University of Sheffield

Department of Computer Science

Formal Analysis of Concurrent Programs

Thesis submitted for degree of PhD

Author:
Alasdair Armstrong

Supervisor:
Georg Struth

May, 2016

Abstract

In this thesis, extensions of Kleene algebras are used to develop algebras for rely-guarantee style reasoning about concurrent programs. In addition to these algebras, detailed denotational models are implemented in the interactive theorem prover Isabelle/HOL. Formal soundness proofs link the algebras to their models. This follows a general algebraic approach for developing correct by construction verification tools within Isabelle. In this approach, algebras provide inference rules and abstract principles for reasoning about the control flow of programs, while the concrete models provide laws for reasoning about data flow. This yields a rapid, lightweight approach for the construction of verification and refinement tools. These tools are used to construct a popular example from the literature, via refinement, within the context of a general-purpose interactive theorem proving environment.

Declaration

This thesis contains material from the following papers:

- A. Armstrong, V. B. F. Gomes and G. Struth. *Lightweight Program Construction in Isabelle/HOL.* In D. Giannakopoulou and G. Salaün (eds.), SEFM 2014, LNCS 8702. [AGS14c]
- A. Armstrong, V. B. F. Gomes and G. Struth. *Algebras for program correctness in Isabelle/HOL.* In P. Höfner, P. Jipsen, W. Kahl and M. E. Müller (eds.), RAMiCs 2014, LNCS 8428. [ABFGS14]
- A. Armstrong, V. B. F. Gomes and G. Struth. *Algebraic Principles for Rely-Guarantee Style Concurrency Verification Tools.* In C. Jones, P. Pihlajasaari and J. Sun (eds.), FM 2014, LNCS 8442. [AGS14a]
- A. Armstrong, G. Struth and T. Weber. *Programming and automating mathematics in the Tarski-Kleene hierarchy.* In R. Berghammer, B. Möller and M. Winter (eds.), Journal of Logical and Algebraic Methods in Programming, 83(2):87-102 (March 2014). [ASW14]
- A. Armstrong, G. Struth and T. Weber. *Program Analysis and Verification Based on Kleene Algebra in Isabelle/HOL.* In S. Blazy, C. Paulin-Mohring and D. Pichardie (eds.), ITP 2013, LNCS 7998. [ASW13b]
- A. Armstrong and G. Struth. *Automated Reasoning in Higher-Order Regular Algebra.* In T. Griffin and W. Kahl (eds.), RAMiCS 2012, LNCS 7560. [AS12]

Isabelle theory files supporting this thesis can be found online at:

`http://thesis.alasdair.eu`

Acknowledgements

First and foremost, I would like to thank my supervisor Prof. Georg Struth, for being a wonderful mentor during my PhD experience. I greatly appreciate his assistance and considerable expertise in guiding my research efforts, not to mention his assistance in writing research papers and this very thesis itself. In addition, I would like to thank the other members of my Supervisory team, Prof. Marian Gheorge and Dr. Joab Winkler for their valuable contributions during the course of my PhD studies.

I would like to extend special thanks to Dr. Tjark Weber for helping to provide me with the fantastic opportunity to spend time studying in Uppsala during my PhD, as well as lending his expertise to my research efforts. I would also like to thank everyone in the UPMARC research group in Uppsala for being extremely welcoming and friendly during my time there.

I wish to thank my fellow PhD student, Victor Gomes. His talent for Isabelle and his assistance in writing papers has been extremely helpful over the course of my PhD. Furthermore, I'd like to thank the other regular attendees of our V.T. reading group, Dr. Brijesh Dongol, Dr. James Cranch, Dr. Kirill Bogdanov, and Prof. Lindsay Groves while he visited Sheffield, for valuable discussions on many varied subjects and papers in the field of concurrency and beyond. I would also like to thank Dr. Simon Foster for his insights into interactive and automated theorem proving techniques.

I would like to thank Prof. Ian Hayes and Prof. Cliff Jones for inviting me to their rely-guarantee meeting after FM2014, as well as all the other attendees including Dr. Robert Colvin. Their insights into the rely-guarantee method were extremely enlightening and helped me greatly.

