Law 3.91 (Introduce-Rely-Guar-Invariant)**.** *For any predicate p and relations r and q, such that $\vdash r \Rightarrow p \Rightarrow p\text{`}$ and $\vdash p_0 \Rightarrow p$,*

$$\textbf{rely}\ r \cdot [p_0, p\text{`} \wedge q] \sqsubseteq \textbf{guar}{-}\textbf{inv}\ p \cdot \textbf{rely}\ r \cdot [p_0, q]$$

3.6 Arranging rely and guarantee commands

Rely and guarantee commands can be nested in any order, but only a rely nested inside of a guarantee corresponds to a rely-guarantee specification in the sense of [26]. The problem in nesting a guarantee inside of a rely is that to show **rely** $r \cdot (\mathbf{guar}\ g \cdot [p, q]) \sqsubseteq d$, law 3.76 (Rely-Refinement) requires one to show **guar** $g \cdot [p, q] \sqsubseteq [idrel]\ d \parallel \langle r \vee idrel \rangle^\star$, which requires the interference r as well the implementation d to satisfy the guarantee g. For a rely nested within a guarantee, only d is required to satisfy the guarantee. As a general rule, a guarantee within a rely should be avoided. There are however, situations in which this is acceptable: if the rely is *idrel*, or the more general case where the rely implies the guarantee [48].

To prevent nesting of a guarantee inside of a rely, the most general laws for introducing parallelism discussed in Section 3.8 considers that a command or specification is already nested inside of rely command (see laws 3.96 and 3.97). By using the general laws instead of more specific ones (see laws 3.94 and 3.95), the designer can ensure that the rely command is kept in the innermost part of a program. We illustrate in Section 3.8.1 the consequences of choosing the incorrect rule during a derivation.

Law 3.92 (Guarantee-Plus-Rely)**.** *For any relations g, z and r, and commands c and d, such that* **rely** $(r, z) \cdot c \sqsubseteq d$ *and* **guar** $g \cdot d \sqsubseteq d$*,*

$$\mathbf{guar}\ g \cdot \mathbf{rely}\ (r, z) \cdot c \sqsubseteq d$$

3.7 Trading postconditions with rely and guarantee

A specification surrounded by a rely context of r and a guarantee context of g incorporates the reflexive transitive closure of $g \vee r$ in the postcondition.

Law 3.93 (Trade-Rely-Guarantee)**.** *For any predicate p, relations g, r and q,*

$$\mathbf{guar}\ g \cdot \mathbf{rely}\ r \cdot [p, q] \sim \mathbf{guar}\ g \cdot \mathbf{rely}\ r \cdot [p, q \wedge (g \vee r)^{**}]$$

This law is used to eliminate redundant information from a postcondition. The redundancy comes from the fact that parts of a postcondition may be inferred from the frame and

surrounding rely and guarantee commands. In the absence of a rely command, the standard route to eliminate this kind of redundancy is to use law 3.62g (Distribute-Guarantee). In the case where a rely command stands between the specification command and the frame, this route does not work. In this case, the rely command has to be taken into account as illustrated in the refinement step $\mathscr{R}_{11}$ on page 188.

3.8 Introducing parallelism

We now turn our attention to introduction of parallelism in a program. Rely and guarantee commands play a key role in such introduction because they define the interface between a program and its environment, delimiting the scenarios where an abortive behaviour is acceptable. The reason why rely and guarantee commands are needed in the process of introducing parallelism is better understood via an example. Consider the program $[x' = 1] \Cap [y' = 2]$. As each specification modifies a different variable, we can provide a parallel implementation for this program. An intuitive, but inconsistent, way of deriving a parallel implementation is to replace the strict conjunction by a parallel composition. To understand the problem in this transformation, we need to remember that a specification aborts in any non-stuttering environment, i.e. specifications have an implicit rely condition of *idrel*. The conjunction of commands does not abort in a stuttering environment, but $[x' = 1] \parallel [y' = 2]$ aborts, because each branch updates the state and generates interference, thus breaking the rely of the other branch. Therefore, this transformation introduces a new behavior in the implementation instead of constraining the behaviour of the abstract program. Nevertheless, we can still refine the conjunction to a parallel composition if we take into account the issue of interference, which is well handled by the rely and guarantee commands.

The next law captures the essence of rely-guarantee approach to develop a parallel program. It allows a conjunction of two programs to be implemented by the parallel composition of its parts. This law generalises the parallel introduction law from [26] because it applies to a conjunction of commands rather than a conjunction of postconditions.

Law 3.94 (Introduce-Parallel-SConj)**.** *For any predicate p, relations z, g_0 and g_1, and commands c_0 and c_1 such that $\vdash p \Rightarrow stops(c_0, z) \wedge stops(c_1, z)$,*

$$\{p\} ; (c_0 \Cap c_1) \sqsubseteq [z] \ (\textbf{guar}\ g_0 \cdot \textbf{rely}\ (g_1, z) \cdot \{p\} ; c_0) \parallel (\textbf{guar}\ g_1 \cdot \textbf{rely}\ (g_0, z) \cdot \{p\} ; c_1)$$

The next law specialises the conjunction of commands to a specification. This law is useful to introduce parallelism from a specification that is not nested in a rely command.

Law (contrib.) 3.95 (Introduce-Parallel-Spec)**.** *For any predicates p, p_0 and p_1, and relations q, q_0, q_1, g_0 and g_1, such that $\vdash p \Rightarrow p_0 \wedge p_1$ and $\vdash p \wedge (q_0 \wedge q_1) \Rightarrow q$,*

$$[p, q] \sqsubseteq (\textbf{guar}\ g_0 \cdot \textbf{rely}\ g_1 \cdot [p_0, q_0]) \parallel (\textbf{guar}\ g_1 \cdot \textbf{rely}\ g_0 \cdot [p_1, q_1])$$

Note that the introduction of parallelism can make a feasible specification infeasible. This can occur because the previous law poses no restrictions on the choice of g_0 and g_1 with respect to the stability of $[p_0, q_0]$ and $[p_1, q_1]$[3]. A similar situation occurs in law 3.25 (Sequential), which splits a specification into a sequential composition of specifications; there, the choice of *mid* has an impact on the feasibility of the refined program.

If a specification or a conjunction of commands is already enclosed in a rely command and one needs to further split the specification into a parallel composition, the usage of the previous laws would cause the nesting of a guarantee inside of a rely command. The next laws offer arrangements that are particularly useful for developments where one aims to have nested parallelism. They ensure that rely commands are kept inside of the scope of guarantee commands during the development; this is necessary to allow further refinement using the laws offered in this chapter.

Law 3.96 (Introduce-Parallel-SConj-Nested)**.** *For any predicates p, p_0 and p_1, relations z, r, g_0 and g_1, such that $\vdash p \Rightarrow p_0 \wedge p_1$, and commands c_0 and c_1 such that $\vdash p_0 \Rightarrow stops(c_0, z)$ and $\vdash p_1 \Rightarrow stops(c_1, z)$,*

$$\begin{aligned}&\textbf{rely}\ (r, z) \cdot \{p\}\ ;\ (c_0 \Cap c_1) \sqsubseteq \\ &(\textbf{guar}\ g_0 \cdot \textbf{rely}\ (g_1 \vee r, z) \cdot \{p_0\}\ ;\ c_0) \parallel (\textbf{guar}\ g_1 \cdot \textbf{rely}\ (g_0 \vee r, z) \cdot \{p_1\}\ ;\ c_1)\end{aligned}$$

Aside. *Law 3.94 (Introduce-Parallel-SConj) is special case of law 3.96, where the implicit rely condition is idrel.*

Law 3.97 (Introduce-Parallel-Spec-Nested)**.** *For any predicates p, p_0 and p_1, and relations q, q_0, q_1, g_0, g_1 and r, such that $\vdash p \Rightarrow p_0 \wedge p_1$ and $\vdash p \wedge (q_0 \wedge q_1) \Rightarrow q$,*

$$\textbf{rely}\ r \cdot [p, q] \sqsubseteq (\textbf{guar}\ g_0 \cdot \textbf{rely}\ g_1 \vee r \cdot [p_0, q_0]) \parallel (\textbf{guar}\ g_1 \cdot \textbf{rely}\ g_0 \vee r \cdot [p_1, q_1])$$

[3] As it will be discussed later in this chapter, stability is required for introducing control structures and assignment from specifications.

Aside. *Law 3.95 (Introduce-Parallel-Spec) is special case of law 3.97, where the implicit rely condition is idrel.*

3.8.1 Example: nested parallelism

This example explores the range of choices for introducing parallel composition and investigates the consequences of each choice. As example, we consider the development of a program that performs four assignments in parallel. The implicit type of the variables in this program is **N**. The program to be developed is:

$$(x{:=}x+2 \parallel y{:=}y+2) \parallel (z{:=}x+2 \parallel w{:=}w+1)$$

To abstract this program using the postcondition command we need to consider that the assignment to z may happen before or after the assignment to x. To cater for both scenarios, we can specify that the final value of z will be at least higher than $x+2$. The next specification offers a starting point for this derivation.

$$[true, (x' = x+2) \wedge (y' = y+2) \wedge (x+2 \leq z') \wedge (w' = w+1)]$$

The refinement can be done by splitting the initial specification into a parallel composition of two specifications, each of them responsible for setting the final value of two variables, and then splitting each of these specifications into a nested parallel composition, where each branch sets the value of a single variable. The first attempt to reproduce this intuition is shown below, and reveals that care must be taken when introducing nested parallelism.

$[true, (x' = x+2) \wedge (y' = y+2) \wedge (x+2 \leq z') \wedge (w' = w+1)]$

$\sqsubseteq$ by 3.95 (Introduce-Parallel-Spec)

(**guar** *idset* $\{z, w\} \wedge (x+2 \leq x') \cdot$ **rely** *idset* $\{x, y\} \cdot [(x' = x+2) \wedge (y' = y+2)]) \parallel$
(**guar** *idset* $\{x, y\} \cdot$ **rely** *idset* $\{z, w\} \wedge (x+2 \leq x') \cdot [(x+2 \leq z') \wedge (w' = w+1)]$)

$\sqsubseteq$ by 3.95 (Introduce-Parallel-Spec), 3.62c (Distribute-Guarantee)

and 3.62b (Distribute-Guarantee)

$$\left(\begin{array}{l}\textbf{guar}\ idset\ \{z,w\} \wedge (x+2 \leq x') \cdot \textbf{rely}\ idset\ \{x,y\} \cdot \\ \quad (\textbf{guar}\ idset\ \{z,w,y\} \wedge (x+2 \leq x') \cdot \textbf{rely}\ idset\ \{x\} \cdot [true, x' = x+2]) \parallel \\ \quad (\textbf{guar}\ idset\ \{z,w,x\} \cdot \textbf{rely}\ idset\ \{y\} \cdot [true, y' = y+2])\end{array}\right) \parallel$$
$$\left(\begin{array}{l}\textbf{guar}\ idset\ \{x,y\} \cdot \textbf{rely}\ idset\ \{z,w\} \wedge (x+2 \leq x') \cdot \\ \quad (\textbf{guar}\ idset\ \{x,y,w\} \cdot \textbf{rely}\ idset\ \{z\} \wedge (x+2 \leq x') \cdot [true, x+2 \leq z']) \parallel \\ \quad (\textbf{guar}\ idset\ \{x,y,z\} \cdot \textbf{rely}\ idset\ \{w\} \cdot [true, w' = w+1])\end{array}\right)$$

The use of law 3.95 (Introduce-Parallel-Spec) to introduce nested parallelism results in a program containing a guarantee command nested inside of a rely command. Recall from Section 3.6 (Arranging rely and guarantee commands) that a guarantee nested inside of a rely block requires the environment to satisfy the guarantee condition. This arrangement is only feasible when the rely implies the guarantee condition, which is not the case in this application. To prevent this arrangement from occurring, the existing rely command has to be pushed towards the innermost part of the program in the moment of introducing nested parallelism. This can be done by using law 3.97 (Introduce-Parallel-Spec-Nested). Next, we present the correct derivation.

$[true, (x' = x+2) \wedge (y' = y+2) \wedge (x+2 \leq z') \wedge (w' = w+1)]$

$\sqsubseteq$ by 3.95 (Introduce-Parallel-Spec)

$(\textbf{guar}\ idset\ \{z,w\} \wedge (x+2 \leq x') \cdot \textbf{rely}\ idset\ \{x,y\} \cdot [(x' = x+2) \wedge (y' = y+2)]) \parallel$
$(\textbf{guar}\ idset\ \{x,y\} \cdot \textbf{rely}\ idset\ \{z,w\} \wedge (x+2 \leq x') \cdot [(x+2 \leq z') \wedge (w' = w+1)])$

$\sqsubseteq$ by 3.97 (Introduce-Parallel-Spec-Nested) and 3.87a (Rely-Monotonic)

$$\left(\begin{array}{l}\textbf{guar}\ idset\ \{z,w\} \wedge (x+2 \leq x') \cdot \\ \quad (\textbf{guar}\ idset\ \{z,w,y\} \wedge (x+2 \leq x') \cdot \textbf{rely}\ idset\ \{x\} \cdot [true, x' = x+2]) \parallel \\ \quad (\textbf{guar}\ idset\ \{z,w,x\} \cdot \textbf{rely}\ idset\ \{y\} \cdot [true, y' = y+2])\end{array}\right) \parallel$$
$$\left(\begin{array}{l}\textbf{guar}\ idset\ \{x,y\} \cdot \\ \quad (\textbf{guar}\ idset\ \{x,y,w\} \cdot \textbf{rely}\ idset\ \{z\} \wedge (x+2 \leq x') \cdot [true, x+2 \leq z']) \parallel \\ \quad (\textbf{guar}\ idset\ \{x,y,z\} \cdot \textbf{rely}\ idset\ \{w\} \cdot [true, w' = w+1])\end{array}\right)$$

$\sqsubseteq$ by 3.62b (Distribute-Guarantee), 3.62c (Distribute-Guarantee)

and 3.60a (Guarantee-Monotonic)

$$\begin{pmatrix} (\textbf{guar}\ idset\ \{z, w, y\} \wedge (x + 2 \leq x') \cdot \textbf{rely}\ idset\ \{x\} \cdot [true, x' = x + 2]) \parallel \\ (\textbf{guar}\ idset\ \{z, w, x\} \cdot \textbf{rely}\ idset\ \{y\} \cdot [true, y' = y + 2]) \end{pmatrix} \parallel \\ \begin{pmatrix} (\textbf{guar}\ idset\ \{x, y, w\} \cdot \textbf{rely}\ idset\ \{z\} \wedge (x + 2 \leq x') \cdot [true, x + 2 \leq z']) \parallel \\ (\textbf{guar}\ idset\ \{x, y, z\} \cdot \textbf{rely}\ idset\ \{w\} \cdot [true, w' = w + 1]) \end{pmatrix}$$

The derivation can continue by introducing assignments using laws presented in Section 3.12 (Control structures and assignment). The key point of the example is to show the consequence of choosing a wrong law for introducing nested parallelism. In that case, an uninformed user might continue the refinement from a bad design decision only to discover that after eliminating the innermost rely and guarantee commands there would be no way of get rid of the outermost rely and guarantee commands.

3.9 Expressions and tests

A successful test evaluates its argument non-atomically and can only perform stuttering program steps. The evaluation performed by the test command is sensitive to interference, and aborts if the evaluation of its argument results in undefined.

Lemma 3.98 (Introduce-Test)**.** *For any boolean expression b,*

$$[\mathit{defined}\ b, [\![b]\!]_r \wedge \mathit{idrel}] \sqsubseteq [[b]]$$

Remark. *This lemma establishes a semantic connection between relational and non-atomic evaluation, and is needed in proofs involving the introduction of control structures in programs that are not subject to interference.*

If a boolean expression b satisfies the single reference property (Section 3.2.1) with respect to a relation r, the value of e is its value in the state in which the unstable variable is sampled, or if there is no unstable variable the value of e is stable under interference. In both cases, given an initial state s, the evaluation state is related to the initial state by r^{**}. This property is encoded by the following law, which is applied in proof of law 3.100 (Rely-Test).

Lemma 3.99 (Test-Single-Reference)**.** *For any predicate p, relations q and r, and boolean expression b, such that b satisfies the single reference property with respect to r,*

$$[p, q] \sqsubseteq [[b]] \parallel \langle r \vee \mathit{idrel}\rangle^\star \ \Leftrightarrow\ [p, q] \sqsubseteq \langle r \vee \mathit{idrel}\rangle^\star \,;\, [[b]] \,;\, \langle r \vee \mathit{idrel}\rangle^\star$$

The next law plays a similar role to law 3.98, but can be used if a specification is wrapped in a rely command. The side conditions of this law ensure that the relational evaluation captures all possible outcomes of a non-atomic evaluation.

Law 3.100 (Rely-Test)**.** *For any relation r, predicate p that is preserved by r, boolean expression b that has the single reference property with respect to r and predicate* b_0 *such that* $\vdash p \wedge [\![b]\!]_r \Rightarrow b_0$ *and* $\vdash p \wedge r \Rightarrow (b_0 \Rightarrow b_0\text{`})$ *and* $\vdash p \Rightarrow$ *defined b,*

$$\textbf{rely}\ r \cdot [p, r^{**} \wedge b_0\text{`}] \sqsubseteq [[b]]$$

Aside. *Remember from Section 3.2.1 that an expression e satisfies the single reference property with respect to a relation r, if e has at most a single variable that is unstable under the relation r, and e has at most a single occurrence of such a variable.*

3.10 Local variables

The command (**var** $x \cdot c$) shadows the global variable x within its body c; in other words, references to x within c are directed to the local variable x and the global variable of same name becomes inaccessible. The local variable x is not affected by interference external to **var** $x \cdot c$.

Recall that local variable blocks are monotonic with respect to the refinement of its body[4], and have distributive and absorption properties over strict conjunction and guarantees[5]. The next two laws on local variables are about their introduction in a program. Since the introduction of local variables affects the frame of a program, these laws require the frame of a program to have already been declared to enable their application.

Lemma 3.101 (Introduce-Variable-Frame)**.** *For any variable x, set of variables Y, and command c, assuming x is not in Y and is unrestricted in c,*

$$Y\text{:}\ c \sqsubseteq \textbf{var}\ x \cdot (\{x\} \cup Y)\text{:}\ c$$

[4]Lemma 3.27c.
[5]Lemma 3.54, and laws 3.62 and 3.63.

The next law states that local variables are shielded from the interference in its outer scope.

Law 3.102 (Introduce-Variable-Rely)**.** *For any relations z and r, variable x and set of variables Y, such that x is not in Y and is unrestricted in* **rely** $(r, z) \cdot c$*, then*

$$Y\colon (\textbf{rely}\ (r, z) \cdot c) \sqsubseteq \textbf{var}\ x \cdot (\{x\} \cup Y)\colon (\textbf{rely}\ (\mathit{idset}\ \{x\} \wedge r, z) \cdot c)$$

3.10.1 Example: shadowing

This example discusses shadowing of global variables using the local variables. Shadowing refers to the situation where a global variable becomes inaccessible within the scope of a local variable block of same name. Prior to shadowing a variable, we have to make it unrestricted within the the command where it will be shadowed. In the case where the command is a specification, this can be done by splitting the specification into a sequential composition, and then using appropriate laws to manipulate the frame of the specification. The process is illustrated next. For readability, relations are represented by their characteristic expressions.

$$\begin{array}{rl} & \textbf{guar}\ (x \le x') \cdot \{x\}\colon [\mathit{true}, x' = x + 2] \\ \mathscr{R}_1 \equiv & \sqsubseteq \text{ by 3.25 (Sequential) and 3.62a (Distribute-Guarantee)} \\ & \textbf{guar}\ (x \le x') \cdot \{x\}\colon [\mathit{true}, x' = x + 1]; \\ & \textbf{guar}\ (x \le x') \cdot \{x\}\colon [\mathit{true}, x' = x + 1] \\ \mathscr{R}_2 \equiv & \sqsubseteq \text{ by 3.25 (Sequential) and 3.62a (Distribute-Guarantee)} \\ & \textbf{guar}\ (x \le x') \cdot \{x\}\colon [\mathit{true}, x' = x + 1]; \\ & \textbf{guar}\ (x \le x') \cdot \{x\}\colon [\mathit{true}, \mathit{idset}\ \{x\}]; \\ & \textbf{guar}\ (x \le x') \cdot \{x\}\colon [\mathit{true}, x' = x + 1] \end{array}$$

◁

At this point, we refine a specification that is only allowed to stutter by another one that can perform internal computations on a local variable. In this situation, even though for an external observer a program may appear inactive, it may be actively performing internal computations. The left-hand side of the refinement $\mathscr{R}_3$ corresponds to the program marked with $\lhd$ in the result of the refinement $\mathscr{R}_2$.

$$
\begin{array}{lll}
\mathscr{R}_3 & \equiv & \sqsubseteq \text{ by 4.7c (Monotonic-Derived)} \\
 & & \textbf{guar } (x \leq x') \cdot \emptyset\colon [\mathit{true}, \mathit{idset}\ \{x\}] \\
\mathscr{R}_4 & \equiv & \sim \text{ by 4.9 (Trade-Spec-Frame)} \\
 & & \textbf{guar } (x \leq x') \cdot \emptyset\colon [\mathit{true}, \mathit{true}] \\
\mathscr{R}_5 & \equiv & \sqsubseteq \text{ by 3.101 (Introduce-Variable-Frame)} \\
 & & \textbf{guar } (x \leq x') \cdot \textbf{var } x \cdot \{x\}\colon [\mathit{true}, \mathit{true}] \\
\mathscr{R}_6 & \equiv & \sqsubseteq \text{ by 3.60a (Guarantee-Monotonic)} \\
 & & \textbf{guar } \mathit{idset}\ \{x\} \cdot \textbf{var } x \cdot \{x\}\colon [\mathit{true}, \mathit{true}] \\
\mathscr{R}_7 & \equiv & \sim \text{ by 3.62d (Distribute-Guarantee)} \\
 & & \textbf{var } x \cdot \{x\}\colon [\mathit{true}, \mathit{true}]
\end{array}
$$

From this point, the development can continue by strengthening the restrictions on the final value of the local variable x. There is no relationship between the final value of the local variable x and the global variable.

3.11 Restricting access to variables

Given a command c, a way of syntactically controlling interference is to delimit the variables that are in the scope of c that can be referenced by c. RG-WSL offers the command (**uses** $X \cdot c$) to restrict the set of variables that c can use to those in the set X. Any command c can be refined by constraining the variables it can use.

Lemma 3.103 (Introduce-Uses)**.** *For any command c and set of variables X,*

$$c \sqsubseteq \textbf{uses } X \cdot c$$

The syntactic control of interference is formalised by the next two laws. The first of these trades the refinement of a specification by a parallel program for the refinement of the same

specification by a sequential program. Law 3.105 refines a specification enclosed in a **rely** by a specification enclosed in a **uses** command.

Lemma 3.104 (Uses-Atomic-Effective)**.** *For any predicate p, relation q, command c, and set of variables X,*

$$(\ [p, q] \sqsubseteq (\mathbf{uses}\ X \cdot c) \parallel \langle idset\ X \rangle^\star) \ \Leftrightarrow\ (\ [p, q] \sqsubseteq \langle idset\ X \rangle^\star\ ;\ (\mathbf{uses}\ X \cdot c)\ ;\ \langle idset\ X \rangle^\star)$$

Law 3.105 (Rely-Uses)**.** *For any predicate p, relation q, and set of commands X, such that* $[p, q]$ *tolerates interference idset X,*

$$\mathbf{rely}\ idset\ X \cdot [p, q] \sqsubseteq \mathbf{uses}\ X \cdot [p, q]$$

If a program is enclosed in a **uses** command, it carries this syntactic restriction throughout its development. The command **uses** itself is not code and must be eliminated in the last stage of the development, when its body contains only code. The notions of *code* and *free variables for code* are necessary to formalise the elimination of the **uses** block and are defined next. The elimination of the **uses** command is briefly discussed in [48], but is not formalised there.

Definition (contrib.) 3.106 (Code)**.** *Let c and d be commands, x a variable, e an expression, and b a boolean expression,*

$$\frac{}{code\ (x{:=}e)} \qquad \frac{}{code\ [[b]]} \qquad \frac{code\ c}{code\ (\mathbf{var}\ x \cdot c)}$$

$$\frac{code\ c}{code\ (\mathbf{while}\ b\ \mathbf{do}\ \ c)} \qquad \frac{code\ c \qquad code\ d}{code\ (\mathbf{if}\ b\ \ \mathbf{then}\ c\ \ \mathbf{else}\ d)} \qquad \frac{code\ c \qquad code\ d}{code\ (c\ ;\ d)}$$

$$\frac{code\ c \qquad code\ d}{code\ (c \parallel d)}$$

Definition (contrib.) 3.107 (Free-Variables-Code)**.** *Let c, c_1 and c_2 be code, x a variable, e an expression, and b a boolean expression. The set of free variables of programs which qualify as code is inductively defined as follows:*

$$\frac{}{\mathit{free}\ [[b]]\ (\mathit{free\text{-}exp}\ b)} \qquad \frac{}{\mathit{free}\ (x{:=}e)\ (\{x\} \cup \mathit{free\text{-}exp}\ e)}$$

$$\frac{\mathit{free}\ c_1\ X_1 \qquad \mathit{free}\ c_2\ X_2}{\mathit{free}\ (c_1\ ;\ c_2)\ (X_1 \cup X_2)} \qquad \frac{\mathit{free}\ [[b]]\ X_0 \qquad \mathit{free}\ c_0\ X_1 \qquad \mathit{free}\ c_1\ X_2}{\mathit{free}\ (\textbf{if}\ b\ \ \textbf{then}\ c_0\ \ \textbf{else}\ c_1)\ (X_0 \cup X_1 \cup X_2)}$$

$$\frac{\mathit{free}\ c_1\ X_1 \qquad \mathit{free}\ c_2\ X_2}{\mathit{free}\ (c_1 \parallel c_2)\ (X_1 \cup X_2)} \qquad \frac{\mathit{free}\ [[b]]\ X_0 \qquad \mathit{free}\ c_1\ X_1}{\mathit{free}\ (\textbf{while}\ b\ \textbf{do}\ \ c)\ (X_0 \cup X_1)}$$

$$\frac{\mathit{free}\ c\ X}{\mathit{free}\ (\textbf{var}\ x \cdot c)\ (X - \{x\})}$$

The next lemma formalises the strategy suggested to eliminate **uses** blocks in [48], which consists into syntactically discharge this constraint.

Lemma (contrib.) 3.108 (Elimination-Uses)**.** *For any command c, which consists only of code, and set of variables X, such that all free variables of c are in X,*

$$\textbf{uses}\ X \cdot c \sqsubseteq c$$

Proof. The assumption that c is code prevents further refinements to extend the set of free variables of c. Thus, it is only necessary to show that c already satisfies the constraint imposed by the **uses** block. This can be done by syntactically checking that all free variables in c are contained in X. □

Remark. *The strategy for eliminating the **uses** constructor assumes that once code is reached, the user will not try to further refine the code. Such additional refinement steps are possible by expanding the definition of the commands classed as code and further refining the sub-components.*

Law (contrib.) 3.109 (Distribute-Uses)**.** *For any sets of variables X and Y, relation g, predicate p and command c,*

$$\mathbf{guar}\ g \cdot \mathbf{uses}\ X \cdot c \sqsubseteq \mathbf{uses}\ X \cdot (\mathbf{guar}\ g \cdot c) \tag{3.109a}$$

$$\mathbf{guar{-}inv}\ p \cdot (\mathbf{uses}\ X \cdot c) \sqsubseteq \mathbf{uses}\ X \cdot \mathbf{guar{-}inv}\ p \cdot c \tag{3.109b}$$

$$Y\colon (\mathbf{uses}\ X \cdot c) \sqsubseteq \mathbf{uses}\ X \cdot Y\colon c \tag{3.109c}$$

Once a **uses** block has been introduced, the user can opt to use lemma 3.26 (Assoc-Comm-Dist) to distribute the **uses** block through its body, or to monotonically refine the body without distributing the **uses** block. The first strategy leads to a situation where multiple instances of the **uses** block need to be eliminated separately, whereas the second strategy is more efficient as it only requires a single proof to eliminate the **uses** block. Note that the user is allowed to refine the body of a **uses** block without respecting the restriction imposed by this constructor, but in such case the **uses** block cannot be eliminated. An **uses** block whose body includes free variables outside those in X cannot perform program steps unless it aborts.

The next rule complements the definition of unrestricted variables given in Section 2.11 to state that if a variable is not free in a code c, then it is unrestricted.

Definition (contrib.) 3.110 (Unrestricted-Free)**.** *Let x be a variable, c a command, and S a set of variables,*

$$\frac{\mathit{free}\ c\ S \qquad x \notin S \qquad \mathit{code}\ c}{\mathit{unrest}(x, c)}$$

3.12 Control structures and assignment

The laws in this section introduce while loops, conditionals and assignments from specifications wrapped in rely and guarantee commands. A limitation of the laws that introduce control structure is that they require the guard to satisfy the single reference property with respect to the rely condition. Thus, if the guard does not respect the single reference property, the offending occurrences of unstable variables must be read and stored in local variables before introducing control structures.

In general, refining programs requires one to collect information about the state of a program along the derivation to then use this information to discharge proof obligations in later stages of the development. The guard of a control structure generally provides relevant

information to the context of its body. In the sequential refinement calculus, for example, one can assume the validity or negation of the guard in specific parts of the body of the control structure. However, if the control structure is subject to interference, the extraction of information from its guard (b) requires additional care. This is because the predicate denoted by the guard ($[\![b]\!]_r$) may not be stable under the environment actions, but may imply a weaker predicate b_0 that is stable. Similarly, the negation of the guard ($[\![\neg b]\!]_r$) may not be stable under the environment actions, but may imply a weaker predicate b_1 that is stable. This weakening is useful for the purpose of enriching the context within the body of the control structure. In page 182, for example, the refinement step $\mathscr{R}_{20}$ introduces a loop whose guard is $ok < ot \wedge ok < et$, but since only the condition $ok < ot$ is stable under interference $(et' \leq et) \wedge idset\ \{ot, ok, v\}$, only this part of the guard is used to enrich the precondition of the specification nested in the loop.

Law 3.111 (Rely-Conditional)**.** *For any predicates p, b_0 and b_1, relations r and q, such that $[p, q]$ tolerates interference r, and boolean expression b, such that b satisfies the single reference property with respect to r and $\vdash p \wedge [\![b]\!]_r \Rightarrow b_0$ and $\vdash p \wedge r \Rightarrow (b_0 \Rightarrow b_0\text{`})$, and $\vdash p \wedge [\![\neg b]\!]_r \Rightarrow b_1$ and $\vdash p \wedge r \Rightarrow (b_1 \Rightarrow b_1\text{`})$, and $\vdash p \Rightarrow$ defined b,*

$$\textbf{rely}\ r \cdot [p, q] \sqsubseteq (\textbf{if}\ b\ \ \textbf{then rely}\ r \cdot [p \wedge b_0, q]\ \ \textbf{else rely}\ r \cdot [p \wedge b_1, q])$$

Law 3.112 (Rely-Loop)**.** *For predicates p, b_0 and b_1, relations r, w and q, and set of variables X, such that p is preserved by r and w is well-founded on p and $\vdash$ depends-only (w, X) and $\vdash p \wedge r^{**} \Rightarrow {w^{**}}_X$ and boolean expression b, such that b satisfies the single reference property with respect to r and $\vdash p \wedge [\![b]\!]_r \Rightarrow b_0$ and $\vdash p \wedge r \Rightarrow (b_0 \Rightarrow b_0\text{`})$, and $\vdash p \wedge [\![\neg b]\!]_r \Rightarrow b_1$, and $\vdash p \wedge r \Rightarrow (b_1 \Rightarrow b_1\text{`})$ and $\vdash p \Rightarrow$ defined b,*

$$\textbf{rely}\ r \cdot [p, p\text{`} \wedge b_1\text{`} \wedge {w^{**}}_X] \sqsubseteq \textbf{while}\ b\ \textbf{do}\ \ \textbf{rely}\ r \cdot [p \wedge b_0, p\text{`} \wedge w]$$

Remark. *Remember from definition 2.17 on page 34 that the notation ${r^{**}}_X$ is the transitive closure of r enlarged with $(idset\ X)$. To apply the law above, the loop invariant must be encoded using the predicate p, and the well-founded relation w should only concern the variables relevant to the falsification of the loop condition. A potential mistake in the application of this law is to attempt to encode the postcondition in the well-founded*

relation. The postcondition should be encoded using the loop invariant and the negation of the condition of the loop.

To introduce an assignment from a specification in the context of a guarantee, one must ensure that the assignment implements the postcondition and also satisfies the guarantee. These checks allow the elimination of the guarantee constructor. The precondition must ensure that the expression appearing in the right-hand side of the assignment is well-defined.

Law 3.113 (Assignment-Guarantee)**.** *For any predicate p, relation g, variable x and expression e such that* $\vdash ((p \wedge (\lambda s\ s'.\ s'\ x = [\![e]\!]_v\ s) \wedge \mathit{idset}\ \overline{\{x\}}) \Rightarrow (q \wedge (g \vee \mathit{idrel})))$ *and also* $\vdash p \Rightarrow \mathit{defined}\ e$*,*

$$\mathbf{guar}\ g \cdot [p, q] \sqsubseteq x{:=}e$$

Remark. *For nested guarantee contexts, one has to flatten the nested guarantees using law 3.62c (Distribute-Guarantee) before applying law 3.113 (Assignment-Guarantee). Otherwise, the refinement will lead to an assignment nested in a guarantee command.*

Law 3.114 (Assignment-Rely-Guarantee)**.** *For any variable x, expression e, set of variables X, predicate p and relations g and q, such that* $[p, q]$ *tolerates interference idset X, and* $\vdash p \Rightarrow \mathit{defined}\ e$*, and* $\vdash ((p \wedge (\lambda s\ s'.\ s'\ x = [\![e]\!]_v\ s) \wedge \mathit{idset}\ \overline{\{x\}}) \Rightarrow (q \wedge (g \vee \mathit{idrel})))$*, and* $\mathit{free\text{-}exp}\ e \cup \{x\} \subseteq X$*,*

$$\{x\}\colon (\mathbf{guar}\ g \cdot \mathbf{rely}\ \mathit{idset}\ X \cdot [p, q]) \sqsubseteq x{:=}e$$

The next law is adequate for situations where the rely condition is not in the shape *idset X*. Its proof obligations use the single-reference property (Section 3.2.1) to allow assignment to be introduced in situations where its right-hand side includes up to one occurrence of a single unstable variable. The key idea is that, if the expression on the right-hand of the assignment is stable under interference, then the assignment can be introduced.

The proof of next law published in [48] has minor slips. These discussed and fixed in Section A.3 (Law 3.115 (Assignment-Single-Reference)) on page 260.

Law 3.115 (Assignment-Single-Reference)**.** *For any variable x, expression e, predicate p, and relations r and q, such that* $[p, q]$ *tolerates interference r,* $\vdash p \Rightarrow \mathit{defined}\ e$*, e satisfies the single reference property with respect to r and is preserved by r, i.e.* $\mathit{tol\text{-}interf}(\mathit{true}, \lambda s\ s'.\ [\![e]\!]_v\ s = [\![e]\!]_v\ s', r)$*, and* $\vdash p \wedge (\lambda s\ s'.\ s'\ x = [\![e]\!]_v\ s) \wedge \mathit{idset}\ \overline{\{x\}} \Rightarrow q$

$$\textbf{rely}\ r \cdot [p, q] \sqsubseteq x{:=}e$$

3.13 Discussion and summary of contributions

This chapter introduced a total of 115 laws and definitions, including many of which are not present in [48]. Novel laws and definitions are accompanied by the keyword **contrib** and are summarised in Table 3.2. Each additional law or definition shown in this table is used at least once in the mechanisation of top-level laws. This means that we introduced no irrelevant laws and definitions. For a full appreciation of the application of these laws and definitions, we invite the reader to consult the proofs available in Appendix A.1.

Recall that we use the term *lemma* to refer to properties whose proof directly follows from the semantics of RG-WSL and the term *law* to refer to properties whose proof can be derived without directly appeal to the semantics. A key contribution of our work is that all the definitions and laws presented in this chapter are mechanised using Isabelle/HOL. Moreover, we prove that all laws are derived from the set of lemmas discussed in this thesis. The lemmas themselves are a key part of the mechanisation, but in general, we have not proved them from the semantics. Instead, Chapter 4 discusses the use of locales for encoding the lemmas as local assumptions of the algebra.

Reference	Name	Type	Page
3.24	Relational-Composition-Split	Lemma	75
3.30	Redundant-Pre-Post	Law	79
3.44	Term-Test	Lemma	82
3.34	Iteration-Commute	Lemma	80
3.45	Term-Precondition	Law	82
3.46	Distribute-Stops-Sequential	Lemma	83
3.63	Distribute-Guarantee-Var	Law	89
3.69	Distribute-Guarantee-Invariant	Law	92
3.81	Rely-Idrel-Specification	Law	97
3.86	Unrestricted-Rely	Lemma	99
3.106	Code	Definition	110
3.107	Free-Variables-Code	Definition	111
3.108	Elimination-Uses	Lemma	111
3.109	Distribute-Uses	Law	112
3.110	Unrestricted-Free	Definition	112

Table 3.2 Summary of novel laws and definitions

The main contributions of this chapter are:

1. Identification and formalisation of relevant lemmas about RG-WSL that are missing in [48]. Each lemma presented in this chapter is used at least once in the mechanisation. There are no lemmas that are defined and never used in proofs;

2. Investigation of the route to be taken by a designer in order to derive programs featuring nested parallelism. We illustrate the pitfalls involved with nested parallelism in Section 3.8.1.

3. Extension of the concept of unrestriction to cover derived commands;

4. Notion of code and free variables for code. We apply these concepts to design a method for eliminating the **uses** command from implemented code. The application of the proposed method is illustrated in Section 6.4;

5. A novel interpretation for $stops(c, r)$ that provides a direct link between the concept formalised by *stops* and the notion of program steps (π) discussed in Section 2.7 (Operational semantics on page 39);

6. Investigation of the stability of *stops* and its impact on the theory;

7. A generalisation of *stops*, used to tweak the definition of the rely command to sheds light into an alternative proof for law 3.84 (Distribute-Rely-Sequential).

3.13.1 Intricate aspects of using the R/G refinement calculus

Indistinguishable rely conditions

The definition of rely does not distinguish between a relation that only allows stuttering interference from a relation that does not allow interference at all. The next equivalences formalise this observation, and also show that adding stuttering interference to a rely condition is equivalent to add no interference at all.

Law 3.116 (Rely-Equivalences)**.** *For any relations z and r, and command c,*

$$\textbf{rely}\ (idrel,\ z) \cdot c = \textbf{rely}\ (false,\ z) \cdot c$$
$$\textbf{rely}\ r \cdot c = \textbf{rely}\ (r,\ false) \cdot c$$

Stability of *stops*

We investigated the stability of $stops(c, r)$ under r and discovered that $stops(c, r)$ is not stable under r in general. Recall that stability is formalised by the definition Tolerate-Interference introduced on page 72.

Law 3.117 (Stops-Not-Stable)**.** *The predicate $stops(c, r)$ is not stable under r in general.*

$$\neg\ (\forall\, c\ r.\ tol\text{-}interf\,(stops(c, r),\ true,\ r))$$

Proof. This can be proved by showing that $\exists c\ r.\ \neg\ tol\text{-}interf\,(stops(c,\ r),\ true,\ r)$. To instantiate c use the command $\{x = 0\}$ and to instantiate r use the relation $x < x'$. The proof that $\neg\ tol\text{-}interf\,(stops(c,\ r),\ true,\ r)$ follows immediately from definition 3.16 (Tolerate-Interference). □

3.13.2 Stronger definition of rely command

Here we discuss a small change in the definition of the rely command that might lead to a proof of law 3.84 (Distribute-Rely-Sequential on page 98) that is independent from lemma 3.83 (Distribute-Rely-Post-Assertion on page 97). The importance of law 3.84 can be succinctly described by the fact that it distributes the rely command over the components of sequential composition. Thus, it provides a mechanism to distribute the permission of making assumptions about the environment to sub-components of a program.

In the first attempt to get rid of the controversial lemma 3.83 in the proof of law 3.84 we dropped the termination condition in the definition of the rely command. While this simplification eliminates the need for lemma 3.83, it creates another problem. Dropping the terminating requirement on the rely command causes this command to be given permission to abort when put run with infinite but fair interference satisfying r. In general, we want the rely command to terminate even in situations where the interference does not cease to exist.

For a second attempt to eliminate lemma 3.83 from the proof of law 3.84 we suggest to tweak the definition of the rely command. But this time, instead of dropping (weakening) the termination condition as we experimented, we suggest to make it stronger. For that, we introduce a generalisation of stops that takes into account the state in which a program is required to terminate.

Definition 3.118 (Weakest Precondition)**.** *Let p and q be predicates, r a relation, and c a command.*

$$(\vdash p \Rightarrow wp(c, r, q)) \;\Leftrightarrow\; (\{p\} \,;\, \langle true\rangle^\star \sqsubseteq [r]\; c \,;\, \{q\})$$

The command $\{wp(c, r, q)\}$ denotes the weakest precondition for termination of c in a state satisfying q when running in an environment r. Note that *stops* is a special case of the definition of wp, that is, $stops(c, r) = wp(c, r, true)$. A novel definition for the rely command is proposed next. It leaves the behavioural restriction of the rely command unaffected, but strengthens the terminating condition.

Definition (contrib.) 3.119 (Rely-WP)**.** *Let r and z be relations and c a command,*

$$\textbf{rely}\ (r,\, z) \cdot c \ \equiv\ \bigsqcap \left\{ d \;\middle|\; \begin{array}{l} (\{wp(c,\, true,\, z)\}\ ;\ c \sqsubseteq [z]\ d \parallel \langle r \vee idrel \rangle^{\star}) \wedge \\ \forall\ q.\ \left(\begin{array}{c} tol\text{-}interf(true,\, q,\, z \vee r) \\ \longrightarrow \\ \vdash wp(c,\, q,\, z) \Rightarrow wp(d,\, q,\, z \vee r) \end{array} \right) \end{array} \right\}$$

This definition determines that **rely** $(r,\, z) \cdot c$ must stop when running in an environment $z \vee r$ from states where c stops in an environment z. Up to this point, it looks like the original definition. The novelty comes from the added requirement: considering the same initial state, any predicate stable under r that c is able to establish at the end of its execution in an environment z must also be established by **rely** $(r,\, z) \cdot c$ at the end of its execution in an environment $z \vee r$.

We believe this formulation is a potential candidate to eliminate the controversial lemma 3.83 from the proof of law 3.84. The adjustment in the definition of rely has to be accompanied by the introduction of properties about *wp*. In this direction we further contribute with two laws about *wp*, that highlight some essential properties of this operator.

Law (contrib.) 3.120 (WP-Monotonic)**.** *For any predicates q, q_0 and q_1, relations r, r_0 and r_1, and commands c and d,*

$$c \sqsubseteq [r]\ d \implies \vdash wp(c,\, r,\, q) \Rightarrow wp(d,\, r,\, q) \tag{3.120a}$$

$$\vdash r_0 \Rightarrow r_1 \vee idrel \implies \vdash wp(c,\, r_1,\, q) \Rightarrow wp(c,\, r_0,\, q) \tag{3.120b}$$

$$\vdash q_0 \Rightarrow q_1 \implies \vdash wp(c,\, r,\, q_0) \Rightarrow wp(c,\, r,\, q_1) \tag{3.120c}$$

Law (contrib.) 3.121 (WP-Transference-CQ)**.** *For relation r, predicate q and command c,*

$$wp(c,\, r,\, q) = wp(c\ ;\ \{q\},\, r,\, true)$$

The generalisation of *stops* proposed in Definition 3.118 is just one out of five definitions that were investigated to replace *stops*. For the alternative definitions, we refer the reader to theory *12-Extensions* in Appendix A.1. The reason we have not investigated the impacts of the proposed definition for the rely command is because of time constraints. To carry out the investigation of the adequacy of a new definition for the rely command, we have to revisit all proofs involving this concept and also to build a consistent infrastructure to support the use of *wp*. The distributivity of the rely command over a sequential composition is a well-known property in the literature of rely guarantee (e.g. [26, 43]). Considering the definition of the

rely command as a "black-box", there should be no doubts that law 3.84 must hold. The challenge is, of course, to demonstrate that this property holds in this algebraic incarnation of rely-guarantee and, if it does not hold, to fine-tune the definition of the rely command to make this property to hold.

Chapter 4

Rely-guarantee in Isabelle/HOL

> The separation of practical and theoretical work is artificial and injurious. Much of the practical work done in computing, both in software and in hardware design, is unsound and clumsy because the people who do it have not any clear understanding of the fundamental design principles of their work. Most of the abstract mathematical and theoretical work is sterile because it has no point of contact with real computing. [...] This separation cannot happen.
>
> Christopher Strachey

This chapter discusses the infrastructure developed to encode the refinement calculus from Chapter 3 as a refinement algebra in Isabelle/HOL. It explains conventions adopted to keep the mechanisation manageable, the proof style used to formalise derivations, and decisions we took to enhance the level of automation of the mechanisation. These decisions are practical contributions of our encoding: without careful design and introduction of redundancy in specific points of the mechanisation, the theory would have become quite clumsy to use to derive the examples from Chapter 6, because it would offer little support from Isabelle to automatically complete proofs.

A key contribution of this chapter is that it concludes the discussion started in Chapter 2 about the encoding of RG-WSL. Recall from Section 2.3.4 that our encoding of RG-WSL differs from that proposed in [48], in the sense that the argument taken by unbounded choice is required to be a countable set of commands, instead of an arbitrary set of commands. We analyse alternative choices for encoding RG-WSL, and compare these to our encoding.

This chapter also discusses a few incorrect paper proofs from [48], presenting a sketch of the mechanised proof in these cases. The chapter ends with a brief parallel between relations and lattices.

4.1 Methodology

Our objective is to take the algebraic approach to encode rely-guarantee refinement calculus as presented in Chapter 3 using Isabelle/HOL [109]. As a consequence of this decision, our mechanisation departs from a set of laws taken for granted, which is a reduced set of laws sufficiently expressive to derive other laws. In this chapter we refer to laws taken for granted as *local assumptions*. In doing so, we differ from the usual nomenclature adopted in the literature [54, 43], which employs the term *axioms* to refer to properties introduced without a proof. The term "axiom" is not technically honest to our mechanisation: we do not use axiomatisation within Isabelle, but locales to create local contexts where properties without an underlying proof can be introduced.

We represent programs via *datatypes*, i.e. syntactic entities, thus the algebraic laws express semantic properties over syntactic terms. The syntax is necessary to formalise the operational semantics discussed in Chapter 2. We use the operational semantics to prove the soundness of about a dozen refinement laws via stratified forward simulation. Recall from Figure 2.7 on page 60 that forward simulation is not a general proof method for refinement: it is possible for a concrete program to refine an abstract one, even though the abstract does not simulate the concrete. The general strategy to prove the soundness of a refinement law is to expand the definition of refinement (Definition 2.73 on page 55) and reason in terms of a denotational semantics. Our mechanisation can be extended to support this proof strategy by adding a proof-oriented denotational semantics to the theory. The mechanisation provided in Appendix A.1 abstracts the denotational semantics as a constant. Thus, it does not contain enough details to prove the soundness of the local assumptions.

4.1.1 Naming conventions

To organise the refinement laws in Isabelle, we developed a naming convention to help the user to search laws by the key commands or operators involved, and count the number of theorems of a certain type. The most frequent elements of our convention are summarised in Table 4.2, which maps infixes to their meanings. These infixes are included in the label of the laws, but have no semantics within Isabelle.

Additionally, we use Isabelle predefined attributes to enhance levels of automation. The following attributes can be enclosed within square brackets and added after the label of a law. These attributes affect automated proof commands such as *safe*, *simp* and *auto*, and

Term	**Meaning**
Ref	main symbol in the conclusion is refinement
Tr	main symbol in the conclusion is trace equality
Pre	precondition command
Spec	specification command
Dist	distributivity property
Mono	monotonic property
Trans	transitive property
Assoc	associative property
Comm	commutative property
Idem	idempotent property
Ident	identitity
ParC	parallel composition
SeqC	sequential composition
SConj	strict conjunction
UCh	unbounded choice
Iter	iteration
Guar	guarantee
Rely	rely
Term	termination

Table 4.2 Naming convention

thus are used at discretion[1]. Attributes can only be added to a law if the law matches the characteristic shape required by the attribute.

simp can be applied to laws whose main symbol is an equality. As general advice, the left-hand side of a simplification rule should be more complex than its right-hand side in some sense (*e.g.*, number of unique function symbols). Upon invocation of simplification tactics such as *simp* and *auto*, Isabelle tries pattern match the left-hand side of each simplification rule against the goal, and when it succeeds, it rewrites it by the respective right-hand side of the successful rule.

intro! automatically used by the safe proof method. This attribute should be applied only to introduction laws that preserve provability, i.e., that trade a provable goal by other that is not stronger than it.

Great care needs to be taken, however, as unrestricted use of attributes might render Isabelle tools to loop (i.e., rewrites must adhere to some notion of simplification to prevent one rewriting undoing the previous). This is a hard task that is achieved heuristically and through experience. It sets the difference between easy/hard proof. It may also bias the rewriting direction towards specific choices. All these are key proof engineering techniques to make theories usable and productive. Without such fine tuning, our mechanisation would be very hard to use.