Thanks to Prof. John Derrick and Dr. Anthony Simons for providing me with excellent opportunities and assistance in furthering my research horizons during the course of my studies. I am very grateful for such opportunities.

I also wish to thank the excellent flatmates I had the good fortune to spend my time living with during my PhD: Sarah, Dave, Mike, Dom and Belinda, thank you all for putting up with me! I'd also like to thank all my many other friends in Sheffield and elsewhere for making the time outside my studies here so enjoyable.

Last but not least, I would like to thank my family for all their support during my PhD.

Contents

Chapter 1

Introduction

1.1 Introduction

Over the past decade multi-core processors have become ubiquitous, with multi-threaded, concurrent programming becoming the norm for many applications. The importance and timeliness of concurrency verification is therefore of key importance. In this setting, extensions of Hoare logic supporting concurrency are becoming increasingly important for both the development and verification of such concurrent programs. One such extension of Hoare logic for concurrent programs is Jones' rely-guarantee method [Jon81]. This method provides a compositional approach to verifying concurrent programs—the verification or development of large concurrent programs can be reduced to the independent verification or development of individual subprograms.

For simple imperative programs, extensions of Kleene algebra give a succinct and lightweight setting for reasoning about control flow. Algebras such as Kleene algebra with tests (KAT) [Koz00] and Kleene algebra with domain (KAD) [DMS06] provide a formalisation of propositional Hoare logic, that is, Hoare logic without the assignment rule. These algebras give a stratified approach to program verification and refinement: At the most abstract, algebras are used to derive inference rules and refinement laws, as well as reason about control flow. At the most concrete, detailed models support fine-grained reasoning about data flow and, in the case of concurrency, interference. This approach therefore imposes a clean and desirable separation of concerns between the abstract control flow and concrete data flow of a program.

The applicability of Kleene algebras and variants has been demonstrated via numerous applications, for instance, compiler optimisation [KP00], program construction [BS10], transformation and termination [DMS11], static analysis [FD07] and concurrency control [Coh00].

To make this algebraic method applicable for concrete program development and verification tasks, its integration into tools is essential. To capture the flexibility of the method, a number of features are desirable. First, solid mathematical models for fine-grained program behaviour must be implemented. Second, one would like an abstract layer at which inference rules and refinement laws can be easily derived. Third, a high degree of proof automation is mandatory for the analysis of large, real-world programs. For all these features, interactive theorem proving tools, such as *Isabelle* [NPW02], appear to provide both the requisite flexibility and power. The overall approach taken in this thesis for developing such program verification and construction tools in Isabelle is described in Section 1.2.

As mentioned, for sequential programs, the applicability of algebra, and variants of Kleene algebra in particular, is well known. For concurrent programs, the situation is much less clear. In this thesis, new algebras are considered which can provide for concurrent programs the same kind of lightweight and succinct setting that variants of Kleene algebra provide for simple imper-

ative programs. It is perhaps fair to state that *interference* is the essence of concurrency. As such, algebras that allow reasoning about interference must be developed to bring the benefits of the algebraic approach into this concurrent world. Jones' rely guarantee method provides much of the inspiration for the development of these algebras. These algebras are implemented in Isabelle and are linked with detailed infinite language-based semantics, such that the verification, and construction of concurrent programs via refinement can be done within an interactive theorem proving environment.

1.2 Principles of Algebraic Tool Design

It has previously not been entirely clear how the world of abstract algebras can be utilised in the construction of concrete tools for the development and verification of real-world programs. Isabelle offers features suitable for engineering algebraic hierarchies, which can then be linked to concrete models via soundness proofs. This allows for the stratified approach described in Section 1.1. Such hierarchies and the methods used to implement them in Isabelle are described in Chapter 3 and Chapter 4. In this section, it is shown how program verification and construction tools can be developed along two axes. For the first axis consider how one can go from the abstract algebraic layer to a concrete tool for program verification or refinement. This is shown in Figure 1.1.

Algebra	Intermediate Semantics	Concrete Semantics
Control flow	Abstract data flow	Concrete data flow
Verification conditions	—	Verification tools
Refinement laws	—	Construction tools

Figure 1.1: Algebraic tool design principles

As an example, consider Kozen's *Kleene Algebra with tests* [Koz00] (KAT). In KAT one can derive all the rules of propositional Hoare logic for simple while programs, that is, Hoare logic without the assignment rule. By extending KAT with residuals, as in Pratt's *action algebra* one can even obtain refinement rules for simple while programs. This algebraic level, where verification and refinement rules can be derived is depicted as the leftmost column of Figure 1.1.

The intermediate semantics—the middle column of Figure 1.1—would in this example be binary relations. Binary relations form a model of KAT, and are a widely used standard semantics for simple while programs.

By instantiating the intermediate semantics, in this case binary relations,

with a concrete notion of program stores one obtains the rightmost column, the concrete semantics. For simple while programs, this might be represented in Isabelle as a record of program variables. For developing and verifying programs involving pointers and dynamic memory, it also could be instantiated with a formalisation of the heap. At this level, it is possible to link into already well-developed and extensive Isabelle libraries for dealing with such data types.

As mentioned, this process, whereby the type of KAT is instantiated to the type of binary relations over that signature, and finally to the type of binary relations over some specific notion of program store, gives the first axis upon which program verification and construction tools are based.

For the second axis, domain specific algebras are considered for different aspects of program analysis. For example:

- KAT embeds a Boolean algebra of *tests* or *assertions* into a Kleene algebra to enable reasoning about while loops and if statements.
- SKAT, schematic Kleene algebra with tests, extends KAT with axioms for data flow, specifically assignment. It allows for the equivalence of Manna's flowchart schemes to be proven algebraically.
- ω-algebras extend Kleene algebras with an infinite iteration operator, and therefore are an important step towards supporting notions of total correctness, where programs are not guaranteed to terminate.
- DRA, demonic refinement algebra, is a variant of KAT for total correctness. DRA introduces a *strong iteration operator* which stands for finite or infinite operation in order to handle potentially non-terminating programs.
- Action algebras have residuals for the sequential composition operator. In this setting refinement rules can be derived in addition to verification rules.
- bi-Kleene algebra extends Kleene algebras with a notion of parallel composition in addition to the usual sequential composition. Previously, *concurrent Kleene algebra* sought to capture the interaction between these two composition operators as an exchange law. In this thesis, extensions of bi-Kleene algebra called *rely-guarantee* algebras are used to capture the notion of interference between concurrent programs.
- In a quantale, every isotone function is guaranteed to have a least and greatest fixpoint. In this setting one can derive rules for recursion via standard fixpoint theory.

The complete relationship between the above algebraic structures (and more) is given in Figure 3.1. This overall approach to developing program verification and refinement tools is described in full detail for sequential programs using SKAT in Chapter 5, before being expanded to cover concurrent programs with the rely-guarantee method in Chapters 6 through 8. Essentially the algebra is used to provide a clean separation of concerns between the abstract level of refinement laws, and the concrete level where data flow can be considered. The use of algebra in this way allows verification and construction tools to be developed in a rapid fashion—given one tool, often a new one can be developed for a slightly different application domain with relatively little effort, simply by changing several axioms, or by using another model.

1.3 Research Contributions and Thesis Overview

The primary aim of this thesis is to use the approach in Section 1.2 to develop tools for rely-guarantee reasoning within the Isabelle/HOL theorem prover. To achieve this algebras and models for rely-guarantee reasoning are conceived and developed within Isabelle, and used to construct prototype verification tools. These tools have been used to verify canonical examples from the literature.

In this section, the content and research contributions contained within this thesis are summarised on a chapter by chapter basis.

- Chapter 2 presents an overview of existing work on the rely-guarantee method. Furthermore, the rely guarantee method is briefly compared and contrasted to other methods for reasoning about concurrent programs, such as the Owicki-Gries method and concurrent separation logic.

- Chapter 3 presents in full detail the hierarchy of algebras briefly described in Section 1.2. Furthermore, the introduction of these algebras is used to introduce Isabelle/HOL, including showing basic proofs, as well as the techniques and methods for formalising algebraic hierarchies within Isabelle.

  As part of the work involved in this thesis, the algebraic hierarchies used have been formalised in Isabelle, and some have been released as libraries for the Isabelle *Archive of Formal Proofs*. Specifically files for variants of idempotent semirings (dioids) and Kleene algebra were released [ASW13a], as well as files for Kozen's *Kleene algebra with tests* and von Wright's *demonic Refinement algebra* [AGS14b].