4.1.2 Encoding lemmas

Lemmas from Chapters 2 and 3 are taken as local assumptions to form the refinement algebra discussed in this chapter. We group the local assumptions based on their subject: basic laws, associativity, commutativity, distributivity, monotonicity, etc. These groups resemble the structure of the previous chapter. Each group is encoded using a locale and inherits definitions given in previously defined locales. At the base of the hierarchy of local assumptions is the locale Trace_Semantics (Figure 4.1), which introduces key concepts about the semantics, such as the abstract notion of denotational semantics (*trace-program*). The concept of refinement is introduced in the locale Algebra_Core (Figure 4.2), which

[1] *Safe* splits a sub-goal into simpler sub-goals and break the premises into independent pieces of information. This transformation is expected to preserve the provability of a conjecture. *Simp* performs simple arithmetic and logical simplifications, and is generally fine-tuned by listing the laws to be applied. Simplification rules are applied by default both by *simp* and *auto*. This last command performs a more aggressive transformation on the goal and may transform an originally provable goal into one whose provability is not guaranteed.

introduces refinement and trace equality, and also introduces most of the lemmas discussed in Section 3.3 (Basic refinement laws)[2]. The encoding of assumptions is available in the theory *05-RG-Algebra* in Appendix A.

locale *Trace-Semantics* =
fixes *trace-program* :: *Command* ⇒ *Traces* ([⟦-⟧]) **and**
trace-semantics :: *Command* ⇒ *relation* ⇒ *Traces* ([⟦-⟧[-])
defines $⟦c⟧[r] \equiv \{ t.\ t \in ⟦c⟧ \wedge \vdash env\ t \Rightarrow_r (r \vee_r idrel)\}$
assumes $⟦(\bigsqcap acset\ C)⟧[r] = \bigcup\{ ⟦c⟧[r] \mid c\ .\ c \in C\}$

Figure 4.1 Trace Semantics

locale *Algebra-Core* = *Trace-Semantics* +
fixes *refinement* :: *Command* ⇒ *relation* ⇒ *Command* ⇒ 𝔹 ((- ⊑[-]/ -) [51,51,51] 50)
and *trace-eq* :: *Command* ⇒ *relation* ⇒ *Command*⇒ 𝔹 ((- ∼[-]/ -) [51,51,51] 50)
defines
$c \sqsubseteq[r]\ d \equiv ⟦d⟧[r] \subseteq ⟦c⟧[true]$ **and**
$c \sim[r]\ d \equiv c \sqsubseteq[r]\ d \wedge d \sqsubseteq[r]\ c$
assumes
— Semantics and Forward Simulation
$\forall n.\ a \preceq[n,r]\ c \Longrightarrow a \sqsubseteq[r]\ c$ **and**
— Basic refinement laws
$[defined\ b, ⟦b⟧_r \wedge_r idrel] \sqsubseteq[true]\ [[b]]$ **and**
$pred\ p \Longrightarrow [p, idrel \wedge_r p`] \sqsubseteq[true]\ skip$ **and**
$pred\ p \Longrightarrow [p, q] \sqsubseteq[true]\ \langle p, q\rangle$ **and**
$pred\ p \Longrightarrow ([p,q] \sqsubseteq[idrel]\ c) = ([p,q] \sqsubseteq[true]\ c)$ **and**
$⟦pred\ p_0; pred\ p_1; \vdash p_0 \Rightarrow_r p_1⟧ \Longrightarrow \{p_0\} \sqsubseteq[true]\ \{p_1\}$ **and**
$⟦pred\ p_0; pred\ p_1⟧ \Longrightarrow \{p_0\}\ ;_c\ \{p_1\} \sim[true]\ \{p_0 \wedge_r p_1\}$ **and**
$⟦pred\ p; pred\ mid⟧ \Longrightarrow [p, q_0\ ;_r\ q_1] \sqsubseteq[true]\ [p, q_0 \wedge_r mid`]\ ;_c\ [mid, q_1]$ **and**
$⟦\ pred\ p_0; pred\ p_1; \vdash p_0 \Rightarrow_r p_1\ ; \vdash p_0 \wedge_r q_1 \Rightarrow_r q_0\ ⟧ \Longrightarrow \langle p_0, q_0\rangle \sqsubseteq[true]\ \langle p_1, q_1\rangle$ **and**
$⟦\ pred\ p_0; pred\ p_1; \vdash p_0 \Rightarrow_r p_1\ ; \vdash p_0 \wedge_r q_1 \Rightarrow_r q_0\ ⟧ \Longrightarrow [p_0, q_0] \sqsubseteq[true]\ [p_1, q_1]$

Figure 4.2 Algebra Core

The encoding of assumptions via locales provides a modular representation where sets of assumptions can be plugged together to form a larger proof context. Using this approach, we cannot refer to definitions and assumptions of a locale unless we are within its context. To be

[2] Isabelle adopts mixfix annotations to define the syntax of operators together with their precedence. In Figure 4.2, the syntax _ ⊑[_] _ is assigned to *refinement*, whose priority is *50*. This choice forces the user to put parenthesis around equalities involving this symbol, because equality has the same precedence. The numbers within square brackets are used to declare the priority of each argument in isolation, and are used to fine-tune the use of parenthesis to solve ambiguity between operators. These numbers are set-up generally experimentally.

in such context the user has two options: either she must interpret the locale with respect to the top-level proof context, or expand the context of the locale. For the first approach, the user has to discharge all assumptions of the locale first. This lifts the laws and definitions that are originally confined to the scope of the locale to become available in the context under which the locale was interpreted. To follow this approach we need a model of the trace semantics suitable for mechanical proofs, as many of the lemmas from Chapter 3 can only be argued using a denotational semantics. Since we do not have such a model, we took a pragmatic approach in the mechanisation: we prove laws by expanding the context of locales. For that, we use the reserved word **context** followed by the name of the locale and a block defined via **begin/end** to introduce additional definitions or prove laws within the context of a specific locale. Thus, derived laws are correct modulo the validity of the local assumptions. To minimise the reliance on local assumptions, we used stratified forward simulation to prove about a dozen lemmas discussed in Chapter 3.

4.1.3 Proof style

The proof style we adopted is called *procedural* and provides a precise control of the proof engine, but it does not show the state of the proof in the proof script, thus making our proofs difficult to be followed outside Isabelle. The proof state can be seen in Isabelle by selecting either the *state* tab or *output* tabs, and then moving the input cursor through the lines of the proof script. We illustrate the procedural proof style in Figure 4.3. For convenience, in this example we show the proof state after the application of key laws, and we systematically replace the actual identifier of the laws by their respective names in this thesis.

Alternatively, Isabelle admits a *structured* proof style via *Isar* [109], which makes proofs suitable for reading outside Isabelle. See an example of structured proof in Figure 4.4. For convenience, we systematically replace the actual identifier of the laws by their respective names in this thesis. The decision for mostly using the procedural style is due to the familiarity of the author with this proof style, and the fact that for large proof exercises such as in our work, the procedural style is more productive. This has already been observed (and followed) by other large developments at the repository of formal proofs[3] and seL4 verified microkernel [69].

Many of the proofs in this document can be automatically found by *sledgehammer*, an Isabelle's proof finder. For certain cases where sledgehammer can find a proof, it can also provide a structured proof. The ability to use sledgehammer to find proof hints at an adequate

[3] http://afp.sourceforge.net

theorem *Conjunction-Atomic-Procedural-Subproof*:
$(magic \sqcap (\langle g_1\rangle \;;_c\; \langle g_1\rangle^\omega \Cap \langle g_0\rangle \;;_c\; \langle g_0\rangle^\omega)) \sim (magic \sqcap \langle g_0 \wedge_r g_1\rangle ;_c (\langle g_0\rangle^\omega \Cap \langle g_1\rangle^\omega))$
apply (*rule Substitution (4.8c)* , *simp*)
— Proof state: $\langle g_1\rangle \;;\; \langle g_1\rangle^\omega \Cap \langle g_0\rangle \;;\; \langle g_0\rangle^\omega \sim \langle g_0 \wedge g_1\rangle \;;\; (\langle g_0\rangle^\omega \Cap \langle g_1\rangle^\omega)$
apply (*rule Transitivity-Trace (4.6b)*)
apply (*rule Assoc-Comm-Dist (3.26d)*)
— Proof state: $\langle g_0\rangle \;;\; \langle g_0\rangle^\omega \Cap \langle g_1\rangle \;;\; \langle g_1\rangle^\omega \sim \langle g_0 \wedge g_1\rangle \;;\; (\langle g_0\rangle^\omega \Cap \langle g_1\rangle^\omega)$
apply (*rule Transitivity-Trace (4.6b)*)
apply (*rule Conjunction-Properties (3.54c)* , *simp+*)
— Proof state: $(\langle g_0\rangle \Cap \langle g_1\rangle) \;;\; (\langle g_0\rangle^\omega \Cap \langle g_1\rangle^\omega) \sim \langle g_0 \wedge g_1\rangle \;;\; (\langle g_0\rangle^\omega \Cap \langle g_1\rangle^\omega)$
apply (*rule Transitivity-Trace (4.6b)*)
apply (*rule Substitution (4.8a)*)
by (*rule Conjunction-Properties (3.54b)* , *simp*)

Figure 4.3 Procedural proof example.

theorem *Conjunction-Atomic-Structured-Subproof*:
$(magic \sqcap (\langle g_1\rangle \;;_c\; \langle g_1\rangle^\omega \Cap \langle g_0\rangle \;;_c\; \langle g_0\rangle^\omega)) \sim (magic \sqcap \langle g_0 \wedge_r g_1\rangle ;_c (\langle g_0\rangle^\omega \Cap \langle g_1\rangle^\omega))$
proof −
have $\langle g_0\rangle \Cap \langle g_1\rangle \sim \langle g_0 \wedge_r g_1\rangle$
by (*metis Relation-Properties (4.1h) Conjunction-Properties (3.54b)*)
hence $\langle g_0\rangle \;;_c\; \langle g_0\rangle^\omega \Cap \langle g_1\rangle \;;_c\; \langle g_1\rangle^\omega \sim \langle g_0 \wedge_r g_1\rangle ;_c (\langle g_0\rangle^\omega \Cap \langle g_1\rangle^\omega)$
using *Conjunction-Properties (3.54c) Substitution (4.8a) Transitivity-Trace (4.6b) Predicate (2.19) Typographic conventions (2.12)* **by** *metis*
hence $\langle g_1\rangle \;;_c\; \langle g_1\rangle^\omega \Cap \langle g_0\rangle \;;_c\; \langle g_0\rangle^\omega \sim \langle g_0 \wedge_r g_1\rangle ;_c (\langle g_0\rangle^\omega \Cap \langle g_1\rangle^\omega)$
using *Assoc-Comm-Dist (3.26d) Transitivity-Trace (4.6b)* **by** *blast*
thus
$(magic \sqcap (\langle g_1\rangle \;;_c\; \langle g_1\rangle^\omega \Cap \langle g_0\rangle \;;_c\; \langle g_0\rangle^\omega)) \sim (magic \sqcap \langle g_0 \wedge_r g_1\rangle ;_c (\langle g_0\rangle^\omega \Cap \langle g_1\rangle^\omega))$
using *Substitution (4.8c)* **by** *simp*
qed

Figure 4.4 Structured proof example.

fine tuning of theory attributes, lemma shape and granularity. All this effort is crucial proof engineering in order to make theories usable and as highly automated as possible.

4.2 Proof engineering

This section presents part of the setup necessary to enable modular proofs in Isabelle/HOL. The laws in this section serve to organize the mechanical reasoning into layers, so that algebraic reasoning about high-level commands is possible without expanding their definition. This section also introduces a mechanical formulation of the single reference property, which is necessary to formalise side conditions of laws that introduce control structures in programs.

4.2.1 Relational operators

Most refinement laws generate proof obligations when applied. For example, the application of law 3.84 (Distribute-Rely-Sequential) to distribute the rely over a sequential composition of commands adds two new goals to the proof state: *pred p* and $\vdash p \Rightarrow \mathit{stops}(c_0 \,;\, c_1, z)$. More often than not, the proof obligations involve reasoning over logical interpretation of relations ($\vdash$). We provide a collection of laws to algebraically reason about relations and their logical interpretation. To a certain extent, these laws prevent the need for expanding definitions given in Figure 2.4 on page 34. The next set of laws illustrates some of the properties that are provided to reason about relations, post-state notation, implication, conjunction and closure of relations. For a complete list of these property, we refer the reader to theory *02-Relations* in Appendix A.1.

Law 4.1 (Relation-Properties)**.** *For any sets of variables X and Y, relation g, q, r and s, and predicates p, p_0 and p_1,*

$$\mathit{idrel} \vee \mathit{idset}\, X = \mathit{idset}\, X \tag{4.1a}$$

$$\mathit{idset}\, X \wedge \mathit{idset}\, Y = \mathit{idset}\, (X \cup Y) \tag{4.1b}$$

$$(g \vee \mathit{idrel})^{**} = g^{**} \tag{4.1c}$$

$$\mathit{idrel} \vee (g \,;\, g^{**}) = g^{**} \tag{4.1d}$$

$$(p_0 \wedge p_1)\text{`} = p_0\text{`} \wedge p_1\text{`} \tag{4.1e}$$

$$r \Rightarrow q \Rightarrow s = q \wedge r \Rightarrow s \tag{4.1f}$$

$$p \Rightarrow p\text{`} = (p \Rightarrow p\text{`})^{**} \tag{4.1g}$$

$$\mathit{true} \wedge q = q \tag{4.1h}$$

$$q \wedge \mathit{true} = q \tag{4.1i}$$
$$q \Rightarrow r \Rightarrow q = \mathit{true} \tag{4.1j}$$
$$q \Rightarrow r \vee q = \mathit{true} \tag{4.1k}$$
$$q \wedge r \Rightarrow q = \mathit{true} \tag{4.1l}$$
$$q \wedge r \Rightarrow r = \mathit{true} \tag{4.1m}$$

While reasoning over logical evaluation ($\vdash$), it is generally useful to decompose proofs into sub-goals. For example, the next law is often used to split a proof obligation in the shape $\vdash r \Rightarrow q \wedge s$, and also to reason monotonically using logical interpretation.

Law 4.2 (Log-Interp-Imp-Monotonic)**.** *For any relations r, q and s, and sets of variables X and Y,*

$$\vdash r \Rightarrow q \wedge \vdash r \Rightarrow s \implies \vdash r \Rightarrow q \wedge s \tag{4.2a}$$
$$\vdash q \Rightarrow q' \wedge \vdash r \Rightarrow r' \implies \vdash q \wedge r \Rightarrow q' \wedge r' \tag{4.2b}$$
$$\vdash r_0 \Rightarrow {r_0}' \wedge \vdash r_1 \Rightarrow {r_1}' \implies \vdash r_0 \,;\, r_1 \Rightarrow {r_0}' \,;\, {r_1}' \tag{4.2c}$$
$$Y \subseteq X \implies \vdash \mathit{idset}\ X \Rightarrow \mathit{idset}\ Y \tag{4.2d}$$

We also offer laws to decompose the proof that a relation is a predicate. For example, the next laws are useful to show that relations formed from conjunction or disjunction of simpler predicates are still a predicate

Law 4.3 (Monotonic-Predicate)**.** *For any relations p_0 and p_1,*

$$\mathit{pred}\ p_0 \wedge \mathit{pred}\ p_1 \implies \mathit{pred}\ (p_0 \wedge p_1)$$
$$\mathit{pred}\ p_0 \wedge \mathit{pred}\ p_1 \implies \mathit{pred}\ (p_0 \vee p_1)$$

Laws to manipulate relations can be introduced on-demand while proving refinement laws. Thus, the set of laws to manipulate relations is deliberately incomplete. This is easy to notice in proofs involving relations that have quantifiers. In general, to reason over such relations one has to expand logical evaluation and also the definitions given in Figure 2.4. In practice, the user can always expand the definitions and apply simplification tactics such as *simp* and *auto*. If the algebraic style is preferred, the user can extend the set of laws to manipulate relations on-demand.

To enhance the automation of proofs involving relations, redundant laws can be provided and attributes such as simp can be assigned to the laws. In our experience, attributes were added in a late stage of the mechanisation, in an experimental manner, i.e. analysing the effect of adding attributes on a case-by-case basis. The next guideline was observed when introducing laws to enhance automation, and assigning the attributes to them. Some of these observations are also noted in [38].

1. Avoid marking commutative and associative laws with the simp attribute, because both left and right hand side are equally complex;

2. Offer some degree of redundancy in laws. Human interaction during proofs can be counter-balanced by introducing redundant laws. For example, we offer laws 4.1h and 4.1i to minimise the manual application of commutative laws;

3. Create simplification laws to absorb tautologies. Laws such as laws 4.1j and 4.1k speed up proofs, as they prevent the need for expanding logical interpretation ($\vdash$);

4. Create introduction rules to trade complex for simpler goals. Law 4.2a is a perfect example of this type of strategy. Beyond increasing the modularity of proof scripts, these laws can be used to increase automation by tagging laws that preserve provability with the attribute intro!.

5. Create zooming rules to jump between operators' level of abstraction. We used zooming rules to prevent the expansion of the concept of stability (Definition 3.16 on page 72). Law 3.17 (Closure-Interference) is a good representative of zooming rules.

4.2.2 Single reference property

The single reference property holds for an expression e and a relation r if e has at most a single reference to a variable modified by r [48]. The concept is discussed in Section in 3.2.1, but is not given a formal definition there.

For the mechanisation of laws 3.100 (Rely-Test), 3.111 (Rely-Conditional), 3.112 (Rely-Loop), and 3.115 Assignment-Single-Reference), which require the concept of single reference, we are forced to come up with a mathematical representation. The proposed formalisation requires the user to identify the unstable variable x if such a variable exists in an expression e, and then to prove that all remaining free variables are stable with respect to the relation r and that the unstable variable happens at most once.

Definition 4.4 (Single reference property)**.** *Let b an expression and r a relation,*

$$SRF\ (b, r)\ \equiv\ \begin{pmatrix} \vdash r \Rightarrow idset\ (free\text{-}exp\ b) \vee \\ \exists x.\ count(x, b) \leq 1 \wedge \vdash r \Rightarrow idset\ (free\text{-}exp\ b - \{x\}) \end{pmatrix}$$

where count is defined by induction on the structure of Exp as

$$\begin{aligned} count(v, N\,n) &\equiv 0 \\ count(v, V\,x) &\equiv \mathit{if}\ v = x\ \mathit{then}\ 1\ \mathit{else}\ 0 \\ count(v, UOp\ uop\ e) &\equiv count(v, e) \\ count(v, BOp\ bop\ e_1\ e_2) &\equiv count(v, e_1) + count(v, e_2) \end{aligned}$$

To illustrate the application of the concept we discharge a proof obligation that arises in the refinement step $\mathscr{R}_{20}$ in the derivation of a concurrent version of FINDP on page 192. We use the symbol $\Leftarrow$ to indicate that we are applying a weakening rule, that is, a law that matches a certain conclusion, and trades it by another one that implies the original. Next, *ok*, *ek*, *ot* and *v* denote program variables, *b* is boolean expression and *r* is relation. Let $b = (oc < ot \wedge oc < et)$ and $r = ((et' \leq et) \wedge idset\ \{ot, oc, v\})$. The proof obligation require us to prove $SRF\ (b, r)$.

$$\begin{array}{ll} & SRF\ (b, r) \\ = & \text{by Definition 4.4} \\ & \vdash r \Rightarrow idset\ (free\text{-}exp\ b) \vee \\ & \exists x.\ count(x, b) \leq 1 \wedge \vdash r \Rightarrow idset\ (free\text{-}exp\ b - \{x\}) \\ \Leftarrow & \text{by logical simplification (introduce disjunction)} \\ & \exists x.\ count(x, b) \leq 1 \wedge \vdash r \Rightarrow idset\ (free\text{-}exp\ b - \{x\}) \\ \Leftarrow & \text{by logical simplification (introduce existential quantifier)} \\ & count(et, b) \leq 1 \wedge \vdash r \Rightarrow idset\ \{ot, ok\} \end{array}$$

Any expression respects the single reference property with respect to the relation *idrel*. This happens because $idrel = idset\ UNIV$, thus the first component of the disjunction in the definition of *SRF* always evaluate to *true*.

4.2.3 Monotonicity and substitution

Substitution laws smooth the proof process by encoding the intuition of substitutions into the theory of refinement. Typical usage of substitution laws occur when one has a refinement proof to complete and wants to replace one of the terms at one side by other term that is semantically equivalent. Using substitution laws this is encoded via laws 4.5a and 4.5b. For example, using 4.5a, the user has to prove that program a is equivalent to c in the context r, and also show that $a \sqsubseteq[r]\ d$ to complete the original proof that $c \sqsubseteq[r]\ d$.

Law 4.5 (Substitution-Refinement)**.** *For any commands a, c and d, and relation r,*

$$a \sim[r]\ c \wedge a \sqsubseteq[r]\ d \implies c \sqsubseteq[r]\ d \tag{4.5a}$$

$$c \sim[r]\ d \wedge a \sqsubseteq[r]\ c \implies a \sqsubseteq[r]\ d \tag{4.5b}$$

Trace equality is reflexive, transitive and symmetric. These are examples of laws where the attributes simp and sym are added to the label of the law.

Law 4.6 (Transitivity-Trace)**.** *For any commands a, c and d, and relation r,*

$$c \sim c \tag{4.6a}$$

$$a \sim[r]\ c \wedge c \sim[r]\ d \implies a \sim[r]\ d \tag{4.6b}$$

One of the benefits of defining derived commands in terms of the primitive ones is that some properties enjoyed by the primitive commands, e.g. monotonicity for strict conjunction ($\Cap$), are also enjoyed by derived commands, e.g. the guarantee command (**guar** $g \cdot c$). To prevent the need for expanding derived commands in proofs to apply these properties, we have to add extra laws in the mechanisation to make explicit that the monotonic properties also hold for the derived commands. In general, properties of monotonicity are overlooked in the description of rely-guarantee refinement calculus [48], but are essential to allow modular development of programs, such as those presented in Chapter 6. Some top-level monotonic laws are shown next.

Law 4.7 (Monotonic-Derived)**.** *For any relations z and r, predicate p, boolean expression e, sets of variables W and Y, and commands a, b, c and d,*

$$c \sqsubseteq[r]\ d \implies \textbf{while}\ e\ \textbf{do}\ \ c \sqsubseteq[r]\ \textbf{while}\ e\ \textbf{do}\ \ d \tag{4.7a}$$

$$a \sqsubseteq[r]\ b \wedge c \sqsubseteq[r]\ d \implies (\textbf{if}\ e\ \ \textbf{then}\ a\ \ \textbf{else}\ c) \sqsubseteq[r]\ (\textbf{if}\ e\ \ \textbf{then}\ b\ \ \textbf{else}\ d) \tag{4.7b}$$

$$W \subseteq Y \wedge c \sqsubseteq[r]\ d \implies Y{:}\ c \sqsubseteq[r]\ W{:}\ d \tag{4.7c}$$

The next law illustrates some cases of substitution of terms via trace equality. In most cases the application of this law is preceded by the application of law 4.5 to substitute the left or right-hand side of a refinement proof, or the application of law 4.6b to prove trace equivalence by transitivity.

Law 4.8 (Substitution)**.** *For any relation r, variable x, set of variables Y, and commands a, b, term c and d,*

$$a \sim[r]\ b \implies a\ ;\ c \sim[r]\ b\ ;\ c \tag{4.8a}$$

$$c \sim[r]\ d \implies a\ ;\ c \sim[r]\ a\ ;\ d \tag{4.8b}$$

$$a \sim[r]\ b \wedge c \sim[r]\ d \implies a \sqcap c \sim[r]\ b \sqcap d \tag{4.8c}$$

$$a \sim b \wedge c \sim d \implies a \parallel c \sim b \parallel d \tag{4.8d}$$

$$c \sim[r]\ d \implies \mathbf{guar}\ g \cdot c \sim[r]\ \mathbf{guar}\ g \cdot d \tag{4.8e}$$

$$a \sim[z]\ b \implies \mathbf{rely}\ (r, z) \cdot a \sim \mathbf{rely}\ (r, z) \cdot b \tag{4.8f}$$

4.2.4 Shortening proofs

The next laws are used to shorten proofs by preventing the expansion of derived commands.

Law 4.9 (Trade-Spec-Frame)**.** *For any predicate p, relation r and set of variables X and Y, such that* $Y \subseteq \overline{X}$*,*

$$X\colon [p, q]\ \sim\ X\colon [p, \mathit{idset}\ Y \wedge q]$$

Law 4.10 (Skip-Iteration)**.** *For any command c,*

$$c^\star\ \sqsubseteq\ \mathit{skip}$$

Law 4.11 (Dist-Guarantee-Var)**.** *For any variable x, program c and relation g, such that* $\vdash$ *depends-only* $(g, \overline{\{x\}})$*,*

$$\mathbf{guar}\ g \cdot \mathbf{var}\ x \cdot c\ \sim\ \mathbf{var}\ x \cdot (\mathbf{guar}\ g \cdot c)$$

Law 4.12 (Trade-Spec-Guarantee)**.** *For any predicate p and relations g and q,*

$$\textbf{guar}\ g \cdot [p, g^{**} \wedge q] \ \sim\ \textbf{guar}\ g \cdot [p, q]$$

Law 4.13 (Distribute-Frame-Sequential)**.** *For any set of variables Y, Y_1 and Y_2, such that $Y_1 \subseteq Y$ and $Y_2 \subseteq Y$, and commands c, c′, d and d′, and relation r such that $c \sqsubseteq[r] c'$ and $d \sqsubseteq[r] d'$,*

$$Y\colon (c\ ;\ d) \sqsubseteq[r]\ Y_1\colon c'\ ;\ Y_2\colon d'$$

Law 4.14 (Omega-Star-Atomic)**.** *For any relations r, rx and ry, such that $\vdash rx \wedge ry \Rightarrow r$,*

$$\langle r\rangle^\star \sqsubseteq \langle rx\rangle^\omega \Cap \langle ry\rangle^\star$$

4.2.5 Representation issues

Isabelle represents set comprehension via the *Collect* constructor. This representation makes proofs involving sets sometimes cumbersome, and required us to create additional laws, such as 4.15a,4.15b and 4.15c to handle internal representation issues in Isabelle. Also, the conversion between the derived command $\sqcap$ (Definition 3.6 on page 68) and the primitive command $\bigsqcap$ sometimes require us to handle internal representation issues of Isabelle. The next laws increase the ability of **sledgehammer** (Isabelle's automatic proof finder) to succeed when trying to prove a goal that involves non-deterministic choice.

Law 4.15 (Representation-Nondeterministic-Choice)**.** *For any programs c_1, c_2 and d,*

$$\{c_1 \parallel d \mid P\ c_1\} \ =\ \{x \parallel d \mid x \in \{x \mid P\ x\}\} \tag{4.15a}$$

$$\{c_1 \parallel d, c_2 \parallel d\} \ =\ \{c \parallel d \mid c \in \{c_1, c_2\}\} \tag{4.15b}$$

$$\{c_1\ ;\ d, c_2\ ;\ d\} \ =\ \{c\ ;\ d \mid c \in \{c_1, c_2\}\} \tag{4.15c}$$

4.3 Conflicts between semantics and local assumptions

RG-WSL as presented in Figure 2.1 on page 21 cannot be formalised in Isabelle/HOL. Recall from Section 2.3.4 on page 28 that the problem amounts to the use of the type constructor *set*

in the constructor of unbounded choice ($\bigsqcap$). To encode RG-WSL, we have restricted the argument of unbounded choice to be a countable set of commands. This section discusses the divergence between the operational semantics and the mechanised laws pointed at the end of Section 2.3.4.

To better understand the issue, we turn our attention to the operational semantics for unbounded choice (law 2.46 on page 47). The premise of this rule requires C to be a countable set. This assumption is strictly necessary to allow one to infer that *acset C* denotes a countable set. The constructor *acset* is defined in the theory *Countable-Set-Type*, which is distributed with Isabelle/HOL. It takes a set of a type α and returns a representation of this set using the type α *cset*. The success of the application of *acset* depends on its argument to be a countable set.

To encode the local assumptions of the algebra, we have the option of dropping the requirement on the set taken by unbounded choice to be countable. On one hand, if we drop this restriction, we introduce a discrepancy between the local assumptions and the operational semantics of the language. On the other hand, if we enforce this restriction via the introduction of provisos in the local assumptions, the derivation of laws involving the rely command from local assumptions is compromised, because it would demand us to prove that the set used to define the rely command (Definition 3.72 on page 94) is countable, but this set is not countable (see proof on Appendix A.4.3).

The position taken in this thesis is that of dropping the restriction on countable sets for the local assumptions of the algebra. Before justifying this design decision in Section 4.3.3, we discuss two alternative encodings for RG-WSL in Isabelle/HOL.

4.3.1 Semantic encoding

The conflict between semantics and local assumptions can be prevented by eliminating the syntax and replacing the encoding of RG-WSL using datatypes by an encoding where each programming constructor is a function from their arguments to an abstract type $'a$, as done in [35]. In this formulation, a possible encoding for RG-WSL using locales is shown in Figure 4.5.

This formulation does not restrict the argument taken by unbounded choice to be countable. It circumvents this restriction by avoiding a syntactic representation of RG-WSL. The drawback of this approach is that the lack of syntax means that one has to give up on the operational semantics and the laws that establish a foundation for the algebra in terms of the semantics provided in Chapter 2.

locale *RG-WSL* =
fixes
Atomic :: *relation* ⇒ *relation* ⇒ *'a* **and**
Post :: *relation* ⇒ *'a* **and**
Pre :: *relation* ⇒ *'a* **and**
UCh :: *'a set* ⇒ *'a* **and**
SConj :: *'a* ⇒ *'a* ⇒ *'a* **and**
SeqComp :: *'a* ⇒ *'a* ⇒ *'a* **and**
ParComp :: *'a* ⇒ *'a* ⇒ *'a* **and**
Conditional :: *Exp* ⇒ *'a* **and**
Uses :: *vname set* ⇒ *'a* **and**
State :: *vname* ⇒ *vvalue* ⇒ *'a*

Figure 4.5 Semantic encoding for RG-WSL

4.3.2 Stratification

RG-WSL can be split in two syntactic layers: one containing inner commands that can be used inside of unbounded choice (*ICommand*); and other of outer commands that operate on the same level as unbounded choice (*Command*) as shown in Figure 4.6.

datatype *ICommand* = *iAtomic relation relation* (⟨-,-⟩$_i$)
| *iPost relation* ([-]$_i$)
| *iPre relation* ({-}$_i$)
| *iStrictConj ICommand ICommand* (**infix** ⩓$_i$ *70*)
| *iSeqComp ICommand ICommand* (**infix** ;$_i$ *80*)
| *iParComp ICommand ICommand* (**infix** ‖$_i$ *70*)
| *iConditional Exp* ([[-]]$_i$)
| *iUses vname set ICommand* ((**uses**$_i$ - ·/ -) [*71*,*71*] *70*)
| *iState vname vvalue ICommand* ((**state**$_i$ -↦- ·/ -) [*71*,*71*,*71*] *70*)
datatype *Command* = *DemNonDetChoiceSet ICommand set* ((⨅-) *75*)
| *Atomic relation relation* (⟨-,-⟩)
| *Post relation* ([-])
| *Pre relation* ({-})
| *StrictConj ICommand ICommand* (**infix** ⩓ *70*)
| *SeqComp ICommand ICommand* (**infix** ;$_c$ *80*)
| *ParComp ICommand ICommand* (**infix** ‖ *70*)
| *Conditional Exp* ([[-]])
| *Uses vname set ICommand* ((**uses** - ·/ -) [*71*,*71*] *70*)
| *State vname vvalue ICommand* ((**state** -↦- ·/ -) [*71*,*71*,*71*] *70*)

Figure 4.6 Nesting control for RG-WSL

Using the syntax proposed in Figure 4.6, the operational semantics can be given to both for inner and outer layers. The key insight of this design is that it introduces nesting control for unbounded choice. In the case illustrated in Figure 4.6, no nesting is allowed. One can think of that language as *RG-WSL*$^{(0)}$ in the sense that it allows no nesting of unbounded choice.

Several layers can be devised between the inner and outer language, such that a number of levels of nesting is allowed. For example, one can denote by $\mathit{RG\text{-}WSL}^{(k)}$ a formulation of RG-WSL that allow up to k nested occurrences of unbounded choice. The impact of limiting the nesting non-deterministic choice is that it constrains the nesting of derived commands based on it, such as the rely command, conditionals, local variable and assignment. To illustrate the expressiveness of $\mathit{RG\text{-}WSL}^{(k)}$ we list some programs from RG-WSL and show the minimum value of k to represent them in $\mathit{RG\text{-}WSL}^{(k)}$. Next we assume that $c \in \mathit{RG\text{-}WSL}^{(0)}$,

$$
\begin{array}{ll}
(\textbf{rely}\ (r,\, z) \cdot c) \in \mathit{RG\text{-}WSL}^{(k)}, & \text{for } 1 \leq k \\
(\textbf{rely}\ (r_0,\, z \vee r_1) \cdot (\textbf{rely}\ (r_1,\, z) \cdot c)) \in \mathit{RG\text{-}WSL}^{(k)}, & \text{for } 2 \leq k \\
(\textbf{var}\ m \cdot \textbf{if}\ x < y\ \ \textbf{then}\ m{:=}y\ \ \textbf{else}\ m{:=}x) \in \mathit{RG\text{-}WSL}^{(k)}, & \text{for } 2 \leq k.
\end{array}
$$

The drawback of implementing $\mathit{RG\text{-}WSL}^{(k)}$ for any $k \geq 0$, is that it requires redundancy in the semantics and also in the local assumptions of the algebra, because each layer requires its own laws. Ignoring the tedious aspects involved with the mechanisation of several layers, the stratification scales on-demand. That is, one can systematically add or remove layers as necessary for expressing an intended program.

4.3.3 Justifying design decision

We use the experiment on stratification to draw conclusions about the consistencyof our encoding using countable sets. We conjecture that *any concrete derivation based on the encoding of RG-WSL using countable sets has a corresponding refinement using* $\mathit{RG\text{-}WSL}^{(k)}$ *for some value of* k.

To see why this holds, consider the refinement chain: $C_0 \sqsubseteq[r]\ C_1 \sqsubseteq[r]\ \ldots \sqsubseteq[r]\ C_n$, where each program C_i contains a finite number of nested occurrences of unbounded choice. Let $C_j \in \{C_0, \ldots, C_n\}$ be such that it has the highest number of nested occurrences of unbounded choice among all elements in $\{C_0, \ldots, C_n\}$. Let the number of nested occurrences of unbounded choice in C_j be k. It is possible to systematically convert each $C_i \in \mathit{RG\text{-}WSL}$ into $C_i{}' \in \mathit{RG\text{-}WSL}^{(k)}$, and find a derivation for $C_n{}'$ from $C_0{}'$ using laws over $\mathit{RG\text{-}WSL}^{(k)}$.

Thus, although local assumptions do not include a proviso to ensure that the set taken by non-deterministic choice is countable, we can map each concrete derivation in our algebra to a respective derivation using a stratified version of RG-WSL. This mapping allow us to compensate for the conflict between the semantics and local assumptions.

4.4 Discussion and summary of contributions

In this chapter we discussed the encoding of the refinement calculus presented in Chapter 3 in Isabelle/HOL. We encoded the algebra using a hierarchy of locales, where each locale is used to group local assumptions that are related by a specific subject (*e.g.* monotonicity, commutativity, etc.). This chapter also closes the discussion about the encoding of RG-WSL, and analyses two alternative encodings that could have been used to represent RG-WSL. Next we summarise the contributions of this chapter.

1. Guidelines for enhancing the level of automation of the mechanisation. Section 4.2.1 discussed the rationale behind the use of attributes that enable automatic application of laws in our mechanisation, and the creation of auxiliary laws to increase the efficiency of tools such as *sledgehammer* to find proofs;

2. Discussion and formalisation of single reference property in Section 4.2.2;

3. Additional refinement laws to facilitate proofs. Sections 4.2.3-4.2.5 presented laws that prevent the need for expanding derived commands to apply monotonic and distributive properties;

4. Discussion of the conflict between semantics and local assumptions in the mechanised theory. Section 4.3 revisits the issue involving the formalisation of non-deterministic choice, and analyses the limitations of alternative encodings that prevent the problem.

Although the laws presented in this chapter do not increase the expressiveness of the algebra (they are all derived from local assumptions), they are essential to smooth the path for the application of mechanised theory.

Encoding of relations The level of automation for proofs involving logic interpretation ($\vdash$) can be improved. The current encoding does not limit the expressiveness of the theory, but requires considerable interaction from the user to discharge proof obligations. This can be minimised by: (i) introducing more laws to reason about relations, or (ii) instantiating the type of relations to be a lattice. The second approach appears to be more effective, as it would allows the user to directly benefit from properties already included in the libraries of Isabelle/HOL. In our experiments, we did not succeed to instantiate relations as a lattice, but we proved the next law that hints at this possibility.

Law 4.16 (Lattice-Exchange)**.** *For any relations q and r,*

$$q \wedge r = \mathit{inf}\ q\ r \tag{4.16a}$$
$$q \vee r = \mathit{sup}\ q\ r \tag{4.16b}$$
$$\neg\, r = -\, r \tag{4.16c}$$
$$q \Rightarrow r = \mathit{sup}\ (-\, q)\ r \tag{4.16d}$$
$$\mathit{true} = \mathit{top} \tag{4.16e}$$
$$\mathit{false} = \mathit{bot} \tag{4.16f}$$

Type constraints. Another direction for improving the mechanisation is to introduce a mechanism to automatically discharging proof obligations involving type constraints. Recall from Section 2.4.1 (Predicates on page 34) that we use a single type to encode relations used as preconditions and postconditions. This simplification is compensated by introducing provisos in the local assumptions to ensure that relations used to instantiate preconditions are predicates (see Figure 4.2 for examples). Sometimes the same proviso is proved a number of times in a proof, as it is reintroduced every time a law manipulating a precondition is used. These provisos are trivial to be proved, but perhaps there might be a more effective way of managing them without polluting proofs with their details.

Chapter 5

Extensions to rely-guarantee algebra

> Formalization is an experimental science.
>
> Dana Scott

This chapter extends the rely-guarantee algebra with indexed parallel composition, assignment to indexed arrays and two abstractions: *eguard* and reachable values. The primitive **eguard** *r* delivers a mechanism to restrict environment steps of a program; this command is necessary to complete the proof of law 3.102 (Introduce-Variable-Rely on page 108). Reachable evaluations provide an over-approximation for the use of history variables, and allows the designer to refer to non-atomic evaluation of expressions within postconditions. Reachable evaluation is inspired in the convention known as *possible values*, which was first discussed in [66], where there is a need of modelling the transference of value between a global variable *x* and another variable *read-x*, and the value acquired by *read-x* may be the initial or final value or *x*, as well any intermediate value that *x* may have held as result of interference prior the transference.

5.1 Indexed parallelism

We define a generalisation of the binary parallel composition to a collection of programs. Before presenting the actual definition of indexed parallelism mechanised in this thesis (Def. 5.3), we discuss two hypothetical definitions and analyse their problems. To formalise these definitions we consider two components: a set of indices S and a parameterised program F, which is not a program itself but a function from indices to programs. The i-th program of the composition is represented by $F\ i$. The first definition is introduced next.

Definition 5.1 (Indexed Parallel Composition)**.** *Let n be a positive natural number, and F a total function from naturals to commands.*

$$\|_S \cdot F = \begin{cases} skip & for\ S = \emptyset, \\ (F\ n) \parallel (\|_{\{1..n-1\}} \cdot F) & for\ S = 1..n. \end{cases}$$

The set of indices S is finite, but its size is arbitrary. The base case, $S = \emptyset$, returns the identity for parallel composition, i.e. *skip*; in the inductive case, $S{=}\{1..n\}$, the n-th program is composed in parallel with the indexed parallel composition of the first $n{-}1$ programs. To illustrate this definition, consider the parameterised program F_{12} where $F_{12}\ (1) = a$ and $F_{12}\ (2) = b$. Expanding the definition of indexed parallelism we can show that $\|_{\{1,2\}} \cdot F_{12} = b \parallel (a \parallel skip)$.

Definition 5.1 requires indices to be positive natural numbers. For certain algorithms, it is more convenient to use a different type to represent indices, e.g. for the derivation of Floyd-Warshall in Section 6.6, programs are indexed according to the position of the cells (i, j) of a matrix where they operate on. Thus, to facilitate the representation of algorithms, the next definition allows indices of a generic type.

Definition 5.2 (Indexed Parallel Composition)**.** *Let S be a finite set of indices of type 'a, and F a total function of the type 'a ⇒ Command.*

$$\|_S \cdot F = \begin{cases} skip & for\ S = \emptyset, \\ F\ e \parallel (\|_{S'} \cdot F) & for\ S = S' \cup \{e\} \wedge e \notin S'. \end{cases}$$

Returning to our example, we can now represent $(b \parallel (a \parallel skip))$ as $(\|_{\{a,b\}} \cdot id)$, because it will expand as $(id\ b \parallel ((id\ a) \parallel skip))$. Note that Definition 5.2 only makes sense from a semantic point of view. The reason is that $b \parallel (a \parallel skip) \neq a \parallel (b \parallel skip)$, and for the definition over sets to be consistent from a syntactic perspective, we would need these terms to be syntactically equivalent. Thus, the equality in the definition using sets is not syntactic, but is semantic equivalent ($\sim$ in our notation). To provide a syntax for indexed parallelism, sets must be accompanied by an ordering relation to establish a unique order for parallel composition. In the mechanisation of indexed parallelism we use the next definition based on lists instead of sets, since lists already impose an order over the elements and are finite by definition.

Definition 5.3 (Indexed Parallel Composition)**.** *Let L be a list of indices of the type 'a, and F be a total function of the type 'a ⇒ Command.*

$$\|_L \cdot F = \begin{cases} skip & for\ L = [], \\ F\ c \parallel (\|_{CL} \cdot F) & for\ L = c \cdot CL. \end{cases}$$

It is worth emphasising that programs and parameterised programs (F) have different types. The former is an element of RG-WSL, whereas the latter is a function from indices to programs. In the context of parameterised programs, it makes sense to refer to *parameterised relations*, which themselves are not relations, but functions from indices to relations. In the discussion that follows, we refer to a parameterised program as the *body* of an indexed parallel composition.

5.1.1 Monotonicity and substitution

Handling indexed parallelism is pretty much similar to handling binary parallelism. It is possible to refine the body of an indexed parallel composition using laws for monotonicity and substitution, and also distribute a guarantee over an indexed composition. Special care must be taken if the guarantee is parameterised by an index: specific laws for pushing the guarantee inside the indexed parallel composition and to move a nested guarantee to outside of the indexed parallel composition are needed and usual distributivity laws do not work.

Law 5.4 (Monotonic-Indexed-Parallelism)**.** *For any parameterised programs F and D, and list of indices Idx, such that F i ⊑ D i for all i ∈ set Idx,*

$$(\|_{Idx} \cdot F) \sqsubseteq (\|_{Idx} \cdot D)$$

To refine the parameterised program F in the indexed parallel composition $\|_{Idx} \cdot F$ by D, it is sufficient to ensure that pointwise D refines F for all the indices in Idx. Similarly, substitution of F by D requires parameterised programs to be pointwise trace equivalent for all indices in Idx.

Law 5.5 (Substitution-Indexed-Parallelism)**.** *For any parameterised programs F and D, and lists of indices Idx, such that F i ∼ D i for all i ∈ set Idx,*

$$(\|_{Idx} \cdot F) \sim (\|_{Idx} \cdot D)$$

Non-parameterised guarantees distributes over indexed parallelism.

Law 5.6 (Distribute-g-Parallel)**.** *For any context function F and relation g,*

$$(\mathbf{guar}\ g \cdot \|_{Idx} \cdot F) \sim (\|_{Idx} \cdot (\lambda c.\ \mathbf{guar}\ g \cdot F\ c))$$

If the guarantee condition is parameterised by an index, the previous law cannot be applied. In this case, the next two laws formalise the trade of a guarantee to inside or outside of the indexed parallel composition. To remove the inner parameterised guarantee g_0, one needs to use an external and non-parameterised guarantee g_1 that is stronger than the conjunction of the guarantees of all programs composed in parallel.

Law 5.7 (Pull-Out-Parameterised-Guar)**.** *For any parameterised program F, list of indices Idx, relation g_1 and parameterised relation g_0, such that ($\vdash g_1 \Rightarrow g_0\ i$), for all $i \in$ set Idx,*

$$(\|_{Idx} \cdot (\lambda c.\ \mathbf{guar}\ g_0\ c \cdot F\ c)) \sqsubseteq (\mathbf{guar}\ g_1 \cdot \|_{Idx} \cdot F)$$

To push a non-parameterised guarantee g_0 inside of an indexed parallel composition one can apply law 5.6 (Distribute-g-Parallel). Alternatively, one can use a parameterised guarantee g_1 that is stronger than g_0 for each index in *Idx*.

Law 5.8 (Push-In-Parameterised-Guar)**.** *For any context function F, list of indices Idx, relation g_0 and parameterised relation g_1, such that $\vdash g_1\ i \Rightarrow g_0$ for all $i \in$ set Idx,*

$$(\mathbf{guar}\ g_0 \cdot \|_{Idx} \cdot F) \sqsubseteq (\|_{Idx} \cdot (\lambda c.\ \mathbf{guar}\ g_1\ c \cdot F\ c))$$

5.1.2 Introducing indexed parallelism

The introduction of indexed parallelism in a development mirrors the introduction of binary parallelism discussed in Section 3.8. Two cases are provided, but only the first strictly necessary: parameterised rely and guarantee conditions for a nested specification. The reason

for presenting the second case is to tailor the proof obligations for the scenario where rely and guarantee conditions are not parameterised, reducing the effort necessary to introduce indexed parallel composition. We illustrate the first law in the derivation of Floyd-Warshall, and the second in the derivation of Sieve.

Law 5.9 (Introduce-Multi-Parallel-Parameterised)**.** *For any predicate p, injective list of indices Idx*[1]*, non-parameterised relations Q and R, and parameterised relations g, r and q such that* $\vdash g\ i \Rightarrow r\ j$ *for all* $i, j \in$ *set Idx satisfying* $i \neq j$*, and* $\vdash R \Rightarrow r\ i \vee$ *idrel for all* $i \in Idx$*, and* $\vdash (\lambda s\ s'.\ \forall i{\in}\mathit{set}\ \mathit{Idx}.\ q\ i\ s\ s') \Rightarrow Q$*,*

$$\mathbf{rely}\ R \cdot [p, Q] \sqsubseteq \|_{Idx} \cdot (\lambda i.\ \mathbf{guar}\ g\ i \cdot \mathbf{rely}\ r\ i \cdot [p, q\ i])$$

Proof. By induction on the structure of *Idx*. To prepare law 5.9 for the inductive proof both the rely and postcondition must be formulated in terms of *Idx*. This formulation establishes an interface between the induction hypothesis and the conclusion of the law. The first step is thus to strengthen the postcondition Q and weaken the rely condition R using laws 3.19a (Consequence) and 3.87a (Rely-Monotonic). This leaves the conjecture in the right shape for a proof by induction, as

$$\begin{array}{ll} & \mathbf{rely}\ (\lambda s\ s'.\ \forall i{\in}\mathit{set}\ \mathit{Idx}.\ r\ i\ s\ s') \cdot [p, \lambda s\ s'.\ \forall i{\in}\mathit{set}\ \mathit{Idx}.\ q\ i\ s\ s'] \\ \sqsubseteq & \|_{Idx} \cdot (\lambda i.\ \mathbf{guar}\ g\ i \cdot \mathbf{rely}\ r\ i \cdot [p, q\ i]) \end{array}$$

The base case ($Idx = []$) requires us to show that $\mathbf{rely}\ true \cdot [p, true] \sqsubseteq skip$. This follows from law 3.88 (Rely-Specification) as $\vdash p \Rightarrow stops(skip, true)$ and $[p, true] \sqsubseteq skip \parallel \langle true \rangle^\star$. For the inductive case ($Idx = a \cdot list$), we must show:

$$\begin{array}{ll} & \mathbf{rely}\ (\lambda s\ s'.\ \forall i{\in}\mathit{set}\ (a \cdot \mathit{list}).\ r\ i\ s\ s') \cdot [p, \lambda s\ s'.\ \forall i{\in}\mathit{set}\ (a \cdot \mathit{list}).\ q\ i\ s\ s'] \\ \sqsubseteq & \|_{(a\ \cdot\ list)} \cdot (\lambda i.\ \mathbf{guar}\ g\ i \cdot \mathbf{rely}\ r\ i \cdot [p, q\ i]) \end{array}$$

using the inductive hypothesis:

$$a \notin \mathit{set}\ \mathit{list} \wedge (\forall i\ j.\ \{i, j\} \subseteq \mathit{set}\ (a \cdot \mathit{list}) \wedge i \neq j \longrightarrow \vdash g\ i \Rightarrow r\ j) \wedge \begin{pmatrix} & \mathbf{rely}\ (\lambda s\ s'.\ \forall i{\in}\mathit{set}\ \mathit{list}.\ r\ i\ s\ s') \cdot [p, \lambda s\ s'.\ \forall i{\in}\mathit{set}\ \mathit{list}.\ q\ i\ s\ s'] \\ \sqsubseteq & \|_{list} \cdot (\lambda i.\ \mathbf{guar}\ g\ i \cdot \mathbf{rely}\ r\ i \cdot [p, q\ i]) \end{pmatrix}$$

[1] Injective lists do not contain repeated elements.

$\quad$ **rely** $(\lambda s\ s'.\ \forall i{\in}set\ (a \cdot list).\ r\ i\ s\ s') \cdot [p, \lambda s\ s'.\ \forall i{\in}set\ (a \cdot list).\ q\ i\ s\ s']$

$\sqsubseteq$ by logical equivalence as $(\forall i{\in}set\ (a \cdot list).\ P\ i) = (P\ a \wedge (\forall i{\in}set\ list.\ P\ i))$

$\quad$ **rely** $r\ a \wedge (\lambda s\ s'.\ \forall i{\in}set\ list.\ r\ i\ s\ s') \cdot [p, \lambda s\ s'.\ q\ a\ s\ s' \wedge (\forall i{\in}set\ list.\ q\ i\ s\ s')]$

$\sqsubseteq$ by law 3.97 (Introduce-Parallel-Spec-Nested)

$\quad$ (**guar** $g\ a \cdot$ **rely** $r\ a \vee (r\ a \wedge (\lambda s\ s'.\ \forall i{\in}set\ list.\ r\ i\ s\ s')) \cdot [p, q\ a]) \parallel$

$\quad$ (**guar** $r\ a \cdot$ **rely** $g\ a \vee (r\ a \wedge (\lambda s\ s'.\ \forall i{\in}set\ list.\ r\ i\ s\ s')) \cdot [p, \lambda s\ s'.\ \forall i{\in}set\ list.\ q\ i\ s\ s'])$

$\sqsubseteq$ by laws 3.27g (Monotonic-WSL) and 3.87a (Rely-Monotonic)

$\quad$ (**guar** $g\ a \cdot$ **rely** $r\ a \cdot [p, q\ a]) \parallel$

$\quad$ (**guar** $r\ a \cdot$ **rely** $g\ a \vee (r\ a \wedge (\lambda s\ s'.\ \forall i{\in}set\ list.\ r\ i\ s\ s')) \cdot [p, \lambda s\ s'.\ \forall i{\in}set\ list.\ q\ i\ s\ s'])$

$\sqsubseteq$ by laws 3.27g (Monotonic-WSL), 3.87a (Rely-Monotonic) and ind. hypothesis

$\quad$ (**guar** $g\ a \cdot$ **rely** $r\ a \cdot [p, q\ a]) \parallel$

$\quad$ (**guar** $r\ a \cdot$ **rely** $(\lambda s\ s'.\ \forall i{\in}set\ list.\ r\ i\ s\ s') \cdot [p, \lambda s\ s'.\ \forall i{\in}set\ list.\ q\ i\ s\ s'])$

$\sqsubseteq$ by laws 3.27g (Monotonic-WSL), 3.60b (Guarantee-Monotonic) and ind. hypothesis

$\quad$ (**guar** $g\ a \cdot$ **rely** $r\ a \cdot [p, q\ a]) \parallel$ (**guar** $r\ a \cdot \parallel_{list} \cdot (\lambda i.$ **guar** $g\ i \cdot$ **rely** $r\ i \cdot [p, q\ i]))$

$\sqsubseteq$ by laws 3.27g (Monotonic-WSL), 3.62c (Distribute-Guarantee)

$\quad$ and 5.8 (Push-In-Parameterised-Guar) and ind. hypothesis

$\quad$ (**guar** $g\ a \cdot$ **rely** $r\ a \cdot [p, q\ a]) \parallel (\parallel_{list} \cdot (\lambda i.$ **guar** $g\ i \cdot$ **rely** $r\ i \cdot [p, q\ i]))$

$\sim$ by Definiton 5.3 (Indexed Parallel Composition)

$\quad \parallel_{(a \cdot list)} \cdot (\lambda i.$ **guar** $g\ i \cdot$ **rely** $r\ i \cdot [p, q\ i])$ $\square$

Remark. *The assumption ($\forall i\ j.\ \{i, j\} \subseteq set\ Idx \wedge i \neq j \longrightarrow \vdash g\ i \Rightarrow r\ j$) requires each guarantee to imply the rely of all sibling programs. The guard $i \neq j$ weakens this assumption by not requiring the guarantee of each program to imply its own rely. To get rid of this guard in the inductive case, the list of indices Idx is required to be injective.*

The injective requirement on the list of indices Idx does not prevent law 5.9 being used to model parallel compositions with multiple instances of a same program. This can be done by defining the parameterised relations g, r and q in such a way that given the indices of replicas, say i,j $\in$ set Idx ($i \neq j$), one has $g\ i = g\ j$, $q\ i = q\ j$ and $r\ i = r\ j$. For example, for $p = true$ and a relation Q denoting $((x \leq x') \wedge (y + 1 \leq y'))$, and $R = idrel$, law 5.9 allows

the derivation of the parallel composition

$$\|_{[1,\,2,\,3]} \cdot (\textbf{guar}\ g\ i \cdot \textbf{rely}\ r\ i \cdot [p, q\ i])$$

where g, r and q are defined to meet the constraints:

$$\begin{array}{lll} g\ 1 = (x \leq x') \wedge idset\ \overline{\{x\}} & r\ 1 = (x \leq x') & q\ 1 = (x \leq x') \\ g\ 2 = (x \leq x') \wedge idset\ \overline{\{x\}} & r\ 2 = (x \leq x') & q\ 2 = (x \leq x') \\ g\ 3 = idset\ \{x\} & r\ 3 = idset\ \{y\} & q\ 3 = (y' = y + 1) \end{array}$$

Thus, the indices 1 and 2 result in instances of a same program.

For the case where rely and guarantee conditions are independent of the index of the programs, relations are not parameterised and the assumption $\vdash g \Rightarrow r$ replaces the assumption $\forall i\, j.\ \{i, j\} \subseteq set\ Idx \wedge i \neq j \longrightarrow \vdash g\ i \Rightarrow r\ j$ of law 5.9. A consequence of this simplification is that the list of indices does not need to be injective. Since the proof of the next two laws are analogous to the proofs already discussed, we omit them here.

Law 5.10 (Introduce-Multi-Parallel)**.** *For any predicate p, list of indices Idx, non parameterised relations g, r, R and Q, and parameterised relation q such that* $\vdash g \Rightarrow r$ *and* $\vdash R \Rightarrow r \vee idrel$*, and* $\vdash (\lambda s\ s'.\ \forall i \in set\ Idx.\ q\ i\ s\ s') \Rightarrow Q$*,*

$$\textbf{rely}\ R \cdot [p, Q] \sqsubseteq \|_{Idx} \cdot (\lambda i.\ \textbf{guar}\ g \cdot \textbf{rely}\ r \cdot [p, q\ i])$$

5.2 Eguard

In a step of the derivation of a search algorithm in Section 6.4, we needed to introduce two local variables in a program containing a rely command. We wanted the rely condition resulting from this transformation to record the fact that local variables are unaffected by external interference. The desired transformation is shown next and corresponds to the refinement step $\mathscr{R}_2$ on page 186.

$$\begin{array}{ll} & \{t\}\colon \textbf{rely}\ idset\ \{v, t\} \cdot [true, post_0(t, t, T_p)] \\ \sqsubseteq & \textbf{var}\ ot \cdot \textbf{var}\ et \cdot \{t, ot, et\}\colon \textbf{rely}\ idset\ \{v, t, ot, et\} \cdot [true, post_0(t, t, T_p)] \end{array}$$

This refinement can be justified using law 3.102 (Introduce-Variable-Rely). To prove this law we needed first proving law 3.90 (Strengthen-Rely-In-Context), shown below for convenience.

$$\textbf{rely}\ (r, z) \cdot c \ \sqsubseteq[rx]\ \ \textbf{rely}\ (rx \wedge r, z) \cdot c$$

To prove this law we have to instantiate an existentially quantified program, say b, to be the same as another program, say d, except that program b only allows environment steps (ϵ) satisfying the relation $rx \vee idrel$, that is $[\![b]\!] = [\![d]\!][rx]$. While trying to characterise such a program we discovered that RG-WSL is not sufficiently expressive to encode it. The reason being because there is no command allowing one to constrain environment steps. In [48], the proof of this law follows as if it were possible to constrain environment steps of a program through set comprehension. This would only be possible if we had encoded programs as an abbreviation for sets of traces. However, we characterise programs as syntactic objects. Recall from Chapter 2 that our syntactic encoding is necessary to formalise the operational semantics of RG-WSL. We cannot define the operational semantics without a syntax.

To overcome this expressiveness weakness we propose a new primitive command, *eguard*. The next definition extends the operational semantics (see Section 2.7) to cover this new primitive. The command (**eguard** r) takes a relation r, used to constrain environment steps (rule 5.11a). It does not impose restrictions on program and termination steps (rules 5.11b and 5.11c, respectively). The lack of a rule to infer **eguard** $r_{\sigma\times}$ means that *eguard* cannot abort. This semantics is ideal for our purpose of proving law 3.90: the program ((**eguard** $rx \vee idrel$) $\Cap d$) only aborts when d aborts, and it behaves exactly as d, except that it only allows environment steps satisfy the relation $rx \vee idrel$.

Definition 5.11 (Semantics-Eguard)**.** *Let r be a relation, and σ and σ' states, and remember $\upsilon\ \sigma$ means a termination step, $\pi\ \sigma\ \sigma'$ means a program step and $\epsilon\ \sigma\ \sigma'$ means an environment step. Morever, remember that $\sigma, \sigma' \models r$ means that the relation r is satisfied by the pair of before and after states (σ, σ')*

$$\frac{\sigma, \sigma' \models r}{\textbf{eguard}\ r \xrightarrow{\epsilon\ \sigma\ \sigma'} \textbf{eguard}\ r} \tag{5.11a}$$

$$\frac{}{\textbf{eguard}\ r \xrightarrow{\pi\ \sigma\ \sigma'} \textbf{eguard}\ r} \tag{5.11b}$$

$$\frac{}{\textbf{eguard}\ r \xrightarrow{\upsilon\ \sigma} \mathit{nil}} \tag{5.11c}$$

Three lemmas are introduced to reason about *eguard*. These lemmas are used in the proof of law 3.90 (Strengthen-Rely-In-Context), which is discussed at the end of this section. We prove the first lemma using forward simulation (Section 2.10), and for the other two we offer an informal proof sketch and encode them as local assumptions of the algebra.

Lemma 5.12 (Strengthen-Eguard)**.** *For any relations r_1, r_2 and r, such that $\vdash r_1 \wedge r_2 \Rightarrow r$,*

$$\textbf{eguard}\ r \sqsubseteq (\textbf{eguard}\ r_1) \Cap (\textbf{eguard}\ r_2)$$

Proof. By law 2.79 (Refinement-Stratified-Forward-Simulation), one has to show that:

$$\textbf{eguard}\ r \preceq[n,\mathit{true}]\ (\textbf{eguard}\ r_1) \Cap (\textbf{eguard}\ r_2)$$

for all $n \in \mathbf{N}$. The proof is done by induction on n. For $n = 0$ the proof is trivial and follows from the base case of Definition 2.78 (Stratified-Forward-Simulation). For the inductive case, one has to show that

$$\textbf{eguard}\ r \preceq[\mathit{Suc}\ n,\mathit{true}]\ (\textbf{eguard}\ r_1) \Cap (\textbf{eguard}\ r_2)$$

from the assumptions

$$\vdash r_1 \wedge r_2 \Rightarrow r \tag{5.12a}$$

$$\textbf{eguard}\ r \preceq[n,\mathit{true}]\ (\textbf{eguard}\ r_1) \Cap (\textbf{eguard}\ r_2) \tag{5.12b}$$

By the inductive case of Definition 2.78, this holds if

$$\forall\, \alpha\ d'.\ ((\textbf{eguard}\ r_1) \Cap (\textbf{eguard}\ r_2) \xrightarrow{\alpha} d') \longrightarrow (\exists c'.\ (\textbf{eguard}\ r) \xrightarrow{\alpha} c' \wedge c' \preceq[n,\mathit{true}]\ d')$$

The proof follows by case analysis on the type of transition α. Three cases have to be considered: environment step (ϵ), termination step (υ) and program step (π). For each case we provide a witness for c' and discuss the proof of the left and right conjunct in the scope of the existential quantifier. Next we refer to the left-hand side of the implication as the *premise*.

Program step Choose c' to be **eguard** r. The left conjunct is eliminated by rule 5.11b. The right conjunct is proved by case analysis[2] on the premise. Two cases have to be considered: (i) (**eguard** r_1) $\Cap$ (**eguard** r_2) aborts; or (ii) **eguard** $r_1 \xrightarrow{\alpha} d_1{}'$ and **eguard** $r_1 \xrightarrow{\alpha} d_2{}'$ and $d' = d_1{}' \Cap d_2{}'$. Since neither **eguard** r_1 nor **eguard** r_2 can abort, their conjunction cannot abort and case (i) is spurious. Thus, only case (ii) needs to be considered. Applying rule 5.11b twice, we have that $d_1{}' =$ **eguard** r_1 and $d_2{}' =$ **eguard** r_2, and thus $d' =$ (**eguard** r_1) $\Cap$ (**eguard** r_2). Therefore, $c' \preceq[n,true]\ d'$ follows from the inductive hypothesis 5.12b.

Term. step Choose c' to be *nil*. By rule 5.11c we prove the left conjunct. By case analysis on the premise, one can infer that $d' = nil$ and therefore the right conjunct holds because forward simulation is a reflexive relation.

Env. step Let $\alpha = \epsilon\ \sigma\ \sigma'$ and choose c' to be **eguard** r. By case analysis on the premise, either (i) the strict conjunction aborts or (ii) it does an environment step. Since *eguard* cannot abort, case (i) is spurious. By case analysis on (ii), both branches of strict conjunction have to agree on the environment step. Therefore, it has to be the case that **eguard** $r_1 \xrightarrow{\epsilon\ \sigma\ \sigma'}$ **eguard** r_1 and **eguard** $r_2 \xrightarrow{\epsilon\ \sigma\ \sigma'}$ **eguard** r_2, and then $d' =$ (**eguard** r_1) $\Cap$ (**eguard** r_2). By case analysis each of these transitions, one can infer that $\sigma, \sigma' \models r_1$ and $\sigma, \sigma' \models r_2$. Thus, it follows from assumption 5.12a that $\sigma, \sigma' \models r$. Hence, the left conjunct holds by rule 5.11a and the right conjunct follows from the inductive hypothesis 5.12b. □

The next lemma uses *eguard* to establish a relationship between the traces of a program in a restricted environment ($[\![c]\!][r]$) and traces of a program in an unrestricted environment ($[\![c]\!][true]$). The proof of this lemma is beyond the expressiveness of our mechanisation. We provide a proof sketch, but its mechanisation would requires the algebraic manipulation of

[2] *Case analysis* on inductive assumptions is a proof technique available for inductive definitions. It corresponds to exhausting all possible causes which can make the inductive assumption to hold, and then show that each of these causes suffices to prove the conclusion of the original conjecture. Such proof technique holds because an inductive definition corresponds to the smallest set closed over the inference rules of that definition, and then an inductive assumption only holds if there is a "good" reason for it to hold [70]. In Isabelle, such proof technique is available through the so-called *rule-inversion mechanism* [85]. For example, if the premise contains $c \Cap d \xrightarrow{\alpha} d'$ and we know that α is not a termination step, then it must be the case that this transition happened either because: (i) rule 2.49 was triggered; or (ii) rule 2.51 (abortion) was triggered, and cause the triggering of rule 2.44. If we know that c and d cannot abort, then we know that their conjunction cannot abort as well. Therefore, case (ii) is impossible (i.e., spurious), and we only need to consider case (i).

the concept of denotational semantics ($[\![\_]\!]$), which we encoded as an uninterpreted constant in Isabelle. Thus, the next equivalence was encoded as a local assumption by the algebra.

Lemma 5.13 (Traces-Eguard)**.** *For any command d relation r,*

$$[\![d]\!][r] = [\![d \Cap (\textbf{eguard } r \lor \mathit{idrel})]\!][\mathit{true}]$$

Proof sketch. *By definition 2.71 (Denotational semantics), $[\![d]\!][r]$ only includes traces of d whose environment transitions (ϵ) satisfy r or stutter. Since the operational semantics for (***eguard** *r) only allows environment steps which satisfy r (rule 5.11b), and the operational semantics for strict conjunction requires both branches to agree on labelled state transitions (π, ϵ, υ), we can infer that any trace of $((\textbf{eguard } r \lor \mathit{idrel}) \Cap c)$ can only contain environment transitions that satisfy r or stutter.*

The next lemma allows the user to introduce or remove a syntactic restriction on a sibling program (d) that runs in parallel with a program (c) guarded by **eguard** r.

Lemma 5.14 (Conjunction-Parallel-Eguard)**.** *For any relations r and commands c and d,*

$$((c \Cap (\textbf{eguard } r)) \parallel d) \sim ((c \Cap (\textbf{eguard } r)) \parallel (\langle r \rangle^{\omega} \Cap d))$$

Proof sketch. *The refinement from left to right is straightforward. It uses law 3.29e (Zeros-and-units) to substitute (d) by $(\langle \mathit{true} \rangle^{\omega} \Cap d)$ on the left-hand side, and law 3.19b (Consequence) to strengthen the atomic command from $\langle \mathit{true} \rangle^{\omega}$ to $\langle r \rangle^{\omega}$.*

The refinement from right to left holds because whenever d attempts to do a program step that does not satisfy r, its transition is blocked by the sibling program $c \Cap (\textbf{eguard } r)$. Such blocking is explained by the matching mechanism used in the rule for parallel composition (see Definition 2.52 on page 48 and rule 2.53 on page 49). Thus, any program step that d can make must also be a program step of $\langle r \rangle^{\omega} \Cap d$. For environment and termination steps, the refinement holds because the command $\langle r \rangle^{\omega}$ does not restrict any of these steps, thus the strict conjunction preserves the behaviour already allowed by d.

Law 5.15 (Refinement-Eguard)**.** *Let c and d be commands, and r a relation,*

$$(c \sqsubseteq [r]\ d) = (\ c \sqsubseteq d \Cap (\textbf{eguard } r \lor \mathit{idrel}))$$

The previous law follows from Definition 2.73 (Refinement-in-Context) and lemma 5.13 (Traces-Eguard). A revised proof of the derivation of law 3.90, discussed in the begin of this section, is available in Appendix A.3 (Revised paper proofs) on page 257. The revised proof has been mechanised and uses *eguard* in a challenging instantiation step of the derivation.

5.3 Revised abortive conditions

While investigating properties about parallel composition we draw our attention to an intriguing case: the parallel composition between *magic* and *abort*. Using the abortive conditions for parallel composition introduced in Section 2.7.2, we proved by forward simulation that $(magic \parallel abort) \sim abort$. This equivalence raised a red flag, as it could be combined with law 3.29f (Zeros-and-units) to show that $magic \sim abort$, which is false and not what we want.

The problem behind the proof of $magic \sim abort$ amounts to the abortive conditions presented in Section 2.7. More precisely, the abortive conditions for parallel composition state that the composition aborts if any branch can abort. To prevent the abortion of the parallel composition in $magic \parallel abort$, we introduce an extra requirement in the small-step semantics: for the parallel composition to abort, not only does one branch has to abort, but the other branch has to agree with the abortion of its environment. We model that all commands except *magic* do not prevent their environment from aborting.

The fact that a program c does not prevent its environment from aborting from a state σ is formalised as $c_{\sigma \times env}$. The small-step semantics receives the following additional rules. They state each command does not prevent their environment from aborting.

1. A precondition does not prevent its environment from aborting.

$$\frac{}{\{p\}_{\sigma \times env}} \tag{5.16}$$

2. The atomic command and the command *eguard r* do not prevent their environments from aborting.

$$\frac{}{\langle p, q \rangle_{\sigma \times env}} \quad \frac{}{\textbf{eguard } r_{\sigma \times env}} \tag{5.17}$$

3. As long as there is one command within C which does not prevent the environment from aborting, the unbounded choice does not prevent the environment from abort. This prevents *magic* from conceding permission for its environment to abort, because the proviso is not vacuously true.

$$\frac{\mathit{countable}\ C \qquad c \in C \qquad c_{\sigma \times env}}{\bigsqcap \mathit{acset}\ C_{\sigma \times env}} \tag{5.18}$$

4. As long as the first component of a sequential composition does not prevents its environment from aborting, the composition does not prevent the environment from aborting. This allows, for example, the program *abort* ; *magic* to give permission for its environment to abort.

$$\frac{c_1 \sigma \times env}{c_1 \mathbin{;} c_2 \sigma \times env} \tag{5.19}$$

5. For strict conjunction, both branches need to agree in order to allow the environment to abort.

$$\frac{c_1 \sigma \times env \qquad c_2 \sigma \times env}{c_1 \mathbin{⨅} c_2 \sigma \times env} \tag{5.20}$$

6. If c allows the environment to abort from a state $\sigma(y := v)$, then **state** $y \mapsto v \cdot c$ allows the environment to abort in a state σ.

$$\frac{c_{\sigma(y := v) \times env}}{\mathbf{state}\ y \mapsto v \cdot c_{\sigma \times env}} \tag{5.21}$$

7. If c allows the environment to abort, then **uses** $X \cdot c$ also allows the environment to abort.

$$\frac{c_{\sigma \times env}}{\mathbf{uses}\ X \cdot c_{\sigma \times env}} \tag{5.22}$$

The abortive conditions for parallel composition are patched with proviso $c_{i\sigma \times env}$ to restrict the abortive behaviour of the composition. The patch is simple: instead of relying in

a single branch to infer that a composition aborts, both branches are required to agree on the abortion; one branch actively aborts and the other branch allows the sibling to abort.

$$\frac{c_1\sigma\times \qquad c_2\sigma\times env}{c_1 \parallel c_2\sigma\times} \qquad \frac{c_2\sigma\times \qquad c_1\sigma\times env}{c_1 \parallel c_2\sigma\times} \qquad \frac{c_1\sigma\times env \qquad c_2\sigma\times env}{c_1 \parallel c_2\sigma\times env} \tag{5.23}$$

Recall that indexed parallelism is not introduced as a primitive command. Instead, it is recursively defined using binary parallelism. Therefore, the patch for the semantics of binary parallel composition is the only one we need to concern about.

5.4 Assignment to indexed arrays

Arrays are a common datatype in programming languages, and provide a powerful mechanism to store and index a predefined number of values. In this thesis, arrays are used in the formalisation of two concurrent programs discussed in Chapter 6: Floyd-Warshall and Findp. The rely-guarantee refinement calculus as presented in [48] is capable of handling reading on arrays assuming atomic fetch of an array, but it does not include refinement laws to introduce assignments to cells of an array in a program. This section extends the algebra with a formalisation of assignment to indexed arrays.

Similarly to [41], we model arrays using a recursive datatype, where each dimension is encoded as a list. Recall that arrays are formalised via the constructor *VArray* in Definition 2.1 on page 23. Lists in Isabelle are zero-based, i.e. their index starts from zero. Thus, arrays in RG-WSL are also zero-based. Isabelle uses the operator ! (exclamation mark) to access positions of a list, thus the first element of the list $[a, b, c]$ is retrieved using $[a, b, c]$! *0*. Next, we propose a definition for assignment to indexed arrays which is inspired on Definition 3.12 (Assignment), introduced on page 69.

Definition 5.24 (Assignment-Array)**.** *Assignment of an expression e to an indexed position l, where* $l = [i_0, i_1, ..., i_{n-1}]$*, of a n-dimensional array A, is modelled as the non-deterministic choice between all possible evaluations of the expression e followed by an atomic update of the target cell of A to the evaluated value of e.*

$$xl{:=}e \;\equiv \bigsqcap \mathit{acset}\ \{[[N\ v = e]]\ ;\ \langle(\lambda s\ s'.\ s'\ x = (s\ x)[l{\leftarrow}v]) \wedge \mathit{idset}\ \overline{\{x\}}\rangle \mid \mathit{True}\}$$

In this definition, v is a logical constant that is existentially quantified in the scope of the non-deterministic choice ($\bigsqcap$). The terms x, l and e represent, respectively, the name of the

array to be updated, a list of natural numbers that index a single cell in the array x, and the expression to be assigned. The assignment is composed of two stages: evaluation and update, which are related via sequential composition. In the evaluation stage, the expression e is evaluated to the value v. Next, the array is updated atomically, but only a single cell of the array is modified during the update.

The expression $(s\ x)[l\leftarrow v]$ in the atomic command reflects the update of a single cell of the array (x) and is used to update the state (s). It is an abbreviation for the function Substitute-Cell (defined next), which takes three arguments: an array A, a list of natural numbers l that index a cell within the array, and a value v that is used to replace the old value stored in the indexed cell. The update $(A)[l\leftarrow v]$ is defined by recursion on the list of indices (l): for each recursive call, the task of updating a n-dimensional array is reduced to the task of updating a $(n-1)$-dimensional array. The base case consists of a 0-dimensional array, that is, a scalar value, whose update consists in replacing the old value by the new value. In the recursive case, every time the head (x) of the list of indices ($x \cdot xs$) is taken, the function Substitute-Cell returns an array that is equal to the argument A, except for the sub-array indexed from x which may differ. Attempts to update a non-existent position in an array leaves the array unchanged[3].

The next definition makes use of Isabelle's notation for updates of lists, where $l[i := x]$ is equal to list l, except for position i which holds the value x provided that $i < \mathit{length}\ l$; if $\mathit{length}\ l \leq i$, $l[i := x] = l$.

Definition 5.25 (Substitute-Cell)**.** *Let A be the encoding of an array as a vvalue, and lv a list of indices (where each index is a natural number), and let v be a vvalue; and recall from Definition 2.2 (Project-Argument) that* $[\![\mathit{VArray}\ v]\!]_a = v$.

$$(A)[lv\leftarrow v] = \begin{cases} v & \mathit{if}\ lv = [], \\ \mathit{VArray}\ ([\![A]\!]_a[x := ([\![A]\!]_a\ !\ x)[xs\leftarrow v]]) & \mathit{if}\ lv = x \cdot xs. \end{cases}$$

To see how this definition works, take for example the array

$$A = (\mathit{VArray}\ [\mathit{VArray}\ [p_{11}, p_{12}, p_{13}], \mathit{VArray}\ [p_{21}, p_{22}, p_{23}], \mathit{VArray}\ [p_{31}, p_{32}, p_{33}]])$$

[3]Recall that variables are not declared in RG-WSL. Thus, whenever we need to handle arrays in a program, we must include in the precondition information about the dimension of the array. Section 6.6 illustrates how the dimension of an array is included in a precondition.

whose visual representation as a matrix was presented on Section 2.2 on page 23. If we want to replace the value p_{23}, indexed by the list $[1, 2]$, by the value *VNat 5*, we can denote the updated array by $(A)[[1, 2] \leftarrow VNat\ 5]$.

Now we present a law that establishes the conditions for introducing assignments to indexed arrays in programs nested into rely and guarantee commands. The law deals with a special case: when the expression (e) to be assigned to the indexed cell is stable under interference. The law is introduced as a local assumption in the algebra and its application is illustrated in the derivation of Floyd-Warshall in Section 6.6. To highlight the assumptions, we enumerate them.

Lemma 5.26 (Assignment-Array)**.** *For any array x, list of natural numbers l, expression e, predicate p and relations g, q and r, such that*

1. $[p, q]$ *tolerates interference r; and*
2. $\vdash p \Rightarrow$ *defined e; and*
3. $\vdash p \wedge (\lambda s\ s'.\ s'\ x = (s\ x)[l \leftarrow [\![e]\!]_v\ s]) \wedge idset\ \overline{\{x\}} \Rightarrow q \wedge (g \vee idrel)$*; and*
4. *e is preserved by r, that is,* $\vdash p \wedge r \Rightarrow (\lambda s\ s'.\ [\![e]\!]_v\ s = [\![e]\!]_v\ s')$,

the following holds,

$$\{x\}\colon (\textbf{guar}\ g \cdot \textbf{rely}\ r \cdot [p, q]) \sqsubseteq xl{:=}e$$

The inspiration for the provisos comes from the context of application of this law[4]. We expect the proof of this law to mirror the structure of the proof of law 3.115 (Assignment-Single-Reference), which is discussed in detail on Section A.3 on page 260. However, as this law does not assume the expression e to satisfy the single reference property with respect to r, the proof has to take a different path to justify the distribution of interference over the test command (cf. proof of the sub-goal G1.1 on page 262). We hope the necessary details to formally justify this distribution can be extracted from the fact that e is stable under r.

[4]We explicitly mark this law as a lemma to emphasize that that it has been taken as an assumption of the mechanisation, instead of being proved in Isabelle/HOL.

5.5 Reachable evaluations

In the sequential refinement calculus, any program can be abstracted using pre and postconditions. On the other hand, for rely-guarantee algebra, there are concurrent programs that cannot be expressed accurately modulo the introduction of auxiliary variables [106, 63, 48]. A canonical example is a program that copies the value of shared variables into a local variable. Using rely-guarantee, one has no clean way of providing an abstract specification for $(\mathbf{var}\ x \cdot x{:=}w)$ without making assumptions on how the environment changes w. For example, the specification

$$\mathbf{var}\ x \cdot \mathbf{rely}\ \mathit{true} \cdot [x' = w \vee x' = w']$$

fails to capture an intermediate value of w being read by the assignment. A way of characterising this program is to appeal to auxiliary variables and assume that the environment will record in a history variable all the updates to w. Next, the operator *drop n* eliminates the first n elements of a list (e.g. $\mathit{drop}\ 2\ [a, b, c, d] = [c, d]$), and the operator *set* returns the set of elements of a list (e.g. $\mathit{set}\ [a, b, a] = \{a, b\}$).

$$\begin{array}{r} \mathbf{var}\ x \cdot \mathbf{rely}\ (w \neq w' \longrightarrow hw' = hw \frown [w]) \wedge \mathit{idset}\ \{x\} \cdot \\ [x' \in \mathit{set}\ (\mathit{drop}\ (\mathit{length}\ hw)\ hw') \vee x' = y] \end{array} \tag{5.27}$$

This specification uses an auxiliary variable (hw) to record all values that have been stored in the variable w. The rely condition assumes that whenever the environment changes w, it atomically logs the previous value that w had into hw. The specification command has a relational perspective of hw, and uses this perspective to state that the final value of x corresponds to one of the values that w has stored since the begin of the computation.

Auxiliary variables (e.g. hw) are a specification artefact, necessary to compensate for the limited expressiveness of specification languages[5], but they do not play an essential role in the refined code. As general rule, auxiliary variables do not affect the control flow of a program and are only assigned to, but not read from in the final code, which motivates some program logics, such as [92, 106], to offer a rule for their elimination. Normal usage of auxiliary variables is to record progress of a process (e.g., how many times a variable has been assigned to, or which stage of an algorithm a program is executing), or to formulate a specification

[5] Limited in the sense that certain programs cannot be specified using solely the variables effectively involved in the final computation, as we discuss for the assignment involving shared variables.

in a way that it is stable under interference. Nevertheless, indiscriminate usage of auxiliary variables jeopardise the compositionality of rely-guarantee specifications [63]. Furthermore, they obfuscate specifications. For example, to abstract the assignment $x{:=}y * z + w$ using history variables we need to use a quite unwieldy relation in the postcondition:

$$x \in \left\{ v \;\middle|\; \begin{array}{r} \exists\, y_0\ z_0\ w_0.\ (y_0 \in \mathit{set}\ (\mathit{drop}\ (\mathit{length}\ \mathit{hy})\ \mathit{hy}') \vee y_0 = y) \wedge \\ (z_0 \in \mathit{set}\ (\mathit{drop}\ (\mathit{length}\ \mathit{hz})\ \mathit{hz}') \vee z_0 = z) \wedge \\ (w_0 \in \mathit{set}\ (\mathit{drop}\ (\mathit{length}\ \mathit{hw})\ \mathit{hw}') \vee w_0 = w) \wedge \\ v = y_0 * z_0 + w_0 \end{array} \right\}$$

This section introduces a notion of *reachable evaluations*: a syntactic abbreviation that facilitates the use of the multi-step evaluation within relations. The concept is inspired by the notion of possible values [64], a convention proposed in [66] to refer to intermediate values that a variable can acquire during the execution of a program. Using the notion of reachable evaluations, one can obtain a relation that serves to abstract the program (**var** $x \cdot x{:=}w$) without requiring history variables, as shown in 5.27. The intuition behind the concept is that, generally a designer will wrap a concurrent specification inside of a rely command, and the information provided by the rely can be used to predict the states that a program can go through when it is interrupted by the environment. By predicting these states, one can predict which values shared variables can acquire as a result of updates caused by the environment. Then, by combining the set of values that variables can acquire in the lifetime of an evaluation, one can predict the potential results of non-atomic evaluation and use it inside of a relation. There is a *caveat* though: the concept of reachable evaluations *over-approximates* the set of evaluations allowed by history variables. It relies on predicting rather than inspecting the effect of interference via history variables, thus it considers values that may not occur in the actual history of a variable for a given execution trace.

To understand this over-approximation, consider again the example 5.27, but this time enrich the rely condition with the knowledge that w monotonically increases. Suppose that the task is to abstract the assignment **var** $x \cdot x{:=}w$ in the context of the parallel composition:

$$\{w = 4\}\ ;\ (\textbf{var}\ x \cdot x{:=}w \parallel (w{:=}5\ ;\ w{:=}6\ ;\ w{:=}8\ ;\ w{:=}12\ ;\ w{:=}13\ ;\ w{:=}15))$$

Figure 5.1 compares the accuracy provided by the usage of history variables versus that provided by reachable evaluations. Inspecting the history variable (*hw*), one can observe that the environment effectively updates the variable w to the values *5*, *6*, *8*, *12*, *13* and *15* (in this order). If we are to predict the set of reachable evaluations for the variable w, knowing that it

is subject to interference that monotonically increases this variable, then we have to consider the evaluation of *w* in every state that can be related to the initial state via the rely condition. Thus, this approach includes values such as *7*, *9* and *11* that never appear in the history of variable *w*. Next we introduce the formal definition of reachable evaluations.

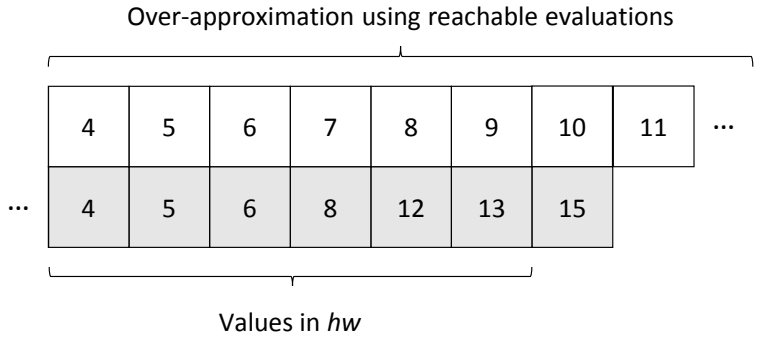


Figure 5.1 Reachable evaluations *versus* history variables.

Definition 5.28 (Reachable-Evaluations)**.** *Let r be a relation, s a state, and exp an expression. Additionally, let* α *be a labelled state transition of the form* $\epsilon\ \sigma\ \sigma'$ *or* $\pi\ \sigma\ \sigma'$*, and let t denote a trace, and remember that: pre extracts the before state (e.g.* $pre\ \epsilon\ \sigma\ \sigma' = \sigma$*), hd returns the head of a list (e.g.* $hd\ [a, b, c] = a$*), env abstracts the environment of a trace*[6]*, and* $exp \xrightarrow{t}_{e*} (N\ v)$ *denotes that the expression exp evaluates for the value v via trace t.*

$$RE\ r\ s\ exp = \{v \mid \exists t.\ (exp \xrightarrow{t}_{e*} N\ v) \wedge (pre\ (hd\ t) = s) \wedge (\vdash env\ t \Rightarrow r)\}$$

The set *RE r s exp* denotes all potential evaluations (*v*) of the expression *exp* from the initial state *s*, where the evaluation is subject to interference *r*. It is important to understand that the motivation for reachable evaluations is to abstract assignments that contain one or more unstable variables *without* using auxiliary variables. The weakness of this definition is that it allows the user to bypass interference, as she can choose one potential evaluation instead of sampling the variables to evaluate the expression *exp*. Thus, one has to be aware that specifications using reachable evaluations record the intention of an assignment, but do not force the user to sample the variables to implement the desired assignment. To facilitate the usage of the concept of reachable evaluations in specifications, we introduce a definition that we call *loose equality*.

[6]To refresh the concept of *env*, see Definition 2.70 on page 54.

Definition 5.29 (Loose-Equality)**.** *For any variable name x, relation r and expression exp,*

$$(x \doteq_r exp) \quad \equiv \quad \lambda s\ s'.\ s'\ x \in RE\ r\ s\ exp$$

Using this definition, the program **var** $x \cdot x{:=}w$ running in an environment z is specified as

$$\textbf{var}\ x \cdot \textbf{rely}\ z \cdot [true, x \doteq_z w]$$

We provide only one lemma to reason about reachable evaluations. It eliminates reachable evaluations from a specification command. This lemma is formalised as a local assumption of the algebra and its use is illustrated in the next example.

Lemma 5.30 (RE-Intended-Assignment)**.** *For any predicate p, relations g, r and q, variable x and expression e such that*

1. $\vdash p \Rightarrow$ *defined e; and*
2. *p is stable under r; and*
3. $\vdash r \Rightarrow$ *idset* $\{x\}$*; and*
4. $\vdash p \wedge x \doteq [r]\ e \wedge$ *idset* $\overline{\{x\}} \Rightarrow q \wedge (g \vee$ *idrel*$)$,

the following holds,

$$\{x\}\colon (\textbf{guar}\ g \cdot \textbf{rely}\ r \cdot [p, q]) \sqsubseteq x{:=}e$$

Proof sketch. *If the environment does a step which does not satisfy the relation r the left-hand side aborts, and in this case the refinement holds trivially. Therefore we only need to care about refinement on an environment r. The relation* $(x \doteq_r e)$ *specify that x is a reachable evaluation of the expression e in an environment r. This relation establishes an over-approximation: it sets a lower bound in the sense that it accepts the final value of x to be equal to any evaluation of e from which the variables are sample in states coherent with the operational semantics for expression evaluation. However, it does not impose an upper bound to restrict which environment transitions are allowed during the evaluation. This law establishes a refinement by ruling out some of the possibilities of interference allowed in the evaluation of e: it requires any environment transition chosen in the evaluation to be compliant with the traces of* $x{:=}e$.

5.5.1 Example: parallel assignments

This is an adapted version of example 3.8.1 to illustrate the application of reachable expressions to encode multi-step evaluation of expressions via a relation. Consider the following program where an expression containing multiple unstable variables is assigned to z.

$$(x{:=}x+2 \parallel y{:=}y+2) \parallel (z{:=}x+y+w \parallel w{:=}w+1)$$

To abstract this program we leave the final value of z under-specified. A way to achieve this, while still recording the intentional assignment, is shown next.

$$[\mathit{true}, (x'=x+2) \wedge (y'=y+2) \wedge (z \doteq_{\mathit{idset}\ \{z\}} x+y+w) \wedge (w'=w+1)]$$

We now split the initial specification into a parallel composition of four specifications, where each branch sets the value of a single variable.

$$\begin{array}{ll}
 & [\mathit{true}, (x'=x+2) \wedge (y'=y+2) \wedge (z \doteq_{\mathit{idset}\ \{z\}} x+y+w) \wedge (w'=w+1)] \\
\sqsubseteq & \text{by 3.95 (Introduce-Parallel-Spec), 3.97 (Introduce-Parallel-Spec-Nested),} \\
 & \quad \text{3.87a (Rely-Monotonic), 3.62b (Distribute-Guarantee),} \\
\sqsubseteq & \quad \text{3.62c (Distribute-Guarantee), and 3.60a (Guarantee-Monotonic)} \\
 & \begin{pmatrix} (\textbf{guar}\ \mathit{idset}\ \{z,w,y\} \cdot \textbf{rely}\ \mathit{idset}\ \{x\} \cdot [\mathit{true}, x'=x+2]) \parallel \\ (\textbf{guar}\ \mathit{idset}\ \{z,w,x\} \cdot \textbf{rely}\ \mathit{idset}\ \{y\} \cdot [\mathit{true}, y'=y+2]) \end{pmatrix} \parallel \\
 & \begin{pmatrix} (\textbf{guar}\ \mathit{idset}\ \{x,y,w\} \cdot \textbf{rely}\ \mathit{idset}\ \{z\} \cdot [\mathit{true}, z \doteq_{\mathit{idset}\ \{z\}} x+y+w]) \parallel \\ (\textbf{guar}\ \mathit{idset}\ \{x,y,z\} \cdot \textbf{rely}\ \mathit{idset}\ \{w\} \cdot [\mathit{true}, w'=w+1]) \end{pmatrix} \\
\sqsubseteq & \text{by 3.59 (Introduce-Guarantee), 3.114 (Assignment-Rely-Guarantee)} \\
 & \text{and 5.30 (RE-Intended-Assignment)} \\
 & (x{:=}x+2 \parallel y{:=}y+2) \parallel (z{:=}x+y+w \parallel w{:=}w+1)
\end{array}$$

The relation $\mathit{idset}\ \{z\}$, used to predicate the values of x, y and w in the loose equality, just tells that z is stable; it does not restrict the values that x, y and w can acquire. Even such a weak specification still records the intentional assignment of the expression $x+y+w$ to

z. Such strategy to specify assignment involving multiple unstable variables works only if the user sticks to the application of law 5.30 (RE-Intended-Assignment) to eliminate loose equality. This was the law used in this example.

Of course, a user can expand the definition of loose equality and the definition of reachable evaluations, and then use rules 2.41 and 2.42 to prove that a different assignment (e.g. $x{:=}e$) still meets the specification. But, in the case where the user reasons in terms of expression evaluation, even the most restrictive prediction cannot rule out the possibility of a user pick an aleatory value among the predicted evaluations that does not reflect the actions of the environment over the state.

5.5.2 Discussion

To further discuss the notion of reachable evaluations we consider a clear-cut example. Let $\mathit{St\text{-}EF}_{xy}$ be the state where variable e evaluates to $(\mathit{VInt}\ x)$ and variable f evaluates to $(\mathit{VInt}\ y)$. Additionally, let *RelyEF* the reflexive-transitive relation illustrated Figure 5.2. We use *RelyEF* to analyse two aspects of the definition *RE* which were not discussed before:

i *RE* rules out unrealistic evaluation paths. For example, it does not allow one to infer that *8* is a reachable evaluation for the expression $e + f$ from the initial state $\mathit{ST\text{-}EF}_{00}$. Such evaluation would only be possible if state $\mathit{St\text{-}EF}_{52}$ had a transition to state $\mathit{St\text{-}EF}_{33}$ (or vice-versa), but such transitions do not exist. Thus, once an evaluation starts, *RE* keeps track of the transitions already taken.

ii *RE* separates the evaluation of expressions into two stages: sampling and evaluation, e.g. $e * e + f$ can sample variables in the order e, f, e, even though multiplication has higher precedence than addition;

Our approach to compute the set of reachable evaluations of an expression is essentially different from that used in abstract interpretation [85] to compute the set of possible values for variables and expressions in points of a program. During the computation of the set of reachable expressions we keep track of the relationship between variables (see item (i) above), while in the technique of abstract interpretation this relationship is lost to make it possible to record a potentially infinite amount of set of states in a finite amount of space. We also abstract away from the control flow of the environment by using a relation (representing interference) and use the operational semantics (for expression evaluation) to estimate the set of reachable evaluations of an expression. The *over-approximation* of the set of reachable evaluations in Definition 5.28 is influenced by the accuracy of the relation used to abstract

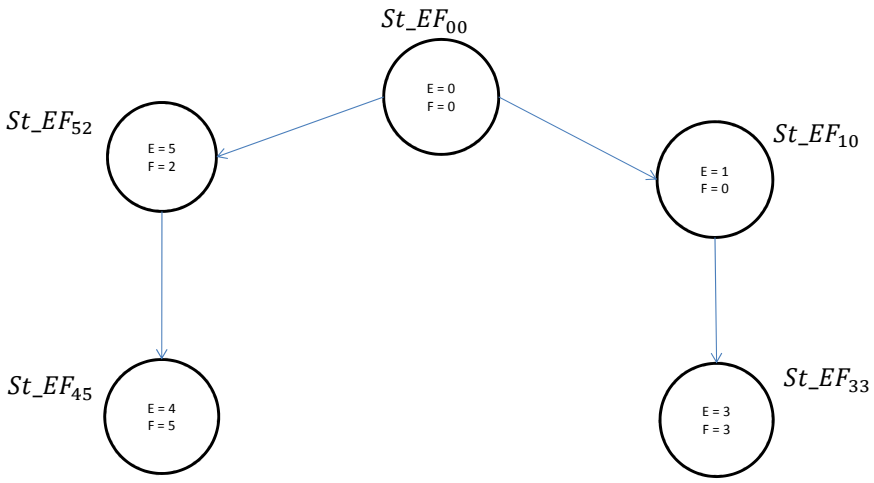


Figure 5.2 Graphical representation of relation RelyEF.

the state: the more accurate the relation, the more accurate is the set of reachable evaluations. For the technique of abstract interpretation, such imprecision comes from the encoding of sets of states as abstract values.

Although the concept of reachable evaluations does not provide the accuracy given by the use of auxiliary variables, it can be argued that it leads to clearer specifications. As we have only introduced one law to handle the concept of reachable evaluations in proofs, it is premature to judge its value as a specification tool. We believe that for further exploration of the concept, a good start point is to investigate if the definition of reachable evaluations can be used to prove properties about the possible values of a variable ($\widehat{x}$) and possible evaluations of an expression ($\widehat{e}$), which are notations discussed in [64].

The concept of reachable evaluations may also proven useful to investigate expression decomposition and refinement of expressions [25]. This would work in the following way: an expression e' refines an expression e in an environment r if the set of evaluations that e' produces under interference r is contained in the set of evaluations produced by e under the same circumstances.

5.6 Sequential laws

The next laws provide special cases to facilitate the introduction of control structures in sequential programs. The conventional route for applying laws 3.111 (Rely-Conditional) and 3.112 (Rely-Loop) for a specification not enclosed in a rely command is to introduce a rely command with the rely condition of $r = idrel$ using law 3.81 (Rely-Idrel-Specification). A drawback of using laws 3.111 and 3.112 in the refinement of sequential programs is to have to manually instantiate b_0 and b_1 and rewrite the transitive-reflexive closure of a well-founded relation (w^{**}) using the notion of transitive-reflexive closure on a closed subset ($w^{**}{}_{UNIV}$).

Law 5.31 (Sequential-Conditional)**.** *For any predicate p, relation q and boolean expression b, such that* $\vdash p \Rightarrow$ *defined b,*

$$[p, q] \sqsubseteq (\textbf{if } b \textbf{ then } [p \wedge [\![b]\!]_r, q] \textbf{ else } [p \wedge [\![\neg b]\!]_r, q])$$

Law 5.32 (Sequential-Loop)**.** *For any predicate p, boolean expression b and relation w that is well-founded on p, such that* $\vdash p \Rightarrow$ *defined b,*

$$[p, p\text{`} \wedge [\![\neg b]\!]_r\text{`} \wedge w^{**}] \sqsubseteq \textbf{while } b \textbf{ do } [p \wedge [\![b]\!]_r, p\text{`} \wedge w]$$

5.7 Type system

The algebra discussed in this thesis allows for derivations that are not type consistent, such as

$$\begin{array}{ll} & [true, x' = 1] \\ \sqsubseteq & \text{by law 3.25 (Sequential)} \\ & [true, x' = False] \,;\, [true, x' = 1] \end{array}$$

There are a number of programming languages used in industry that support concurrency and are weakly typed (e.g. C [68], PHP [102], etc.). The lack of type consistency is caused by the fact our representation of state does not record the type of variables. The fact that the algebra allows the derivation of weakly typed programs does not raise any concerns about its consistency, but makes it harder to prove properties about programs.

In some occasions it is essential to know the type of variables to eliminate projection functions, such as $[\![\_]\!]_n$ and $[\![\_]\!]_i$ (Definition 2.2), in proofs. We met one of such situations in the derivation of Floyd-Warshall, an algorithm to solve the all-pairs shortest-path problem, in Section 6.6. There, we need to know the type and dimension of a matrix of adjacency to prove properties involving the application of operators such as *min* and *max* over the cells of that matrix.

Next, we provide three definitions that can be used to embed information about types in the precondition of programs. The key motivation for these definitions is not to enforce type consistency, but to compensate for the lack of knowledge about types that is usual in derivations using RG-WSL.

Definition 5.33 (Type-Vvalue)**.** *The implicit type of a vvalue (Definition 2.1) can be described via the constructors of the following datatype.*

datatype *vvalue-type* = *type-VNat*
| *type-VInt*
| *type-VBool*
| *type-VNatSet*
| *type-option vvalue-type*
| *type-array vvalue-type*

Definition 5.34 (Infer-Type)**.** *Given a vvalue, its type can be inferred using the following inductive rules.*

$$\frac{x = \mathit{VNat}\ i}{\mathit{Type}\ x\ \mathit{type\text{-}VNat}} \qquad \frac{x = \mathit{VInt}\ i}{\mathit{Type}\ x\ \mathit{type\text{-}VInt}}$$

$$\frac{x = \mathit{VBool}\ b}{\mathit{Type}\ x\ \mathit{type\text{-}VBool}} \qquad \frac{x = \mathit{VNatSet}\ s}{\mathit{Type}\ x\ \mathit{type\text{-}VNatSet}}$$

$$\frac{x = \mathit{VNone} \vee \mathit{Type}\ x\ t}{\mathit{Type}\ x\ (\mathit{type\text{-}option}\ t)}$$

$$\frac{x = \mathit{VArray}\ v \qquad \forall i{<}\mathit{length}\ v.\ \mathit{Type}\ (v\ !\ i)\ t}{\mathit{Type}\ x\ (\mathit{type\text{-}array}\ t)}$$

For example, consider the array B:

$$B = VArray\ [VArray\ [VNat\ 0, VNat\ 7], VArray\ [VNat\ 8, VNat\ 2], VArray\ [VNat\ 4, VNat\ 5]]$$

Using Definition 5.34 (Infer-Type) we can prove that B is a bi-dimensional array of natural numbers, i.e. its type is *type-array* (*type-array type-VNat*).

Definition 5.35 (Dimension-Array)**.** *The dimension of an array (M) can be recorded using the relation dim, defined by recursion on the dimension of the array.*

$$dim\ M\ [] = true$$
$$dim\ M\ (a \cdot ls) = (\lambda s\ s'.\ length\ [\![M]\!]_a = a \wedge (\forall c{<}a.\ dim\ (VArray\ [\![[\![M]\!]_a\ !\ c]\!]_a)\ ls\ s\ s'))$$

Using this definition, we can record the dimension of the array B via the relation $dim\ B\ [3, 2]$.

5.8 Discussion and summary of contributions

5.8.1 Contributions

In this chapter we discussed extensions to the rely guarantee algebra formalised in Chapter 4. We introduced two derived commands: one to represent indexed parallelism, and other to represent assignment to indexed arrays. These commands are accompanied by relevant laws. We also introduced a new primitive command (**eguard** r) to constrain environment steps. This new command, and its lemmas, are necessary to complete the proof of a derived law that relates a newly introduced local variable and the rely command.

This chapter also solves a latent problem in the abortive conditions for parallel composition. The problem was due to the possibility of a single branch to force a parallel composition to abort, independent of the other branch. Under this conditions, the semantics could be used to prove the trace equivalence between the programs *magic* and *abort*, which is false and not what we want. Our solution to this problem consisted on introducing of an agreement clause in the operational semantics to characterise that a program can only abort if its environment allows it to abort. The undesired equivalence was rule out by defining *magic* to be a program that does not give permission for its environment to abort.

We also discussed a syntactic abstraction (reachable evaluations) to over-approximate the non-atomic evaluation of expressions when writing specifications, and also introduce tailored laws to facilitate the introduction of conditionals and loops in sequential programs.