- Chapter 5 illustrates an alternative to the approach given in Section 1.2 by mechanising Angus and Kozen's *schematic Kleene algebra with tests*

(SKAT) [AK01] in Isabelle. This approach differs from that in Section 1.2 by using a more syntactic/grammar based on term algebras. Despite these differences, this chapter serves as an illustration of how Isabelle can be used in the construction of program verification tools without the additional complexity of concurrency.

Schematic Kleene algebra is intended to algebraically model Manna's flowchart schemes [Man74]. The chapter presents the techniques used to mechanise SKAT in Isabelle before formalising a complex flowchart equivalence proof due to Manna. While Manna's proof is based on diagrammatic visual reasoning, the proof here is a mechanisation of Angus and Kozen's algebraic proof.

The chapter then presents a novel approach to implementing a program verification tool in Isabelle for simple while programs using SKAT. This approach uses *Kleene Modules* to separate the type of assertions from those of tests. While this method is used out of necessity due to how SKAT is defined, it provides several conceptual benefits. This chapter represents the first concrete verification tool based on such algebras.

- Chapter 6 presents a basic algebra and model for verifying concurrent programs with the rely-guarantee method. Algebraic axioms from which the rely-guarantee rules can be derived are investigated. It is noted that there exists a key mismatch between algebraic methods and traditional methods for verifying concurrent programs. Algebraic methods are inherently compositional by nature, which seems at odds with the pervasive interference in shared memory concurrency, that makes compositionality a challenge. To address this, operations are considered that break the algebraic compositionality in such a way to express interesting properties of concurrent programs. These ideas are elaborated on in further chapters.

- Chapter 7 presents an enhanced variant of the rely-guarantee algebra presented in Chapter 6. This algebra includes operators for more fine-grained reasoning about interference. Furthermore it supports deriving correct programs by refinement, unlike the algebra in Chapter 6. From this algebra both Jones-style inference rules for program verification, and equivalent rules for refinement can be derived.

  In addition to this, an infinite language/trace based model for this Algebra is described in detail. This model effectively provides a denotational semantics for program verification and construction using the rely-guarantee method. Linking this model with the algebra constitutes a soundness proof for this model. Both the algebra and the model are implemented in Isabelle. This constitutes the first mechanisation (that the author is aware of) of a refinement calculus for

rely-guarantee with a denotational semantics in an interactive theorem prover. Previous work on the mechanisation of rely-guarantee (in both Coq and Isabelle) mostly do not appear to support constructing correct programs via refinement, and are based on operational semantics [Nie03, MPdS13].

The infinite trace model in this chapter is influenced by prior work by Brookes [Bro93] and Dingel [Din02]. The algebra has some similarities with prior work on refinement calculi for rely-guarantee such as in [HJC13]. In contrast to prior work, it can be shown that certain operations that make their semantics more complex; such as pervasive stuttering and mumbling are actually unnecessary for constructing a working tool for refinement.

- Chapter 8 details how more concrete parts of the model from Chapter 7 operate. Specifically a description of how tests (in the sense of Kozen's KAT) work within the model. Furthermore, it contains details of how the assignment statement is implemented in the model.

  In addition, this chapter contains an example of the algebra from Chapter 7 being used for a refinement proof within Isabelle. A derivation of the common FINDP example is given, used before in [Nie01, HJC13] for example. This example shows that refining concurrent programs within Isabelle can be done relatively straightforwardly via the use of custom Isabelle tactics.

Chapter 2

The Rely-Guarantee Method

2.1 Introduction

As mentioned in Chapter 1, extensions of Hoare logics are becoming increasingly important for the verification and development of many kinds of programs, including concurrent and multiprocessor programs. Additionally, as previously claimed in Chapter 1, is perhaps fair to state that *interference* is the essence of concurrency. In the context of shared-variable concurrency, such interference manifests as the state of one process being changed by another. This is pictorially demonstrated in Figure 2.1. It is certainly the case that no non-trivial program can sensibly operate under arbitrary interference, so in order to verify and develop concurrent programs, methods must be developed and used that take specific interferences from another process or environment into account.