The notion of reachable expression is illustrated with a simple derivation, and compared to the notion of possible values [66, 64], which is discussed in the literature.

Finally, this chapter discusses few auxiliary definitions that can be used to record type-related information in derivations. The need for recording such kind of information is observed in the mechanisation of Chapter 6, especially in the derivation of the Floyd-Warshall algorithm, where it is necessary to know the type and dimension of an array in order to eliminate projection functions in proofs.

For convenience, we list the contributions of this chapter.

1. Formalisation of indexed parallelism, including laws for its introduction and refinement (Section 5.1). The laws to reason about indexed parallelism are more involved than intuition.

2. New primitive command to specify programs that constrain environment steps. We illustrate its use by discussing the mechanised proof of law Strengthen-Rely-In-Context (Section 5.2).

3. Identification and patch of a problem in the abortive conditions of parallel composition (Section 5.3).

4. Formalisation of assignment to indexed positions of arrays (Section 5.4);

5. Introduction of an abstraction to reason about potential evaluations of expressions (Section 5.5).

6. Tailored laws to facilitate introduction of conditionals and loops in sequential programs (Section 5.6).

7. Minimalist type system to record meta-data in specifications (Section 5.7).

5.8.2 Further extensions

In general, code can only be derived from specifications that are stable with respect to interference. This has a practical impact on the way that specifications are formulated in rely-guarantee, since it limits the accuracy that can be enforced in the specification of a program. In certain situations, as in the specification of a consumer and producer [106, 30], the limited expressiveness implies that without auxiliary variables one cannot rule out implementations such as *skip*, or implementations that add multiple copies of an element to a buffer.

To increase expressiveness of specifications while preventing the use of auxiliary variables, what is needed (among other extensions) is an abstraction to specify intermediate properties that a process is required to establish independent of environment actions. The key idea is that each trace in the set of traces of **iprop** $q \cdot c$ should include a fragment $\pi\ (\sigma_i,\sigma_{i+1}),..., \pi(\sigma_{j-1}, \sigma_j)$ such that $\sigma_i, \sigma_j \models q$ holds, i.e. the relation q is established. The formulation of this property requires the use of an existential quantifier to allow the choice of i and j to vary according to the trace. For this reason, it is not clear to us how this abstraction can be introduced in a refinement calculus.

The concept of intermediate properties could be used, for example, to express that a producer is required to add a single element to a buffer, although the environment might remove the element before the insertion terminates. In practice, the current approach for dealing with situations where stability forces the specification to be too vague is to either: (i) select an initial specification that already has implementation constructs on it; or (ii) to use auxiliary variables to formulate a specification in a way that it satisfies the criteria of stability. The use of programming constructs (sequential composition, atomic blocks, parallel composition, etc.) to specify programs is illustrated in [32], while the usage of auxiliary variables is illustrated in [106, 30]. Considering these two alternatives, it can be argued that the use of programming constructors to describe specifications may lead to more clarity than the use of auxiliary variables to abstract algorithms.

Chapter 6

Applying the refinement calculus

> He who seeks for methods without having a definite problem in mind seeks for the most part in vain.
>
> David Hilbert

This chapter presents four derivations to illustrate different aspects of the rely-guarantee refinement algebra discussed in the previous chapters. These are taken from the literature on rely-guarantee: Findp (sequential and concurrent versions) comes from [48], Sieve comes from [65] and Floyd-Warshall comes from [32].

For Findp we present the derivation of a sequential and a concurrent version. The general structure of these refinements is the same presented in [48], but the discussion there has minor slips which cause some steps of refinement to be in short of assumptions to discharge proof obligations. We fix these derivations, and observe a potential good practice to be followed in subsequent refinements. We also use the derivation of a concurrent version of Findp to illustrate a refinement path which restricts the access to variables in a program using the novel laws presented in Section 3.11.

The derivation of a concurrent version for Findp shows that *interference* does not necessarily have a negative connotation: it can actually mean that programs *cooperate* to complete a task in faster time than a single program would be able to, i.e. programs can anticipate the termination of others. We also use Findp to discuss a limitation of the algebra to derive programs which depend on the environment to terminate. In short, the use of unfair parallelism as foundation for the algebra limits the ability of deriving this kind of program.

We use Sieve to illustrate the introduction of indexed parallelism in a context where rely and guarantee conditions are symmetric, i.e., they coincide. Sieve provides a classical example of data race, where programs *compete* to assign values to a shared variables without

explicit synchronisation. This kind of development is characteristic of programs that present some form of *idempotence* with respect to memory writings, i.e., programs compete to set a variable to the same value. Different from Findp, in Sieve the termination of one branch of parallel composition does not anticipate the termination of other branches.

Floyd-Warshall illustrates a development with asymmetric rely and guarantee conditions, and suggests that rely-guarantee may be useful to derive parallel versions of dynamic programming algorithms, such as solutions for the Knapsack problem. The derivation of Floyd-Warhsall motivated us to think about assignment to arrays (Section 5.4), and also revealed the weakness of representing the state as total function from variable names to *vvalue*. In short, this choice of representation imposes on the developer the need for including additional information in the precondition stating the type and dimension of an array; this would not be necessary if we had used a strongly typed language, where variables are declared. To cope with such weakness, we introduced a minimalist type system in Section 5.7. A comparison between the examples and a discussion about the bottlenecks of the mechanised theory are presented at the end of the chapter.

6.1 Typographic conventions

The examples in this chapter are automatically converted to Latex using a set of rules defined in Isabelle via the command **notation**. These rules encode notation conventions which allow for a more natural representation of programs, and prevent distracting the reader with details of the actual encoding in Isabelle. The motivation for formalising these conventions within Isabelle is that they ensure the consistence between the Isabelle theories and the examples as presented in this chapter. Changes in the theory are automatically reflected in parts of this thesis; moreover, the algorithms presented in the appendix are generated automatically from the Isabelle sources.

Conventions are activated using anti-quotations[1], via the new print modes *algorithm* and *expression*, which can be mutually active. The *algorithm* mode outputs Latex in a format compatible with the package *algorithmicx*[2], and the print mode *expression* converts annotated lambda expressions into predicates and simplify expressions by omitting quotes around the name of variables and constructors which are "obvious", such as datatype constructors *V* preceding the name of variables and *N* preceding values. To benefit from these print modes, programs need to be annotated with meta-data related to formatting. This meta-data does not

[1] For the definition of anti-quotation, see Section 1.6.4 on page 13.
[2] https://www.ctan.org/pkg/algorithmicx

interfere in proofs; they are completely erased by Isabelle in proofs after the user invoke the *simp* proof method.

The automatic conversion to Latex ensures that the text in this chapter reflects its formalisation in Isabelle. There is a price to pay for such a simplification of notation: the formalisation in Isabelle uses a richer representation with respect to the type of constants and unary and binary operators, which is elided in this chapter. Instead of explicitly displaying the type of each *vvalue* and the operands over *vvalue*, we mention in the relevant sections what is the type of the variables. For example:

- *vvalue* constructors are omitted, e.g. *5* instead of *VNat 5*;
- expressions are written using mathematical symbols rather than actual syntax, e.g. *len v* instead of $len_e\ (V\ ''v'')$;
- projection functions are omitted and binary relations are displayed in usual mathematical notation, e.g. $i \leq i'$ means $\lambda\ s\ s'.\ [\![s\ ''i'']\!]_i \leq [\![s'\ ''i'']\!]_i$;
- variable names are displayed without quotes, e.g. $x{:=}y$ means $''x'' := (V\ ''y'')$;
- the validity of a predicate in the after state is denoted by priming the predicate, e.g. *gi-satp* $\Rightarrow$ *gi-satp'* means *gi-satp* $\Rightarrow$ *gi-satp*‘;
- for readability, sometimes we write **parallel** c **and** d **end parallel** instead of $c \parallel d$, where c and d are programs;
- lambda expressions in the body of indexed parallelism is omitted, e.g. $\|_{i\in}\ S \cdot F\ i$ means $\|_S \cdot (\lambda\ i.\ F\ i)$;
- indentation is used to suppress parenthesis.

We omit the application of laws for reflexivity, transitivity, symmetry, monotonicity and substitution in the discussion that follows. These laws are at the very heart of the mechanisation and are used to navigate throughout the parts of a program under refinement, selecting specific components to be refined. Because the application of these laws is intuitive and is pervasive in the derivations, we do not mention them in the examples here. Refinement steps are marked using the notation $\mathscr{R}_i$ to allow reference to them in the text. This way of marking refinement steps is taken from [32].

To minimise repetition, we follow the convention adopted in the literature [78, 48] of marking the term to be refined next with $\lhd$. Thus, a refinement with no explicit left side applies

to the most recent command marked with ◁. Since commands can be nested into blocks which are split over more than one line, a single marker can be ambiguous to distinguish between the refinement of a nested command or the whole block. To avoid ambiguity, in the cases where a block is split over consecutive lines, and we are refining the whole block, all lines are marked with ◁.

6.2 Reading advice

To facilitate the reading of the material in this chapter, we summarised in Appendix B the laws that are referenced along the chapter. Those pages can be used as a reference sheet to speed up the reading of this chapter, as they prevent the tedious task of searching for laws in different chapters. Laws are identified by their respective number, and their names. We hope that after consulting a law a few times, it becomes easier to remember it from its name, thus minimising the need for switching between pages.

6.3 Findp: Sequential

This example is taken from [48] and illustrates the derivation of a sequential program. It motivates the introduction of arrays in RG-WSL, and serves to discuss the benefit of mechanical versus pen-and-paper derivations.

Findp finds the lowest index t in the domain of an array v, such that the indexed element $v[t]$ satisfies a predefined property p. In the implementation we are to develop, the search starts from the lowest index and moves upwards; thus, if it finds an indexed element that satisfies p, it terminates the search. If no element in the array satisfies p, then the algorithm sets t to be the position immediately after the last in the array. Recall that in RG-WSL arrays are zero-based. Thus, if the first element of the array v satisfies p, then t is set to 0; if no element in the array v satisfies p, then t is set to the length of the array[3].

6.3.1 Abbreviations

Table 6.1 introduces part of the abbreviations used in the derivation of the sequential version of Findp. Recall from Section 2.3.1 on page 25 that unary operators are defined via the Cartesian product of a function of type $vvalue \Rightarrow vvalue$ and a function of type $vvalue \Rightarrow \mathbb{B}$.

[3] We will reuse the definition of length of an array introduced in example 2.5 (*len*) on page 27 on Chapter 2.

The first denotes the shallow-embedded operator, whereas the second denotes its precondition (i.e. for which values the operation is well-defined). The unary operator P wraps the property p of Findp, so that it can be used in deep embedded expressions (*Exp*), such as the condition of an *if-then-else* command. We follow [48] and assume that p is well-defined for every element (x) of the searching array v. Undefinedness could be modelled by defining $T_P(x)$ to be a function different from the one we choose. We discuss at the end of this section the impact of modelling undefinedness for the property p. The operator p itself is introduced as an uninterpreted constant. That means that the derivation does not depend on the actual property under test.

$$
\begin{array}{rcl}
T_P(x) & \equiv & \mathit{True} \\
P(e) & \equiv & \mathit{UOp}\ (p, T_P)\ e \\
\mathit{notp}(v, s, t) & \equiv & \forall\, i{\in}s.\ i < t \longrightarrow \neg\ p\ (v\ !\ i) \\
\mathit{domain}(v) & \equiv & \{x \mid x < \mathit{length}\ v\} \\
\mathit{satp}(v, t) & \equiv & t \in \mathit{domain}(v) \wedge p\ (v\ !\ t) \\
\mathit{post}_0 & \equiv & (t' = \mathit{length}\ v \vee \mathit{satp}(v, t')) \wedge \mathit{notp}(v, \mathit{domain}(v), t') \\
\mathit{gi\text{-}satp} & \equiv & (t = \mathit{length}\ v) \vee \mathit{satp}(v, t) \\
\mathit{gi\text{-}notp} & \equiv & \mathit{notp}(v, \mathit{domain}(v), k) \wedge \mathit{bnd}(k, v)
\end{array}
$$

Table 6.1 Abbreviations for sequential Findp (Part I)

The predicate $\mathit{notp}(v, s, t)$ takes an array of natural numbers (v), a subset of the domain of the array (s), and an index (t). It states that there is no index (i) in s, such that $i < t$, for which $p\ (v\ !\ i)$ holds. The set $\mathit{domain}(v)$ represents the domain of the array v. The predicate $\mathit{satp}(v, t)$ is true if t is within the domain of v and $p\ (v\ !\ t)$ holds. The relation post_0 is used to define the initial specification for Findp. It demands the final value of t to be such that it either lies outside of the domain of the array, or it is least index in the domain of v such that $p\ (v\ !\ t)$ holds. The predicates *gi-notp* and *gi-satp* are used during the development to shift constraints from the initial postcondition to guarantee invariants.

The remaining abbreviations are given in Table 6.2. The predicate $\mathit{bnd}(k, v)$ is a loop invariant. It establishes boundaries the loop variable k. The predicate *init* is used to record information about the initialisation of variable t to be used at a later stage of the development. The relation *post-dec* records what is left to be done once initial restrictions from post_0 are shifted using guarantee invariants. The well-founded relation w is used to establish the termination of the loop used to iterate over the array. The boolean expressions *c-while* and *c-if* are used to introduce a loop and a conditional, respectively.

$$\begin{array}{rcl} \mathit{bnd}(k, v) & \equiv & 0 \leq k \wedge k \leq \mathit{length}\ v \\ \mathit{init} & \equiv & t \leq \mathit{length}\ v \\ \mathit{post\text{-}dec} & \equiv & t' \leq k' \\ w & \equiv & (0 \leq t' - k') \wedge (t' - k' < t - k) \\ \mathit{c\text{-}while} & \equiv & k < t \\ \mathit{c\text{-}if} & \equiv & P(v[k]) \end{array}$$

Table 6.2 Abbreviations for sequential Findp (Part II)

6.3.2 Derivation

This derivation of Findp is a revised version of the pen-and-paper version published in [48]. A problem with the original derivation is that it does not record relevant information about variables at the moment of their initialisation. This information is needed to discharge proof obligations in later steps of the derivation.

The initial specification for Findp states that it must terminate in a state where the variable t (a natural number) is in the domain of the array v, or t must store the value *length v* (that is, the position immediately after the last in the array). Moreover, there must not exist any index i in the domain of v that precedes t such that $P(v[i])$ holds. Formally, this is captured by the following specification:

$$\{t\}\colon [\mathit{true}, \mathit{post}_0] \tag{6.1}$$

The derivation discussed in this section is composed of five transformations. The first introduces and guarantee invariant (*gi-stap*) that requires t to store either the index immediately after the last in the array, or an index such that $P(v[t])$ holds. The second transformation introduces and initialises a loop counter k (k:=*0*), and a guarantee invariant (*gi-notp*) that requires that no index i ($0 \leq i < k$) exists such that $P(v[i])$ holds. The third transformation introduces a loop to increment k. The fourth transformation introduces a conditional, to detect if $P(v[t])$ holds, and the last transformation updates the loop counter and the variable t accordingly to the result of the conditional.

Initialising variable *t*

This refinement step initialises the variable t and introduces an invariant (*gi-satp*) to constrain updates to t. The step $\mathscr{R}_1$ splits $[\mathit{true}, \mathit{post}_0]$ into a sequential composition. For the application of law 3.25 (Sequential) we choose $q_0 \equiv \mathit{idset}\ \overline{\{t\}}$, $q_1 \equiv \mathit{gi\text{-}satp}' \wedge \mathit{post}_0$ and

$mid \equiv gi\text{-}satp \wedge init$. As these values can be visually extracted from the derivation chain, we generally do not mention them. Apart from few exceptions, we omit the proof of side conditions generated by the application of refinement laws. This is done to keep the discussion of the derivations at a reasonable size. All details of the derivation, including the omitted proof obligations can be find in Appendix A.1.

$$
\begin{array}{lll}
 & \{t\}\colon [true, post_0] & \\
\mathscr{R}_1 \equiv & \sqsubseteq \text{ by 3.25 (Sequential)} & \\
 & \{t\}\colon [true, idset\ \overline{\{t\}} \wedge (gi\text{-}satp \wedge init)'\,] \,;\, [gi\text{-}satp \wedge init, gi\text{-}satp' \wedge post_0] & \triangleleft
\end{array}
$$

Step $\mathscr{R}_2$ distributes the frame ($\{t\}$) over the nested sequential composition. The first component of the sequential composition ($\{t\}\colon [true, idset\ \overline{\{t\}} \wedge (gi\text{-}satp \wedge init)'\,]$) establishes the invariant and is implemented by the assignment $t{:=}\ len\ v$.

$$
\begin{array}{lll}
\mathscr{R}_2 \equiv & \sqsubseteq \text{ by 4.13 (Distribute-Frame-Sequential), 3.113 (Assignment-Guarantee)} & \\
 & t{:=}\ len\ v; & \\
 & \{t\}\colon [gi\text{-}satp \wedge init, gi\text{-}satp' \wedge post_0] & \triangleleft
\end{array}
$$

Step $\mathscr{R}_3$ strengthens the postcondition. Since law 3.19a (Consequence) is not aware of the surrounding frame, we have to include the frame in the postcondition to strengthen it, otherwise we end up with proof obligations that cannot be discharged. Following the strengthening step, we use law 4.12 (Trade-Spec-Guarantee) to eliminate the frame from the postcondition for the sake of clarity. Step $\mathscr{R}_4$ uses law 3.68 (Trade-Guarantee-Invariant) to turn the invariant into a guarantee invariant. Step $\mathscr{R}_5$ widens the precondition by removing the predicate *gi-satp* from it.

$$
\begin{array}{lll}
\mathscr{R}_3 \equiv & \sqsubseteq \text{ by 3.19a (Consequence), expanding } post_0 & \\
 & \{t\}\colon [gi\text{-}satp \wedge init, (t' = length\ v \vee satp(v, t')) \wedge notp(v, domain(v), t')] & \\
 & \sqsubseteq \text{ by 3.19a (Consequence) and Definition 2.19 (Post-state notation)} & \\
 & \{t\}\colon [gi\text{-}satp \wedge init, (idset\ \overline{\{t\}} \wedge gi\text{-}satp') \wedge notp(v, domain(v), t')] & \\
 & \sim \text{ by 4.9 (Trade-Spec-Frame)} & \\
 & \{t\}\colon [gi\text{-}satp \wedge init, gi\text{-}satp' \wedge notp(v, domain(v), t')] & \\
\mathscr{R}_4 \equiv & \sqsubseteq \text{ by 3.68 (Trade-Guarantee-Invariant)} & \\
 & \{t\}\colon \mathbf{guar{-}inv}\ gi\text{-}satp \cdot [gi\text{-}satp \wedge init, notp(v, domain(v), t')] &
\end{array}
$$

$\mathscr{R}_5 \equiv \sqsubseteq$ by 3.19a (Consequence)

$\{t\}$: **guar**−**inv** *gi-satp* · $[init, notp(v, domain(v), t')]$ ◁

The purpose of this transformation was to simplify the postcondition at the cost of introducing a guarantee invariant. This kind of transformation is useful to prepare for the introduction of loops, because it gives more flexibility to the user to be able to express a postcondition through a loop invariant, as intricate parts of the postcondition can be moved to a guarantee.

Introducing a loop counter

This step introduces a local variable (k) to iterate over the indices of the array. As we will develop a program that searches the array upwards, we initialise the local variable to value *0*.

The application of law 3.101 require us to prove that the variable k is *unrestricted* in **guar**−**inv** *gi-satp* · $[init, notp(v, domain(v), t')]$. This is shown by using the inductive rules for unrestriction given in Definition 2.81 (Unrestricted-Variable), and by invoking the definition of *depends-only* and logical interpretation ($\vdash$). Following the introduction of the local variable k, the frame is pushed into the guarantee invariant.

$\mathscr{R}_6 \equiv \sqsubseteq$ by 3.101 (Introduce-Variable-Frame), 3.62c (Distribute-Guarantee)

var k

guar−**inv** *gi-satp* · $\{k, t\}$: $[init, notp(v, domain(v), t')]$ ◁

Step $\mathscr{R}_7$ refines the body of the local variable block. It initialises the loop counter c with the lowest index of the array v. Additionally, it introduces and establish an invariant (*gi-notp*), which is turned into a guarantee invariant later. The motivation for the guarantee invariant is to make the postcondition simpler, and thus facilitate the introduction of a while loop. Appendix C.2 contains the proof of Proposition C.1, which is used to justify the proof obligation raised by the application of law 3.25 (Sequential) in the refinement step $\mathscr{R}_7$. The paper proof serves the purpose of illustrating how a non-trivial proof obligation can be discharged using the laws we provided and laws to manipulate quantifiers. In general, we omit the discussion of proof obligations in this chapter to keep the discussion focused on the design decision behind the derivation. All details of the derivations, including the omitted proof obligations can be find in Appendix A.1.

$\mathscr{R}_7 \equiv \sqsubseteq$ by law 3.25 (Sequential) and proposition C.1

$$\textbf{guar}-\textbf{inv}\ \mathit{gi\text{-}satp} \cdot$$
$$\{k,t\}\colon [\mathit{init}, (k' = 0) \wedge \mathit{idset}\ \overline{\{k,t\}} \wedge (\mathit{gi\text{-}notp} \wedge \mathit{init})']\,;$$
$$[\mathit{gi\text{-}notp} \wedge \mathit{init}, \mathit{gi\text{-}notp}' \wedge \mathit{post\text{-}dec} \wedge \mathit{idset}\ \overline{\{c,t\}}]$$

$\mathscr{R}_8 \equiv$ $\sqsubseteq$ by 3.62a (Distribute-Guarantee), 3.113 (Assignment-Guarantee)

$$k{:=}0\,;$$
$$\textbf{guar}-\textbf{inv}\ \mathit{gi\text{-}satp} \cdot \{k,t\}\colon [\mathit{gi\text{-}notp} \wedge \mathit{init}, \mathit{gi\text{-}notp}' \wedge \mathit{post\text{-}dec} \wedge \mathit{idset}\ \overline{\{k,t\}}]$$

$\mathscr{R}_9 \equiv$ $\sim$ by 4.12 (Trade-Spec-Guarantee), and 3.65 (Frame)

$$k{:=}0\,;$$
$$\textbf{guar}-\textbf{inv}\ \mathit{gi\text{-}satp} \cdot \{k,t\}\colon [\mathit{gi\text{-}notp} \wedge \mathit{init}, \mathit{gi\text{-}notp}' \wedge \mathit{post\text{-}dec}]$$

$\mathscr{R}_{10} \equiv$ $\sqsubseteq$ by 3.68 (Trade-Guarantee-Invariant), 3.62c (Distribute-Guarantee)

$$k{:=}0\,;$$
$$\textbf{guar}-\textbf{inv}\ \mathit{gi\text{-}satp} \cdot \textbf{guar}-\textbf{inv}\ \mathit{gi\text{-}notp} \cdot \{k,t\}\colon [\mathit{gi\text{-}notp} \wedge \mathit{init}, \mathit{post\text{-}dec}]$$

$\mathscr{R}_{11} \equiv$ $\sqsubseteq$ by 3.19a (Consequence)

$$k{:=}0\,;$$
$$\textbf{guar}-\textbf{inv}\ \mathit{gi\text{-}satp} \cdot \textbf{guar}-\textbf{inv}\ \mathit{gi\text{-}notp} \cdot \{k,t\}\colon [\mathit{init}, \mathit{post\text{-}dec}] \qquad \triangleleft$$

Introducing a while loop

This refinement introduces a while loop to search for the first element of v to satisfy the property p. In step $\mathscr{R}_{13}$, we are required to show that $w \equiv (0 \leq t' - k') \wedge (t' - k' < t - k)$ is well-founded from states satisfying *init*. As discussed in Section 2.12.3, we did not succeed in proving wellfoudness of relations in Isabelle/HOL. Thus, we take this proof obligation as an assumption of the derivation. It is easy however, to provide a convincing argument of why the relation w is well-founded: at each iteration, the gap between the variables k and t has to decrease, either because t decreases or because k increases. As the relation w does not allow t to become smaller than k, this process of reduction is finite and stops after, at most, $t - k$ iterations.

$\mathscr{R}_{12} \equiv$ $\sqsubseteq$ by 3.19a (Consequence)

$$\textbf{guar}-\textbf{inv}\ \mathit{gi\text{-}satp} \cdot \textbf{guar}-\textbf{inv}\ \mathit{gi\text{-}notp} \cdot$$
$$\{k,t\}\colon [\mathit{init}, \mathit{init}' \wedge \neg \mathit{c\text{-}while}' \wedge w^{**}]$$

$\mathscr{R}_{13} \equiv$ $\sqsubseteq$ by 5.32 (Sequential-Loop)

guar$-$**inv** *gi-satp* $\cdot$ **guar**$-$**inv** *gi-notp* $\cdot$

$\{k, t\}$: (**while** *c-while* **do** $[init \wedge c\text{-}while, w]$)

$\mathscr{R}_{14} \equiv$ $\sqsubseteq$ by 3.62f (Distribute-Guarantee)

while *c-while* **do**

guar$-$**inv** *gi-satp* $\cdot$ **guar**$-$**inv** *gi-notp* $\cdot$ $\{k, t\}$: $[init \wedge c\text{-}while, w]$ $\triangleleft$

At this point, it is worth to look back at the remark added to law 3.112 (Rely-Loop) on page 113. That observation makes easier for a novice to understand how to introduce loops in programs, and also applies to law 5.32 (Sequential-Loop).

Introducing a conditional

This transformation introduces a conditional to assess if the position k of the array v satisfies the property p. If it does, then it requires t to be updated with the value of k, which causes the loop to terminate; otherwise, it requires k to be incremented by one. The increment allows the next position of v to be assessed during the next iteration of the loop.

Before introducing the conditional we need to complement the syntax of concrete expressions with an operator to read an indexed position of an array. Recall from Section 2.3.2 (Expression language) that unary and binary operators can be defined on demand. The operator we need can be modelled via the type $Exp \Rightarrow Exp \Rightarrow Exp$.

The first argument taken by this operator is an expression that denotes an array. The second argument is an expression that denotes the first index to be applied to the array. Thus, to access the content at position k of an array v we write $v[k]$. Indexation is defined as[4]

$$e_1[e_2] \equiv BOp\ (\lambda va\ vb.\ [\![va]\!]_a\ !\ [\![vb]\!]_n, \lambda va\ vb.\ [\![vb]\!]_n < length\ [\![va]\!]_a)\ e_1\ e_2 \qquad (6.2)$$

This presentation style is the same used to discuss the encoding of mod_n on page 27. The leftmost lambda expression formalises the underlying mathematical operator, that is, it returns the indexed element of list denoted by the array. The second lambda expression introduces the precondition for the underlying mathematical operation to succeed: the position to be retrieved must be in the domain of the array. Recall from Section 2.3.3 (Definedness) on

[4]For convenience, we present this operator using the standard notation (e.g. $A[i]$) in this thesis, while in Isabelle/HOL we use a different syntax ($A\ \sharp_e\ i$) to prevent ambiguity with the existing notation for lists.

page 27 that if a binary expression fails to satisfy the precondition of its underlying binary operator, then the evaluation results in *undefined*.

Refinement step $\mathscr{R}_{15}$ illustrates the introduction of a conditional. To discharge the proof obligations involved in the application of law 5.31 (Sequential-Conditional) we are required to prove, assuming the precondition to hold, that the condition (*c-if*) of the *if-then-else* command is defined. Formally, we have to prove that

$$\vdash \mathit{init} \wedge \mathit{c\text{-}while} \Rightarrow \mathit{defined\ c\text{-}if}$$

Expanding *c-while*, *init* and *c-if*, and applying *defined* (Definition 2.6) this becomes:

$$\vdash (t \leq \mathit{length}\ v) \wedge (k < t) \Rightarrow (k < \mathit{length}\ v)$$

This highlights the importance of recording and propagating knowledge about initialisation of variables, as well as the importance of machine checked proofs. Had the precondition not included *init*, we would not be able to prove that k is in the domain of the array v. The pen-and-paper derivation [48] did not record and propagate such information.

$\mathscr{R}_{15}$ $\equiv$ $\sqsubseteq$ by 5.31 (Sequential-Conditional)

guar$-$**inv** *gi-satp* $\cdot$ **guar**$-$**inv** *gi-notp* $\cdot$ $\lhd$

$\{k, t\}$: **if** *c-if* **then** $[\mathit{init} \wedge \mathit{c\text{-}while} \wedge \mathit{c\text{-}if}, w]$ **else** $[\mathit{init} \wedge \mathit{c\text{-}while} \wedge \neg \mathit{c\text{-}if}, w]$ $\lhd$

Aside. *In the cases where a block is split over consecutive lines, and we are refining the whole block, all lines are marked with* $\lhd$*. This is done to prevent ambiguity, as explained on Section 6.1.*

Step $\mathscr{R}_{16}$ distributes the frame $\{k, t\}$ to the branches of the conditional; reduces the frame for each branch separately; replicates the frames inside of the postconditions; strengthens the postconditions; and removes the frames from the postconditions.

$\mathscr{R}_{16}$ $\equiv$ $\sqsubseteq$ by 3.62e (Distribute-Guarantee), 3.60a (Guarantee-Monotonic),
4.12 (Trade-Spec-Guarantee), 3.19a (Consequence)

if *c-if* **then** $\lhd$

guar$-$**inv** *gi-satp* $\cdot$ **guar**$-$**inv** *gi-notp* $\cdot$ $\lhd$

$\{t\}$: $[\mathit{init} \wedge \mathit{c\text{-}while} \wedge \mathit{c\text{-}if}, t' = k]$ $\lhd$

else ◁

guar−**inv** *gi-satp* · **guar**−**inv** *gi-notp* · ◁

$\{k\}$: $[init \wedge c\text{-}while \wedge \neg c\text{-}if, k' = k + 1]$ ◁

Introducing assignments

Step $\mathscr{R}_{17}$ is the last in the derivation. It implements the specifications in the branches of the conditional using assignments. Again, we stress the importance of recording and propagate information about the state. Had the precondition not included *init* and *c-while*, it would not be possible for us to discharge the proof obligations of law 3.113 (Assignment-Guarantee), used to introduce the assignment.

$\mathscr{R}_{17} \equiv \sqsubseteq$ by 3.113 (Assignment-Guarantee)

if *c-if* **then** t:=k **else** k:=k+*1*

In the case of the *then-branch*, the lack of information about the initialisation of t and its relationship to v leads to the situation discussed in the section *Example: reasoning compositionally about logical interpretation* on page 38. In short, we cannot prove that the assignment satisfies the guarantee invariant *gi-satp* if we do not record the relationship between k, t and *length* v in the precondition.

In [48], the authors attempt to avoid this problem by flattening the nested guarantee invariants: this is not a valid transformation, because nested guarantee invariants impose a stronger requirement on its body than flattened guarantee invariants. See remark on law 3.69 (Distribute-Guarantee-Invariant) on page 92 for a comparison between flattened versus nested guarantee-invariants.

6.3.3 Discussion

The derivation of the sequential version of Findp suggests that refraining from widening preconditions in the course of a derivation may be a good practice, because to discharge proof obligations one may depend on information encoded by preconditions. Additionally, a too weak choice for *mid* in the application of law 3.25 (Sequential) may leave the subsequent precondition without information about the preceding part of a program. In the derivation of sequential Findp, the choice of *mid* in the first refinement step is decisive to allow the introduction of the conditional. Thus, care should be taken to preserve the maximum of

information in the course of a derivation. This is simple to follow in practice: (i) whenever applying laws 3.25 and 3.84, the user should choose *mid* such that it transfers the maximum of information from the postcondition of the preceding specification to the precondition of the subsequent specification; (ii) whenever applying laws 3.111 and 3.112 to introduce control structures, the user should choose b_0 and b_1 such that they encode the strongest stable predicate that can be inferred from the condition of the control structure being introduced.

Although we had not experimented to develop a version of Findp considering the property p to be undefined for certain elements of the array, it is not difficult to see what would be the outcome of such modelling. It would impact on the introduction of the conditional, because in step $\mathscr{R}_{15}$ we are required to prove from the precondition that the boolean expression tested by the *if-then-else* is defined. To ensure that $P(v[k])$ is well defined, we would need to use nested conditionals: the external to test if the property p is defined over the element $v[k]$, and the nested one to test if that element satisfies the property p.

Findp also serves to discuss how we pulled apart the notion of undefinedness of the algebra from that of the underlying logic (Isabelle/HOL). For example, consider again indexation over arrays (e.g. $v[k]$). The underlying mathematical representation for this operator is indexation over lists in Isabelle, which is represented by exclamation mark (!), and happens to be defined for empty lists in Isabelle/HOL. For example, in Isabelle/HOL one can prove that the following equality holds, even if k is higher than the size of the list v.

$$v\ !\ k = v\ !\ k$$

Even though this expression is defined in the underlying logic, we cannot derive programs that use it in a test or an assignment, for example. This is because the formal definition given to the indexation operator does not allow the application of an index to an non-existent position of an array. Thus, the attempt to refine a program which uses this expression as a boolean condition would generate a proof obligation that cannot be discharged. It is the user's task when modelling operators to decide if she wants to reflect the underlying logic or be more strict with respect to undefinedness, as we were in this case.

Finally, it is important to remember that our representation of states does not record the dimension of variables. Perhaps, if we had used a richer representation for the state, we would not need to care about recording as much information along the derivation as we have to, because we would have at our disposal the type and dimension of variables to be used when discharging proof obligations. For this example, though, a richer representation of the state would not make a difference.

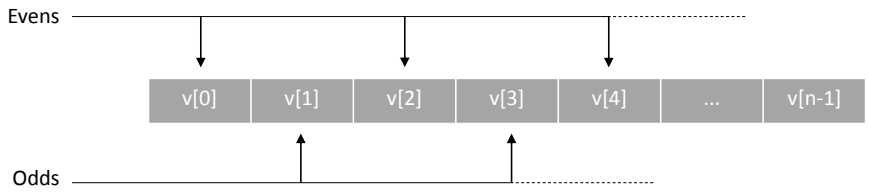


Figure 6.1 Index partitioning for concurrent Findp

6.4 Findp: Concurrent

The concurrent version of Findp illustrates the development of a non-trivial parallel search algorithm involving relies, guarantees, parallelism, local variables, conditionals and iteration. The key insight of this development is to partition the set of indices into even and odd numbers and use two processes to perform the search, as illustrated in Figure 6.1. These programs cooperate to determine the lowest index t to satisfy a property p. To prevent data races in the access to t, each program maintains a local instance of t (et, ot), which is used to update the value of the global t just before the algorithm terminates. These variables hold, for each program, the lowest index known to satisfy p within its range of indices. Initially, both programs assume that no element satisfies p (i.e. $et = ot = length\ v$).

Two local variables are used to iterate over the array, ek and ok. The search starts from the lowest indices ($ek = 0$ and $ok = 1$) and move towards the last position of the array. Thus, if a program finds an index such that the property p holds, it is surely the lowest index in the range searched by that program. Before programs move to the next searching position, they consult the local instance of t hold by their sibling. If that instance holds a value that is smaller than the next position to be searched, then there is no need to continue searching: the lowest index to satisfy p has been found by the sibling program and the search is interrupted, i.e. the unsuccessful branch anticipates its termination. If the sibling program holds a higher value for the local instance of t than the next position to be searched, then the search continues.

A relevant observation about the concurrent version of Findp is that each parallel program is responsible for establishing its own termination. Programs can cooperate to terminate earlier, but no individual program depends on actions of the environment to terminate (i.e., processes are wait-freedom). As it was discussed in Section 2.12.4, the algebra presented in this thesis is not suitable for developing programs whose termination necessarily depends on actions of the environment.

The issue involving initialisation of variables already discussed in Section 6.3 occurs in the derivation of the concurrent version of Findp published in [48] as well. The derivation is fixed following similar steps to those discussed.

6.4.1 Abbreviations

This section introduces definitions used to formalise the derivation of a concurrent version of Findp. Most of the definitions are parameterised versions of those used in the derivation of the sequential version of Findp. Parameterisation is useful to prevent duplication of definitions that are common to the development of the parallel processes searching *odd* and *even* numbers. For convenience, the definitions used in the sequential and concurrent versions of Findp are presented side by side on Appendix C.3.2. We leave the comparison between the development of the sequential and concurrent version to be discussed at the end of this section. The representation as parameterised abbreviations serves to reinforce the symmetry of the development of the *odd* and *even*-branch of the parallel composition and justify why we have not included the full development in this chapter, but just the development of the *odd*-branch.

$$
\begin{array}{rcl}
T_P(x) & \equiv & \mathit{True} \\
P(e) & \equiv & \mathit{UOp}\ (p, T_P)\ e \\
\mathit{notp}(v, s, t) & \equiv & \forall i{\in}s.\ i < t \longrightarrow \neg\ p\ (v\ !\ i) \\
\mathit{domain}(v,\Phi) & \equiv & \{x \mid x < \mathit{length}\ v \wedge \Phi(x)\} \\
\mathit{satp}(v, t, \Phi) & \equiv & t \in \mathit{domain}(v,\Phi) \wedge p\ (v\ !\ t) \\
\mathit{post}_0(x, y, \Phi) & \equiv & ((\mathit{min}(x', y') = \mathit{length}\ v) \vee \mathit{satp}(v, \mathit{min}(x', y'), T_P)) \wedge \\
 & & \mathit{notp}(v, \mathit{domain}(v,\Phi), \mathit{min}(x', y')) \\
\mathit{gi\text{-}satp} & \equiv & (\mathit{min}(ot, et) = \mathit{length}\ v) \vee \mathit{satp}(v, \mathit{min}(ot, et), T_P) \\
\mathit{gi\text{-}notp}(x, \Phi) & \equiv & \mathit{notp}(v, \mathit{domain}(v,\Phi), x) \wedge \mathit{bnd}(x, v) \wedge \Phi(x)
\end{array}
$$

Table 6.3 Abbreviations for concurrent Findp (Part I)

Table 6.3 introduces part of the abbreviations used in the derivation. The function $T_P(x)$ states that the property p of Findp is defined for every element x in the searching array. The unary operator $P(e)$ encapsulates the property p using the type introduced Definition 2.3 on page 26. The predicate $\mathit{notp}(v, s, t)$ takes an array of natural numbers (v), a subset of the domain of the array (s), and an index (t). It states that there is no index (i) in s, such that $i < t$, for which $p\ (v\ !\ i)$ holds. The definition $\mathit{domain}(v,\Phi)$ takes two parameters: an array of natural numbers (v), and a predicate over naturals (Φ). The predicate (Φ) is used to select a specific subset of the domain; e.g. we start with an abstract specification where $\Phi = T_P$, but during the split of the specification into a parallel composition we instantiate Φ to the predicates *odd* and *even*. The predicate $\mathit{satp}(v, t, \Phi)$ states that $v[t]$ satisfies the property p, and t is within $\mathit{domain}(v,\Phi)$. The relation $\mathit{post}_0(x, y, \Phi)$ is a generalised version of the

homonymous relation used in the derivation of a sequential version of *Findp*, and is used to define the top-level specification for the algorithm (Eq. 6.3). The parameters x and y in this definition are variables, used to store the result of the main computation performed by Findp. The predicates *gi-satp* and *gi-notp*(x, Φ) are used during the derivation to introduce guarantee invariants, using the same strategy applied in the derivation of the sequential version of Findp.

The predicate $bnd(k, v)$ is a loop invariant that establishes a lower and upper bound for the loop variable k. The predicate $init(t)$ is used to record information about the initialisation of t. The relation *post-dec*(x) is used to record what is left to be done once most of the original specification is shifted into guarantee invariants. The well-founded relation $w(t, k)$ is used to ensure the termination of the loops used to iterate over the array. The boolean expressions *c-while*(k) and *c-if*(k) are parameterised on a loop variable (k), and are used to introduce a loop and a conditional, respectively. The predicates $b_0(k, t)$ and $b_1(k, ot, et)$ are used for loop introduction in the context of an interfering environment (cf. law 3.111). The predicate $prew(k, t)$ is used to record the precondition of the body of a while loop.

$$
\begin{array}{rcl}
bnd(k, v) & \equiv & 0 \leq k \wedge k \leq length\ v + 1 \\
init(t) & \equiv & t \leq length\ v \\
post\text{-}dec(x) & \equiv & min(ot', et') \leq x' \\
w(t, k) & \equiv & (0 \leq t' - k' + 1) \wedge (t' - k' < t - k) \\
c\text{-}while(k) & \equiv & k < ot \wedge k < et \\
c\text{-}if(k) & \equiv & P(v[k]) \\
b_0(k, t) & \equiv & k < t \\
b_1(k, ot, et) & \equiv & (ot \leq k) \vee (et \leq k) \\
prew(k, t) & \equiv & b_0(k, t) \wedge init(t) \\
g(x, y) & \equiv & (x' \leq x) \wedge idset\ \{y\}
\end{array}
$$

Table 6.4 Abbreviations for concurrent Findp (Part II)

The relation $g(x, y)$ plays two roles during the development. It represents both the guarantee and rely conditions for the development of each branch. It states that the variable taken as first parameter (x) decreases monotonically, while the second parameter (y) is kept stable. Depending on the intended use, the parameters x and y are instantiated to appropriate variables.

6.4.2 Derivation

The derivation is structured in six transformations. The main insight is to split the search into two processes: one to search *odd* indices, and the other to search *even* indices. The first transformation splits the result variable (t) in two local instances (ot and et); this is done to prevent the need for synchronising access to the variable t. The second transformation splits the (sequential) specification in a parallel composition of specifications. For conciseness, the remaining transformations focus on the development of the *odd* branch. The third transformation introduces a loop counter for the *odd* branch (ok); the fourth transformation introduces a *while* loop to iterate over *odd* indices. The fifth transformation introduces a conditional to checks if the current position of the array ($v[ok]$) satisfies the property p, and the last transformation updates the loop variable and the the local copy of the result variable (ot).

At the end of this section we discuss an alternative derivation path, which could be used to justify the derivation using an argument based on syntactic restrictions over variables.

Preventing data races

The strategy to implement a sequential version of Findp was to use a local variable to search the array upwards from the first position up to the max index, and upon identification of the first element to satisfy the property p to update t with its index. Now that we will split the search between two processes, there is a danger of competition (i.e. data-races) for accessing the variable t. The purpose of this refinement step is to eliminate this danger.

The general idea is to split the global variable t in two local variables, et and ot. The redundancy is used to prevent complications in the algorithm that would arise if we had to synchronise the access to the variable t between the two processes. The abstract specification (6.1) for this derivation is the equivalent to the one used to derive the sequential version, and is provided next.

$$\{t\}\colon [true, post_0(t, t, T_P)] \tag{6.3}$$

Since this specification is not nested into a rely command, we can use law 3.101 (Introduce-Variable-Frame) to introduce local variables. Instead of following this path, we will take a different one to illustrate a law that has not been exercised yet. Step $\mathscr{R}_1$ introduces a rely condition, and step $\mathscr{R}_2$ applies law 3.102 (Introduce-Variable-Rely) twice to introduce the local variables ot and et. This law uses the rely condition to record that local variables are not subject to external interference. Each application generates a proof obligation involving

unrestriction. For example, for the first application this proof obligation is

$$\mathit{unrest}(\mathit{ot}, \textbf{rely}\ \mathit{idset}\ \{v, t\} \cdot [\mathit{true}, \mathit{post}_0(t, t, T_P)])$$

$\{t\}$: $[\mathit{true}, \mathit{post}_0(t, t, T_P)]$

$\mathscr{R}_1 \equiv \sqsubseteq$ by 3.81 (Rely-Idrel-Specification) and 3.87a (Rely-Monotonic)

$\{t\}$: **rely** $\mathit{idset}\ \{v, t\} \cdot [\mathit{true}, \mathit{post}_0(t, t, T_P)]$

$\mathscr{R}_2 \equiv \sqsubseteq$ by 3.102 (Introduce-Variable-Rely)

var ot

var et

$\{t, \mathit{ot}, \mathit{et}\}$: **rely** $\mathit{idset}\ \{v, t, \mathit{ot}, \mathit{et}\} \cdot [\mathit{true}, \mathit{post}_0(t, t, T_P)]$ ◁

Step $\mathscr{R}_3$ reformulates the original specification in terms of *ot* and *et*.

$\mathscr{R}_3 \equiv \sqsubseteq$ by 3.19a (Consequence)

$\{t, \mathit{ot}, \mathit{et}\}$: **rely** $\mathit{idset}\ \{v, t, \mathit{ot}, \mathit{et}\} \cdot [\mathit{true}, \mathit{post}_0(\mathit{ot}, \mathit{et}, T_P) \wedge (t' = \mathit{min}(\mathit{ot}', \mathit{et}'))]$ ◁

The next design decision is to separate the algorithm into two phases: computation of the final values of the local instances *et* and *ot*, and update of the global variable *t*. This is done in step $\mathscr{R}_4$. Care is necessary when splitting a specification using law 3.89 (Rely-Sequential): the composed specification must not be weaker than the original. To illustrate this point, consider the refinement step $\mathscr{R}_4$ given next.

$\mathscr{R}_4 \equiv \sqsubseteq$ by 3.89 (Rely-Sequential)

$\{t, \mathit{ot}, \mathit{et}\}$: **rely** $\mathit{idset}\ \{v, t, \mathit{ot}, \mathit{et}\} \cdot [\mathit{true}, \mathit{post}_0(\mathit{ot}, \mathit{et}, T_P)]$; ◁

rely $\mathit{idset}\ \{v, t, \mathit{ot}, \mathit{et}\} \cdot [\mathit{true}, (t' = \mathit{min}(\mathit{ot}, \mathit{et})) \wedge \mathit{idset}\ \{\mathit{ot}, \mathit{et}\}]$ ◁

If we had chosen a weaker q_1, such as

$$q_1 \equiv (t' = \mathit{min}(\mathit{ot}, \mathit{et}))$$

we would end up with the next proof obligation

$$\vdash \mathit{post}_0(\mathit{ot}, \mathit{et}, T_P)\ ;\ q_1 \Rightarrow \mathit{post}_0(\mathit{ot}, \mathit{et}, T_P) \wedge (t' = \mathit{min}(\mathit{ot}', \mathit{et}'))$$

The problem to discharge that proof is that the right-hand side of the implication constrains the final value of *ot* and *et*, but q_1 leaves the value of these variables unrestricted. Consequently, the relational composition $(post_0(ot, et, T_P)\ ;\ q_1)$ also leaves these values unrestricted, and therefore the implication does not hold. The choice we made was

$$q_1 \equiv (t' = min(ot, et)) \wedge idset\ \{ot, et\}$$

which states that *ot* and *et* remain unchanged. The next refinement step implements the right-hand side specification using law 3.114 (Assignment-Rely-Guarantee).

$\mathscr{R}_5 \equiv$ $\sqsubseteq$ by 4.13 (Distribute-Frame-Sequential), 3.87a (Rely-Monotonic) and 3.114 (Assignment-Rely-Guarantee)

$\{ot, et\}$: **rely** $idset\ \{v, ot, et\} \cdot [true, post_0(ot, et, T_P)]$; ◁

$t{:=}min(ot, et)$

Refinement steps $\mathscr{R}_6$-$\mathscr{R}_8$ initialise the local variables *et* and *ot*. They differ from the derivation published in [48] with respect to recording and propagating information about the initialisation of variables. Likewise the sequential development, we use the relations $init(ot)$ and $init(et)$ to record relevant information about the initial values of *ot* and *et*. The predicate *gi-satp* will be used later to introduce a guarantee invariant. Its purpose will be to simplify the postcondition, by shifting restrictions from the postcondition to the guarantee.

$\mathscr{R}_6 \equiv$ $\sqsubseteq$ by 3.89 (Rely-Sequential) and 4.13 (Distribute-Frame-Sequential)

$\{ot, et\}$: **rely** $idset\ \{v, ot, et\} \cdot [idset\ \{v\} \wedge (init(ot) \wedge init(et) \wedge gi\text{-}satp)^{\prime}]$;

$\{ot, et\}$: **rely** $idset\ \{v, ot, et\} \cdot [init(ot) \wedge init(et) \wedge gi\text{-}satp, post_0(ot, et, T_P)]$

$\mathscr{R}_7 \equiv$ $\sqsubseteq$ by 3.89 (Rely-Sequential) and 4.13 (Distribute-Frame-Sequential)

$\{ot\}$: **rely** $idset\ \{v, ot, et\} \cdot [idset\ \{et, v\} \wedge ((ot' = length\ v) \vee satp(v, ot', odd))]$;

$\{et\}$: **rely** $idset\ \{v, ot, et\} \cdot [idset\ \{ot, v\} \wedge ((et' = length\ v) \vee satp(v, et', even))]$;

$\{ot, et\}$: **rely** $idset\ \{v, ot, et\} \cdot [init(ot) \wedge init(et) \wedge gi\text{-}satp, post_0(ot, et, T_P)]$

$\mathscr{R}_8 \equiv$ $\sqsubseteq$ by 3.59 (Introduce-Guarantee) and 3.114 (Assignment-Rely-Guarantee)

$ot{:=}len(v)$;

$et{:=}len(v)$;

$\{ot, et\}$: **rely** $idset\ \{v, ot, et\} \cdot [init(ot) \wedge init(et) \wedge gi\text{-}satp, post_0(ot, et, T_P)]$ ◁

We continue the derivation strengthening the specification command to introduce the invariant *gi-satp* and remove restrictions imposed by $post_0(ot, et, T_P)$. The information encoded by the rely and guarantee conditions is used to allow the strengthening of the postcondition in a way that appears to be locally weakening the postcondition. Failure to include such information in the postcondition leads the user to face proof obligations that cannot be discharged, such as the one discussed in step $\mathscr{R}_4$.

$\mathscr{R}_9 \equiv \sqsubseteq$ by 3.19a (Consequence)

$\{ot, et\}$: **rely** *idset* $\{v, ot, et\}$ ·

$[init(ot) \wedge init(et) \wedge gi\text{-}satp, gi\text{-}satp' \wedge notp(v, domain(v,T_P), min(ot', et')) \wedge$

$(idset\ \{v, ot, et\} \vee idset\ \overline{\{ot, et\}})^{**}]$

$\mathscr{R}_{10} \equiv \sqsubseteq$ by 3.91 (Introduce-Rely-Guar-Invariant)

$\{ot, et\}$: **guar**$-$**inv** *gi-satp* · **rely** *idset* $\{v, ot, et\}$ · ◁

$[init(ot) \wedge init(et) \wedge gi\text{-}satp, notp(v, domain(v,T_P), min(ot', et')) \wedge$ ◁

$(idset\ \{v, ot, et\} \vee idset\ \overline{\{ot, et\}})^{**}]$ ◁

After introducing a guarantee invariant we apply law 3.93 (Trade-Rely-Guarantee) to eliminate the closure of the information encoded by the guarantee and rely conditions from the specification command.

$\mathscr{R}_{11} \equiv \sqsubseteq$ by 3.67 (Distribute-Guarantee-Frame), 3.93 (Trade-Rely-Guarantee) and 3.19a (Consequence)

guar$-$**inv** *gi-satp* · $\{ot, et\}$: **rely** *idset* $\{v, ot, et\}$ · ◁

$[init(ot) \wedge init(et), notp(v, domain(v,T_P), min(ot', et'))]$ ◁

Introducing parallelism

This transformation is the epitome of rely-guarantee refinement. Step $\mathscr{R}_{12}$ splits the specification command into two parallel branches. Since the specification is nested within a rely command, it would not be wise to apply law 3.95 (Introduce-Parallel-Spec) to introduce parallelism, because this would lead to a program where a guarantee is nested into the scope of a rely command. Instead, a law that takes into account the wrapping rely command should to be used. We illustrate the introduction of parallelism via law 3.97 (Introduce-Parallel-Spec-Nested).

$\mathscr{R}_{12} \equiv \sqsubseteq$ by 3.97 (Introduce-Parallel-Spec-Nested)

guar−**inv** *gi-satp* · $\{ot, et\}$: $\triangleleft$

parallel $\triangleleft$

guar $g(ot, et) \wedge idset\ \{v\}$ · $\triangleleft$

rely $g(et, ot) \wedge idset\ \{v\} \vee idset\ \{v, ot, et\}$ · $\triangleleft$

$[init(ot), notp(v, domain(v,odd), min(ot', et'))]$ $\triangleleft$

and $\triangleleft$

guar $g(et, ot) \wedge idset\ \{v\}$ · $\triangleleft$

rely $g(ot, et) \wedge idset\ \{v\} \vee idset\ \{v, ot, et\}$ · $\triangleleft$

$[init(et), notp(v, domain(v,even), min(ot', et'))]$ $\triangleleft$

end parallel $\triangleleft$

Following the introduction of parallelism, the guarantee invariant and frame are distributed over the branches of parallel composition via step $\mathscr{R}_{13}$. Such distribution allows the user to continue benefiting from the information contained in the frame and in the guarantee invariants to strengthen postconditions. Moreover, it prevents the accidental introduction of assignment without observing the guarantee invariant. Should this happen, the user needs to expand the definition of assignment and distribute the guarantee invariant over it, showing that the atomic update which composes the assignment respects the guarantee invariant, in order to eliminate the guarantee invariant.

$\mathscr{R}_{13} \equiv \sqsubseteq$ by 3.62b (Distribute-Guarantee), 3.87a (Rely-Monotonic),
3.62c (Distribute-Guarantee), 3.60a (Guarantee-Monotonic)

parallel

$$\begin{array}{l}
\quad \textbf{guar}-\textbf{inv}\ gi\text{-}satp \cdot \{ot, et\}\text{: }\textbf{guar}\ g(ot, et) \cdot \quad \triangleleft \\
\qquad \textbf{rely}\ g(et, ot) \wedge idset\ \{v\} \cdot \quad \triangleleft \\
\qquad\quad [init(ot), notp(v, domain(v,odd), min(ot', et'))] \quad \triangleleft \\
\textbf{and} \\
\quad \textbf{guar}-\textbf{inv}\ gi\text{-}satp \cdot \{ot, et\}\text{: }\textbf{guar}\ g(et, ot) \cdot \\
\qquad \textbf{rely}\ g(ot, et) \wedge idset\ \{v\} \cdot \\
\qquad\quad [init(et), notp(v, domain(v,even), min(ot', et'))] \\
\textbf{end parallel}
\end{array}$$

Introducing loop counters

This transformation introduces two loop counters, one per each branch of the parallel composition. Without compromising the discussion, we focus on the development of the *odds*-branch. The reason is because the development of the *evens*-branch mirrors that of the *odds*-branch. Intermediate steps of the full-development can be seen in Appendix C.3.

Step $\mathscr{R}_{14}$ simplifies the specification by removing a variable from the frame and strengthening the guarantee condition. Each law and definition mentioned in this step is used twice: the first time to unfold and merge the guarantee and frame; the second time to split the simplified guarantee into a frame nested inside of a guarantee.

$$\begin{array}{l}
\mathscr{R}_{14} \equiv \sim \text{by equivalence } (idset\ \overline{\{ot, et\}} \wedge g(ot, et) = idset\ \overline{\{ot\}} \wedge (ot' \leq ot)), \\
\qquad\qquad 3.62\text{c (Distribute-Guarantee) and 3.65 (Frame)} \\
\quad \textbf{guar}-\textbf{inv}\ gi\text{-}satp \cdot \textbf{guar}\ (ot' \leq ot) \cdot \quad \triangleleft \\
\qquad \{ot\}\text{: }\textbf{rely}\ g(et, ot) \wedge idset\ \{v\} \cdot \quad \triangleleft \\
\qquad\quad [init(ot), notp(v, domain(v,odd), min(ot', et'))] \quad \triangleleft
\end{array}$$

The next transformation introduces a local variable, *ok*, used in later stages of the derivation to iterate over the array *v*.

$$\begin{array}{l}
\mathscr{R}_{15} \equiv \sqsubseteq \text{ by 3.102 (Introduce-Variable-Rely)} \\
\quad \textbf{guar}-\textbf{inv}\ gi\text{-}satp \cdot \textbf{guar}\ (ot' \leq ot) \cdot \quad \triangleleft \\
\quad \textbf{var}\ ok \cdot \{ok, ot\}\text{: }\textbf{rely}\ g(et, ot) \wedge idset\ \{ok, v\} \cdot \quad \triangleleft \\
\qquad [init(ot), notp(v, domain(v,odd), min(ot', et'))] \quad \triangleleft
\end{array}$$

The next transformation distributes the guarantee, guarantee invariant and frame over the local variable block. The distribution of these commands over the local variable block allows the user to continue benefiting from the information encoded by them while strengthening the postcondition.

$\mathscr{R}_{16} \equiv$ $\sqsubseteq$ by 3.62c (Distribute-Guarantee) and 4.11 (Dist-Guarantee-Var)

var *ok*

guar−**inv** *gi-satp* · **guar** $(ot' \leq ot)$ · $\triangleleft$

$\{ok, ot\}$: **rely** $g(et, ot) \wedge idset\ \{ok, v\}$ · $\triangleleft$

$[init(ot), notp(v, domain(v,odd), min(ot', et'))]$ $\triangleleft$

The next step introduces a guarantee invariant. The guarantee invariant allow us to express the postcondition in a simpler way at the cost of adding atomic restrictions to the implementation.

$\mathscr{R}_{17} \equiv$ $\sqsubseteq$ by 3.89 (Rely-Sequential),4.13 (Distribute-Frame-Sequential), 3.62a (Distribute-Guarantee) and 3.64 (Guarantee invariant)

guar−**inv** *gi-satp* · **guar** $(ot' \leq ot)$ ·

$\{ok, ot\}$: **rely** $g(et, ot) \wedge idset\ \{ok, v\}$ ·

$[init(ot), idset\ \{v\} \wedge (init(ot) \wedge gi\text{-}notp(ok, odd))']$;

guar−**inv** *gi-satp* · **guar** $(ot' \leq ot)$ ·

$\{ok, ot\}$: **rely** $g(et, ot) \wedge idset\ \{ok, v\}$ ·

$[init(ot) \wedge gi\text{-}notp(ok, odd), gi\text{-}notp(ok, odd)' \wedge post\text{-}dec(ok) \wedge$

$(idset\ \overline{\{ok, ot\}} \vee g(et, ot) \wedge idset\ \{ok, v\})^{**}]$

$\mathscr{R}_{18} \equiv$ $\sqsubseteq$ by 3.64 (Guarantee invariant), 3.65 (Frame), 3.62c (Distribute-Guarantee), 3.93 (Trade-Rely-Guarantee), 3.114 (Assignment-Rely-Guarantee) and 3.91 (Introduce-Rely-Guar-Invariant)

ok:=*1*

guar−**inv** *gi-satp* · **guar** $(ot' \leq ot)$ · $\triangleleft$

guar−**inv** *gi-notp*(ok, odd) · $\triangleleft$

$\{ok, ot\}$: **rely** $g(et, ot) \wedge idset\ \{ok, v\} \cdot [init(ot), post\text{-}dec(ok)]$ $\triangleleft$

Introducing a while loop

This transformation introduces a while loop to perform the search of the lowest index *ot*, in the range of odd numbers, to satisfy the property *P*. The introduction of a loop requires the specification command to be formatted into a predefined shape, specified by the left-hand side of law 3.112 (Rely-Loop). This preparatory step is carried out in $\mathscr{R}_{19}$.

$\mathscr{R}_{19} \equiv$ $\sqsubseteq$ by 3.19a (Consequence)

guar−**inv** *gi-satp* · **guar** $(ot' \leq ot)$ ·

guar−**inv** *gi-notp*(ok, odd) · $\{ok, ot\}$: **rely** $g(et, ot) \wedge idset\ \{ok, v\}$ ·

$[init(ot), init(ot)' \wedge b_1(ok, ot, et)' \wedge w(ot, ok)^{**}_{\{ok,\, ot\}}]$

$\mathscr{R}_{20} \equiv$ $\sqsubseteq$ by 3.112 (Rely-Loop)

guar−**inv** *gi-satp* · **guar** $(ot' \leq ot)$ · ◁

guar−**inv** *gi-notp*(ok, odd) · $\{ok, ot\}$: ◁

while *c-while*(ok) **do** ◁

rely $g(et, ot) \wedge idset\ \{ok, v\}$ · ◁

$[prew(ok, ot), init(ot)' \wedge w(ot, ok)]$ ◁

The introduction of a loop in the previous step generates 13 proof obligations. Except for one, these are just tedious. The exception is a proof to show that $w(ot, ok)$ is well-founded on $init(ot)$. We did not succeed to prove this property in terms of definition 2.22 (Wellfounded-Precondition), thus this property is taken as an assumption in this derivation. As $w(ot, ok)$ is essentially the same relation *w* used in the development of a sequential version of Findp, the informal proof about well-foundedness is equally valid here. The next refinement step distributes the frame, guarantee and guarantee invariants over the while loop, and also applies law 3.19a (Consequence).

$\mathscr{R}_{21} \equiv$ $\sqsubseteq$ by 3.65 (Frame), 3.64 (Guarantee invariant), 3.62c (Distribute-Guarantee),
3.93 (Trade-Rely-Guarantee), 3.19a (Consequence),
3.87a (Rely-Monotonic) and 3.62f (Distribute-Guarantee)

while *c-while*(ok) **do**

guar−**inv** *gi-satp* · **guar** $(ot' \leq ot)$ · ◁

guar−**inv** *gi-notp*(ok, odd) · ◁

$\{ok, ot\}$: **rely** *idset* $\{ok, ot, v\}$ · $[prew(ok, ot), w(ot, ok)]$ ◁

Introducing a conditional

This transformation introduces a conditional within the while loop, and distributes the guarantee, guarantee invariant and frame over the conditional.

$\mathscr{R}_{22} \equiv$ $\sqsubseteq$ by 3.111 (Rely-Conditional)

if *c-if-odd* **then** $\triangleleft$

guar−**inv** *gi-satp* · **guar** $(ot' \leq ot)$ · $\triangleleft$

guar−**inv** *gi-notp*(ok, odd) · $\triangleleft$

$\{ok, ot\}$: **rely** *idset* $\{ok, ot, v\}$ · $[prew(ok, ot) \wedge$ *c-if-odd*$, w(ot, ok)]$ $\triangleleft$

else $\triangleleft$

guar−**inv** *gi-satp* · **guar** $(ot' \leq ot)$ · $\triangleleft$

guar−**inv** *gi-notp*(ok, odd) · $\triangleleft$

$\{ok, ot\}$: **rely** *idset* $\{ok, ot, v\}$ · $[prew(ok, ot) \wedge \neg$*c-if-odd*$, w(ot, ok)]$ $\triangleleft$

Introducing assignments

This transformation uses law 3.114 (Assignment-Rely-Guarantee) to implement the branches of the conditional.

$\mathscr{R}_{23} \equiv$ $\sqsubseteq$ by 3.64 (Guarantee invariant), 3.62c (Distribute-Guarantee),
3.67 (Distribute-Guarantee-Frame), 3.114 (Assignment-Rely-Guarantee)

if *c-if-odd* **then** $ot{:=}ok$ **else** $ok{:=}ok + 2$

The full intermediate programs, including both parallel branches for searching even and odd numbers, can be seen in Appendix C.3 on page 278.

Alternative derivation

In this section, we exercise an alternative derivation path to introduce the conditional and reach the implementation. The start point is the right-hand side of refinement $\mathscr{R}_{21}$. The insight behind this derivation is to eliminate the rely context surrounding the specification command at the cost of introducing syntactic restrictions in the development. This style of development follows from the application of law 3.105 (Rely-Uses). The derivation discussed

in this section illustrates the novel laws introduced in Section 3.11 (Restricting access to variables).

$\mathscr{R}_{24} \equiv \sqsubseteq$ by 3.105 (Rely-Uses)

guar−**inv** *gi-satp* · **guar** $(ot' \leq ot)$ ·

guar−**inv** *gi-notp*(ok, odd) ·

$\{ok, ot\}$: **uses** $\{ok, ot, v\}$ · $[prew(ok, ot), w(ot, ok)]$

$\mathscr{R}_{25} \equiv \sqsubseteq$ by 3.109 (Distribute-Uses)

uses $\{ok, ot, v\}$ ·

guar−**inv** *gi-satp* · **guar** $(ot' \leq ot)$ ·

guar−**inv** *gi-notp*(ok, odd) · $\{ok, ot\}$: $[prew(ok, ot), w(ot, ok)]$

$\mathscr{R}_{26} \equiv \sqsubseteq$ by 5.31 (Sequential-Conditional), 3.65 (Frame)

3.64 (Guarantee invariant) and 3.62e (Distribute-Guarantee)

uses $\{ok, ot, v\}$ ·

if *c-if-odd* **then**

guar−**inv** *gi-satp* · **guar** $(ot' \leq ot)$ ·

guar−**inv** *gi-notp*(ok, odd) · $\{ok, ot\}$: $[prew(ok, ot) \wedge$ *c-if-odd*$, w(ot, ok)]$

else

guar−**inv** *gi-satp* · **guar** $(ot' \leq ot)$ ·

guar−**inv** *gi-notp*(ok, odd) · $\{ok, ot\}$: $[prew(ok, ot) \wedge \neg$*c-if-odd*$, w(ot, ok)]$

$\mathscr{R}_{27} \equiv \sqsubseteq$ by 3.64 (Guarantee invariant), 3.62c (Distribute-Guarantee),

3.67 (Distribute-Guarantee-Frame), 3.113 (Assignment-Guarantee)

uses $\{ok, ot, v\}$ · **if** *c-if-odd* **then** $ot{:=}ok$ **else** $ok{:=}ok + 2$

$\mathscr{R}_{28} \equiv \sqsubseteq$ by 3.108 (Elimination-Uses)

if *c-if-odd* **then** $ot{:=}ok$ **else** $ok{:=}ok + 2$

Step $\mathscr{R}_{28}$ concludes the derivation by eliminating the uses block. The application law 3.108 (Elimination-Uses) generates two proof obligations:

1. *code* (**if** *c-if*(ok) **then** $ot{:=}ok$ **else** $ok{:=}ok + 2$)
2. $\exists S.$ *free* (**if** *c-if*(ok) **then** $ot{:=}ok$ **else** $ok{:=}ok + 2$) $S \wedge S \subseteq \{ok, ot, v\}$

These are trivial to discharge by applying the inductive definitions of code (Definition 3.106) and free variables for code (Definition 3.107).

6.4.3 Discussion

Since we have developed both a concurrent and a sequential version of Findp, we may wonder if the development of the sequential version of Findp was useful in guiding the development of the concurrent version. Based on our experiments, we believe it was. A simple comparison between the definitions used in the development of the sequential and concurrent versions of Findp (see Appendix C.3.2) suffices to see that the sequential case can be seen as a special case of the concurrent, where two process would be cooperating by performing the exact same task. As this formulation would require additional memory for no benefit, the development of the sequential version is not formulated as special case of the concurrent version with two identical processes searching for the lowest index to satisfy p.

We noted that, the more generalised the definitions are, the more reusable the lemmas about them become. Eventually, if we had provided fully parameterised definitions we would expect to fully reuse the proof of one branch to prove the other branch. Currently, mirroring is achieved by means of copy-and-paste. Little effort is required to systematically change variable names in the proof script and adapt the initialisation of variables.

Even though we calibrated the mechanisation to enhance Isabelle's support to automatically discharge proof obligations, the derivation of Findp shows that there is considerable room for fine-tuning the lemmas for reasoning about relations. Most of the proof obligations involve logical interpretation ($\vdash$), and the majority of them still requires user assistance to be discharged. This proof effort accounts for most of the proof interaction necessary to carry out the derivation of Findp. Although it does not impose any theoretical barrier to the use of the mechanisation, it reduces the productivity by demanding the user to deal with details that are sometimes just tedious.

Finally, an interesting observation about Findp is that it can be used to investigate laws about reachable evaluations. To create the scenario where laws about reachable evaluations can be validated, we need more than two processes cooperating. The initial split of the indices of v into a set of even and odd indices leads to a situation where one branch needs to consult the local copy of t hold by the sibling process in order to allow its early termination. If we then decide to further split the sets of odd and even numbers using additional criteria, we could split each of the local copies of t (*ot*,*et*) into further local copies (e.g. *l-ot* and *r-ot*). Cooperation among the four processes would be achieved by means of early termination.

For that, each process would need consult local instances of t hold by its siblings. The burden is, the law for introducing a while loop can only handle expressions that have at most a single reference to an unstable variable. In this case there would be multiple unstable variables occurring in the test of a while. To work around this constraint, we could use local variables to store copies of the unstable variables and use these copies to construct the boolean condition for the while. An alternative approach to this development path is to investigate the use of reachable evaluations to devise introduction laws for loops (and conditionals) where the expression under test does not satisfy the single reference property.

6.5 Sieve

The sieve of Eratosthenes is an algorithm for computing the prime numbers in the range $S \equiv \{2..maxn\}$ by eliminating multiples of $x \in 2..\lfloor\sqrt{n}\rfloor$. The key aspect behind the parallelisation of this algorithm is that the parallel algorithm does not use locks to prevent data races. Parallel programs *compete* to write to a shared variable, but all the data races actually attempt to write the same value to the variable, and thus the concurrent writes are *idempotent*.

The derivation presented here is a reproduction of the partial derivation of Sieve published in [65]. It serves the purpose of illustrating indexed parallelism with symmetric and non-parameterised rely and guarantee relations, and also to discuss a mechanism to introduce parallelism from a set rather than a list of indices.

6.5.1 Abbreviations

The first definition in Table 6.5 is formalised using definite description (THE). The expression $THE\ x.\ P\ x$ returns the value of x such that $P\ x$ holds, provided there exists a unique such x; otherwise, it denotes an arbitrary value whose type is the same as that of x. For the definition of square root, the implicit type of the parameter n is $\mathbf{N}$. The remaining definitions on the Table are straightforward.

6.5.2 Derivation

The initial specification requires that at the end of the derivation, the set s must contain no composite numbers in the range of $2..maxn$. Step $\mathscr{R}_1$ strengthens the postcondition. Step $\mathscr{R}_2$ introduces a guarantee and step $\mathscr{R}_3$ uses the fact that $guar_1$ is reflexive and transitive to trade

$$
\begin{array}{rcl}
\lfloor\sqrt{n}\rfloor & \equiv & \mathit{THE}\ r.\ r * r \leq n \wedge n < (r+1) * (r+1) \\
\mathit{rangen} & \equiv & \{i \mid 2 \leq i \wedge i \leq \lfloor\sqrt{\mathit{maxn}}\rfloor\} \\
c(i) & \equiv & \{j * i \mid 2 \leq j \wedge i * j \leq \mathit{maxn}\} \\
C & \equiv & \bigcup\{c(i) \mid i \in \mathit{rangen}\} \\
\mathit{post}_0 & \equiv & s' = s - C \\
\mathit{post}_1 & \equiv & s' \cap C = \emptyset \\
\mathit{guar}_1 & \equiv & s' \subseteq s \wedge s - s' \subseteq C \\
\mathit{guar}_2 & \equiv & s' \subseteq s
\end{array}
$$

Table 6.5 Abbreviations for Sieve.

this relation out of the postcondition.

$$
\begin{array}{rcl}
 & & [\mathit{pre}_0, \mathit{post}_0] \\
\mathscr{R}_1 & \equiv & \sqsubseteq \text{ by 3.19a (Consequence)} \\
 & & [\mathit{pre}_0, \mathit{guar}_1 \wedge \mathit{post}_1] \\
\mathscr{R}_2 & \equiv & \sqsubseteq \text{ by 3.59 (Introduce-Guarantee)} \\
 & & \textbf{guar}\ \mathit{guar}_1 \cdot [\mathit{pre}_0, \mathit{guar}_1 \wedge \mathit{post}_1] \\
 & & = \text{ by simplification as } \mathit{guar}_1 = {\mathit{guar}_1}^{**} \\
 & & \textbf{guar}\ \mathit{guar}_1 \cdot [\mathit{pre}_0, {\mathit{guar}_1}^{**} \wedge \mathit{post}_1] \\
\mathscr{R}_3 & \equiv & \sqsubseteq \text{ by 4.12 (Trade-Spec-Guarantee)} \\
 & & \textbf{guar}\ \mathit{guar}_1 \cdot [\mathit{pre}_0, \mathit{post}_1] \qquad \triangleleft
\end{array}
$$

We will now introduce an indexed parallel composition, but before we can do that we need to consider a small representation issue. Indexed parallelism is defined using lists of indices, but in the table of abbreviations for Sieve we do not have such a list. Instead, we defined *rangen* as a set of indices. To reuse this definition, we will introduce an operator that converts a set into a list. This operator will be defined using Hilbert's epsilon operator, that in Isabelle is represented by the syntax *SOME*. Given a finite set S, $\overrightarrow{S}$ denotes an injective list whose range is the set S. The order of the elements in $\overrightarrow{S}$ is left unspecified. Formally,

$$\overrightarrow{S} = (\mathit{SOME}\ l.\ \mathit{set}\ l = S \wedge |S| = \mathit{length}\ l) \tag{6.4}$$

The notation $|S|$ denotes the cardinality of the set S. The rule we need to reason about $\overrightarrow{S}$ in derivations is introduced next.

$$\frac{\mathit{finite}\ S}{\mathit{set}\ \overrightarrow{S} = S} \tag{6.5}$$

Using this rule, the derivation continues as shown next.

$$\begin{array}{lll}
\mathscr{R}_4 & \equiv & \sim \text{ by 3.81 (Rely-Idrel-Specification)} \\
 & & \mathbf{guar}\ \mathit{guar}_1 \cdot \mathbf{rely}\ \mathit{idrel} \cdot [\mathit{pre}_0, \mathit{post}_1] \\
\mathscr{R}_5 & \equiv & \sqsubseteq \text{ by 5.10 (Introduce-Multi-Parallel}^5\text{)} \\
 & & \mathbf{guar}\ \mathit{guar}_1 \cdot (\mathbf{guar}\ \mathit{guar}_2 \cdot \|_{i\in}\ \overrightarrow{\mathit{rangeh}} \cdot \mathbf{rely}\ \mathit{guar}_2 \cdot [\mathit{pre}_0, s' \cap c(i) = \emptyset]) \\
\mathscr{R}_6 & \equiv & \sqsubseteq \text{ by 5.6 (Distribute-g-Parallel) and 3.62c (Distribute-Guarantee)} \\
 & & \|_{i\in}\ \overrightarrow{\mathit{rangeh}} \cdot \mathbf{guar}\ \mathit{guar}_1 \cdot \mathbf{rely}\ \mathit{guar}_2 \cdot [\mathit{pre}_0, s' \cap c(i) = \emptyset]
\end{array}$$

Step $\mathscr{R}_5$ introduces an indexed parallel composition with non-parameterised and symmetric rely and guarantee relations. Step $\mathscr{R}_6$ pushes the external guarantee command into the scope of the indexed parallel composition and merges the nested guarantee commands. Note that in the last step of the derivation the rely and guarantee conditions are not symmetric anymore, as the overall guarantee becomes stronger.

We stop the derivation at this level of abstraction. In theory, it is possible to continue the derivation by reifying the representation of s from set of naturals to an array of booleans S, where $(S\ [i] = \mathit{True}) \Leftrightarrow (i \in s)$. Thus, the length of the array S could be the successor of greatest number in s, considering that arrays in RG-WSL are zero-based. To delete an element i from this concrete representation, it would be necessary to set $S\ [i]$ to *False*. At this level of representation, the body of the indexed parallel composition could be developed using the refinement laws for introducing a loop to visit each element of $c(i)$ and eliminate those elements present in S.

6.5.3 Discussion

The key motivation for this example is to illustrate the introduction of indexed parallelism using non-parameterised rely and guarantee conditions. Proof obligations in this (partial) development are trivial to discharge using the mechanisation.

[5] We omit lambda expressions in the body of indexed parallelism, e.g. $\|_{i\in}\ S \cdot F\ i$ means $\|_S \cdot (\lambda\ i.\ F\ i)$. This is in conformance with the conventions discussed Section 6.1 (Typographic conventions).

The derivation of Sieve can be used to reflect on relationship between data representation and the realisation of guarantee conditions. The algebra does not distinguish between concrete and abstract types, thus a user might continue the derivation using sets. In practice, it is unrealistic to expect a programming language to offer this type, or even to offer atomic operations to perform insertion and exclusion over a type representing a generic set of natural numbers, such as allowed by RG-WSL. On the other hand, if we reify the representation from a set of natural numbers to an array of booleans, we might expect the hardware to offer a primitive to read and set each position of the array atomically. The change in the data representation would affect the formulation of the specification. A putative specification considering a representation of the set s using array of booleans is given next.

$$\|_{i\in}\ \overrightarrow{rangeh} \cdot \textbf{var}\ j \cdot j{:=}2\ ;\ (\textbf{while}\ i * j \leq maxn\ \textbf{do}\ \ )$$

where

$$\begin{aligned} guar_{1R} &\equiv \forall k{\leq}maxn.\ S' \,!\, k \longrightarrow S \,!\, k \wedge (S \,!\, k \wedge \neg\, S' \,!\, k \longrightarrow k \in C) \\ guar_{2R} &\equiv \forall k{\leq}maxn.\ S' \,!\, k \longrightarrow S \,!\, k \end{aligned}$$

To continue developing Sieve from this point, it becomes necessary to generalise the definition of assignment to indexed positions of an arrays to allow the list of indices to contain expressions. This would be necessary because the position to be indexed would be dynamically determined by a loop counter, e.g.

$$\|_{i\in}\ \overrightarrow{rangeh} \cdot \textbf{var}\ j \cdot j{:=}2\ ;\ (\textbf{while}\ i * j < maxn\ \textbf{do}\ \ S[j]{:=}False\ \ ;j{:=}j+1)$$

The definition introduced in Section 5.4 fits situations where the indices to be applied to each dimension of an array are statically determined. This is not the case here, because of the use of a variable to determine the index.

6.6 Floyd-Warshall algorithm

The Floyd-Warshall algorithm is a dynamic programming formulation to solve the all-pairs shortest-paths problem on a directed, weighted graph that does not contain negative-weight cyles. The algorithm iterates over the set of nodes to compute the weight of the shortest path between the pairs of nodes. Prior to the first iteration, the weight of the shortest-paths between nodes of the graph is given by an adjacency matrix; at each iteration k ($0 \leq k < n$),

the distance between each pair of nodes (v_i, v_j), $1 \leq i,j \leq n$, is recalculated to account for the inclusion of v_{k+1} in the set of nodes that can occur in the shortest path from v_i to v_j; the iteration terminates when the full set of nodes has been considered.

The algorithm we are to develop operates on a directed graph $G = (V, E)$, where $V \subseteq \{v_1, v_2, v_3, ..., v_n\}$ is an indexed set of vertices and $E \subseteq V \times V$ is a set of directed edges. Weights are recorded by a $n \times n$ adjacency matrix W, defined as:

$$W[i,j] = \begin{cases} 0, & \text{if } i = j \\ \text{the weight of the edge } (v_i, v_j), & \text{if } i \neq j \text{ and } (v_i, v_j) \in E \\ \infty, & \text{if } i \neq j \text{ and } (v_i, v_j) \notin E. \end{cases} \tag{6.6}$$

To formalise the matrix of adjacency (W) in Isabelle, we use *VNone* to represent infinite (∞). Thus, distances between nodes are *vvalue* of the form (*VInt* α) or *VNone*. Effectively, this corresponds to modelling option types within *vvalue* using *VNone* as the distinguished value *None*. Assuming that the graph G has no negative-weight cyles, let $\delta\ (i, j, k)$ be the weight of the shortest-path in G from v_i to v_j using intermediate vertices only from $\{v_1, ..., v_k\}$ if such path exists in the graph represented by W. Otherwise, let $\delta(i,j,k) = \infty$. Abstractly, we can define δ as

$$\delta(i,j,k) = \begin{cases} W[i,j], & \text{for } k{=}0 \\ min\ (\delta(i,j,k{-}1),\ \delta(i,k,k{-}1) + \delta(k,j,k{-}1)), & \text{for } 1 \leq k. \end{cases} \tag{6.7}$$

To formalise δ we define addition and minimum over our representation of option types. For addition (*fsum*), we define that adding infinite (*VNone*) to another value results in infinite. Thus, if the shortest path from v_i to v_j using only intermediate nodes from $\{v_1, ..., v_k\}$ amounts to an infinite weight, any attempt of using this path to connect other nodes will also result in a path of infinite weight. For minimum of two numbers, (*fmin*), we define that the minimum between infinite and any value x is x. This is used to bias the choice of minimum paths towards connected paths in the definition of δ.

fun *fsum* :: *vvalue* ⇒ *vvalue* ⇒ *vvalue*
where
fsum (*VInt x*) (*VInt y*) = *VInt* (*x* + *y*)
fsum x VNone = *VNone*
fsum VNone x = *VNone*

fun *fmin* :: *vvalue* ⇒ *vvalue* ⇒ *vvalue*
where
fmin (*VInt x*) (*VInt y*) = *VInt* (*min x y*)
fmin x VNone = *x*
fmin VNone x = *x*

The encoding of δ is given next. It takes a matrix of adjacency (w), a source node (i), a target node (j) and a value (k) bounding the set of nodes allowed to appear as intermediate nodes in the shortest path.

fun *shortestPath* :: *vvalue* × *nat* × *nat* × *nat* ⇒ *vvalue* (δ - [*91*] *90*)
where
$\delta\ (W, i, j, 0) = [\![[\![W]\!]_a!i]\!]_a!j$
$\delta\ (W, i, j, (Suc\ k)) = fmin\ (\delta\ (W, i, j, k))\ \ (fsum\ (\delta\ (W, i, (Suc\ k), k))\ (\delta\ (W, (Suc\ k), j, k)))$

The functions *fsum* and *fmin* are the underlying mathematical representations used to define the deep-embedded operators $+_{\infty n}$ and $min_{\infty n}$. These operators are defined using the format illustrated in the definition of mod_n on page 26. To prevent distracting the reader with minor details of the actual encoding of Floyd-Warshall in Isabelle, we program anti-quotations to print $+_{\infty n}$ as $+$, and $min_{\infty n}$ as *min* along the derivation of the algorithm. This allow us to keep the discussion focused on the key aspects for this derivation.

6.6.1 Abbreviations

Table 6.6 presents abbreviations used along the derivation of a concurrent version of Floyd-Warshall algorithm. Recall that exclamation mark is Isabelle's native operator for list indexing. Moreover, remember that we are omitting projection functions, e.g. we write $M\ !\ i\ !\ j$ instead of $[\![[\![M]\!]_a\ !\ i]\!]_a\ !\ j$. Note that the precondition ($prec\ M$) contains type-related information. This unusual formulation is necessary because in RG-WSL the type of the variables is not declared at the begin of a program, and knowledge about type and dimension of variables is sometimes necessary in proofs to get rid of projection functions, such as $[\![\_]\!]_a$.

The *Index* set contains the indices of cells (i, j) in the domain of the matrix M. The predicate $(type(M))$ states that each cell of M is an integer or a distinguished value representing ∞. The predicate $(null\_diag(M))$ states that the diagonal of M is null. The predicate $(no\_nwc(M))$ states that M has no negative cycles. The predicate $(prec(M))$ is the top-level precondition for Floyd-Warshall. In particular, it assumes M to be square matrix of dimension n.

The relation Q_D is the top-level postcondition for Floyd-Warshall. It requires the matrix D to record the weight of the shortest path between the nodes of the graph whose matrix of adjacency is W. The relation $iter\text{-}Q_D$ is used to trade recursion by iteration, and provides a non-recursive version of $inv\text{-}Q_D{}'$. The parameterised relation $Q_D\text{-}cell\ (i, j)$ is used to specify the postcondition of individual processes, and comes on the scene in the refinement step that introduces indexed parallel composition. The parameters taken by this relation represent the

$$
\begin{array}{rcl}
\mathit{Index} & \equiv & \{(i,j) \mid (1 \leq i \wedge i \leq n) \wedge (1 \leq j \wedge j \leq n)\} \\
\mathit{type}(M) & \equiv & \forall\, i\, j.\ (i,j) \in \mathit{Index} \longrightarrow \mathit{Type}\ (M\ !\ i\ !\ j)\ (\mathit{type\text{-}option\ type\text{-}VInt}) \\
\mathit{null\_diag}(M) & \equiv & \forall\, i.\ (i,i) \in \mathit{Index} \longrightarrow M\ !\ i\ !\ i = 0 \\
\mathit{no\_nwc}(M) & \equiv & \forall\, i\, j.\ (i,j) \in \mathit{Index} \longrightarrow \\
 & & \mathit{fmin}(M\ !\ i\ !\ i, \mathit{fsum}(M\ !\ i\ !\ j, M\ !\ j\ !\ i)) = M\ !\ i\ !\ i \\
\mathit{prec}(M) & \equiv & \mathit{dim}(M, [n+1, n+1]) \wedge \mathit{type}(M) \wedge \mathit{null\_diag}(M) \wedge \mathit{no\_nwc}(M) \\
Q_D & \equiv & \forall\, i\, j.\ (i,j) \in \mathit{Index} \longrightarrow D'\ !\ i\ !\ j = \delta\ (W, i, j, n) \\
\mathit{iter}\text{-}Q_D & \equiv & \forall\, i\, j.\ (i,j) \in \mathit{Index} \longrightarrow Q_D\text{-}\mathit{cell}(i,j) \\
\mathit{inv}\text{-}Q_D & \equiv & \forall\, i\, j.\ (i,j) \in \mathit{Index} \longrightarrow D\ !\ i\ !\ j = \delta\ (W, i, j, \mathit{min}(n,k)) \\
Q_D\text{-}\mathit{cell}\ (i,j) & \equiv & D'\ !\ i\ !\ j = \mathit{fmin}(D\ !\ i\ !\ j, \mathit{fsum}(D\ !\ i\ !\ k', D\ !\ k'\ !\ j)) \\
w & \equiv & k < k' \wedge k' \leq n \\
\mathit{g\text{-}cell}\ (c_i, c_j) & \equiv & (\forall i.\ (i,i) \in \mathit{Index} \longrightarrow D\ !\ k\ !\ i = D'\ !\ k\ !\ i \wedge D\ !\ i\ !\ k = D'\ !\ i\ !\ k) \wedge \\
 & & (\forall\, i\, j.\ (i,j) \in \mathit{Index} \wedge (i,j) \neq (c_i, c_j) \longrightarrow D\ !\ i\ !\ j = D'\ !\ i\ !\ j) \wedge \\
 & & \mathit{idset}\ \overline{\{D\}} \wedge (\mathit{prec}(D) \Rightarrow \mathit{prec}(D)') \\
\mathit{r\text{-}cell}\ (c_i, c_j) & \equiv & (\forall i.\ (i,i) \in \mathit{Index} \longrightarrow D\ !\ k\ !\ i = D'\ !\ k\ !\ i \wedge D\ !\ i\ !\ k = D'\ !\ i\ !\ k) \wedge \\
 & & (\forall\, i\, j.\ (i,j) = (c_i, c_j) \longrightarrow D\ !\ i\ !\ j = D'\ !\ i\ !\ j) \wedge \\
 & & \mathit{idset}\ \overline{\{D\}} \wedge (\mathit{prec}(D) \Rightarrow \mathit{prec}(D)')
\end{array}
$$

Table 6.6 Abbreviations for Floyd-Warshall.

row (i) and column (j) of the cell where the indexed process operates. The well-founded relation w states that k must monotonically increase between the before and after state, and its value must not exceed n. This relation is used to establish the termination of the only loop used in the derivation of Floyd-Warshall.

The parameterised relations $g\text{-}cell\ (c_i, c_j)$ and $r\text{-}cell\ (c_i, c_j)$ are the individual guarantee and rely condition for the parallel processes. These relations are parameterised by the index of the process. Thus, these relations are structurally similar to all processes, even though each process instantiate (c_i, c_j) differently.

6.6.2 Derivation

For the derivation of Floyd-Warshall, we took inspiration from [32], where this algorithm is developed also using a refinement calculus for shared variable. There, array indexing is one-based. The development for zero-based indices requires adjustment on the definition of the recursive function δ (Definition 6.7), as well in definitions where matrices are accessed.

For the sake of clarity, we simulate one-based index in this development. For this, we embed a matrix of dimension $n \times n$ into a matrix of dimension $(n+1) \times (n+1)$, discarding the top row and the leftmost column, as illustrated in Figure 6.2. The cost for simplifying the derivation is additional space to store the matrix of adjacency and the result of the computation. The additional space grows linearly with the dimension of the matrix of adjacency.

a)

0	∞	-2	∞
4	0	3	∞
∞	∞	0	2
∞	-1	∞	0

b)

	0	∞	-2	∞
	4	0	3	∞
	∞	∞	0	2
	∞	-1	∞	0

Figure 6.2 Simulating one-based indexing from zero-based indexing. The matrix at the right (b) shows the encoding of the matrix of adjacency (a) discarding the top row and the leftmost column of the matrix in order to obtain the first relevant cell indexed as (1,1) instead of (0,0).

The initial specification for Floyd-Warshall algorirthm is $\{D\}$: $[prec(W), Q_D]$. This specification requires the matrix D to record the weight of the shortest-path between the nodes of the graph whose matrix of adjacency is W. For the purpose of discharging proof obligations when making assignments to indexed positions of D, we need to know the dimension and the type of the matrix of adjacency. This is recorded using the precondition $prec(W)$, which also records the fact that W has no cycles of negative weight (*no-nwc M*) and that the distance from a node to itself is recorded as zero (*null-diag M*).

The derivation is structured in six transformations. The first insight is that for all $(i, j) \in Index$, the computation of $\delta\ (W, i, j, k)$ can be achieved by means of iteration over k. Thus, the first four transformations deal with the introduction of a loop counter (k), and establishment of an invariant for the loop. For the last two transformations we use an example to explain the insight behind rely and guarantee conditions used to introduce parallelism.

Duplicating matrix of adjacency and introducing a loop counter

This transformation prepares the specification for the introduction of a while loop. It introduces and initialises the local variables k, which is used as a loop counter for iterating over the set of intermediate nodes which can be used to construct the shortest path between any

two arbitrary nodes i and j in the graph abstracted by the matrix of adjacency W. Additionally, this transformation initialises the matrix D with a copy of the matrix of adjacency W. The duplication of the matrix of adjacency is relevant for the formal development of the algorithm. Although the original matrix of adjacency is not modified along the algorithm, its original value has to be remembered because is used within an invariant introduced in this step ($inv\text{-}Q_D$). This invariant correlates the loop counter k and the matrix D.

$$
\begin{array}{lll}
 & \{D\}\colon [prec(W),\, Q_D] & \\
\mathscr{R}_1 \;\equiv & \sqsubseteq \text{ by 3.101 (Introduce-Variable-Frame)} & \\
 & \textbf{var } k & \\
 & \quad \{k, D\}\colon [prec(W),\, Q_D] & \triangleleft
\end{array}
$$

The first refinement step introduces the local variable k via the application of law 3.101 (Introduce-Variable-Frame). The application generates a proof obligation for the user to prove that k are unrestricted in $[prec(W),\, Q_D]$. This proof follows from the definition of unrestricted variables (Definition 2.81 on page 61).

The next transformation introduces the invariant $inv\text{-}Q_D \wedge prec(D)$, which state that: (i) the dimension and type of the matrix D, used to store the result of the computation of the shortest paths, remains constant along the program after initialisation of D; (ii) there are no cycles with negative weight in D; (iii) the matrix D has a null diagonal; and (iv) the matrix D holds the shortest paths between pairs of nodes, such that intermediate nodes can only be drawn from $\{V_1, ..., V_k\}$.

$$
\begin{array}{lll}
\mathscr{R}_2 \;\equiv & \sqsubseteq \text{ by 3.25 (Sequential) and 4.13 (Distribute-Frame-Sequential)} & \\
 & \{k, D\}\colon [prec(W),\, idset\ \overline{\{k, D\}} \wedge (inv\text{-}Q_D \wedge prec(D))\,']; & \\
 & \{k, D\}\colon [inv\text{-}Q_D \wedge prec(D),\, idset\ \overline{\{k, D\}} \wedge ((inv\text{-}Q_D \wedge prec(D))' \wedge (n \leq k)')] & \\
\mathscr{R}_3 \;\equiv & \sqsubseteq \text{ by 4.9 (Trade-Spec-Frame)} & \\
 & \{k, D\}\colon [prec(W),\, (inv\text{-}Q_D \wedge prec(D))\,']; & \triangleleft \\
 & \{k, D\}\colon [inv\text{-}Q_D \wedge prec(D),\, (inv\text{-}Q_D \wedge prec(D))' \wedge (n \leq k)'] & \triangleleft
\end{array}
$$

The next transformation initialises the invariant introduced in $\mathscr{R}_2$. The initialisation is achieved by setting k to zero and assigning the matrix of adjacency to D. First, step $\mathscr{R}_4$ splits the left specification into two: the first initialises k, the second initialises D. Then, step $\mathscr{R}_5$ applies law 3.113 (Assignment-Guarantee) to both specifications to introduce assignments.

Recall that there is no formal distinction between arrays and scalar variables in RG-WSL[6]. Although direct copy of arrays is supported by RG-WSL, it is not compatible with most standard programming languages as they handle arrays via references (memory pointers). It can be argued that a more conventional way of initialising arrays is to use a loop per dimension of the array and initialise the cells separately. The issue with this approach is that if we use a loop to initialise D we are not able to infer that W and D have the same dimension at the end of the initialisation, and this has to be ensured by the invariant $prec(D)$. To work around this limitation we copy the array W as we would copy a normal scalar variable. To remove this limitation from the algebra, the representation of the state (Definition 2.1) would need to be extended to record the type and dimension of variables. Variables would also need to be explicitly declared in programs.

$$
\begin{aligned}
\mathscr{R}_4 \;\equiv\;& \sqsubseteq \text{ by 3.25 (Sequential) and 4.13 (Distribute-Frame-Sequential)} \\
& \{k\}\colon [prec(W),\, (k' = 0) \wedge prec(W)'] \;; \\
& \{D\}\colon [prec(W),\, idset\ \{k, W\} \wedge (D' = W)] \;; \\
& \{k, D\}\colon [inv\text{-}Q_D \wedge prec(D),\, (inv\text{-}Q_D \wedge prec(D))' \wedge (n \leq k)'] \\
\mathscr{R}_5 \;\equiv\;& \sqsubseteq \text{ by 3.65 (Frame) and 3.113 (Assignment-Guarantee)} \\
& k{:=}0; \\
& D{:=}W; \\
& \{k, D\}\colon [inv\text{-}Q_D \wedge prec(D),\, (inv\text{-}Q_D \wedge prec(D))' \wedge (n \leq k)'] \qquad \triangleleft
\end{aligned}
$$

Introducing a while loop

This transformation introduces a while loop using k as the control variable. The purpose of the loop is to compute the all-pairs shortest paths by monotonically expanding the set of nodes $\{V_1, V_2, ..., V_k\}$ from where nodes can be draw to derive the shortest path.

The introduction of the loop uses law 5.32 (Sequential-Loop). Introduction of loops requires careful thought about the encoding of a postcondition via a loop invariant and the negation of the loop condition. A potential mistake for the uninitiated in refinement is to attempt to encode the postcondition via the well-founded relation. In this application, the loop invariant is $inv\text{-}Q_D \wedge prec(D)$ and the expression chosen to be the loop condition is

[6] Definition 2.1 (Basic types) on page 23 formalises arrays and scalar values using a single type (*vvalue*).

$k < n$.

$$
\begin{array}{lll}
\mathscr{R}_6 & \equiv & \sqsubseteq \text{ by 3.19a (Consequence)} \\
 & & \{k, D\}\colon [\mathit{inv}\text{-}Q_D \wedge \mathit{prec}(D),\ (\mathit{inv}\text{-}Q_D \wedge \mathit{prec}(D))' \wedge (\neg k < n)' \wedge w^{**}] \\
\mathscr{R}_7 & \equiv & \sqsubseteq \text{ by 5.32 (Sequential-Loop)} \\
 & & \{k, D\}\colon \textbf{while } k < n \textbf{ do} \\
 & & \qquad [\mathit{inv}\text{-}Q_D \wedge \mathit{prec}(D) \wedge k < n,\ (\mathit{inv}\text{-}Q_D \wedge \mathit{prec}(D))' \wedge w] \\
\mathscr{R}_8 & \equiv & \sqsubseteq \text{ by 3.65 (Frame) and 3.62f (Distribute-Guarantee)} \\
 & & \textbf{while } k < n \textbf{ do} \\
 & & \quad \{k, D\}\colon [\mathit{inv}\text{-}Q_D \wedge \mathit{prec}(D) \wedge k < n,\ (\mathit{inv}\text{-}Q_D \wedge \mathit{prec}(D))' \wedge w] \qquad \triangleleft
\end{array}
$$

Step $\mathscr{R}_6$ strengthens the postcondition and leaves the specification command into the right shape for the application of law 5.32 (Sequential-Loop). The expression $\neg k < n$ corresponds to the negation of the loop condition. The relation $w \equiv k < k' \wedge k' \leq n$ states that k monotonically increases up to the value of n. For the application of law 5.32 in step $\mathscr{R}_7$ we assume that w is well-founded on the predicate $\mathit{inv}\text{-}Q_D \wedge \mathit{prec}(D)$. An informal proof of why w is well-founded is simple to argue: at each iteration it reduces the gap between k and n, and it does not allow k to become bigger than n, thus this relation can only be iterated a finite number of times.

Replacing recursion by iteration

This transformation trades recursion by iteration and illustrates the key insight behind the technique of *dynamic programming* [28]. This technique uses memory-based data structures to store the result of recursive calls, typically indexing results based on the value of the parameters passed to a recursive function. In this case, the recursive function used to solve the all-pairs shortest path problem is δ. The motivation for using this technique is to prevent re-computing the solution for the same problem (i.e. $\delta\ (W, i, j, k)$) multiple times.

The invariant $\mathit{inv}\text{-}Q_D$ is stated in terms of the recursive function δ, while the relation $\mathit{iter}\text{-}Q_D$ does not reference the recursive function δ (Definition 6.7). Instead, it refers to cells of the matrix D whenever it needs to access the result of the computation of $\delta\ (W, i, j, k-1)$.

$$
\begin{array}{lll}
\mathscr{R}_9 & \equiv & \sqsubseteq \text{ by 3.19a (Consequence)} \\
 & & \{k, D\}\colon [\mathit{inv}\text{-}Q_D \wedge \mathit{prec}(D) \wedge k < n, \\
 & & \qquad \mathit{idset}\ \overline{\{k, D\}} \wedge (\mathit{iter}\text{-}Q_D \wedge \mathit{prec}(D)' \wedge (k' = k + 1))] \\
\mathscr{R}_{10} & \equiv & \sim \text{ by 4.9 (Trade-Spec-Frame)} \\
 & & \{k, D\}\colon [\mathit{inv}\text{-}Q_D \wedge \mathit{prec}(D) \wedge k < n, \mathit{iter}\text{-}Q_D \wedge \mathit{prec}(D)' \wedge (k' = k + 1)] \quad \triangleleft
\end{array}
$$

Step $\mathscr{R}_9$ strengthens the postcondition and trades recursion by iteration and step $\mathscr{R}_{10}$ simplifies the postcondition, eliminating the redundant frame from it.

Including node $k + 1$ on the computation of the shortest path

This transformation splits the specification in two parts, the first increments k and the second establishes $\mathit{iter}\text{-}Q_D$. Step $\mathscr{R}_{11}$ uses law 3.25 (Sequential) to split the specification. Note the replication of the frame and the introduction of the fact that k must be kept within the range of relevant indices for each dimension of D, that is, $(1 \leq k \leq n)$. Step $\mathscr{R}_{13}$ introduces an assignment to k.

$$
\begin{array}{lll}
\mathscr{R}_{11} & \equiv & \sqsubseteq \text{ by 3.25 (Sequential) and 4.13 (Distribute-Frame-Sequential)} \\
 & & \{k\}\colon [\mathit{inv}\text{-}Q_D \wedge \mathit{prec}(D) \wedge k < n, \\
 & & \qquad \mathit{idset}\ \overline{\{k\}} \wedge (\mathit{prec}(D) \wedge (1 \leq k \leq n))' \wedge (k' = k + 1)]; \\
 & & \{D\}\colon [\mathit{prec}(D) \wedge (1 \leq k \leq n), \mathit{idset}\ \overline{\{D\}} \wedge \mathit{iter}\text{-}Q_D \wedge \mathit{prec}(D)'] \\
\mathscr{R}_{12} & \equiv & \sim \text{ by 4.9 (Trade-Spec-Frame)} \\
 & & \{k\}\colon [\mathit{inv}\text{-}Q_D \wedge \mathit{prec}(D) \wedge k < n, (\mathit{prec}(D) \wedge (1 \leq k \leq n))' \wedge (k' = k + 1)]\ ; \\
 & & \{D\}\colon [\mathit{prec}(D) \wedge (1 \leq k \leq n), \mathit{prec}(D)' \wedge \mathit{iter}\text{-}Q_D] \\
\mathscr{R}_{13} & \equiv & \sqsubseteq \text{ by 3.113 (Assignment-Guarantee)} \\
 & & k{:=}k + 1 \\
 & & \{D\}\colon [\mathit{prec}(D) \wedge (1 \leq k \leq n), \mathit{prec}(D)' \wedge \mathit{iter}\text{-}Q_D] \quad \triangleleft
\end{array}
$$

Introducing indexed parallelism

This transformation illustrates the introduction of indexed parallelism to establish *iter-*Q_D. The main insight behind the parallelisation of Floyd-Warshall is the observation that, for every iteration of the while loop, there are a set of cells S_k in D that stay stable with respect to the previous iteration. This set is composed of cells indexed by row k or column k. All computations that modify cells in D depend only on cells in S_k. Thus, for any iteration, computations can run in parallel. For the first iteration ($k = 0$), S_k stays stable with respect to the matrix of adjacency W, used to initialise D.

To discuss the stability of S_k, Figure 6.3 on page 209 presents the computation of the all-pairs shortest paths using δ for a simple graph with four nodes. The grey colour is used to highlight cells in S_k for each value of k taken by the function δ. The reason why cells in S_k are stable with respect to the previous iteration is because, for $1 \leq k$, the definition $\delta\ (W, i, j, k)$ (Definition 6.7) expands to

$$fmin(\delta\ (W, i, j, k-1), fsum(\delta\ (W, i, k, k-1), \delta\ (W, k, j, k-1)))$$

and, when $i = k$ or $j = k$, that is, the cell (i, j) is in the set S_k, this expressesion reduces to $\delta\ (W, i, j, k-1)$. Such reduction uses the fact that the distance from a node to itself is zero. In Figure 6.3, the result of $\delta\ (W, i, j, k-1)$ is stored in position (i, j) of the matrix D during iteration k.

The first step of this transformation is to expand *iter-*Q_D, making explicit the universal quantifier over Q_D*-cell* (i, j), for (i, j) in the domain of the matrix D.

$$\begin{array}{lll} \mathscr{R}_{14} & = & \text{by Definition of } \textit{iter-}Q_D \text{ (Table 6.6)} \\ & & \{D\}\colon [prec(D) \wedge (1 \leq k \leq n), \qquad \triangleleft \\ & & \qquad prec(D)' \wedge (\forall\, i\, j.\ (i, j) \in Index \longrightarrow Q_D\text{-}cell(i, j))] \triangleleft \end{array}$$

To parallelise Floyd-Warshall we will use n^2 processes, one per each cell of the matrix of adjacency. For this derivation, we will use rely and guarantee conditions that are parameterised by an index. Processes will be indexed using a pair of naturals, the first corresponding to the row, the second corresponding to the column of the cell that it updates. To introduce indexed parallelism we will apply law 5.9 (Introduce-Multi-Parallel-Parameterised). Step $\mathscr{R}_{15}$ substitutes *Index* by $(set\ \overrightarrow{Index})$ using property 6.5 (introduced on page 198). This change of representation is akin to the one used in Sieve.

a)

	V_1	V_2	V_3	V_4
V_1	0	∞	-2	∞
V_2	4	0	3	∞
V_3	∞	∞	0	2
V_4	∞	-1	∞	0

b)

	V_1	V_2	V_3	V_4
V_1	0	∞	-2	∞
V_2	4	0	2	∞
V_3	∞	∞	0	2
V_4	∞	-1	∞	0

c)

	V_1	V_2	V_3	V_4
V_1	0	∞	-2	∞
V_2	4	0	2	∞
V_3	∞	∞	0	2
V_4	3	-1	1	0

d)

	V_1	V_2	V_3	V_4
V_1	0	∞	-2	0
V_2	4	0	2	4
V_3	∞	∞	0	2
V_4	3	-1	1	0

e)

	V_1	V_2	V_3	V_4
V_1	0	−1	-2	0
V_2	4	0	2	4
V_3	5	1	0	2
V_4	3	-1	1	0

f)

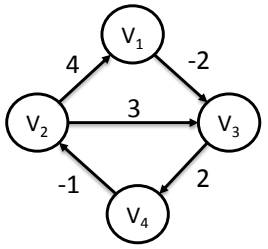


Figure 6.3 Stability of S_k. The graph shown in (f) is encoded via the matrix of adjacency W in (a). Nodes are labelled from V_1 to V_4 and missing edges are characterised by the value ∞ in the matrix of adjacency. Table (b) shows the result of the computation of $\delta\ (W, i, j, k)$ for all $i, j \in \{1,2,3,4\}$ and $k = 1$. In the tables, the leftmost column identifies the source node, and the top row identifies the target node. Thus, the result of $\delta\ (W, i, j, k)$ is presented in position (V_i, V_j). Tables (c), (d) and (e) shows the result of the computation for iterations $k = 2$, $k = 3$ and $k = 4$, respectively.

$$
\begin{aligned}
\mathscr{R}_{15} \;=\;& \text{ by property 6.5 (on page 198)} \\
& \{D\}\colon [prec(D) \wedge (1 \leq k \leq n), \qquad \triangleleft \\
& \qquad prec(D)' \wedge (\forall i\, j.\ (i,j) \in set\ \overrightarrow{Index} \longrightarrow Q_D\text{-}cell(i,j))] \quad \triangleleft
\end{aligned}
$$

Before introducing indexed parallelism, we must choose the rely and guarantee conditions for the individual processes. For this, we take inspiration from the example given in Figure 6.3. On one hand, it is reasonable to propose that each process (i, j) should guarantee that it does not change any cells apart from (i, j) during iteration k. Moreover, if $i = k$ or $j = k$, the content of (i, j) should be left unchanged. On the other hand, it is reasonable to expect the environment to preserve the content of any cell in S_k, and do not change the content of the cell (i, j). These rely and guarantee conditions are almost as accurate as the one used in the mechanised derivation. In practice, we still need to include additional information to

ensure that processes do not change the type and dimension of the matrix D. The introduction of indexed parallelism is formalised in step $\mathscr{R}_{18}$. Steps $\mathscr{R}_{19}$ and $\mathscr{R}_{21}$ distribute indexed parallelism over a guarantee invariant and frame, respectively. Since no specific laws to handle these commands exist, their definitions have to be expanded and contracted to allow law 5.6 (Distribute-g-Parallel) to be applied.

$\mathscr{R}_{16} \equiv$ $\sqsubseteq$ by 3.68 (Trade-Guarantee-Invariant)

$\{D\}$: **guar**−**inv** $prec(D)$ ·

$[prec(D) \wedge (1 \leq k \leq n), \forall i\, j.\ (i,j) \in set\ \overrightarrow{Index} \longrightarrow Q_D\text{-}cell(i,j)]$

$\mathscr{R}_{17} \equiv$ $\sim$ by 3.81 (Rely-Idrel-Specification)

$\{D\}$: **guar**−**inv** $prec(D)$ · **rely** $idrel$ ·

$[prec(D) \wedge (1 \leq k \leq n), \forall i\, j.\ (i,j) \in set\ \overrightarrow{Index} \longrightarrow Q_D\text{-}cell(i,j)]$

$\mathscr{R}_{18} \equiv$ $\sqsubseteq$ by 5.9 (Introduce-Multi-Parallel-Parameterised[7])

$\{D\}$: **guar**−**inv** $prec(D)$ ·

$\|_{(i,j)\in} \overrightarrow{Index}$ · **guar** $g\text{-}cell\ (i,j)$ · **rely** $r\text{-}cell\ (i,j)$ ·

$[prec(D) \wedge (1 \leq k \leq n), Q_D\text{-}cell\ (i,j)]$

$\mathscr{R}_{19} \equiv$ $\sqsubseteq$ by 5.6 (Distribute-g-Parallel) and 3.64 (Guarantee invariant)

$\{D\}$: $\|_{(i,j)\in} \overrightarrow{Index}$ · **guar**−**inv** $prec(D)$ · **guar** $g\text{-}cell\ (i,j)$ · **rely** $r\text{-}cell\ (i,j)$ ·

$[prec(D) \wedge (1 \leq k \leq n), Q_D\text{-}cell\ (i,j)]$

$\mathscr{R}_{20} \equiv$ $\sqsubseteq$ by 3.62c (Distribute-Guarantee) and 3.60a (Guarantee-Monotonic)

$\{D\}$: $\|_{(i,j)\in} \overrightarrow{Index}$ · **guar** $g\text{-}cell\ (i,j)$ · **rely** $r\text{-}cell\ (i,j)$ ·

$[prec(D) \wedge (1 \leq k \leq n), Q_D\text{-}cell\ (i,j)]$

$\mathscr{R}_{21} \equiv$ $\sim$ by 5.6 (Distribute-g-Parallel) and 3.65 (Frame)

$\|_{(i,j)\in} \overrightarrow{Index}$ · $\{D\}$: **guar** $g\text{-}cell\ (i,j)$ · **rely** $r\text{-}cell\ (i,j)$ ·

$[prec(D) \wedge (1 \leq k \leq n), Q_D\text{-}cell\ (i,j)]$

[7] We omit lambda expressions in the body of indexed parallelism, e.g. $\|_{i\in} S \cdot F\ i$ means $\|_S \cdot (\lambda\ i.\ F\ i)$. This is in conformance with the conventions discussed Section 6.1 (Typographic conventions).

Introducing assignment

This transformation implements the assignment to indexed arrays cells and completes the derivation. Its main purpose is to illustrate the usage of law 5.26 (Assignment-Array).

$$\begin{array}{rcl} \mathscr{R}_{22} & \equiv & \sqsubseteq \text{ by 5.26 (Assignment-Array)} \\ & & \|_{(i,j)\in \overrightarrow{Index}} \cdot D[i,j]{:=}min\ (D[i][j], D[i][k]{+}D[k][j]) \end{array}$$

We succeed to discharge part of the proof obligations generated by the application of 5.26. These included the proof that assignment to indexed arrays satisfies the following restrictions imposed by the guarantee condition *g-cell* (i, j):

- Preservation of rows and columns if $i = k$ or $j = k$;
- Non-updates to cells other than (i, j);
- Preservation of type constraints;
- Preservation of the dimension of the matrix D;
- Preservation of a null-diagonal.

One of the proof obligations we have not succeed to prove is to show that the assignment does not introduce negative paths. In short, Isabelle demand us to prove

$$fmin(0, fsum(D\ !\ i\ !\ j, fsum(D\ !\ j\ !\ k, D\ !\ k\ !\ i))) = 0$$

from the premises

$$1 \leq i,j,k \leq n \wedge \vdash dim(D, [n+1, n+1]) \wedge type(D) \wedge null\_diag(D) \wedge no\_nwc(D)$$

The failure in discharging this proof obligation hints at the possibility of the predicate *no-nwc* to be weaker than it actually needs to be in order to formalise that the graph has no negative paths. Instead of formalising notions of graphs for this particular problem, a better approach would be to investigate the possibility of reuse an already existing formalisation of digraphs for Isabelle/HOL, such as [86].

6.6.3 Discussion

Floyd-Warshall provides a simple example to discuss indexed parallelism based on parameterised programs and assignment to indexed positions of an array. This derivation is inspired in the one proposed in [32]. The attempt of using the algebra to develop this example reveals that the formulation of states as total functions from *vname* to *vvalue* is not ideal for practical use of the theory. The lack of type information about variables makes refinement proofs more tedious in practice, by requiring the user to prove type consistency along the derivation. In fact, the problem goes beyond productivity: without a type system certain development paths are eliminated too early. For example, in refinement step $\mathscr{R}_5$, the lack of information about dimension of variables rules out the possibility of initialising the matrix D using a loop.

An exciting aspect about the formulation of Floyd-Warshall is that it suggests that rely-guarantee may be useful to investigate concurrent versions of algorithms formulated using dynamic programming, such as for example the Knapsack problem [28]. To the best of our knowledge, there is no investigation that currently explores the potential of rely-guarantee to verify such class of algorithms.

6.7 Discussion

The examples in this chapter illustrate the practical use of the algebra to derive sequential and concurrent programs. We use them to highlight patterns that are quite common in derivations using the new algebraic rely-guarantee style of [48], such as the replication of a frame and rely condition in a postcondition to provide information that is sometimes essential to discharge proof obligations, or the distribution of guarantees along the development to allow the user to continue benefiting from information encoded by this command.

It can be argued that the frequency that these patterns occurs suggest that there is a need for more tailor made laws. When designing a set of laws, one has a trade-off between smoothing the learning curve for new users versus minimising the size of proof scripts. In our view, the advantage of a reduced set of laws is that it exposes the reasoning patterns and promotes user confidence. Such confidence is vital when proofs are not completed by automated reasoning, since in these moments user interaction is necessary. Next we use a few proof metrics to compare the main derivations presented at this chapter.

Law	Name	Weight (w_i)
3.19b	Consequence	2
3.111	Rely-Conditional	7
5.31	Sequential-Conditional	1
3.112	Rely-Loop	10
5.32	Sequential-Loop	2
4.5a	Substitution-Refinement	0
3.101	Introduce-Variable-Frame	1
3.68	Trade-Guarantee-Invariant	1

Table 6.7 Calibrating weight for measuring total of proof obligations

6.7.1 Proof metrics

In this section we compare the derivations of Findp, Sieve and Floyd-Warshall using four proof metrics: (i) total of proof obligations; (ii) the total of refinement laws used; (iii) the number of distinct laws used and (iv) the total of specification lines. The first three metrics are extracted using the following formula[8]:

$$WTM(t) = \sum_{i=1}^{n} c\ (t, i)$$

where i is the identifier of a law, n is the total of refinement laws in the mechanisation, t is a theory identifier, and c is some complexity function applied to laws in t. For the each metric we instantiate the complexity function c differently, namely:

$$\begin{array}{ll} \text{metric (i)} & c\ (i, t) = occurrences\ (i, t) * w(i) \\ \text{metric (ii)} & c\ (i, t) = occurrences\ (i, t) \\ \text{metric (iii)} & c\ (i, t) = (\mathit{if}\ occurrences\ (i, t) \neq 0\ \mathit{then}\ 1\ \mathit{else}\ 0) \end{array}$$

The function *occurrences* returns the number of occurrences of the law i within the theory t. The function w (weight) provides the total number of proof obligations related to logical interpretation (or that indirectly involve logical interpretation: stability, unrestriction and single reference property) and well-foundedness of relations. These specific proof obligations were chosen because they are the most difficult to discharge, in our opinion. Table 6.7 shows the weight we assigned to a few the mechanised laws for metric.

[8]WTM stands for *weighted theorems per module (theory)*. The formula is taken from [4].

It is important to note that each metric in isolation does not provide sufficient information to compare the effort behind the mechanisation of distinct examples. Even the most elaborate of our metrics, i.e. total of proof obligations, is quite fragile as it does not take into account the complexity of a proof obligation. Table 6.8 presents the metrics (i)-(iv) for each of the examples. In this table we differentiate between the derivation of Findp that uses the rely command until implementing assignments, from the one that switches to the **uses** command (Findp Uses). For the examples considered in the table, the most used law varies between transitivity (law 3.18b) and substitution (law 4.5a).

	Findp Seq.	Findp Rely	Findp Uses	Sieve	Floyd Warshall
Proof obligations	30	121	107	24	27
Applied laws	156	510	578	62	95
Distinct laws	30	40	47	27	28
Specification lines	536	1706	1830	299	744

Table 6.8 Proof metrics for mechanised examples

The first metric suggests that the derivation of a concurrent version of Findp requires up to four times the effort involved in the derivation of its sequential version. We agree with the first perception in part: the total of distinct laws applied in the derivation suggests that the increase in the intellectual work does not grow linear with the increase in the number of proof obligations. More tedious work is required, we agree with that; but it is mostly repetitive. Still considering the first metric, it also suggests that Findp is the most laborious derivation among those considered. For us, this does not necessarily reflect the reality, but might suggest a direction for improving our encoding. The proof obligations in Floyd-Warshall are harder to discharge than those involved in the derivation of a concurrent version of Findp because of the use of type related information in the derivation. At this point, we believe that a different representation of the state would simplify the derivation and make the information on Table 6.8 better for comparison.

6.7.2 Bottlenecks

There are two main bottlenecks that hinder the application of the theory discussed in this thesis to more complex examples. These are: (i) the representation of the state and (ii) the level of automation provided by the mechanisation.

A richer representation of the state may be necessary to apply the algebra to more complex situations. Our derivation of Floyd-Warshall shows, for example, that dimension and type of variables are relevant in proofs.

Our representation of relations poses no limitation to the use of the refinement algebra, but it does not take full advantage of the facilities offered by Isabelle/HOL to increase automation of the theory. The level of automation for reasoning about proof obligations could be increased by lifting laws from HOL library to the type *relation* using the process of pointwise lifting provided by Isabelle's quotient package [57, 33]. This process requires the setup of relations as an abstract type using the specification constructor **typedef**. Alternative ways of representing states and relations are object of current exploration, and will feature in a future paper describing the key aspects of this thesis.

For further exploration on fine-tuning the level of automation of the algebra, it might be useful consider another examples already tackled in the literature, such as Mergesort [67] and Union-Find [59], as a wider range of examples may prevent bias in the process of determining the right attributes for automatically trigger the application of laws in Isabelle/HOL.

Chapter 7

Evaluation

> You can design with quality in mind, but in many respects you can only measure quality in retrospect, because then you can look back and say: "that was a good choice!", "that was a bad choice!".
>
> Ian Phillips, *Where did all errors go? (EDCC 2014)*

This chapter is concerned with the meta-story behind the mechanisation: the pitfalls which are common in this kind of formalisation, the design decisions that turn out to be unwieldy for handling proofs, the threats to the validity of this study, the lessons that can be learned from this project, etc. It also describes the general structure of the mechanisation, so that it can be adapted to fit user-specific purposes.

7.1 Quantitative summary

This section provides a quantitative summary of the mechanisation of rely-guarantee algebra in Isabelle/HOL. The summary provides both an architectural description, as well as quantitative information about each of the theories mechanised. The primary purpose of the quantitative information is to permit a comparison of the mechanised theories among themselves (in terms of size and structure), and clearly state the total of local assumptions that are novel in the sense that are not published in the literature [5, 43, 48, 49, 62].

The mechanisation encompasses 12 theory files (plus examples from Chapter 6). These form a predominantly linear hierarchy, shown in Figure 7.1. At the base of this hierarchy is the formalisation of states (1), where the type of objects that program variables can record is defined, together with the notion of states. Building immediately on top of (1) is a theory of relations (2), where the type relation is introduced and the definitions given in Table 2.4 are

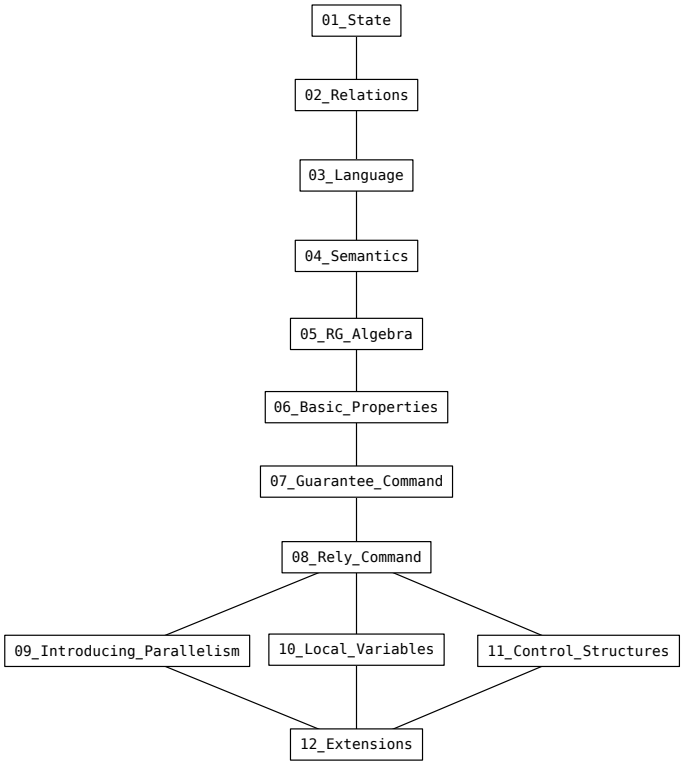


Figure 7.1 Hasse diagram of theories.

formalised together with laws to algebraically reason about relations, which were discussed in Section 4.2.1. The syntactic encoding of RG-WSL is formalised in (3); the semantic related definitions appear in (4); and the core of the algebra, that is, the local assumptions encoding the lemmas from Chapters 2 and 3, are recorded in (5). Theories (6)-(8) encode derived laws, whose organisation resembles their presentation in Chapter 3. To reduce the loading time of the theories we take advantage of the independence between laws about the rely command to split them into separate theories (9)-(11) that can be processed in parallel. Extensions, such as indexed parallelism, assignment to indexed arrays and reachable evaluations are added to the theory in (12), which serves as an user-interface. For derivation of examples, all theories are merged into a single theory which is the only one that has to be *imported* by the applications.

For a smooth introduction to the mechanisation, the recommended entry point is theory *03-Language*, followed by theory *05-RG-Algebra*. This entry point allows the user to explore the theory in a forward and backward manner, delving into details on-demand. The last theory to look should be *04-Semantics*, which concerns the formalisation of the semantics and forward simulation.

Theory	**Types**	**Definitions**	**Assumptions**	**Laws**	**Lines**
01-State	3	0	0	0	120
02-Relations	1	16	0	114	802
03-Language	4	40	0	0	223
04-Semantics	4	25	0	36	1082
05-RG-Algebra	0	8	82	0	211
06-Basic-Properties	0	0	0	118	1170
07-Guarantee-Command	0	0	0	52	1039
08-Rely-Command	0	1	0	49	1346
09-Introducing-Parallelism	0	0	0	7	454
10-Local-Variables	0	0	0	5	87
11-Control-Structures	0	0	0	11	752
12-Extensions	1	9	2	14	1032
Total	13	99	84	406	8318

Table 7.1 Quantitative summary per theory file

For each theory, we provide five measures: i) total of new types; ii) total of definitions (including abbreviations and constants); iii) total of local assumptions; iv) total of proved theorems; and v) lines of specification (including comments). Table 7.1 presents these measures separated by theory.

It can be noticed that local assumptions are centralised in two theories. This simplifies the process of auditing local assumptions. Laws spread over several theories. Isabelle automatically introduces theorems in the theory when a datatype or function is defined. For counting purposes, we only considered theorems which are explicitly added by us. Thus, theorems automatically created by Isabelle are left outside of the counting, since they are not explicitly stated.

We compare our quantitative summary with the one provided by Amstrong [2], who provides an algebraic characterisation of rely-guarantee based on Kleene algebra. There, approximately a total of 10k lines of proof scripts are used, covering 993 laws and 110 definitions. Different from our work, Amstrong formalises a trace model and uses it to prove the consistency of the underlying assumptions of his algebra. Interestingly, Amstrong reports a situation similar to that we discussed in the derivation of the concurrent version of Findp, where different cases of a proof share similarities. The largest of these situations in [2] involves a proof about the semantics that has 9 cases and amounts to several thousand lines of proof scripts. Like our observation in the discussion of Findp, they agree that the proof

Type	Quantity	Percentage
Trace equivalence	42	50%
Refinement	26	31%
Equality	8	9.5%
Others	8	9.5%
Total	84	100%

Table 7.2 Classification of local assumptions

Nature	Quantity	Percentage
Postulated	5	6%
Preexisting	79	94%
Total	84	100%

Table 7.3 Local assumptions

could be shortened by developing a proof engineering mechanism to extract the common aspects of the cases, but the approach taken there is exactly the same taken by us to handle this situation: the proof of the cases uses copy and paste proof segments where appropriate, mutatis mutandis.

7.1.1 Local assumptions

We now turn our attention to the encoding of local assumptions (i.e. laws introduced without proof) for the algebra. Table 7.2 shows that more than half of the local assumptions are formed by equations which establish trace equality between its left and right-hand side. Then, 30.6% of the local assumption are refinement equations. The category "others" includes mostly facts about termination, whose main symbol in the conclusion is logical interpretation ($\vdash$). Every local assumption is used at least once in the formalisation of the algebra, i.e. there are no orphan local assumptions. All the assumptions used in the mechanisation were introduced as lemmas in the previous Chapters[1]. In the mechanisation we identify local assumptions by the initial prefix *Ax*.

[1]The only exceptions to this rule are Definition 3.86 (Unrestricted-Rely) and Definition 3.38 (Stops), which are named in the thesis as definitions rather than lemmas. Not all lemmas discussed in the previous chapter are taken for granted in the mechanisation: we proved about a dozen of them using stratified forward simulation.

Identifier	Page	Name
3.83	97	Distribute-Rely-Post-Assertion
3.46	83	Distribute-Stops-Sequential
3.108	111	Elimination-Uses
5.26	156	Assignment-Array
5.30	160	RE-Intended-Assignment

Table 7.4 Unproved local assumptions

The majority (94%) of the local assumptions is preexisting in the literature [5, 48, 49, 43]; however, a total of 5 local assumptions, representing 6% of the total of assumptions, remains without a formal proof and is not discussed in the literature, and thus had to be postulated. For better understanding these assumptions, we list them on Table 7.4. Next we revisit each of these assumptions, and analyse them in retrospect.

7.1.2 Threats to validity

Algebraic characterisations are correct modulo the validity of their local assumptions. The method of postulating the basis for a theory from a set of basic laws generally concerns those who follow a model-oriented approach. There is a good reason for such concern: if an inconsistency is postulated, then the algebra does not distinguish certain programs that would be distinguished if a model-oriented approach had been used.

Most of the set of local assumptions taken in this work features in peer-reviewed publications in the literature that applies to the new algebraic style of rely-guarantee, such as [49, 65]. Exceptions exist, and are discussed one by one in this section.

Distribute-Rely-Post-Assertion (3.83) In order to prove or to get rid of this lemma, we suspect that it will be necessary to replace the definition of the rely command. Insight into the theory suggests that strengthening the termination condition for the rely command (see Definition 3.119 on page 119) may be the right direction to go. One word of advice though: trying to find a proof for this lemma may be a real red herring! We spent more than 1 man-month fiddling with the mechanisation, and many times we felt we were very close to produce a proof for this lemma, just to find at the end of this period that more experimentation was necessary.

Distribute-Stops-Sequential (3.46) This lemma was created to enable the proof of law 3.84 (Distribute-Rely-Sequential), which is key to allow the distribution of the rely command to the components of a sequential composition. The lemma is implicitly used in [48], even though it is never announced as a property. Refinement from right to the left is straightforward. On the other hand, the refinement from left to the right is more complicated than intuition and does not follow from the definition of *stops*. We suspect that the best way to cope with the proof of this lemma is to replace the definition of *stops* by that of weakest predicate (Definition 3.118 on page 118) and tweak this definition to enforce that this lemma holds by definition.

In our experience, proofs involving *stops* tend to be tricky, especially because of the scarcity of laws to reason about this concept. We overcome such scarcity to a reasonable degree to mechanise the laws from [48]; however, the inherent difficult in handling *stops* in proofs suggests that the study of alternative characterisations for the rely command may be well worthwhile. A reformulation of the rely command may reduce the effort necessary to extend the algebra, and hopefully eliminate the reliance on properties such as lemmas 3.83 and 3.46. In [43], for example, Hayes introduces a notion of *rely quotient*: a binary command that is inspired in the definition of division in integer arithmetic, but that applies to commands. This command is then used to formulate the rely command as the weakest command that behaves as its body when running in parallel with a program used to specify the environment. Based on the reported paper proofs, that formulation appears to lead to simplification in the development of the algebra.

Elimination-Uses (3.108) This is a fragile assumption. It formalises a strategy discussed in [48] to eliminate the uses command from around an executable program. When we introduced this lemma, we stated that we assume that once code is reached, the user will not try to further refine code. This assumption is external to the mechanisation, but is important because the traces of **uses** $X \cdot x{:=}e$ do not include program transitions that modify variables outside of X, but if the user expands the definition of assignment and strengthens the atomic command, then the refined program might modify variables outside of X. Currently, because we do not have a denotational semantics, this cannot be used to exploit the algebra. The way out of this situation is to change the classification of the uses constructor: rather than a specification constructor, it might be better to consider it an implementation constructor. Thus, at the moment of the compilation a syntactic check should be made to: i) confirm that its body is code (Definition 3.106 on page 110), and ii) confirm that all free variables

occurring in the body are indeed in X. If (i) or (ii) are violated, then the compiler should refuse to proceed the compilation.

Assignment-Array (5.26) The proof of this lemma is likely to mirror the structure of the proof of law 3.114 (Assignment-Rely-Guarantee). However, rather than proving this simplified version, it may be worth to tweak the definition of assignment first. The problem of the current definition is that indices have to be a list of naturals, but this only suffices to describe algorithms where indexing of arrays is static, like in the derivation of Floyd-Warshall. For situations where the position of a cell is dynamically determined, like within an iteration over the cells of the array, the list of indices has to be relaxed to allow each dimension to be indexed by an expression rather a natural number. Obviously, such flexibility would introduce more complexity in the proof, but this complexity can be tamed by introducing restrictions. A reasonable restriction is to require indices to not contain any shared variable. This would force the user to read and store shared variables in local variables prior to their usage to create expressions occurring in the indices of an array.

RE-Intended-Assignment (5.30) We expect the proof of this lemma to mirror the structure of the proof of law 3.115 (Assignment-Single-Reference). This lemma does not play a key role in the mechanisation: it is only used to discuss the general intuition behind the abstraction of assignments in Section 3.8.1. In practice, in can be removed without damage to the remainder of the theory.

7.2 Lessons learned

Many of the difficulties that arose in this mechanisation are common to works that involve the formalisation of programming languages. We believe that some of our observations might be useful to novice researchers in this area.

7.2.1 Isolate concepts

Before coming up with a generic mechanism to logically evaluate relations ($\vdash$), we provided an implementation of logical evaluation specifically designed for relational implication. The reason for this decision was because in [48] the underlying notion of logical evaluation always appears coupled with relational implication via the notation $p \Rrightarrow q$. It was through experimentation that we decided to decouple the concept of logical interpretation from that of

relational implication: when we attempted to formalise Definition 3.16 (Tolerate-Interference on page 72) there was a need to nest relational implication. At this point, our approach was to introduce an additional operator for relational implication without coupling it with logical interpretation. Later we observed that [25] presents the concept of logical interpretation decoupled from a particular relational operator. Thus, we decided to separate the concerns and allow logical interpretation to be applied to any relation, instead of just implications.

Logical interpretation captures the evaluation independent of the pair of states. The separation of concerns between relational operators and logical interpretation lead to a more uniform treatment of the concept and facilitated the introduction of laws to decompose proofs involving logical interpretation, such as law 4.2a (Log-Interp-Imp-Monotonic).

7.2.2 Favour usability

When encoding RG-WSL we had to decide how to represent expressions (e), pre (p) and post conditions (q). At first, we were unsure how to best represent these entities. So, we decided to characterise them using three distinct types. As the mechanisation evolved, we observed an increasing number of redundant operators to handle pre, post condition and their combination (e.g. $p \wedge_p p$, $p \wedge_{pr} q$, $q \wedge_{rp} p$, $q \wedge_r q$). It became clear to us that a better representation was possible by unifying the type of pre and post conditions. The price for this unification was the introduction of constraints: every local assumption that uses a relation to represent a precondition had to be updated with an additional proviso to state that that relation was a predicate.

While mechanising law 3.113 (Assignment-Guarantee) we discovered the need for evaluating expressions as relations. This time we had the choice of unifying the types of relations and expressions. This is an example of a modelling decision where there is no right or wrong choice. In these occasions, it is important to consider the ultimate goal of the mechanisation: in our case, to facilitate proofs in the new algebraic style proposed in [48]. Little thought is necessary to perceive that the unification of relations and expressions would bound the expressiveness of relations by the expressiveness of the grammar of expressions. Consequently, if we had decided to unify these types we would inflict on the user the burden of extending the grammar of expressions whenever an unforeseen operator (e.g. universal quantifier) is required in a specification. This, rather than facilitate the use of the theory, would make it more complex to use in practice.

Our approach was to keep the type of relations and expressions disjoint. Thus, the user only has to care about introducing binary or unary operators to represent those specification

constructions that will be part of the implementation, e.g. the specification constructor *length* to extract the size of list has to be encoded using the unary operator *len* that applies to arrays, but the universal quantifier disappears on its way to the implementation, and thus does not need to be part of the grammar of expressions.

7.2.3 Benefit from integrated proof tools

Interactive theorem provers such as Isabelle/HOL come with integrated tools to automatically search for a proof for a given conjecture, and alternatively, search for counterexamples. In the case of Isabelle/HOL, two tools come activated by default: the proof finder **sledgehammer** [94], and the counterexample generator **nitpick** [10]. We discuss three situations where a user can benefit from adopting these tools.

Use sledgehammer as a tutor

Initial expertise in proofs in this project was acquired by rewriting proofs originally found by sledgehammer. In particular, a beneficial decision was to split larger paper proofs into a collection of *intermediate laws*, each corresponding to a proof step of the paper proof. For most cases, sledgehammer was able to automatically find a proof of these intermediate laws, and also link them using laws of transitivity and substitution to reconstruct the larger proof we were aiming for.

Our first step into proof interaction was to decompose one line proofs found by sledgehammer, whose purpose was to link the intermediate laws, into a proof script made of a sequence of **apply** commands fed with transitivity and substitution laws. The second step was to split the proof of the intermediate laws themselves into proof scripts. This approach to reconstruct proofs not only serves to teach the user how to manually apply the algebra, it also helps the user to identify the laws that are more frequently chosen by sledgehammer and experiment to add attributes such as [simp] for increasing automation and proof reuse.

Use nitpick to find omissions in the encoding

When we decided to unify the type of pre (unary relation) and postconditions (binary relation) as a single type relation, a few proofs suddenly stopped working. Running nitpick over the broken proofs, we found a few intriguing counterexamples. We use the following law to illustrate one of such counter-examples. This law was necessary before the unification to reason about closure of relations.

$$\{(s, s') \mid p\ s\ s' \longrightarrow p\ s'\ s\}^* = \{(s, s') \mid p\ s\ s' \longrightarrow p\ s'\ s\}$$

Running nitpick on this conjecture, where the type of p is a relation, it generated a counter-example for p, presented as a lambda expression. At first, the counterexample looks unintelligible, but after rewriting the lambda expression (which denotes a relation) as a set of ordered pairs of states, its interpretation becomes clear. The counter-example as presented by nitpick was:

$$\begin{aligned} p = (\lambda x.\ \text{-})(a_1 := (\lambda x.\ \text{-})(a_1 := \mathit{False}, a_2 := \mathit{False}, a_3 := \mathit{True}),\\ a_2 := (\lambda x.\ \text{-})(a_1 := \mathit{True}, a_2 := \mathit{True}, a_3 := \mathit{False}),\\ a_3 := (\lambda x.\ \text{-})(a_1 := \mathit{False}, a_2 := \mathit{True}, a_3 := \mathit{False})) \end{aligned}$$

which is to interpret as the following relation:

$$p = \{(a_1,a_3), (a_2,a_1), (a_2,a_2), (a_3,a_2)\}$$

Looking back to the conjecture *Closure-Implication*, let us call R the set at the right of the equality, i.e. $R \equiv \{(s, s') \mid p\ s\ s' \longrightarrow p\ s'\ s\}$. Thus we have:

$$R = \{(a_1,a_1), (a_2,a_2), (a_3,a_3), (a_1,a_2), (a_2,a_3), (a_3,a_1)\}$$

Hence the left hand side expression, R^*, is:

$$R^* = \{(a_1,a_1), (a_1,a_2), (a_1,a_3), (a_2,a_1)\ (a_2,a_2), (a_2,a_3), (a_3,a_1), (a_3,a_2), (a_3,a_3)\}$$

Using the set representation it becomes easy to understand what is going on here. The counter-example tell us an instance of p such that $R \neq R^*$. Therefore, we can either conclude that the conjecture we are trying to prove is not valid, or that we have missed to include restrictions on the relation p. We reject the chance that the conjecture does not hold, because it was proved before the type unification, and investigate which restrictions can be made to rule out the instance of p provided by nitpick. The restriction that is missing here is to say that *p must be a predicate*. The problem is solved by introducing *pred p* in the premises of the theorem. Before type unification such restriction was embedded in the type of p, therefore it was transparent and overlooked when unifying the types.

This shows that our initial modelling via disjoint types for pre and postconditions actually payed off in guiding the *provability* of the needed theorems. The unification of types, when

justifiable, may require the introduction of explicit assumptions and restrictions that are initially unclear in the earlier modelling due to distinction of types.

Learn to formulate satisfiability questions

This examples illustrates the application of **sledgehammer** to search for instances of quantified variables that meet certain criteria. The insight behind this example is that for some cases the user may be able to benefit from the interaction between Isabelle and SMT solvers to find out suitable ways of eliminating quantifiers to proceed in a proof. Instantiations are one of the most creative and difficult proof steps in proofs in general, and any assistance with this type of step is therefore worthwhile.

To illustrate the usage of **sledgehammer**, we turn our attention to a sub-goal that raises in the proof of law 3.100 (Rely-Test on page 107). To complete the proof we are required to show

$$\mathit{tol\text{-}interf}(p, r^{**} \wedge b_0\text{`}, r)$$

assuming p and b_0 to be *predicates*, and the premises

$$\begin{array}{ll} \text{P1:} & \vdash r \Rightarrow (p \Rightarrow p\text{`}) \\ \text{P2:} & \vdash p \wedge r \Rightarrow (b_0 \Rightarrow b_0\text{`}) \end{array}$$

Expanding *tol-interf* (Definition 3.16 on page 72) we sub-divide the proof into three sub-proofs:

$$\begin{array}{rl} (i) & \vdash r \Rightarrow (p \Rightarrow p\text{`}) \\ (ii) & \vdash p \wedge (r \,;\, r^{**} \wedge b_0\text{`}) \Rightarrow r^{**} \wedge b_0\text{`} \\ (iii) & \vdash p \wedge (r^{**} \wedge b_0\text{`} \,;\, r) \Rightarrow r^{**} \wedge b_0\text{`} \end{array}$$

Subgoal (*i*) follows immediately from P1. Subgoal (*ii*) is trivial and solved expanding the definitions involved. For (*iii*), given that (b_0`) holds before the relational composition, we are able to strengthen the right-hand side of relational composition (_ ; _) by introducing b_0. This transformation leaves the goal (*iii*) in the shape:

$$(iii) \quad \vdash p \wedge (r^{**} \wedge b_0\text{`} \,;\, b_0 \wedge r) \Rightarrow r^{**} \wedge b_0\text{`}$$

We then apply law 4.2a (Log-Interp-Imp-Monotonic) to this goal to split the conjunction in the conclusion. The implication of the left conjunct (r^{**}) is proved by expanding the definitions involved, and using the property:

$$[\![r^{**}\ s\ s'; r\ s'\ s'']\!] \Longrightarrow r^{**}\ s\ s''$$

which is natively available in the package of transitive closure distributed with Isabelle. The sub-goal that remains is:

$$\vdash p \wedge (r^{**} \wedge b_0\,`\ ; b_0 \wedge r) \Rightarrow b_0\,`$$

We then enrich the set of assumptions with $\vdash r^{**} \Rightarrow p \Rightarrow p\,`$, which can be inferred from the application of law 3.17a to assumption P1. Expanding the definitions we are left with the following sub-goal in Isabelle (which includes the premises P1 and P2):

$\bigwedge St\ St'\ s''.$
$pred\ p \Longrightarrow$
$pred\ b_0 \Longrightarrow$
$\forall St\ St'.\ p\ St\ St' \wedge r\ St\ St' \longrightarrow b_0\ St\ St' \longrightarrow b_0\ St'\ St \Longrightarrow$
$\forall St\ St'.\ r^{**}\ St\ St' \longrightarrow p\ St\ St' \longrightarrow p\ St'\ St \Longrightarrow$
$p\ St\ St' \Longrightarrow r^{**}\ St\ s'' \Longrightarrow b_0\ s''\ St \Longrightarrow b_0\ s''\ St' \Longrightarrow r\ s''\ St' \Longrightarrow$
$b_0\ St'\ St$

From this point we take an *unsafe*[2] path: weakening the premises by trading the universal quantifiers by particular instances of them. The difficulty to complete the proof lies on the fact that we do not know which values to use to replace the quantified variables, and neither whether this strategy renders the goal unprovable. The heuristic we illustrate next was only observed after failing a number of times to find suitable instances for the quantified variables. We replace the quantified states using St_1, ..., St_4 as illustrated next:

$\bigwedge St\ St'\ s''.$
$pred\ p \Longrightarrow$
$pred\ b_0 \Longrightarrow$
$p\ St_1\ St_3 \wedge r\ St_1\ St_3 \longrightarrow b_0\ St_1\ St_3 \longrightarrow b_0\ St_3\ St_1 \Longrightarrow$
$r^{**}\ St_2\ St_4 \longrightarrow p\ St_2\ St_4 \longrightarrow p\ St_4\ St_2 \Longrightarrow$
$p\ St\ St' \Longrightarrow r^{**}\ St\ s'' \Longrightarrow b_0\ s''\ St \Longrightarrow b_0\ s''\ St' \Longrightarrow r\ s''\ St' \Longrightarrow$
$b_0\ St'\ St$

[2]Recall from Section 1.6.4 on page 13 that Isabelle uses the terminology *safe* to refer to proof steps that preserve provability, and *unsafe* to refer to those that can render a predicate unprovable.

By applying safe rules, this goal is split into 12 sub-goals. We turn this collection of sub-goals into a satisfiability problem and use **sledgehammer** to answer the question: (Q1) are there any instances of St_1, ..., St_4 such that the 12 sub-goals are collectively satisfied? Formally, this question takes the shape of the following goal

$$
\begin{aligned}
&\bigwedge St\ St'\ s''.\\
&pred\ p \Longrightarrow pred\ b_0 \Longrightarrow\\
&p\ St\ St' \Longrightarrow r^{**}\ St\ s'' \Longrightarrow b_0\ s''\ St \Longrightarrow\\
&b_0\ s''\ St' \Longrightarrow r\ s''\ St' \Longrightarrow (\exists\ St_1\ St_2\ St_3\ St_4.\\
&(\neg\ r^{**}\ St_2\ St_4 \longrightarrow \neg\ b_0\ St'\ St \longrightarrow p\ St_1\ St_3) \wedge\\
&(\neg\ r^{**}\ St_2\ St_4 \longrightarrow \neg\ b_0\ St'\ St \longrightarrow r^{**}\ St_1\ St_3) \wedge\\
&(\neg\ p\ St_2\ St_4 \longrightarrow \neg\ b_0\ St'\ St \longrightarrow p\ St_1\ St_3) \wedge\\
&(\neg\ p\ St_2\ St_4 \longrightarrow \neg\ b_0\ St'\ St \longrightarrow r^{**}\ St_1\ St_3) \wedge\\
&(p\ St_4\ St_2 \longrightarrow \neg\ b_0\ St'\ St \longrightarrow p\ St_1\ St_3) \wedge\\
&(p\ St_4\ St_2 \longrightarrow \neg\ b_0\ St'\ St \longrightarrow r^{**}\ St_1\ St_3) \wedge\\
&(\neg\ r^{**}\ St_2\ St_4 \longrightarrow \neg\ b_0\ St_1\ St_3 \longrightarrow b_0\ St'\ St) \wedge\\
&(\neg\ r^{**}\ St_2\ St_4 \longrightarrow b_0\ St_3\ St_1 \longrightarrow b_0\ St'\ St) \wedge\\
&(\neg\ b_0\ St_1\ St_3 \longrightarrow \neg\ p\ St_2\ St_4 \longrightarrow b_0\ St'\ St) \wedge\\
&(\neg\ b_0\ St_1\ St_3 \longrightarrow p\ St_4\ St_2 \longrightarrow b_0\ St'\ St) \wedge\\
&(\ b_0\ St_3\ St_1 \longrightarrow \neg\ p\ St_2\ St_4 \longrightarrow b_0\ St'\ St) \wedge\\
&(\ b_0\ St_3\ St_1 \longrightarrow p\ St_4\ St_2 \longrightarrow b_0\ St'\ St))
\end{aligned}
$$

In general, sledgehammer's failure to find a proof represents an inconclusive answer for the validity of a conjecture. On the other hand, the discovery of a proof tell us that the conjecture (restriction) is valid (can be meet). For this conjecture, the fact that sledgehammer found a proof tell us that the answer for question Q1 is positive, i.e. the proof strategy of trading the quantifiers by specific values preserves the provability of the goal in *iii*. Knowing this fact, however, does not help us to identify a suitable instantiation for St_1, St_2, St_3 and St_4. To assist the search for particular instantiations, the theorem formulating Q1 is updated to introduce constrains over $St_1..St_4$ which give away hints about potential instantiations.

The first hint we ask for, is if there is an instantiation satisfying $St_1 = St_4$. The answer is positive. After re-examining the predicate we ask for another hint: can St_1 be any of the bounded variables St, St' and s''? The answer is positive. Using this strategy to extract information from Isabelle we manage to discover that $St_1 = s''$ and $St_2 = St$, $St_3 = St'$ and $St_4 = s''$ is a suitable choice to eliminate the universal quantifier yielding a provable goal.

Fine-tune your proof assistants

Both sledgehammer [94] and nitpick [10] can take parameters when called, which allow the user to customize their timeout, list of SMT solvers they interact with, list of lemmas they are allowed to use, etc. By default these tools timeout after 30 seconds. This value is rather short compared to the timeout we mostly used when calling these tools in this mechanisation, which was 720 seconds. We reach this value empirically: sometimes we knew a proof exist, because we had a paper proof available, and we just wanted to see if sledgehammer was able to find that same proof. Thus, we called sledgehammer with increasing amounts of time, and from our experience 720s suffice for most of the cases where sledgehammer in fact succeed.

Sometimes sledgehammer succeed by finding an alternative proof than the one we want it to return. In such situations, we analysed the rationale of the alternative proof by reconstructing the automatic proof via a stepwise application of the laws in Isabelle. To explore the limits of sledgehammer in the cases where it found a proof that was not exactly the one we wanted, we use some of its options to indicate the lemmas it should use, or then to forbid specific lemmas from being used by it. The lemmas we forbid of being used were those that were present on the alternative proof and absent in the proof we wanted. During this experimentation we found that it is not much effective to provide sledgehammer with a set of relevant lemmas that are suggestive of a given proof path. Instead, it appears to be more effective to tell sledgehammer what it *cannot* use if we want it to return a specific proof. The syntax for parameter passing for both sledgehammer and nitpick can be found in the tutorials distributed with Isabelle [12, 11].

7.3 Related work

Next we compare our work to preexisting works that propose a refinement calculus for rely-guarantee or provide mechanised support to reason about rely-guarantee in Isabelle/HOL.

7.3.1 Systematic parallel programming

Dingel [32] formalises a shared-variable and message-passing refinement calculus for rely-guarantee based on a denotational semantics developed by Brookes [21]. Rely-guarantee specifications are encoded as a tuple of four predicates, representing pre, rely, guarantee and post conditions. Message-passing parallelism is emulated by treating a channel as a shared queue: the read operation extracts and assigns the head of the queue to a specified variable,

whereas the writing operation writes the value specified in an expression to the specified queue.

Dingel follows Brookes and describes parallel composition as a fair operator using a trace semantics. This modelling decision allows the derivation of programs whose termination depends on environment actions, and makes it possible to reason about busy wait programs and liveness properties. Evaluation of expressions is not assumed to be atomic, but is assumed that it always terminates (i.e. no treatment of undefinedness is given). In the algebra discussed in this thesis we do not assume that the evaluation of expressions always terminate. Instead, the provisos of the refinement laws are used to prevent the user from introducing undefined expressions in programs.

Dingel also considers assignment to indexed arrays. Similar to our modelling, indices are statically determined (i.e. the rule for introducing assignment to indexed arrays expects the indices to be natural numbers instead of expressions). Different from our work, the theory proposed in [32] has not been mechanised.

7.3.2 Formal analysis of concurrent programs

Amstrong [3, 2] introduces a Kleene algebra to model rely-guarantee and interference based reasoning. All rules except the assignment axiom are derived within the algebra before instantiating it for specific models of concurrency. Similar to [45], the rely command is formalised using a concept that is related to parallel composition via a Galois connection (cf. Definition 7.1):

$$(y \leq x/z) \Leftrightarrow (x \parallel y \leq z)$$

The program $x \,/\, z$ is the weakest program such that when placed with parallel with program x, the parallel composition behaves as z. Rely and guarantee conditions are required to satisfy a set of axioms that allow one to absorb, distribute and reason about interference. These axioms are similar in intent to the lemmas 3.74 (Properties-Finite-Interference) and 3.75 (Distribute-Interference) in this work, and they imply that the rely and guarantee conditions are reflexive-transitive. A rely-guarantee specification in [3, 2] has the following structure:

$$(pre \cdot (rely \parallel c) \leq post) \wedge (c \leq guar)$$

This encoding of rely-guarantee specifications diverges both from the view of this thesis and that of the more recent works by Hayes [43, 45] in the sense that it forces a program (c)

to unconditionally implement the guarantee condition (*guar*), even from states where the precondition (*pre*) does not hold and the environment has failed to meet the rely condition (*rely*).

An important aspect of [3] is that it has been applied to verify simple examples from the literature, such as Findp. Amstrong echoes Nieto's [97] view that even seemingly straightforward concurrency verification tasks can be tedious and complex, and concludes with the position that more work has to be done to manage the complexity of verification proofs within interactive theorem provers. Between the publication of [3] and the time of this writing, an extended discussion of the algebra has been published by Armstrong as a PhD thesis [2]. In this more recent version, the author uses *Eisbach*, a new feature of Isabelle that allows users to construct customised proof tactics. It is reported that the use of these proof tactics reduced part of the tedious work involved in the verification of Findp.

7.3.3 The rely-guarantee method in Isabelle/HOL

Nieto [97, 82] provides the first formalisation of Owicki-Gries and rely-guarantee methods in Isabelle/HOL[3]. The focus of the mechanisation is to provide means of reasoning about implemented code via the backward application of proof rules, rather than support correction by construction as aims the work in this thesis. The formalisation discussed by Nieto includes a synchronisation primitive (**await**) which is used to model mutual exclusion algorithms.

Rely-guarantee specifications are represented as a tuple of four components: two binary relations representing the rely and guarantee conditions, and two predicates denoting pre and post conditions. Indexed parallel composition of an arbitrary number of programs is supported. Interestingly, but not surprisingly, their assumptions for the indexed parallel introduction rule coincide with ours for law 5.9 (Introduce-Multi-Parallel-Parameterised). The main weakness of [82] is the restriction imposed in the language used to write programs: it does not support nested parallelism. Our algebra does not have this limitation.

7.3.4 On the Mechanisation of Rely-Guarantee in Coq

Moreira et. al. provide mechanical support to a rely-guarantee program logic in Coq [77]. Their mechanisation follows closely the formalisation of Coleman and Jones [26], and supports an simple shared-variable programming language extended with parallelism and atomic blocks. Different from [26], and from the work in this thesis, expression evaluation

[3]The encoding discussed in Nieto's thesis has been kept up-to-date with recent developments in Isabelle, and is distributed in the folder of examples of the tool.

in assignments and conditions is assumed to be atomic, and postconditions are modelled as predicates of a single state. Definedness is not discussed in their work, and indexed parallelism is not investigated.

Inline with the work on this thesis, Moreira et. al. use total functions to represent states. They model the co-domain of the state function using a datatype containing all possible valuations for variables, in a similar way to our formalisation in Definition 2.1. Projections functions are also used in their derivations, but they do not discuss the limitations inherent with this modelling decision.

The main criticism of Moreira et. al. to the applicability of their proof system is that it does not include proof rules to reason about the introduction of auxiliary variables in proofs. Comparatively, we did not provide laws to eliminate auxiliary variables from a program, but we do not consider this to be a major limitation to our approach. Auxiliary variables introduced during a development can be easily identified in concrete programs, and can be manually eliminated. Similar to the work on this thesis, Moreira et. al. models parallel composition as an unfair operator. From our perspective, this causes major limitations than the lack of a proof rule to eliminate auxiliary variables.

7.3.5 Generalised rely-guarantee concurrency

Hayes [43] develops a general algebra to reason about concurrency by means of rely and guarantee abstractions. Departing from the usual relational view, processes are used to characterise these abstractions. A surprising consequence of this decision is that one can express rely and guarantee conditions that are not possible when one considers these abstractions as relations, modulo the use of auxiliary variables. For example, compared to the algebra discussed in this thesis, the algebra developed in [43] allows one to formalise a guarantee condition such as $\langle idrel \rangle^\star \,;\, \langle g \rangle \,;\, \langle idrel \rangle^\omega$, that specifies that a state transition satisfying g occurs exactly once, but allows any number of stuttering steps before and after that transition[4].

The key advantage of [43] over the algebra discussed in this thesis is the simplicity of proofs behind the derivation of important laws, such as parallel introduction, distributive laws and nesting of operators. The simplicity results from a combination of factors: (i) formalisation of lattice theoretic properties involving the key primitive constructors; (ii) dropping of the requirement of a rely command to terminate in presence of potentially infinite interference (cf. Requirement 3.71); (iii) dropping of the notion of earlier termination of

[4] In retrospect, the characterisation of rely and guarantee as processes appears to be the right direction for encoding the concept intermediate properties proposed in Section 5.8.2.

parallel processes at the semantic level (cf. rules 2.54 and 2.55); (iv) assumption that rely and guarantee conditions are reflexive; etc.

Strict conjunction (named in [43] as *weak conjunction*) continues to provide the basis for characterising the guarantee command. On the other hand, the rely command is given a different and simpler formulation using the notion of *rely-quotient*. The new operator is defined as:

$$c \mathbin{//} i \equiv \bigsqcap \{d.\ (c \sqsubseteq d \parallel i)\} \tag{7.1}$$

where i is a process encoding the rely condition and c is the body of the rely condition[5]. For example, compared to the algebra discussed in this thesis, an approximation for **rely** $(r, z) \cdot c$ would be

$$(c \mathbin{//} \langle r \vee_r \mathit{idrel} \rangle^\omega) \mathbin{//} \langle z \vee_r \mathit{idrel} \rangle^\omega$$

Algebraic properties that manipulate the notion of rely-quotient are simplified by the fact that it does not demand the termination of d, while Definition 3.72 (Rely) requires the termination of d. This simplification pays off in proofs: the proof of laws for distributing the rely command over several commands including sequential composition becomes quite simple.

Different from the algebra in this thesis, which includes the primitive **state** $x \mapsto v \cdot c$ to implement the notion of hiding of variables in the traces of a program, there is no discussion of local variables in [43] nor in its mechanised version [35]. Inspecting the mechanisation, it can be seen that it gives a semantic rather than a syntactic characterisation of the wide-spectrum language, and thus avoids the syntactic problem we had in the characterisation of RG-WSL. At the moment of this writing, no study has been published showing the practical usage of the algebra in [43] to derive concrete programs.

7.3.6 An algebra of synchronous atomic steps

In [45], Hayes and colleagues develop an abstract algebra that can be instantiated to the relational rely-guarantee from [43] (discussed above) as well as to different processes algebras such as CSP [53] and CCS and SCCS [75].

The underlying programming language is defined in terms of very basic primitives: tests, atomic steps, sequential composition, strict conjunction, parallel composition, demonic and

[5] An indirect definition for rely-quotient is provided by $(c \mathbin{//} i \sqsubseteq d) \Leftrightarrow (c \sqsubseteq d \parallel i)$

angelic non-determinism, fix point operators and special elements to represent the identities for the language constructors. Many of the proofs rely on a key theorem that provides a canonical representation for a command.

Guarantees are encoded based on the notion of strict conjunction and infinite iteration of atomic commands (just like in [43]). Rely is given a novel formulation based on the strict conjunction and the derived command (**assume** r), which behaves silently if the environment transitions respect the relation r between its before and after state but aborts otherwise. Interestingly, this formulation of the rely command leads to a nice feature of the algebra not discussed in [45]: the intricate aspects that emerge as consequence of different arrangements (nesting) of a rely and guarantee command disappear in the abstract algebra. This is because a weak conjunction is associative and commutative, and a rely-guarantee specification in the abstract algebra has the general shape:

$$\{p\} \; ; ((\textbf{rely}\ r) \Cap (\textbf{guar}\ g) \Cap [q])$$

Thus, one can apply commutative and associative laws to switch the arrangement of rely and guarantee constructors. Common to the formalisation in this thesis, and the works in [43, 45], is the fact that there is a clear separation of concerns between pre, rely, guarantee and postcondition, allowing each of these components to be refined in isolation.

To the current author, the lack of practical examples is the main critic to [45]. Even though it is reported that the abstract algebra was mechanised in Isabelle/HOL, several practical issues are left open, such as details of the programming language which have to be made more concrete before the abstract algebra can be put in action.

7.3.7 Algebraic Principles for Program Correctness Tools in Isabelle/HOL

Gomes [39] formalises a KAT based algebraic framework to verify and refine non-deterministic sequential programs, and extends the framework to support the verification of simple shared-variable concurrent programs, as well as a separate extension to support reasoning about sequential programs that manipulate pointers. To support concurrency, the language is extended with parallel composition. The semantics for parallel composition is given by an unfair scheduler. The scheduler unfolds a concurrent program into a non-deterministic program annotated with program counters which play the role as labels.

Although the parallel composition is assigned to an unfair semantics, the ability of referring to program counters in assertions makes their algebra powerful enough to prove simple facts such as progress in mutual exclusion algorithms, and absence of deadlock. Proofs are interactive, though specialised tactics devised using Eisbach are provided to simplify user interaction. The application of the framework to verify concurrent programs is discussed using a mutual exclusion algorithm, and a fragment of code that is responsible for the initialisation phase of a handshake protocol.

The main objective of [39] is to investigate the principles that should be considered in order to build software verification tools where control flow laws and data-level laws are formalised independently. Although such separation is not elaborated our work, we make fewer assumptions with respect to atomicity of expression evaluation and language design. For example, we account for interference during expression evaluation, and we support nested parallelism. None of these is supported by the formalisation proposed in [39].

Chapter 8

Conclusion

> We must not expect to find solutions to all of the problems presented by building computer systems in standard mathematics. Nor — unless we are unbelievably fortunate — will we always find beautiful mathematical solutions first time; but publishing an attempt which does solve a problem could spur others to show the way to a cleaner formulation. In any case, this is a more honest approach than ignoring all aspects of a problem which do not fit our current formalism.
>
> Cliff Jones, *Some mistakes I have made and what I have learned from them, 1998*

This chapter revisits the thesis proposition, and discusses how it was addressed it in the context of the contributions presented in previous chapters. It also discusses limitations of this research with respect to the architecture of recent processors. Finally, Section 8.4 discusses future works.

8.1 Summary and contributions

Recent investigations in formal derivation of concurrent programs have explored the integration of rely-guarantee into a refinement calculus, establishing a programming methodology to derive code for shared-variable programs from specifications written in rely-guarantee [48, 65]. In this thesis, we mechanised the refinement calculus proposed in [48] as a refinement algebra because we were mainly interested in investigating practical aspects of the application of the theory. In the course of the mechanisation we located and fixed inconsistencies present in definitions and proofs from [48], and extended that theory with concepts to enable the derivation of programs involving indexed parallelism and assignment to arrays.

The earliest evidence of the benefit of using Isabelle/HOL in this project was the discovery of a problem in the syntax of the programming language that went unnoticed in the literature [47–49, 65]. The problem and its solution are discussed in Chapter 2, but the solution we propose comes at the cost of introducing a conflict between the semantics and the refinement laws assumed to hold in the characterisation of the algebra. This conflict and its consequences are further explored in Chapter 4, where we discuss the methodology adopted to mechanise the refinement calculus proposed in [48], and justify our design decision with respect to alternative designs.

Chapter 3 introduces the laws and definitions that form the basis of the refinement calculus discussed in [48]. This chapter also includes top-level refinement laws used in derivations or to guide discussions in the next chapters, as well as contributions of our own. The latter are carefully tagged with the keyword **contrib**.

Great care was taken during the encoding of the calculus to prevent redundancy of concepts. The mechanised proofs resemble the algebraic style of pen-and-paper proofs [48], modulo our preference for the procedural style in proofs for the sake of productivity. For most laws, their application generates proof obligations born out of every verification task. In Chapters 4 and 7 we discuss the infrastructure we developed to enable the user's effort to discharge proof obligations to be reduced.

Design decisions behind the mechanisation enable Isabelle/HOL to automatically take care of some of the proof obligations via the use of automatically proof methods such as *simp*. For the cases where proof obligations are not automatic discharged, the user can apply a collection of laws to compositionally reason about logical interpretation and unrestriction (see Chapter 2), as well as stability (see Chapter 3). Sometimes, the invocation of *sledgehammer*, Isabelle's proof finder, may suffice to discharge proof obligations. Ultimately, for proofs involving expandable concepts, the user can unfold the definitions and call the simplifier. In our experience, we found this latter strategy to be specially useful to explore proof obligations before constructing a clearer proof. All design decisions we took cooperate to facilitate the construction of proofs in the new refinement calculus in the style of [48]. Moreover, the availability of built-in tools from Isabelle, such as *find-theorems*, can assist the user in the selection of laws that can be applied at each stage of the development, by means of pattern matching against goals. This contributes to increase the productivity and allows users to assess the whole database of laws almost instantaneously.

The mechanisation exceeds the calculus proposed by Hayes et. al. in [48]. Extensions are the subject of Chapter 5, which covers indexed parallelism, assignment to indexed arrays, corrections in the operational semantics and additional abstractions to reason about

programs that constrain environment steps and approximate non-atomic evaluation using relations. Each extension is illustrated at least once in Chapter 6. Some of our extensions, such as indexed parallelism, are designed so that they facilitate rather than complicate the representation of sophisticated algorithms. For example, previous formalisations of indexed parallelism require indices of processes to be natural numbers [32, 82]. We break this convention and allow indices to be of any type. As result, we simplify the representation of algorithms that operate on non-linear data structures, such as Floyd-Warshall, where the index of a cell of a matrix is used to identify a concurrent process. Had we not provided such flexibility, the formalisation of Floyd-Warshall would involve nested parallelism.

Chapter 6 demonstrate the benefits of the careful balance between shallow and deep-embedding of constructors in practice. The derivation of examples shows that the user only has to care about the syntax of binary and unary operators that are required at the implementation level (code). Together with the fact that states do not have to be formalised prior to the derivation of examples, this minimises the setup phase that precedes formal analysis in other mechanised theories, such as [82, 2]. There is a price to pay though for skipping formalisation of the state: the notion of state offered by our mechanisation is weakly typed. On one side, this provides flexibility to derive programs that are not type consistent, but it can also bring complications to the derivation of algorithms where knowledge about types is relevant in proofs, such as Floyd-Warshall.

On Chapter 7 we revisit the mechanisation and analyse the theories quantitatively. We draw some general lessons that can be applied in future projects involving the encoding of refinement algebras in Isabelle/HOL, and discuss the threats to the validity of our study. The key weakness is the fact that the mechanisation is based on a set of assumptions that are not verified against a semantic model within the mechanisation itself. Nevertheless, the consistency of the set of laws taken for form the basis of the algebra has been verified by manual proofs [48], and many of the fundamental laws from which we built upon has been peer-reviewed [65]. Chapter 7 lists all the assumptions of the algebra for which a proof is still unknown.

Revisited thesis proposition

At the introduction of this thesis we delineate our journey by setting three questions (Section 1.4). The first question (Q1) asks if it the calculus proposed in [48] can be given automated theorem proving support via the algebraic approach discussed in [54]. We answer this question positively, using Chapter 4 to discuss how this can be achieved. The second question (Q2) considers if the theory can be used to derive concurrent programs involving

parallelism and arrays. Similarly, the answer for this question is positive, and is supported by the derivations discussed on Chapter 6. Finally, the last question concerns the prevention of the introduction of undefined expressions in concurrent programs while retaining the ability of extending the grammar of expressions on-the-fly. We propose a solution for this problem in Chapter 2, by carefully combining shallow-embedding and deep-embedding in the design of the grammar of expressions.

8.2 Takeaway message

This document is targeted to researchers and practitioners of formal methods who are willing to explore the practical use and limitations of the rely-guarantee approach. We provide a prototype in which program derivation experiments can be rigorously conducted. Our greatest limitation is that we do not have laws to reason about fair parallelism. Many interesting concurrent programs involve busy waiting and can only be proved correct if parallelism is assumed to be fair. To a certain degree, auxiliary variables can be used to reason about progress in programs subject to unfair parallelism, but this is not investigated in this thesis. It is natural for us to think that the first step towards making the prototype more applicable is to support fair parallelism, and then increase the level of automation of the mechanisation.

For the layman, the takeaway message of this thesis is that formal concurrent program derivation can be done compositionally. Assumptions and commitments defined at top-level specifications can be refined and distributed throughout the development of concurrent programs; assumptions can be widened, and commitments can be narrowed – in the same way that preconditions and postconditions can be manipulated. The rely-guarantee method conquers compositionality by abstracting the code of the program and its environment using binary relations: the guarantee and rely condition. For situations where interaction between processes is limited, as illustrated in the derivations included in the thesis, the control-flow of the environment can be fully abstracted away in the rely condition. However, for cases where the environment's control-flow is relevant to reason about correctness, one may need the so-called auxiliary variables to incorporate the relevant part of the control-flow of the environment into the rely-condition.

Any experienced programmer of a parallel platform knows that expression evaluation in presence of interference is non-deterministic and does not observe the laws of mathematics. This happens because variables can be sampled from different states. For this reason, one has to be careful when designing the conditions used in control structures embedded in concurrent programs. The calculus approaches this issue by imposing syntactic restrictions

on expressions that can appear in concrete programs. Expressions can contain at most one shared variable, and no more than a single reference to it. This restriction does not affect the expressiveness of programs that can be derived using the calculus because expressions can be decomposed into sub-expressions that are evaluated and stored in a portion of memory that is protected from interference. We believe that this approach is useful even for programmers who are unwilling to apply this refinement calculus because it breaks expression evaluation into manageable pieces that are easier to debug.

The mechanisation provided with this thesis is an academic prototype, but it illustrates a few principles that are applied more broadly to the construction of formal tools independent of the theorem prover chosen as a platform, and the underlying theory being encoded. These are: (i) isolation of concepts; (ii) user-centred development; (iii) support of automated proof tools. These principles are at the heart of formal tools that trade expressiveness by automation, such as Dafny [72], Infer [23] and Simulink Design Verifier[1], that have been proved useful in industry. In this sense, the study of our prototype serves to understand better how to make design decisions in the process of constructing formal tools, but there is a long journey to go before our prototype could become attractive for industrial application.

8.3 Limitations

The algebra discussed in this thesis departs from the refinement calculus proposed in [48], and thus its scope is influenced by a mixture of previously taken decisions from [48] and decisions taken along the mechanisation.

The main limitation of this work is that it does not provide a general method to check the consistency of the assumptions of the algebra. Partially counterbalancing this limitation, we implement a method for checking the consistency of few refinement laws via stratified forward simulation (Section 2.10), and use it to reduce the number of assumptions that have to be made to formalise the algebra. The extension of the mechanisation with a traces semantics is left as future research.

Another limitation of this work is that it also does not reflect the architecture of current processors. Nowadays processors are designed with the so-called *weak memory models* [90], that establishes how each processor updates the memory and affect the shared state seen by other processors. In this model, semicolon ceases to be a sequencing operator. From a formal perspective, there is no guarantee that memory writes made by a process will be immediately visible by other processes; instead, each process can hold a local writing buffer, that will

[1] https://uk.mathworks.com/products/sldesignverifier.html

$$
\begin{array}{l}
flag_1 := 0; \\
flag_2 := 0; \\
\left(\begin{array}{l|l}
flag_1 := 1; & flag_2 := 1; \\
\text{if } (flag_2 = 0) \{ & \text{if } (flag_1 = 0) \{ \\
\quad \text{CR1} & \quad \text{CR2} \\
\} & \}
\end{array}\right)
\end{array}
$$

Figure 8.1 Dekker's algorithm for critical region

be committed to the memory in the order that are emitted by the process. For single-thread programs, and programs without data races, no difference in the behaviour is observable. But for concurrent programs, where external observations to the shared state are relevant, the memory does not reflect the writes made by a program until the local cache is flushed into the shared memory. In order to enforce that the shared memory continues to obey a sequentially consistent memory model, developers make usage of memory barriers, also known as memory fences. These force the local buffer to be flushed to the shared memory. In practice, the usage of memory barriers slow down the parallelism between processes by several orders of magnitude, and developers try to use as fewer barriers as possible to reduce loss of performance.

Without memory barriers, classical algorithms such as Hugo Simpson's four-slot [103] and Dekker's algorithm for critical region, shown in Figure 8.1, are not guaranteed to work as expected. In the case of Dekker's algorithm, $flag_1$ and $flag_2$ in the conditional of the if may not reflect the changes made by the environment on these variables.

We share Richard Bornat's view [16] that the time spent on developing theories and verifying algorithms under the assumption of a sequentially consistent model (SC) is not wasted. Verification of concurrent programs is already a difficult task under the assumption of a SC model, and research grounded on the assumption of these models can shed light into research that is establishing the basis for program verification on weak memory models. Additionally, we sustain the hope that the development of tools to automatically introduce fences on code, such as [1] and [19], may allow researchers on program verification to abstract away from such hardware details. In short, these tools decide if the SC semantics coincides with the semantics on a weak memory model; if not, the tool synthesises a minimal number of memory barriers that enforce robustness, thus hiding the weak memory model from the programmer and providing the illusion of sequentially consistency. Obviously, since too many fences slow down the performance, it might be the case that efficient algorithms

designed for SC models might not be efficient when transplanted into modern architectures. Nevertheless, from a functional perspective they would still be correct.

8.4 Future work

Wellfounded relations As discussed in Chapters 2, we did not succeed in using the definition of well-founded relations formalised in this work to prove that the relevant relations used in Chapter 6 are well-founded. This is not a limitation of this work, but a technical issue that can be solved by further experimentation. In particular, it may be useful to investigate how the termination of loops is proved in other mechanised works, such [82, 2].

Fair parallelism Many parallel algorithms need to assume fairness in order to achieve their goals. For example, reasoning about programs whose termination depends on environment actions requires one to model parallel composition as a fair operator: a program ensures that it will achieve its postcondition as long as it will be given a chance to execute; in turn, if it depends on a resource locked by the environment, it can only make progress if the environment is given the chance to execute as well (assuming that the resource will eventually become available). To derive this kind of programs using the algebra, one would need to introduce a semantic characterisation of parallel composition as a fair operator. Alternatively, in [43] the author suggests that it may be possible to reason algebraically about fair parallelism even if the language does not have this operator, by introducing fair parallel composition as a derived command that refines an unfair parallel composition. The set of assumptions necessary for reasoning in this approach are not discussed in [43], but intuitively it would include laws for example to introduce mutual exclusion schemes from a specification command. Research in this direction might benefit from the refinement schemes discussed in [32].

Alternative definition for the rely command As discussed in Section 7.1.2, lemma Distribute-Rely-Post-Assertion (3.83) cannot be derived from the assumptions of the algebra. The problem amounts to the termination condition for the rely command in presence of potentially infinite interference. Directions for solving the problem include dropping this requirement, as done in [43], or then strengthening the termination condition of the rely command, as suggested in Definition 3.119. The last approach is open to experimentation. The ultimate goal one should consider while working in this direction is to enable the proof of law 3.84 (Distribute-Rely-Sequential).

Encoding states using records As noticed in the derivation of Floyd-Warshall in Setcion 6.6, the characterisation of states as a total function from variable names to values does not provide sufficient information about the type and dimension of variables. The solution proposed there was to include information about the type of the variables in the precondition of the abstract specification. We believe that a better approach is possible by changing the formalisation of the state, so that type and dimension are recorded. One direction for that is to formalise the state as a record, as done in [82, 2]. Another approach is to use a total function from variable names to a triple, containing the value, type and dimension of a variable. In this direction, it might be necessary to extend RG-WSL with type declaration for variables. We suspect that the usage of records may lead to a simpler solution, because it can be used to ensure that relations are type consistent, while the use of total states would still require the introduction of well-formed conditions to ensure that relations are type consistent. Extra thinking would be necessary to model local variables using records, since local variables might have the name of a preexisting variable but a different type and dimension.

Encoding of relations We suspect that the level of automation in proofs involving logical interpretation can be improved by instantiating relations as a lattice (see Section 4.4). The practical problem with this characterisation is that type synonyms cannot be used to instantiate type classes. It appears necessary to first wrap the type relation using a datatype in Isabelle, so that it can be instantiated as a lattice. Work in this direction can benefit from law Law 4.16 (Lattice-Exchange) that shows how the definitions involving binary relations can be mapped to lattice operators.

Type constraints Currently, a number of proof obligations involving type constraints are generated during the application of the algebra. We suspect that using Eisbach, Isabelle's proof method language, it may be possible to construct a method that behaves as the method *rule*, except that it automatically identify type constraints involving predicate and eliminates them automatically. In practice, this would allow the size of proof scripts to be systematically reduced.

Proof metrics A recent study involving proof metrics in the context of the seL4 project found out that there is a quadratic relationship between the size of the formal statement of a property and the final size of its formal proof in Isabelle [74]. It would be an interesting experiment to verify if the refinement proofs using the algebra mechanised in this thesis also obey this relationship.

Proof of soundness The most relevant theoretical extension to this work is to investigate the existence of a semantic model which validates the assumptions of the algebra. For this, a denotational semantics has to be formalised. Recent investigation combining program algebras and rely guarantee has suggested that modelling the semantics of parallel composition using the notion of *shuffling* [2] may be one alternative. Additionally, the trace semantics provided by Hayes in [27] may be another alternative if the definition of rely command is restructured in terms of the primitives of the language proposed there.

Bibliography

[1] Alglave, J., Kroening, D., Nimal, V., and Poetzl, D. (2014). Don't sit on the fence: A static analysis approach to automatic fence insertion. In *Proceedings of the 16th International Conference on Computer Aided Verification - Volume 8559*, pages 508–524, New York, NY, USA. Springer-Verlag New York, Inc.

[2] Armstrong, A. (2016). *Formal Analysis of Concurrent Programs*. PhD thesis, University of Sheffield.

[3] Armstrong, A., Gomes, V. B. F., and Struth, G. (2014). *Algebraic Principles for Rely-Guarantee Style Concurrency Verification Tools*, pages 78–93. Springer International Publishing, Cham.

[4] Aspinall, D. and Kaliszyk, C. (2016). *Towards Formal Proof Metrics*, pages 325–341. Springer Berlin Heidelberg, Berlin, Heidelberg.

[5] Back, R.-J. J., Akademi, A., and Wright, J. V. (1998). *Refinement Calculus: A Systematic Introduction*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1st edition.

[6] Ballarin, C. (2006). *Interpretation of Locales in Isabelle: Theories and Proof Contexts*, pages 31–43. Springer Berlin Heidelberg, Berlin, Heidelberg.

[7] Barrett, C. and Tinelli, C. (2007). CVC3. In Damm, W. and Hermanns, H., editors, *Proceedings of the* 19^{th} *International Conference on Computer Aided Verification (CAV '07)*, volume 4590 of *Lecture Notes in Computer Science*, pages 298–302. Springer-Verlag. Berlin, Germany.

[8] Berdine, J., Calcagno, C., and O'Hearn, P. (2006). Smallfoot: Modular automatic assertion checking with separation logic. In de Boer, F., Bonsangue, M., Graf, S., and de Roever, W.-P., editors, *Formal Methods for Components and Objects*, volume 4111 of *Lecture Notes in Computer Science*, pages 115–137. Springer Berlin Heidelberg.

[9] Berghofer, S. and Wenzel, M. (1999). *Inductive Datatypes in HOL — Lessons Learned in Formal-Logic Engineering*, pages 19–36. Springer Berlin Heidelberg, Berlin, Heidelberg.

[10] Blanchette, J. and Nipkow, T. (2010). Nitpick: A counterexample generator for higher-order logic based on a relational model finder. In Kaufmann, M. and Paulson, L., editors, *Interactive Theorem Proving*, volume 6172 of *Lecture Notes in Computer Science*, pages 131–146. Springer Berlin Heidelberg.

[11] Blanchette, J. C. (2016a). Hammering Away: A User's Guide to Sledgehammer for Isabelle/HOL. Technical report, Institut für Informatik, Technische Universität München.

[12] Blanchette, J. C. (2016b). Picking Nits: A User's Guide to Nitpick for Isabelle/HOL. Technical report, Institut für Informatik, Technische Universität München.

[13] Blanchette, J. C., Böhme, S., and Paulson, L. C. (2011). Extending sledgehammer with smt solvers. In *Proceedings of the 23rd International Conference on Automated Deduction*, CADE'11, pages 116–130, Berlin, Heidelberg. Springer-Verlag.

[14] Blanchette, J. C., Hölzl, J., Lochbihler, A., Panny, L., Popescu, A., and Traytel, D. (2014). *Truly Modular (Co)datatypes for Isabelle/HOL*, pages 93–110. Springer International Publishing, Cham.

[15] Blass, A. and Gurevich, Y. (2008). Program termination and well partial orderings. *ACM Trans. Comput. Log.*, 9(3).

[16] Bornat, R. and Amjad, H. (2013). Explanation of two non-blocking shared-variable communication algorithms. *Formal Aspects of Computing*, 25(6):893–931.

[17] Bornat, R., Calcagno, C., O'Hearn, P., and Parkinson, M. (2005). Permission accounting in separation logic. *SIGPLAN Not.*, 40(1):259–270.

[18] Bossi, A., Piazza, C., and Rossi, S. (2008). Action refinement in process algebra and security issues. In King, A., editor, *Logic-Based Program Synthesis and Transformation*, volume 4915 of *Lecture Notes in Computer Science*, pages 201–217. Springer Berlin Heidelberg.

[19] Bouajjani, A., Derevenetc, E., and Meyer, R. (2014). Robustness against relaxed memory models. In *Software Engineering 2014, Fachtagung des GI-Fachbereichs Softwaretechnik, 25. Februar 2014, Kiel, Deutschland*, pages 85–86.

[20] Boyland, J. (2003). Checking interference with fractional permissions. In *Proceedings of the 10th International Conference on Static Analysis*, SAS'03, pages 55–72, Berlin, Heidelberg. Springer-Verlag.

[21] Brookes, S. (1996). Full abstraction for a shared-variable parallel language. *Information and Computation*, 127(2):145 – 163.

[22] Brookes, S. (2004). *A Semantics for Concurrent Separation Logic*, pages 16–34. Springer Berlin Heidelberg, Berlin, Heidelberg.

[23] Calcagno, C., Distefano, D., Dubreil, J., Gabi, D., Hooimeijer, P., Luca, M., O'Hearn, P., Papakonstantinou, I., Purbrick, J., and Rodriguez, D. (2015). *Moving Fast with Software Verification*, pages 3–11. Springer International Publishing, Cham.

[24] Cavalcanti, A., Sampaio, A., and Woodcock, J. (2006). Refinement: an overview. In *Refinement Techniques in Software Engineering*, volume 3167 of *Lecture Notes in Computer Science*, pages 1–17. Springer Berlin Heidelberg.

[25] Coleman, J. W. (2008). Expression decomposition in a rely/guarantee context. In *Verified Software: Theories, Tools, Experiments*, volume 5295 of *Lecture Notes in Computer Science*, pages 146–160. Springer Berlin Heidelberg.

[26] Coleman, J. W. and Jones, C. B. (2007). A structural proof of the soundness of rely/guarantee rules. *J. Log. and Comput.*, 17(4):807–841.

[27] Colvin, R. J., Hayes, I. J., and Meinicke, L. A. (2017). Designing a semantic model for a wide-spectrum language with concurrency. *Formal Aspects of Computing*, pages 1–23.

[28] Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. (2009). *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition.

[29] de Moura, L. and Bjørner, N. (2008). *Z3: An Efficient SMT Solver*, pages 337–340. Springer Berlin Heidelberg, Berlin, Heidelberg.

[30] Dias, D. and Freitas, L. (2014). Designing an unbounded buffer in rely-guarantee. Technical Report CS-TR-1431, Newcastle University.

[31] Dijkstra, E. W. (1976). *A discipline of programming*. Prentice-Hall Englewood Cliffs, N.J.

[32] Dingel, J. (2000). *Systematic Parallel Programming*. Research paper. School of Computer Science, Carnegie Mellon University.

[33] Dongol, B., Gomes, V. B. F., and Struth, G. (2015). *A Program Construction and Verification Tool for Separation Logic*, pages 137–158. Springer International Publishing, Cham.

[34] Dongol, B. and Hayes, I. J. (2012). *Deriving Real-Time Action Systems Controllers from Multiscale System Specifications*, pages 102–131. Springer Berlin Heidelberg, Berlin, Heidelberg.

[35] Fell, J., Hayes, I. J., and Velykis, A. (2016). Concurrent refinement algebra and rely quotients. *Archive of Formal Proofs*, 2016.

[36] for Aeronautics, R. T. C. (2011). DO-178C, software considerations in airborne systems and equipment certification.

[37] Foster, S., Zeyda, F., and Woodcock, J. (2015). Isabelle/utp: A mechanised theory engineering framework. In Naumann, D., editor, *Unifying Theories of Programming*, volume 8963 of *Lecture Notes in Computer Science*, pages 21–41. Springer International Publishing.

[38] Freitas, L. and Whiteside, I. (2014). Proof patterns for formal methods. In *FM 2014: Formal Methods - 19th International Symposium, Singapore, May 12-16, 2014. Proceedings*, pages 279–295.

[39] Gomes, V. B. F. (2016). *Algebraic Principles for Program Correctness Tools in Isabelle/HOL.* PhD thesis, University of Sheffield.

[40] Haftmann, F., Klein, G., Nipkow, T., and Schirmer, N. (2016). *LATEX Sugar for Isabelle Documents*, distributed with the isabelle system edition.

[41] Harrison, J. (1998). Formalizing Dijkstra. In *Proceedings of the 11th International Conference on Theorem Proving in Higher Order Logics*, pages 171–188, London, UK, UK. Springer-Verlag.

[42] Havelund, K. and Pressburger, T. (2000). Model checking java programs using java pathfinder. *International Journal on Software Tools for Technology Transfer*, 2(4):366–381.

[43] Hayes, I. J. (2016). Generalised rely-guarantee concurrency: an algebraic foundation. *Formal Aspects of Computing*, 28(6):1057–1078.

[44] Hayes, I. J., Burns, A., Dongol, B., and Jones, C. B. (2013a). Comparing degrees of non-deterministic in expression evaluation. *The Computer Journal*, 56(6):741–755.

[45] Hayes, I. J., Colvin, R. J., Meinicke, L. A., Winter, K., and Velykis, A. (2016). *An Algebra of Synchronous Atomic Steps*, pages 352–369. Springer International Publishing, Cham.

[46] Hayes, I. J., Jones, C. B., and Colvin, R. J. (2012). Refining rely-guarantee thinking. Technical Report CS-TR-1334, Newcastle University.

[47] Hayes, I. J., Jones, C. B., and Colvin, R. J. (2013b). Reasoning about concurrent programs: Refining rely-guarantee thinking. Technical Report CS-TR-1395, Newcastle University.

[48] Hayes, I. J., Jones, C. B., and Colvin, R. J. (2014). Laws and Semantics for Rely-Guarantee Refinement. Technical Report CS-TR-1425, Newcastle University.

[49] Hayes, I. J. and Meinicke, L. (2014). *Invariants, Well-Founded Statements and Real-Time Program Algebra*, pages 318–334. Springer International Publishing, Cham.

[50] Hennessy, M. and Milner, R. (1985). Algebraic laws for nondeterminism and concurrency. *J. ACM*, 32(1):137–161.

[51] Hoare, C. A. R. (1969). An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580.

[52] Hoare, C. A. R. (1972). Towards a Theory of Parallel Programming. In *Operating Systems Techniques*, volume 9 of *A.P.I.C. Studies in Data Processing*, pages 61–71. Academic Press.

[53] Hoare, C. A. R. (1978). Communicating sequential processes. *Commun. ACM*, 21(8):666–677.

[54] Hoare, C. A. R., Hayes, I. J., Jifeng, H., Morgan, C. C., Roscoe, A. W., Sanders, J. W., Sorensen, I. H., Spivey, J. M., and Sufrin, B. A. (1987). Laws of programming. *Commun. ACM*, 30(8):672–686.

[55] Hoare, T., van Staden, S., Möller, B., Struth, G., and Zhu, H. (2016). Developments in concurrent kleene algebra. *Journal of Logical and Algebraic Methods in Programming*, 85(4):617 – 636. Relational and algebraic methods in computer science.

[56] Holzmann, G. J. (1997). The model checker spin. *IEEE Transactions on Software Engineering*, 23(5):279–295.

[57] Huffman, B. and Kunčar, O. (2013). *Lifting and Transfer: A Modular Design for Quotients in Isabelle/HOL*, pages 131–146. Springer International Publishing, Cham.

[58] Huth, M. and Ryan, M. (2004). *Logic in Computer Science: Modelling and Reasoning About Systems*. Cambridge University Press, New York, NY, USA.

[59] Jones, C. (1981). *Development Methods for Computer Programs Including a Notion of Interference*. Technical monograph. Oxford University Computing Laboratory.

[60] Jones, C. B. (1983a). Specification and design of (parallel) programs. In *IFIP Congress*, pages 321–332.

[61] Jones, C. B. (1983b). Tentative steps toward a development method for interfering programs. *ACM Trans. Program. Lang. Syst.*, 5(4):596–619.

[62] Jones, C. B. (1990). *Systematic Software Development using VDM*. Prentice Hall International, second edition.

[63] Jones, C. B. (2010). The role of auxiliary variables in the formal development of concurrent programs. In Roscoe, A., Jones, C. B., and Wood, K. R., editors, *Reflections on the Work of C.A.R. Hoare*, pages 167–187. Springer London.

[64] Jones, C. B. and Hayes, I. J. (2016). Possible values: Exploring a concept for concurrency. *Journal of Logical and Algebraic Methods in Programming*, 85(5, Part 2):972 – 984.

[65] Jones, C. B., Hayes, I. J., and Colvin, R. J. (2015). Balancing expressiveness in formal approaches to concurrency. *Formal Aspects of Computing*, 27(3):475–497.

[66] Jones, C. B. and Pierce, K. G. (2011). Elucidating concurrent algorithms via layers of abstraction and reification. *Formal Asp. Comput.*, 23(3):289–306.

[67] Jones, C. B. and Yatapanage, N. (2015). *Reasoning about Separation Using Abstraction and Reification*, pages 3–19. Springer International Publishing, Cham.

[68] Kernighan, B. W. (1988). *The C Programming Language*. Prentice Hall Professional Technical Reference, 2nd edition.

[69] Klein, G., Elphinstone, K., Heiser, G., Andronick, J., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., Sewell, T., Tuch, H., and Winwood, S. (2009). sel4: Formal verification of an OS kernel. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, pages 207–220, New York, NY, USA. ACM.

[70] Kozen, D. and Silva, A. (2016). Practical coinduction. *Mathematical Structures in Computer Science*, pages 1–21.

[71] Krauss, A. and Schropp, A. (2010). *A Mechanized Translation from Higher-Order Logic to Set Theory*, pages 323–338. Springer Berlin Heidelberg, Berlin, Heidelberg.

[72] Leino, K. (2010). Dafny: An automatic program verifier for functional correctness. In Clarke, E. and Voronkov, A., editors, *Logic for Programming, Artificial Intelligence, and Reasoning*, volume 6355 of *Lecture Notes in Computer Science*, pages 348–370. Springer Berlin Heidelberg.

[73] Lynch, N. and Vaandrager, F. (1995). Forward and backward simulations. *Information and Computation*, 121(2):214 – 233.

[74] Matichuk, D., Murray, T., Andronick, J., Jeffery, R., Klein, G., and Staples, M. (2015). Empirical study towards a leading indicator for cost of formal software verification. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 722–732.

[75] Milner, R. (1989). *Communication and Concurrency*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

[76] Milner, R. (1999). *Communicating and Mobile Systems: The Pi Calculus*. Cambridge University Press.

[77] Moreira, N., Pereira, D., and Melo de Sousa, S. a. (2013). On the Mechanisation of Rely-Guarantee in Coq. Technical Report DCC-2013-01, Faculdade de Ciências da Universidade do Porto.

[78] Morgan, C. (1994). *Programming from specifications (2nd ed.)*. Prentice Hall International (UK) Ltd., Hertfordshire, UK, UK.

[79] Morris, J. M. (1987). A theoretical basis for stepwise refinement and the programming calculus. *Science of Computer Programming*, 9(3):287 – 306.

[80] Mosses, P. D. (2004). Modular structural operational semantics. *The Journal of Logic and Algebraic Programming*, 60:195 – 228.

[81] Nielson, H. R. and Nielson, F. (1992). *Semantics with applications: a formal introduction*. John Wiley & Sons, Inc., New York, NY, USA.

[82] Nieto, L. P. (2002). *Verification of Parallel Programs with the Owicki-Gries and Rely-Guarantee Methods in Isabelle/HOL*. PhD thesis, Technische Universität München.

[83] Nipkow, T. (1998). Winskel is (almost) right: Towards a mechanized semantics textbook. *Formal Aspects of Computing*, 10(2):171–186.

[84] Nipkow, T. (2002). *Hoare Logics for Recursive Procedures and Unbounded Nondeterminism*, pages 103–119. Springer Berlin Heidelberg, Berlin, Heidelberg.

[85] Nipkow, T. and Klein, G. (2015). *Concrete Semantics: With Isabelle/HOL*. Springer International Publishing.

[86] Noschinski, L. (2015). A graph library for Isabelle. *Mathematics in Computer Science*, 9(1):23–39.

[87] Obua, S. (2006). *Partizan Games in Isabelle/HOLZF*, pages 272–286. Springer Berlin Heidelberg, Berlin, Heidelberg.

[88] OHearn, P. W. (2007). Resources, concurrency, and local reasoning. *Theor. Comput. Sci.*, 375(1-3):271–307.

[89] Oheimb, D. v. (2001). *Analyzing Java in Isabelle/HOL: Formalization, Type Safety and Hoare Logic*. PhD thesis, Technische Universität München.

[90] Owens, S., Sarkar, S., and Sewell, P. (2009). A better x86 memory model: x86-tso. In Berghofer, S., Nipkow, T., Urban, C., and Wenzel, M., editors, *Theorem Proving in Higher Order Logics*, volume 5674 of *Lecture Notes in Computer Science*, pages 391–407. Springer Berlin Heidelberg.

[91] Owicki, S. and Gries, D. (1976a). An axiomatic proof technique for parallel programs I. *Acta Informatica*, 6(4):319–340.

[92] Owicki, S. and Gries, D. (1976b). Verifying properties of parallel programs: An axiomatic approach. *Commun. ACM*, 19(5):279–285.

[93] Owre, S., Rushby, J. M., and Shankar, N. (1992). PVS: a prototype verification system. In *Proceedings of the 11th International Conference on Automated Deduction: Automated Deduction*, CADE-11, pages 748–752, London, UK, UK. Springer-Verlag.

[94] Paulson, L. (2012). Three years of experience with sledgehammer, a practical link between automatic and interactive theorem provers. In Schmidt, R. A., Schulz, S., and Konev, B., editors, *PAAR-2010: Proceedings of the 2nd Workshop on Practical Aspects of Automated Reasoning*, volume 9 of *EPiC Series in Computing*, pages 1–10. EasyChair.

[95] Philippaerts, P., Mühlberg, J. T., Penninckx, W., Smans, J., Jacobs, B., and Piessens, F. (2014). Software verification with VeriFast: Industrial case studies. *Sci. Comput. Program.*, 82:77–97.

[96] Pierce, K. G. (2009). *Enhancing the Usability of Rely-guarantee Conditions for Atomicity Refinement*. PhD thesis, University of Newcastle upon Tyne.

[97] Prensa Nieto, L. (2003). The rely-guarantee method in Isabelle/HOL. In *Proceedings of ESOP 2003*, volume 2618 of *LNCS*. Springer-Verlag.

[98] Roscoe, A. W., Hoare, C. A. R., and Bird, R. (1997). *The Theory and Practice of Concurrency*. Prentice Hall PTR, Upper Saddle River, NJ, USA.

[99] Sampaio, A. (1997). *An Algebraic Approach to Compiler Design*, volume 4 of *AMAST Series in Computing*. World Scientific.

[100] Sangiorgi, D. and Rutten, J. (2011). *Advanced Topics in Bisimulation and Coinduction*. Cambridge University Press, New York, NY, USA, 1st edition.

[101] Schirmer, N. and Wenzel, M. (2009). State spaces – The locale way. In *4th International Workshop on Systems Software Verification (SSV 2009)*, volume 254 of *Electronic Notes in Theoretical Computer Science*, pages 161–179. Elsevier Science B.V.

[102] Schlossnagle, G. (2003). *Advanced PHP Programming*. Sams, Indianapolis, IN, USA.

[103] Simpson, H. (1990). Four-slot fully asynchronous communication mechanism. *Computers and Digital Techniques, IEE Proceedings E*, 137(1):17–30.

[104] Stirling, C. (1988). A generalization of Owicki-Gries's Hoare logic for a concurrent while language. *Theor. Comput. Sci.*, 58(1-3):347–359.

[105] Stirling, C. (1998). *The joys of bisimulation*, pages 142–151. Springer Berlin Heidelberg, Berlin, Heidelberg.

[106] Stølen, K. (1991). Development of parallel programs on shared data-structures. Technical Report UMCS-91-1-1, University of Manchester, UK.

[107] Vafeiadis, V. (2008). Modular fine-grained concurrency verification. Technical Report UCAM-CL-TR-726, University of Cambridge, Computer Laboratory.

[108] van Staden, S. (2015). On rely-guarantee reasoning. In *Mathematics of Program Construction - 12th International Conference, MPC 2015, Königswinter, Germany, June 29 - July 1, 2015. Proceedings*, pages 30–49.

[109] Wenzel, M. (2016). The Isabelle/Isar reference manual.

Appendix A

Rely-guarantee algebra in Isabelle/HOL

The Isabelle mechanisation is provided digitally as print out would be about 300 pages. That includes the source theory files (*.thy) and additional material that is relevant to the use of the mechanisation in Isabelle. The .thy files were produced using the latest stable version of Isabelle available at the moment of writing (Isabelle 2016-1). Isabelle can be downloaded free of charge from the website https://isabelle.in.tum.de. The most recent release of Isabelle at the time of this writing can be obtained also from the folder Isabelle in the CD-ROM attached to this thesis.

A.1 Mechanisation

See CD-ROM attached to the thesis. For a general view of the theory files, see discussion in Section 7.1 on page 217.

A.2 Precedence and associativity

Precedence level	Name	Syntax	Associativity
90	Finite iteration	$\_^\star$	-
90	Infinite iteration	$\_^\infty$	-
90	Omega iteration	$\_^\omega$	-
90	Frame	_: _	-
90	Assignment	_:=_	-
80	Sequential composition	_ ; _	Left
75	Rely	**rely** (_, _) · _	-
70	Parallel composition	_ ∥ _	-
70	Strict conjunction	_ ⋒ _	-
70	Binary non-determinism	_ ⊓ _	Right
70	Uses	**uses** _ · _	-
70	State	**state** _ ↦ _ · _	-
70	Eguard	**eguard** _	-
65	Guarantee	**guar** _ · _	-

Table A.1 Precedence and associativity for commands

Precedence level	Name	Syntax	Associativity
1000	Transitive closure reflexive on given subset	$\_^{**}{}_A$	-
999	Reflexive-transitive closure	$\_^{**}$	-
90	Post-state notation	_‘	-
75	Relational conjunction	_ ∧ _	Left
72	Relational disjunction	_ ∨ _	Left
70	Relational implication	_ ⇒ _	Right
75	Relational composition	_ ; _	Right
40	Relational negation	¬ _	

Table A.2 Precedence and associativity for remaining operators

A.3 Revised paper proofs

While mechanising the theory from previous chapters we found three kind of issues: i) misspellings of mathematical formulas or symbols, e.g. missing parenthesis, equalities where there should be an implication, etc; ii) redundant assumptions and missing laws to justify transformations between proof steps; iii) incorrect proofs, which could not be fixed via introduction of new laws. Issues of the first kind (i) are easy to fix, and of second kind (ii) we discovered the redundant assumptions and missing laws by inspecting the proof state of corresponding propositions with issues. In this section we discuss issues of the third kind (iii), which required us to take different proof paths from those discussed in [48].

Law 3.79 (Distribute-Rely) This law is originally presented as an equivalence in [48]. The reason for the difference in our presentation, which only establishes a refinement, is that we discovered a mistake in the original paper proof. To illustrate the problem, we reproduce part of original paper proof. To fit the terms on one line, we omit the *acset* constructor after $\bigsqcap$.

$$
\begin{array}{ll}
 & \mathbf{rely}\ (r_0,\ z \vee r_1) \cdot (\mathbf{rely}\ (r_1,\ z) \cdot c) \\
= & \text{expanding the external rely command by Definition 3.72 (Rely)} \\
 & \bigsqcap \{\ d \mid \{stops(\mathbf{rely}\ (r_1,\ z) \cdot c,\ z \vee r_1)\}\ ;\ (\mathbf{rely}\ (r_1,\ z) \cdot c) \sqsubseteq [z \vee r_1]\ d \parallel \langle r_0 \vee idrel \rangle^\star \\
 & \qquad \wedge \vdash stops(\mathbf{rely}\ (r_1,\ z) \cdot c,\ z \vee r_1) \Rightarrow stops(d,\ z \vee r_1 \vee r_0)\} \\
= & \text{by law 3.77 (Rely-Stops)} \\
 & \bigsqcap \{\ d \mid \{stops(c,\ z)\}\ ;\ (\mathbf{rely}\ (r_1,\ z) \cdot c) \sqsubseteq [z \vee r_1]\ d \parallel \langle r_0 \vee idrel \rangle^\star \wedge \\
 & \qquad \vdash stops(c,\ z) \Rightarrow stops(d,\ z \vee r_1 \vee r_0)\} \\
\sim & \text{by laws 3.82 (Distribute-Rely-Precondition) and 3.76 (Rely-Refinement)} \\
 & \bigsqcap \{d \mid (\{stops(c,\ z)\}\ ;\ c \sqsubseteq [z]\ (d \parallel \langle r_0 \vee idrel \rangle^\star) \parallel \langle r_1 \vee idrel \rangle^\star \wedge \\
 & \qquad \vdash stops(c,\ z) \Rightarrow stops(d \parallel \langle r_0 \vee idrel \rangle^\star,\ z \vee r_1)) \wedge \\
 & \qquad \vdash stops(c,\ z) \Rightarrow stops(d,\ z \vee r_1 \vee r_0)\}
\end{array}
$$

The problem in the proof sketch above is the application of law 3.82 (Distribute-Rely-Precondition). This law cannot be applied, because it requires the command c inside of the rely block to be guarded by the precondition $\{stops(c,\ z)\}$. The way to circumvent the problem is to weaken the precondition $\{stops(c,\ z)\}$ to $\{true\}$, allowing the precondition

command to be eliminated. The fixed proof is shown next. It continues from the second step above, and thus replaces the third step previously presented.

$$\begin{array}{ll}
 & \bigsqcap \{\, d \mid \{stops(c, z)\} \,;\, (\textbf{rely}\ (r_1, z) \cdot c) \sqsubseteq [z \vee r_1]\ d \parallel \langle r_0 \vee idrel\rangle^\star \wedge \\
 & \qquad \vdash stops(c, z) \Rightarrow stops(d, z \vee r_1 \vee r_0)\} \\
\sqsubseteq & \text{by laws 3.19c (Consequence), 3.29a (Zeros-and-units)} \\
 & \bigsqcap \{d \mid \textbf{rely}\ r_1 \cdot c_z \sqsubseteq [z \vee_r r_1]\ d \parallel \langle r_0 \vee_r idrel\rangle^\star \wedge \\
 & \qquad \vdash stops(c, z) \Rightarrow stops(d, z \vee r_1 \vee r_0)\} \\
\sqsubseteq & \text{by laws 3.78 (Rely-Environment) and 3.76 (Rely-Refinement)} \\
 & \bigsqcap \{d \mid (\{stops(c, z)\} \,;\, c \sqsubseteq [z]\ (d \parallel \langle r_0 \vee idrel\rangle^\star) \parallel \langle r_1 \vee idrel\rangle^\star \wedge \\
 & \qquad \vdash stops(c, z) \Rightarrow stops(d \parallel \langle r_0 \vee idrel\rangle^\star, z \vee r_1)) \wedge \\
 & \qquad \vdash stops(c, z) \Rightarrow stops(d, z \vee r_1 \vee r_0)\} \\
\sqsubseteq & \text{by laws 3.74a (Properties-Finite-Interference), 3.50 (Term-In-Context)} \\
 & \text{and 3.39a (Term-Monotonic)} \\
 & \bigsqcap \{d \mid (\{stops\ c\ z\} \,;_c\, c \sqsubseteq [z]\ (d \parallel\ \langle r_0 \vee_r r_1 \vee_r idrel\rangle^\star \wedge \\
 & \qquad \vdash stops\ c\ z \Rightarrow_r stops\ d\ (z \vee_r r_1 \vee_r r_0)\} \\
= & \text{by Definition 3.72 (Rely)} \\
 & \textbf{rely}\ (r_0 \vee r_1, z) \cdot c
\end{array}$$

Proof of law 3.90 (Strengthen-Rely-In-Context).

$$\textbf{rely}\ (r, z) \cdot c \sqsubseteq [rx]\ \textbf{rely}\ (rx \wedge r, z) \cdot c$$

By Definition 3.72 (Rely) and law 3.28a (Monotonic-WSL) one as to show that

$$\exists b.\ \{stops(c, z)\} \,;\, c \sqsubseteq [z]\ b \parallel \langle r \vee idrel\rangle^\star \wedge \vdash stops(c, z) \Rightarrow stops(b, z \vee r) \wedge b \sqsubseteq [rx]\ d$$

from the premises

$$\text{P1: } \{stops(c, z)\} \; ; \; c \sqsubseteq [z] \; d \parallel \langle rx \wedge r \vee idrel \rangle^{\star}$$
$$\text{P2: } \vdash stops(c, z) \Rightarrow stops(d, z \vee rx \wedge r)$$

Choose b to be $(d \Cap (\textbf{eguard } rx \vee idrel))$. This instantiation of b generates three subgoals:

$$\text{G1: } \{stops(c, z)\} \; ; \; c \sqsubseteq [z] \; (d \Cap (\textbf{eguard } rx \vee idrel)) \parallel \langle r \vee idrel \rangle^{\star}$$
$$\text{G2: } \vdash stops(c, z) \Rightarrow stops(d \Cap (\textbf{eguard } rx \vee idrel), z \vee r)$$
$$\text{G3: } d \Cap (\textbf{eguard } rx \vee idrel) \sqsubseteq [rx] \; d$$

For the first sub-goal (G1):

$$\begin{array}{rl} & \{stops(c, z)\} \; ; \; c \\ \sqsubseteq [z] & \text{assumption P1} \\ & d \parallel \langle rx \wedge r \vee idrel \rangle^{\star} \\ \sqsubseteq & \text{by 5.15 (Refinement-Eguard) and on the left branch} \\ & \text{and 4.14 (Omega-Star-Atomic) on the right branch} \\ & (d \Cap (\textbf{eguard } rx \vee idrel)) \parallel (\langle rx \vee idrel \rangle^{\omega} \Cap \langle r \vee idrel \rangle^{\star}) \\ \sim & \text{by law 5.14 (Conjunction-Parallel-Eguard)} \\ & (d \Cap (\textbf{eguard } rx \vee idrel)) \parallel \langle r \vee idrel \rangle^{\star} \end{array}$$

For the second sub-goal (G2), the application of 5.15 (Refinement-Eguard) and 3.38 (Stops) to assumption P2 allow us to derive assumption P2':

$$\text{P2': } \{stops(c, z)\} \; ; \; \langle true \rangle^{\star} \sqsubseteq d \Cap (\textbf{eguard } z \vee rx \wedge r \vee idrel)$$

Similarly, applying 5.15 (Refinement-Eguard) and 3.38 (Stops) to G2, it becomes G2'

$$\text{G2': } \{stops(c, z)\} \; ; \; \langle true \rangle^{\star} \sqsubseteq (d \Cap (\textbf{eguard } rx \vee idrel)) \Cap (\textbf{eguard } z \vee r \vee idrel)$$

This can be proved as follows

$$\begin{array}{rl} & \{stops(c, z)\} \; ; \; \langle true \rangle^{\star} \\ \sqsubseteq & \text{by assumption P2'} \end{array}$$

$$
\begin{array}{ll}
 & d \Cap (\textbf{eguard}\ z \vee rx \wedge r \vee idrel) \\
\sqsubseteq & \text{by 5.12 (Strengthen-Eguard)} \\
 & d \Cap ((\textbf{eguard}\ rx \vee idrel) \Cap (\textbf{eguard}\ z \vee r \vee idrel)) \\
\sim & \text{by law 3.26c (Assoc-Comm-Dist)} \\
 & (d \Cap (\textbf{eguard}\ rx \vee idrel)) \Cap (\textbf{eguard}\ z \vee r \vee idrel)
\end{array}
$$

The proof of G3 follows immediately from law 5.15 (Refinement-Eguard).

Law 3.112 (Rely-Loop) Whilst mechanising the version of this law presented in [48] we noticed that the assumption $\vdash p \wedge r \Rightarrow {w^{**}}_X$ was not strong enough to justify the following proof step, which uses law 3.19a (Consequence) to strengthen a postcondition.

$$
\begin{array}{ll}
 & (\textbf{rely}\ r \cdot [p, {w^{++}} \wedge p`] \sqcap skip)\ ;\ (\textbf{rely}\ r \cdot [p, p` \wedge {b_1}` \wedge {w^{**}}_X]) \\
\sqsubseteq & \text{by lemma 3.19a (Consequence) as} \vdash p \wedge r \Rightarrow {w^{**}}_X \\
 & (\textbf{rely}\ r \cdot [p, {w^{++}} \wedge p`] \sqcap skip)\ ;\ (\textbf{rely}\ r \cdot [p, {r^{**}} \wedge {b_1}`])
\end{array}
$$

To make this refinement possible, we replaced r by its reflexive-transitive closure, leaving the assumption in the shape $\vdash p \wedge {r^{**}} \Rightarrow {w^{**}}_X$.

Law 3.115 (Assignment-Single-Reference) In the purpose of this proof is to show that for a predicate p, relations r and q, variable x, and expression e,

$$\textbf{rely}\ r \cdot [p, q] \sqsubseteq x{:=}e$$

from the premises:

P1: $tol\text{-}interf\,(p, q, r)$

P2: $\vdash p \Rightarrow defined\ e$

P3: $SRF\ (e, r)$

P4: $tol\text{-}interf\,(true, \lambda s\ s'.\ [\![e]\!]_v\ s = [\![e]\!]_v\ s', r)$

P5: $\vdash p \wedge (\lambda s\ s'.\ s'\ x = [\![e]\!]_v\ s) \wedge idset\ \overline{\{x\}} \Rightarrow q$

From law 3.28c (Monotonic-WSL) and Definition 3.12 (Assignment), we must show that

$$\textbf{rely}\ r \cdot [p, q] \sqsubseteq [[N\ v = e]] \ ; \langle (\lambda s\ s'.\ s'\ x = v) \wedge \mathit{idset}\ \overline{\{x\}} \rangle$$

By law 3.76 (Rely-Refinement), this refinement proof is equivalent to the next two sub-goals

G1: $\vdash \mathit{stops}([p, q], \mathit{idrel}) \Rightarrow \mathit{stops}([[N\ v = e]] \ ; \langle (\lambda s\ s'.\ s'\ x = v) \wedge \mathit{idset}\ \overline{\{x\}} \rangle, \mathit{idrel} \vee r)$

G2: $\{\mathit{stops}([p, q], \mathit{idrel})\} \ ; [p, q]$
$\quad \sqsubseteq [\mathit{idrel}]\ [[N\ v = e]] \ ; \langle (\lambda s\ s'.\ s'\ x = v) \wedge \mathit{idset}\ \overline{\{x\}} \rangle \parallel \langle r \vee \mathit{idrel} \rangle^{\star}$

For the first sub-goal (G1):

$\quad \vdash \mathit{stops}([p, q], \mathit{idrel}) \Rightarrow \mathit{stops}([[N\ v = e]] \ ; \langle (\lambda s\ s'.\ s'\ x = v) \wedge \mathit{idset}\ \overline{\{x\}} \rangle, \mathit{idrel} \vee r)$
$=$ by 3.40b (Term-Equivalences) and 3.4 (Derived commands)
$\quad \vdash p \wedge \mathit{stops}([q], \mathit{idrel}) \Rightarrow \mathit{stops}([[N\ v = e]] \ ; \langle (\lambda s\ s'.\ s'\ x = v) \wedge \mathit{idset}\ \overline{\{x\}} \rangle, \mathit{idrel} \vee r)$
$=$ by 3.41 (Term-Postcondition)
$\quad \vdash p \Rightarrow \mathit{stops}([[N\ v = e]] \ ; \langle (\lambda s\ s'.\ s'\ x = v) \wedge \mathit{idset}\ \overline{\{x\}} \rangle, \mathit{idrel} \vee r)$

This implication follows from laws 3.52 (Term-Sequential-Special-Case), 3.44 (Term-Test), 3.43 (Term-Atomic) and assumptions P1 and P2. For the second sub-goal (G2), we first weaken the environment of the refinement relation from *idrel* to *true* using law 3.18c (Refinement-Preorder), and then proceed the refinement proof from the left-hand side.

$\quad \{\mathit{stops}([p, q], \mathit{idrel})\} \ ; [p, q]$
$\sqsubseteq$ by 3.19c (Consequence)
$\quad [p, q]$
$\sqsubseteq$ by 3.25 (Sequential)
$\quad [p, (r^{**} \ ; p \wedge (\lambda s\ s'.\ [\![e]\!]_v\ s = v) \wedge \mathit{idrel}) \ ; r^{**}] \ ;$
$\quad [\mathit{true}, (r^{**} \ ; (\lambda s\ s'.\ s'\ x = v) \wedge \mathit{idset}\ \overline{\{x\}}) \ ; r^{**}]$

At this point, the proof that remains to be done is

$$[p, (r^{**} \ ; p \wedge (\lambda s\ s'.\ [\![e]\!]_v\ s = v) \wedge \mathit{idrel}) \ ; r^{**}] \ ;$$
$$[\mathit{true}, (r^{**} \ ; (\lambda s\ s'.\ s'\ x = v) \wedge \mathit{idset}\ \overline{\{x\}}) \ ; r^{**}]$$

$$\sqsubseteq\ [[N\,v = e]]\ ;\ \langle(\lambda s\ s'.\ s'\,x = v) \wedge \mathit{idset}\ \overline{\{x\}}\rangle\ \|\ \langle r \vee \mathit{idrel}\rangle^\star$$

To prove this refinement we will apply law 3.99 (Test-Single-Reference), which distributes interference over the test command ($[[\_]]$). In order to do that, we need first to distribute interference on the program at the right-hand side of the refinement symbol. For this we use law 3.75 (Distribute-Interference) as follows

$$\begin{array}{ll} & [[N\,v = e]]\ ;\ \langle(\lambda s\ s'.\ s'\,x = v) \wedge \mathit{idset}\ \overline{\{x\}}\rangle\ \|\ \langle r \vee \mathit{idrel}\rangle^\star \\ \sim & \text{by 3.75 (Distribute-Interference)} \\ & ([[N\,v = e]]\ \|\ \langle r \vee \mathit{idrel}\rangle^\star)\ ;\ (\langle(\lambda s\ s'.\ s'\,x = v) \wedge \mathit{idset}\ \overline{\{x\}}\rangle\ \|\ \langle r \vee \mathit{idrel}\rangle^\star) \end{array}$$

After the substitution, the proof that remains to be done is

$$\begin{array}{ll} & [p,\ (r^{**}\ ;\ p \wedge (\lambda s\ s'.\ [\![e]\!]_v\ s = v) \wedge \mathit{idrel})\ ;\ r^{**}]\ ; \\ & [\mathit{true},\ (r^{**}\ ;\ (\lambda s\ s'.\ s'\,x = v) \wedge \mathit{idset}\ \overline{\{x\}})\ ;\ r^{**}] \\ \sqsubseteq & ([[N\,v = e]]\ \|\ \langle r \vee \mathit{idrel}\rangle^\star)\ ;\ (\langle(\lambda s\ s'.\ s'\,x = v) \wedge \mathit{idset}\ \overline{\{x\}}\rangle\ \|\ \langle r \vee \mathit{idrel}\rangle^\star) \end{array}$$

By monotonicity of sequential composition ($;_c$), we split the proof into the next two sub-proofs.

$$\begin{array}{ll} \text{G1.1:} & [p,\ (r^{**}\ ;\ p \wedge (\lambda s\ s'.\ [\![e]\!]_v\ s = v) \wedge \mathit{idrel})\ ;\ r^{**}] \\ & \sqsubseteq [[N\,v = e]]\ \|\ \langle r \vee \mathit{idrel}\rangle^\star \\ \text{G1.2:} & [\mathit{true},\ (r^{**}\ ;\ (\lambda s\ s'.\ s'\,x = v) \wedge \mathit{idset}\ \overline{\{x\}})\ ;\ r^{**}] \\ & \sqsubseteq \langle(\lambda s\ s'.\ s'\,x = v) \wedge \mathit{idset}\ \overline{\{x\}}\rangle\ \|\ \langle r \vee \mathit{idrel}\rangle^\star \end{array}$$

For the first sub-goal (G1.1), we apply law 3.99 (Test-Single-Reference) to trade this sub-goal by an equivalent where interference is distributed over the test commannd.

$$\begin{array}{ll} & [p,\ (r^{**}\ ;\ p \wedge (\lambda s\ s'.\ [\![e]\!]_v\ s = v) \wedge \mathit{idrel})\ ;\ r^{**}] \\ \sqsubseteq & \langle r \vee \mathit{idrel}\rangle^\star\ ;\ [[N\,v = e]]\ ;\ \langle r \vee \mathit{idrel}\rangle^\star \end{array}$$

This refinement is proved by refining the left-hand side until reach the right-hand side, as shown next.

$$[p,\ (r^{**}\ ;\ p \wedge (\lambda s\ s'.\ [\![e]\!]_v\ s = v) \wedge \mathit{idrel})\ ;\ r^{**}]$$

$\sqsubseteq$ by 3.25 (Sequential)

$[p, r^{**} ; p \wedge (\lambda s\ s'.\ [\![e]\!]_v\ s = v) \wedge idrel] ; [true, r^{**}]$

$\sqsubseteq$ by 3.25 (Sequential) and 3.19a (Consequence)

$[true, r^{**}] ; [defined\ (N\ v = e), [\![N\ v = e]\!]_r \wedge idrel] ; [true, r^{**}]$

$\sqsubseteq$ by 3.1 and 3.4 (Derived commands), 3.29a (Zeros-and-units)

4.1c (Relation-Properties), 3.35 (Iteration-Properties), and 3.98 (Introduce-Test)

$\langle r \vee idrel\rangle^\star ; [[N\ v = e]] ; \langle r \vee idrel\rangle^\star$

For the second goal (G1.2), the proof follows similar structure.

$[true, (r^{**} ; (\lambda s\ s'.\ s'\,x = v) \wedge idset\ \overline{\{x\}}) ; r^{**}]$

$\sqsubseteq$ by 3.25 (Sequential)

$[true, r^{**} ; (\lambda s\ s'.\ s'\,x = v) \wedge idset\ \overline{\{x\}}] ; [true, r^{**}]$

$\sqsubseteq$ by 3.25 (Sequential) and 3.19a (Consequence)

$[true, r^{**}] ; [true, (\lambda s\ s'.\ s'\,x = v) \wedge idset\ \overline{\{x\}}] ; [true, r^{**}]$

$\sqsubseteq$ by 3.1 and 3.4 (Derived commands), 3.29a (Zeros-and-units), 3.21 (Make-Atomic)

4.1c (Relation-Properties), and 3.35 (Iteration-Properties)

$\langle r \vee idrel\rangle^\star ; \langle (\lambda s\ s'.\ s'\,x = v) \wedge idset\ \overline{\{x\}}\rangle ; \langle r \vee idrel\rangle^\star$

$\sim$ by 3.74c (Properties-Finite-Interference)

$\langle (\lambda s\ s'.\ s'\,x = v) \wedge idset\ \overline{\{x\}}\rangle \parallel \langle r \vee idrel\rangle^\star$

The problem in the proof of this law in [48] is that the conclusion of law 3.99 (Test-Single-Reference) is misinterpreted as

$$[[e]] \parallel \langle r \vee idrel\rangle^\star \sqsubseteq \langle r \vee idrel\rangle^\star ; [[e]] ; \langle r \vee idrel\rangle^\star.$$

A.4 Additional material

A.4.1 Compiled PDF of the theory

The file document.pdf in /RG-laws/output/ is generated from the *.thy source files of the mechanisation and includes comments along the theory.

A.4.2 Uncountability of RG-WSL

RG-WSL contains program constructors that take arguments that are formalised as relations, e.g. the postcondition and precondition commands. Any attempt to count programs in RG-WSL, therefore, requires one to establish a mechanism to count relations. However, relations are just functions to booleans, and the space of functions is uncountable. Therefore, RG-WSL is also uncountable.

This argument may be easier to understand if we analyse a simple (hypothetical) language (*TimedSquash*) with two constructs: (i) *bounce* and , (ii) **delay** $\mathbb{R}$ *TimedL*. The first construct (*bounce*) is a terminal command, and the second is a composite command that takes a real number representing a delay to be waited for before executing action defined in the second argument. The language *TimedL* is uncountable because any attempt to count programs in this language would demand a strategy to count real numbers, but such strategy does not exist.

A.4.3 Uncountability of the set used to define the rely command

This section proves that the set used to define the rely command is uncountable. For convenience, the definition of the rely command given on page 94 is repeated below.

Definition 3.72 (Rely)**.** *Let r and z be relations and c a command,*

$$\textbf{rely}\ (r,\, z)\cdot c \;\equiv\; \bigsqcap acset \left\{ d \;\middle|\; \begin{array}{l} (\{stops(c,\, z)\}\ ;\ c \sqsubseteq [z]\ d \parallel \langle r \vee idrel\rangle^{\star}) \wedge \\ (\vdash stops(c,\, z) \Rightarrow stops(d,\, z \vee r)) \end{array} \right\}$$

Assume the set used in the definition of the rely command to be countable. For brevity, we shall refer to this set as S. Thus, for any relations r and z and command c, there must be an enumeration of programs d, such that **rely** $(r,\, z)\cdot c$ is defined as the countable choice over programs d, that is, $S = \{d_0, d_1, ...\}$. Let us now consider the concrete instance **rely** $idrel \cdot [true]$. Our purpose is to show that there exists an uncountable amount of programs that can refine this rely command, therefore, S must contain an uncountable amount of programs as well. For that, we will make an arbitrary choice of d using law 3.28b (Monotonic-WSL on page 78).

$$\textbf{rely}\ idrel \cdot [true] \;=\; \bigsqcap acset \left\{ d \;\middle|\; \begin{array}{l} (\{stops([true],\, idrel)\}\ ;\ [true] \sqsubseteq [idrel]\ d \parallel \langle idrel\rangle^{\star}) \\ \wedge\ (\vdash stops([true],\, idrel) \Rightarrow stops(d,\, idrel)) \end{array} \right\}$$

For the application of law 3.28b we will choose the concrete program $[q]$, where q is an arbitrary relation. Thus, the set used to construct the rely command must include at least, all programs $[q]$ for all concrete instances of q. But it is not possible to enumerate relations (q_0, q_1,...). Consequently, there exists no possible enumeration of the programs ($[q_0]$, $[q_1]$, ...). This contradicts the assumption that the set S is countable. Thus, S is not countable in the general case (arbitrary parameters z, r and c in the definition of the rely command).

A.4.4 Font extension

To see the symbol for strict conjunction (⫛) correctly, we extended the Isabelle font with an additional glyph. The extension of fonts is explained in the file *Font-Extension.pdf* available in the directory RG-laws/.

Appendix B

Summary of laws and definitions used in Chapter 6

Lemma 3.19 (Consequence)**.** *For any predicates p_0 and p_1, and relations q_0 and q_1,*

$$\vdash p_0 \Rightarrow p_1 \wedge \vdash p_0 \wedge q_1 \Rightarrow q_0 \implies [p_0, q_0] \sqsubseteq [p_1, q_1] \tag{3.19a}$$

Law 3.25 (Sequential)**.** *For any predicates p and mid, and relations q, q_0 and q_1, such that $\vdash p \wedge (q_0 \wedge mid' \,;\, q_1) \Rightarrow q$,*

$$[p, q] \sqsubseteq [p, q_0 \wedge mid'] \,;\, [mid, q_1]$$

Law 3.59 (Introduce-Guarantee)**.** *For any relation g and command c,*

$$c \sqsubseteq \mathbf{guar}\ g \cdot c$$

Law 3.60 (Guarantee-Monotonic)**.** *For any relations g_0 and g_1, and command c,*

$$\vdash g_0 \Rightarrow g_1 \vee idrel \implies \mathbf{guar}\ g_1 \cdot c \sqsubseteq \mathbf{guar}\ g_0 \cdot c \tag{3.60a}$$

Law 3.62 (Distribute-Guarantee)**.** *For any predicate p, relations g, g_0, g_1 and q, boolean expression b, commands c and d, and variable x, the following hold.*

$$\mathbf{guar}\ g \cdot c \,;\, d \sim (\mathbf{guar}\ g \cdot c) \,;\, (\mathbf{guar}\ g \cdot d) \tag{3.62a}$$

$$\mathbf{guar}\ g \cdot c \parallel d \sim (\mathbf{guar}\ g \cdot c) \parallel (\mathbf{guar}\ g \cdot d) \tag{3.62b}$$

$$\mathbf{guar}\ g_0 \cdot (\mathbf{guar}\ g_1 \cdot c) \sim \mathbf{guar}\ g_0 \wedge g_1 \cdot c \tag{3.62c}$$

$$\textbf{guar}\ \mathit{idset}\ \{x\} \cdot \textbf{var}\ x \cdot c \ \sim\ \textbf{var}\ x \cdot c \tag{3.62d}$$

$$\textbf{guar}\ g \cdot (\textbf{if}\ b\ \ \textbf{then}\ c\ \ \textbf{else}\ d) \ \sim\ \textbf{if}\ b\ \ \textbf{then guar}\ g \cdot c\ \ \textbf{else guar}\ g \cdot d \tag{3.62e}$$

$$\textbf{guar}\ g \cdot (\textbf{while}\ b\ \textbf{do}\ \ c) \ \sim\ \textbf{while}\ b\ \textbf{do}\ \ \textbf{guar}\ g \cdot c \tag{3.62f}$$

Definition 3.64 (Guarantee invariant)**.** *Let p be a predicate and c a command,*

$$\textbf{guar}{-}\textbf{inv}\ p \cdot c \equiv \textbf{guar}\ p \Rightarrow p\text{`} \cdot c$$

Definition 3.65 (Frame)**.** *Let x be a set of variables and c a command,*

$$x{:}\ c \equiv \textbf{guar}\ \mathit{idset}\ \overline{x} \cdot c$$

Law 3.67 (Distribute-Guarantee-Frame)**.** *For any relation g, set of variables X and command c, the following hold.*

$$\textbf{guar}\ g \cdot X{:}\ c \ \sim\ X{:}\ (\textbf{guar}\ g \cdot c)$$

Law 3.68 (Trade-Guarantee-Invariant)**.** *For any predicates p and p_0 and relation q, such that $\vdash p_0 \Rightarrow p$,*

$$[p_0, p\text{`} \wedge q] \sqsubseteq \textbf{guar}{-}\textbf{inv}\ p \cdot [p_0, q]$$

Definition (contrib.) 3.86 (Unrestricted-Rely)**.** *Let z and r be relations, x a variable, and c a command,*

$$\frac{\vdash \mathit{depends\text{-}only}\ (r, \overline{\{x\}}) \vee \vdash r \Rightarrow \mathit{idrel} \qquad \vdash \mathit{depends\text{-}only}\ (z, \overline{\{x\}}) \vee \vdash z \Rightarrow \mathit{idrel} \qquad \mathit{unrest}(x, c)}{\mathit{unrest}(x, \textbf{rely}\ (r, z) \cdot c)}$$

Law 3.87 (Rely-Monotonic)**.** *For any relations r, r_0 and r_1, such that $\vdash r_0 \Rightarrow r_1 \vee \mathit{idrel}$, and command c*

$$\textbf{rely}\ (r_0, z) \cdot c \ \sqsubseteq\ \textbf{rely}\ (r_1, z) \cdot c \tag{3.87a}$$

Law 3.89 (Rely-Sequential)**.** *For any predicates p and mid, and any relations r, q_0 and q_1, such that $\vdash p \wedge (q_0 \wedge mid' \,;\, q_1) \Rightarrow q$,*

$$\textbf{rely}\ r \cdot [p, q] \sqsubseteq (\textbf{rely}\ r \cdot [p, q_0 \wedge mid']) \,;\, (\textbf{rely}\ r \cdot [mid, q_1])$$

Law 3.91 (Introduce-Rely-Guar-Invariant)**.** *For any predicate p and relations r and q, such that $\vdash r \Rightarrow p \Rightarrow p'$ and $\vdash p_0 \Rightarrow p$,*

$$\textbf{rely}\ r \cdot [p_0, p' \wedge q] \sqsubseteq \textbf{guar}{-}\textbf{inv}\ p \cdot \textbf{rely}\ r \cdot [p_0, q]$$

Law 3.93 (Trade-Rely-Guarantee)**.** *For any predicate p, relations g, r and q,*

$$\textbf{guar}\ g \cdot \textbf{rely}\ r \cdot [p, q] \sim \textbf{guar}\ g \cdot \textbf{rely}\ r \cdot [p, q \wedge (g \vee r)^{**}]$$

Law (contrib.) 3.95 (Introduce-Parallel-Spec)**.** *For any predicates p, p_0 and p_1, and relations q, q_0, q_1, g_0 and g_1, such that $\vdash p \Rightarrow p_0 \wedge p_1$ and $\vdash p \wedge (q_0 \wedge q_1) \Rightarrow q$,*

$$[p, q] \sqsubseteq (\textbf{guar}\ g_0 \cdot \textbf{rely}\ g_1 \cdot [p_0, q_0]) \parallel (\textbf{guar}\ g_1 \cdot \textbf{rely}\ g_0 \cdot [p_1, q_1])$$

Law 3.97 (Introduce-Parallel-Spec-Nested)**.** *For any predicates p, p_0 and p_1, and relations q, q_0, q_1, g_0, g_1 and r, such that $\vdash p \Rightarrow p_0 \wedge p_1$ and $\vdash p \wedge (q_0 \wedge q_1) \Rightarrow q$,*

$$\textbf{rely}\ r \cdot [p, q] \sqsubseteq (\textbf{guar}\ g_0 \cdot \textbf{rely}\ g_1 \vee r \cdot [p_0, q_0]) \parallel (\textbf{guar}\ g_1 \cdot \textbf{rely}\ g_0 \vee r \cdot [p_1, q_1])$$

Lemma 3.101 (Introduce-Variable-Frame)**.** *For any variable x, set of variables Y, and command c, assuming x is not in Y and is unrestricted in c,*

$$Y\colon c \sqsubseteq \textbf{var}\ x \cdot (\{x\} \cup Y)\colon c$$

Law 3.102 (Introduce-Variable-Rely)**.** *For any relations z and r, variable x and set of variables Y, such that x is not in Y and is unrestricted in* **rely** $(r, z) \cdot c$*, then*

$$Y\colon (\textbf{rely}\ (r, z) \cdot c) \sqsubseteq \textbf{var}\ x \cdot (\{x\} \cup Y)\colon (\textbf{rely}\ (idset\ \{x\} \wedge r, z) \cdot c)$$

Law 3.105 (Rely-Uses)**.** *For any predicate p, relation q, and set of commands X, such that $[p, q]$ tolerates interference idset X,*

$$\textbf{rely}\ idset\ X \cdot [p, q] \sqsubseteq \textbf{uses}\ X \cdot [p, q]$$

Lemma (contrib.) 3.108 (Elimination-Uses)**.** *For any command c, which consists only of code, and set of variables X, such that all free variables of c are in X,*

$$\mathbf{uses}\ X \cdot c \sqsubseteq c$$

Law (contrib.) 3.109 (Distribute-Uses)**.** *For any sets of variables X and Y, relation g, predicate p and command c,*

$$\mathbf{guar}\ g \cdot \mathbf{uses}\ X \cdot c \ \sqsubseteq\ \mathbf{uses}\ X \cdot (\mathbf{guar}\ g \cdot c) \tag{3.109a}$$

$$\mathbf{guar{-}inv}\ p \cdot (\mathbf{uses}\ X \cdot c) \ \sqsubseteq\ \mathbf{uses}\ X \cdot \mathbf{guar{-}inv}\ p \cdot c \tag{3.109b}$$

$$Y{:}\ (\mathbf{uses}\ X \cdot c) \ \sqsubseteq\ \mathbf{uses}\ X \cdot Y{:}\ c \tag{3.109c}$$

Law 3.111 (Rely-Conditional)**.** *For any predicates p, b_0 and b_1, relations r and q, such that $[p, q]$ tolerates interference r, and boolean expression b, such that b satisfies the single reference property with respect to r and $\vdash p \wedge [\![b]\!]_r \Rightarrow b_0$ and $\vdash p \wedge r \Rightarrow (b_0 \Rightarrow b_0\text{`})$, and $\vdash p \wedge [\![\neg b]\!]_r \Rightarrow b_1$ and $\vdash p \wedge r \Rightarrow (b_1 \Rightarrow b_1\text{`})$, and $\vdash p \Rightarrow$ defined b,*

$$\mathbf{rely}\ r \cdot [p, q] \sqsubseteq (\mathbf{if}\ b\ \ \mathbf{then\ rely}\ r \cdot [p \wedge b_0, q]\ \ \mathbf{else\ rely}\ r \cdot [p \wedge b_1, q])$$

Law 3.112 (Rely-Loop)**.** *For predicates p, b_0 and b_1, relations r, w and q, and set of variables X, such that p is preserved by r and w is well-founded on p and $\vdash$ depends-only (w, X) and $\vdash p \wedge r^{**} \Rightarrow {w^{**}}_X$ and boolean expression b, such that b satisfies the single reference property with respect to r and $\vdash p \wedge [\![b]\!]_r \Rightarrow b_0$ and $\vdash p \wedge r \Rightarrow (b_0 \Rightarrow b_0\text{`})$, and $\vdash p \wedge [\![\neg b]\!]_r \Rightarrow b_1$, and $\vdash p \wedge r \Rightarrow (b_1 \Rightarrow b_1\text{`})$ and $\vdash p \Rightarrow$ defined b,*

$$\mathbf{rely}\ r \cdot [p, p\text{`} \wedge b_1\text{`} \wedge {w^{**}}_X] \sqsubseteq \mathbf{while}\ b\ \mathbf{do}\ \ \mathbf{rely}\ r \cdot [p \wedge b_0, p\text{`} \wedge w]$$

Law 3.113 (Assignment-Guarantee)**.** *For any predicate p, relation g, variable x and expression e such that $\vdash ((p \wedge (\lambda s\ s'.\ s'\ x = [\![e]\!]_v\ s) \wedge idset\ \overline{\{x\}}) \Rightarrow (q \wedge (g \vee idrel)))$ and also $\vdash p \Rightarrow$ defined e,*

$$\mathbf{guar}\ g \cdot [p, q] \sqsubseteq x{:=}e$$

Law 3.114 (Assignment-Rely-Guarantee)**.** *For any variable x, expression e, set of variables X, predicate p and relations g and q, such that $[p, q]$ tolerates interference idset X, and*

$\vdash p \Rightarrow$ *defined e, and* $\vdash ((p \wedge (\lambda s\ s'.\ s'\ x = [\![e]\!]_v\ s) \wedge$ *idset* $\overline{\{x\}}) \Rightarrow (q \wedge (g \vee$ *idrel*$})))$*, and free-exp* $e \cup \{x\} \subseteq X$*,*

$$\{x\}\colon (\textbf{guar}\ g \cdot \textbf{rely}\ \mathit{idset}\ X \cdot [p, q]) \sqsubseteq x{:=}e$$

Law (contrib.) 3.81 (Rely-Idrel-Specification)**.** *For any predicate p and relation q,*

$$(\textbf{rely}\ \mathit{idrel} \cdot [p, q]) \sim [p, q]$$

Law 4.9 (Trade-Spec-Frame)**.** *For any predicate p, relation r and set of variables X and Y, such that* $Y \subseteq \overline{X}$*,*

$$X\colon [p, q]\ \sim\ X\colon [p, \mathit{idset}\ Y \wedge q]$$

Law 4.11 (Dist-Guarantee-Var)**.** *For any variable x, program c and relation g, such that* $\vdash$ *depends-only* $(g, \overline{\{x\}})$*,*

$$\textbf{guar}\ g \cdot \textbf{var}\ x \cdot c\ \sim\ \textbf{var}\ x \cdot (\textbf{guar}\ g \cdot c)$$

Law 4.12 (Trade-Spec-Guarantee)**.** *For any predicate p and relations g and q,*

$$\textbf{guar}\ g \cdot [p, g^{**} \wedge q]\ \sim\ \textbf{guar}\ g \cdot [p, q]$$

Law 4.13 (Distribute-Frame-Sequential)**.** *For any set of variables Y,* Y_1 *and* Y_2*, such that* $Y_1 \subseteq Y$ *and* $Y_2 \subseteq Y$*, and commands c,* c'*, d and* d'*, and relation r such that* $c \sqsubseteq[r] c'$ *and* $d \sqsubseteq[r] d'$*,*

$$Y\colon (c\ ;\ d) \sqsubseteq[r]\ Y_1\colon c'\ ;\ Y_2\colon d'$$

Law 5.6 (Distribute-g-Parallel)**.** *For any context function F and relation g,*

$$(\textbf{guar}\ g \cdot \|_{Idx} \cdot F) \sim (\|_{Idx} \cdot (\lambda c.\ \textbf{guar}\ g \cdot F\ c))$$

Law 5.9 (Introduce-Multi-Parallel-Parameterised)**.** *For any predicate p, injective list of indices Idx*[1]*, non-parameterised relations Q and R, and parameterised relations g, r and q*

[1] Injective lists do not contain repeated elements.

such that $\vdash g\ i \Rightarrow r\ j$ *for all* $i, j \in$ *set Idx satisfying* $i \neq j$, *and* $\vdash R \Rightarrow r\ i \vee$ *idrel for all* $i \in Idx$, *and* $\vdash (\lambda s\ s'.\ \forall i{\in}\textit{set Idx}.\ q\ i\ s\ s') \Rightarrow Q$,

$$\textbf{rely}\ R \cdot [p, Q] \sqsubseteq \|_{Idx} \cdot (\lambda i.\ \textbf{guar}\ g\ i \cdot \textbf{rely}\ r\ i \cdot [p, q\ i])$$

Law 5.10 (Introduce-Multi-Parallel)**.** *For any predicate p, list of indices Idx, non parameterised relations g, r, R and Q, and parameterised relation q such that* $\vdash g \Rightarrow r$ *and* $\vdash R \Rightarrow r \vee$ *idrel, and* $\vdash (\lambda s\ s'.\ \forall i{\in}\textit{set Idx}.\ q\ i\ s\ s') \Rightarrow Q$,

$$\textbf{rely}\ R \cdot [p, Q] \sqsubseteq \|_{Idx} \cdot (\lambda i.\ \textbf{guar}\ g \cdot \textbf{rely}\ r \cdot [p, q\ i])$$

Lemma 5.26 (Assignment-Array)**.** *For any array x, list of natural numbers l, expression e, predicate p and relations g, q and r, such that*

1. $[p, q]$ *tolerates interference r; and*
2. $\vdash p \Rightarrow$ *defined e; and*
3. $\vdash p \wedge (\lambda s\ s'.\ s'\ x = (s\ x)[l{\leftarrow}[\![e]\!]_v\ s]) \wedge$ *idset* $\overline{\{x\}} \Rightarrow q \wedge (g \vee \textit{idrel})$; *and*
4. *e is preserved by r, that is,* $\vdash p \wedge r \Rightarrow (\lambda s\ s'.\ [\![e]\!]_v\ s = [\![e]\!]_v\ s')$,

the following holds,

$$\{x\}\colon (\textbf{guar}\ g \cdot \textbf{rely}\ r \cdot [p, q]) \sqsubseteq xl{:=}e$$

Lemma 5.30 (RE-Intended-Assignment)**.** *For any predicate p, relations g, r and q, variable x and expression e such that*

1. $\vdash p \Rightarrow$ *defined e; and*
2. *p is stable under r; and*
3. $\vdash r \Rightarrow$ *idset* $\{x\}$; *and*
4. $\vdash p \wedge x \doteq [r]\ e \wedge$ *idset* $\overline{\{x\}} \Rightarrow q \wedge (g \vee \textit{idrel})$,

the following holds,

$$\{x\}\colon (\textbf{guar}\ g \cdot \textbf{rely}\ r \cdot [p, q]) \sqsubseteq x{:=}e$$

Law 5.31 (Sequential-Conditional)**.** *For any predicate p, relation q and boolean expression b, such that* $\vdash p \Rightarrow$ *defined b,*

$$[p, q] \sqsubseteq (\textbf{if } b \textbf{ then } [p \wedge [\![b]\!]_r, q] \textbf{ else } [p \wedge [\![\neg b]\!]_r, q])$$

Law 5.32 (Sequential-Loop)**.** *For any predicate p, boolean expression b and relation w that is well-founded on p, such that* $\vdash p \Rightarrow$ *defined b,*

$$[p, p\text{`} \wedge [\![\neg b]\!]_r\text{`} \wedge w^{**}] \sqsubseteq \textbf{while } b \textbf{ do } [p \wedge [\![b]\!]_r, p\text{`} \wedge w]$$

Appendix C

Applying the refinement algebra (sources)

C.1 Findp Sequential

C.1.1 Derivation

The table of abbreviations is reproduced at the last page of the section.

Algorithm 1 Findp S0

{t}: [true, $post_0$]

Algorithm 2 Findp S1

t:=len(v);
{t}: **guar**−**inv** gi-satp · [init, notp(v, domain(v), t′)]

Algorithm 3 Findp S2

t:=len(v);
var k
    k:=0;
    guar−**inv** gi-satp ·
    guar−**inv** gi-notp · {k, t}: [init, post-dec]

Algorithm 4 Findp S3

```
t:=len(v) ;
var k
    k:=0 ;
    while c-while do
        guar−inv gi-satp ·
        guar−inv gi-notp · {k, t}: [init ∧ c-while, init′ ∧ w]
```

Algorithm 5 Findp S4

```
t:=len(v);
var k
    k:=0 ;
    while c-while do
        if c-if then
            guar−inv gi-satp ·
            guar−inv gi-notp · {t}: [init ∧ c-while ∧ c-if, init′ ∧ (t′ = k)]
        else
            guar−inv gi-satp ·
            guar−inv gi-notp · {k}: [init ∧ c-while ∧ ¬c-if, init′ ∧ (k′ = k + 1)]
```

Algorithm 6 Findp S5

```
t:=len(v);
var k
    k:=0 ;
    while c-while do
        if c-if then
            t:=k
        else
            k:=k + 1
```

C.2 Proof obligations

Proposition C.1 (Proof obligation for $\mathscr{R}_7$)**.** *The next statement holds.*

$$\vdash \begin{pmatrix} \mathit{init} \wedge ((k' = 0) \wedge \mathit{idset}\ \overline{\{k,t\}} \wedge (\mathit{gi\text{-}notp} \wedge \mathit{init})' ; \mathit{gi\text{-}notp}' \wedge \mathit{post\text{-}dec} \wedge \mathit{idset}\ \overline{\{k,t\}}) \\ \Rightarrow_r \\ \mathit{notp}(v, \mathit{domain}(v), t') \end{pmatrix}$$

Proof.

$\mathit{init} \wedge ((k' = 0) \wedge \mathit{idset}\ \overline{\{k,t\}} \wedge (\mathit{gi\text{-}notp} \wedge \mathit{init})' ; \mathit{gi\text{-}notp}' \wedge \mathit{post\text{-}dec} \wedge \mathit{idset}\ \overline{\{k,t\}})$

$\Rightarrow$ by law 4.1m (Relation-Properties)

$(k' = 0) \wedge \mathit{idset}\ \overline{\{k,t\}} \wedge (\mathit{gi\text{-}notp} \wedge \mathit{init})' ; \mathit{gi\text{-}notp}' \wedge \mathit{post\text{-}dec} \wedge \mathit{idset}\ \overline{\{k,t\}}$

$\Rightarrow$ by laws 4.2c (Log-Interp-Imp-Monotonic), 4.1l and 4.1m (Relation-Properties)

$\mathit{idset}\ \overline{\{k,t\}}\ ;\ \mathit{gi\text{-}notp}' \wedge \mathit{post\text{-}dec} \wedge \mathit{idset}\ \overline{\{k,t\}}$

$\Rightarrow$ by laws 4.2b, 4.2c and 4.2d (Log-Interp-Imp-Monotonic)

$\mathit{idset}\ \{v\}\ ;\ \mathit{gi\text{-}notp}' \wedge \mathit{post\text{-}dec} \wedge \mathit{idset}\ \{v\}$

$=$ expanding ($\mathit{idset}\ \{v\}$), *gi-notp* and *post-dec*

$(v' = v)\ ;\ (\mathit{notp}(v, \mathit{domain}(v), k) \wedge \mathit{bnd}(k, v))' \wedge (t' \leq k') \wedge (v' = v)$

$=$ expanding *notp* and the post-state notation (Def. 2.19)

$(v' = v)\ ;\ (\forall i{\in}\mathit{domain}(v').\ i < k' \longrightarrow \neg\ p\ (v'\ !\ i)) \wedge \mathit{bnd}(k', v') \wedge (t' \leq k') \wedge (v' = v)$

$=$ expanding the relational composition and eliminating its existential quantifier

$(\forall i{\in}\mathit{domain}(v').\ i < k' \longrightarrow \neg\ p\ (v'\ !\ i)) \wedge \mathit{bnd}(k', v') \wedge (t' \leq k') \wedge (v' = v)$

$\Rightarrow$ monotonocity of $\leq$

$\forall i{\in}\mathit{domain}(v).\ i < t' \longrightarrow \neg\ p\ (v\ !\ i)$

$=$ contracting *notp*

$\mathit{notp}(v, \mathit{domain}(v), t')$ □

C.3 Findp Concurrent

C.3.1 Derivation

Algorithm 7 Findp C0: Initial specification

{t}: [true, post_0(t, t, T_P)]

Algorithm 8 Findp C1: Preventing data races

var ot ·
var et
ot:=len(v) ; et:=len(v);
guar−**inv** gi-satp ·
{ot, et}: **rely** idset {v, ot, et} · [init(ot) ∧ init(et), notp(v, domain(v,T_P), min(ot′, et′))];
t:=min(ot, et)

Algorithm 9 Findp C2: Introducing parallelism

var ot ·
var et
ot:=len(v) ; et:=len(v);
parallel
guar-inv gi-satp ·
{ot, et}: **guar** g(ot, et) ·
rely g(et, ot) ∧ idset {v} · [init(ot), notp(v, domain(v,odd), min(ot′, et′))]
and
guar-inv gi-satp ·
{ot, et}: **guar** g(et, ot) ·
rely g(ot, et) ∧ idset {v} · [init(et), notp(v, domain(v,even), min(ot′, et′))]
end parallel;
t:=min(ot, et)

Algorithm 10 Findp C3: Introducing loop counters

var ot ·
var et
ot:=len(v) ; et:=len(v);
parallel
var ok
ok:=1;
guar−**inv** gi-satp ·
guar $\mathrm{ot}' \leq \mathrm{ot}$ ·
guar-inv gi-notp(ok, odd) ·
{ok, ot}: **rely** g(et, ot) ∧ idset {ok, v} · [init(ot), post-dec(ok)]
and
var ek
ek:=0;
guar−**inv** gi-satp ·
guar $\mathrm{et}' \leq \mathrm{et}$ ·
guar-inv gi-notp(ek, even) ·
{ek, et}: **rely** g(ot, et) ∧ idset {ek, v} · [init(et), post-dec(ek)]
end parallel;
t:=min(ot, et)

Algorithm 11 Findp C4: Introducing a while loop

```
var ot ·
var et
    ot:=len(v) ; et:=len(v) ;
    parallel
        var ok
            ok:=1 ;
            while c-while(ok) do
                guar−inv gi-satp ·
                guar ot′ ≤ ot ·
                guar-inv gi-notp(ok, odd) ·
                    {ok, ot}: rely idset {ok, ot, v} · [prew(ok, ot), w(ot, ok)]
    and
        var ek
            ek:=0 ;
            while c-while(ek) do
                guar−inv gi-satp ·
                guar et′ ≤ et ·
                guar-inv gi-notp(ek, even) ·
                    {ek, et}: rely idset {ek, et, v} · [prew(ek, et), w(et, ek)]
    end parallel;
    t:=min(ot, et)
```

Algorithm 12 Findp C5: Introducing a conditional

```
var ot ·
var et
    ot:=len(v) ; et:=len(v) ;
    parallel
        var ok
            ok:=1 ;
            while c-while(ok) do
                if c-if(ok) then
                    guar−inv gi-satp ·
                    guar ot′ ≤ ot ·
                    guar-inv gi-notp(ok, odd) ·
                        {ok, ot}: rely idset {ok, ot, v} · [prew(ok, ot) ∧ c-if(ok), w(ot, ok)]
                else
                    guar−inv gi-satp ·
                    guar ot′ ≤ ot ·
                    guar-inv gi-notp(ok, odd) ·
                        {ok, ot}: rely idset {ok, ot, v} · [prew(ok, ot) ∧ ¬c-if(ok), w(ot, ok)]
    and
        var ek
            ek:=0 ;
            while c-while(ek) do
                if c-if(ek) then
                    guar−inv gi-satp ·
                    guar et′ ≤ et ·
                    guar-inv gi-notp(ek, even) ·
                        {ek, et}: rely idset {ek, et, v} · [prew(ek, et) ∧ c-if(ek), w(et, ek)]
                else
                    guar−inv gi-satp · guar (et′ ≤ et) ·
                    guar-inv gi-notp(ek, even) ·
                        {ek, et}: rely idset {ek, et, v} · [prew(ek, et) ∧ ¬c-if(ek), w(et, ek)]
    end parallel;
    t:=min(ot, et)
```

Algorithm 13 Findp C6: Implementing assignments

```
var ot ·
var et
    ot:=len(v) ; et:=len(v);
    parallel
        var ok
            ok:=1 ;
            while c-while(ok) do
                if c-if(ok) then
                    ot:=ok
                else
                    ok:=ok + 2
    and
        var ek
            ek:=0 ;
            while c-while(ek) do
                if c-if(ek) then
                    et:=ek
                else
                    ek:=ek + 2
    end parallel;
    t:=min(ot, et)
```

C.3.2 Abbreviations for Findp

$T_P(x)$	$\equiv$	$True$
$P(e)$	$\equiv$	$UOp\ (p, T_P)\ e$
$notp(v, s, t)$	$\equiv$	$\forall i{\in}s.\ i < t \longrightarrow \neg\ p\ (v\ !\ i)$
$domain(v)$	$\equiv$	$\{x \mid x < length\ v\}$
$satp(v, t)$	$\equiv$	$t \in domain(v) \wedge p\ (v\ !\ t)$
$post_0$	$\equiv$	$(t' = length\ v \vee satp(v, t')) \wedge$
		$notp(v, domain(v), t')$
$gi\text{-}satp$	$\equiv$	$(t = length\ v) \vee satp(v, t)$
$gi\text{-}notp$	$\equiv$	$notp(v, domain(v), k) \wedge bnd(k, v)$
$bnd(k, v)$	$\equiv$	$0 \leq k \wedge k \leq length\ v$
$init$	$\equiv$	$t \leq length\ v$
$post\text{-}dec$	$\equiv$	$t' \leq k'$
w	$\equiv$	$(0 \leq t' - k') \wedge (t' - k' < t - k)$
$c\text{-}while$	$\equiv$	$k < t$
$c\text{-}if$	$\equiv$	$P(v[k])$

$T_P(x)$	$\equiv$	$True$
$P(e)$	$\equiv$	$UOp\ (p, T_P)\ e$
$notp(v, s, t)$	$\equiv$	$\forall i{\in}s.\ i < t \longrightarrow \neg\ p\ (v\ !\ i)$
$domain(v,\Phi)$	$\equiv$	$\{x \mid x < length\ v \wedge \Phi(x)\}$
$satp(v, t, \Phi)$	$\equiv$	$t \in domain(v,\Phi) \wedge p\ (v\ !\ t)$
$post_0(x, y, \Phi)$	$\equiv$	$((min(x', y') = length\ v) \vee satp(v, min(x', y'), T_P)) \wedge$
		$notp(v, domain(v,\Phi), min(x', y'))$
$gi\text{-}satp$	$\equiv$	$(min(ot, et) = length\ v) \vee satp(v, min(ot, et), T_P)$
$gi\text{-}notp(x, \Phi)$	$\equiv$	$notp(v, domain(v,\Phi), x) \wedge bnd(x, v) \wedge \Phi(x)$
$bnd(k, v)$	$\equiv$	$0 \leq k \wedge k \leq length\ v + 1$
$init(t)$	$\equiv$	$t \leq length\ v$
$post\text{-}dec(x)$	$\equiv$	$min(ot', et') \leq x'$
$w(t, k)$	$\equiv$	$(0 \leq t' - k' + 1) \wedge (t' - k' < t - k)$
$c\text{-}while(k)$	$\equiv$	$k < ot \wedge k < et$
$c\text{-}if(k)$	$\equiv$	$P(v[k])$
$b_0(k, t)$	$\equiv$	$k < t$
$b_1(k, ot, et)$	$\equiv$	$(ot \leq k) \vee (et \leq k)$
$prew(k, t)$	$\equiv$	$b_0(k, t) \wedge init(t)$
$g(x, y)$	$\equiv$	$(x' \leq x) \wedge idset\ \{y\}$

Table C.1 Abbreviations for sequential and concurrent Findp (side by side). Left: sequential, right: concurrent