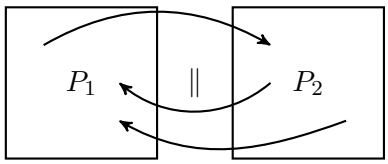


Figure 2.1: Two concurrent processes interfering

In this chapter, prior work on the Owicki-Gries method, the rely-guarantee method, and on concurrent separation logic are all described and contrasted as possible solutions to the problem of verifying and developing programs under interference. Most attention is given to rely-guarantee, but all of these methods are in some sense extensions of Hoare logic aiming to provide additional Hoare-style inference rules to handle concurrent composition. Of course, given the well known complexity of implementing concurrent programs, such rules are often rather non-trivial, and can even offer substantial insight into the nature of concurrency itself.

2.2 The Owicki-Gries Method

The Owicki-Gries method [Owi75, OG76] represents a seminal approach to extending Hoare logic to capture concurrency. The Parallel rule given by Owicki and Gries states that

$$\frac{\{P_1\}X\{Q_1\} \qquad \{P_2\}Y\{Q_2\}}{\{P_1 \wedge P_1\}X\|Y\{Q_1 \wedge Q_2\}}$$

provided the proofs of $\{P_1\}X\{Q_1\}$ and $\{P_2\}Y\{Q_2\}$ are interference free. This interference freedom property is also referred to as the *Einmischungsfrei* property. While the Owicki-Gries method could perhaps be seen as an attempt at a compositional proof method, it is not compositional, as

if the final *Einmischungsfrei* property fails, then the entire program development up till that point may well have to be totally re-done. Nevertheless, the Owicki-Gries method has proven quite successful in practice, even when compared with compositional approaches [dRdBH$^+$01b]. Furthermore it is known that the Owicki-Gries proof rule is unsound for programs which manipulate pointers—as there is a possibility of race conditions involving attempts to update or deallocate shared pointers [Bro04].

2.3 The Rely-Guarantee Method

One of the most popular extensions of Hoare logic for concurrent programs is Jones' rely-guarantee method [Jon81]. A main benefit of this method, especially over Owicki-Gries, is compositionality: the verification of large concurrent programs can be reduced to the independent verification of individual subprograms. The effect of interactions or interference between subprograms is captured by *rely* and *guarantee* conditions. Rely conditions describe the effect of the environment on an individual subprogram. Guarantee conditions, in turn, describe the effect of an individual subprogram on the environment. By constraining a subprogram by a rely condition, the global effect of interactions is captured locally. Compare Figure 2.2 with Figure 2.1. In Figure 2.2 the interference between the programs P_1 and P_2 has been abstracted away into guarantee conditions, which are then used as the rely condition for the opposing process. Note that a program is only required to satisfy its guarantee as long as its rely condition holds. In this thesis, rely and guarantee conditions are collectively referred to as *interference constraints*. In this way, the rely-guarantee method provides compositionality—rather than reasoning about the complicated interference between many concurrent components, we can reason about the components and their interference constraints separately.

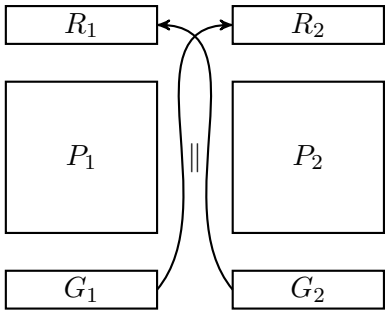


Figure 2.2: Interaction captured by relies and guarantees

The rely-guarantee rules as they appear in this thesis (Figure 6.1 in Chapter 6 and Figure 7.4 in Chapter 7) were first given by Jones in [Jon81,

Jon83]. The remainder of this section attempts to give a (in no way complete) overview of related work done on the rely-guarantee method since then.

Stirling [Sti88] presents rely-guarantee (although not named as such) as extension of the Owicki-Gries method. He extends the Hoare triples with sets of *environment invariants*, giving quintuples which are very much in the style of Jones. These environment invariants are used to encode the properties that are guaranteed to be preserved by both the environment a program runs in and by the program itself. By showing that the results of the interference freedom test in the Owicki-Gries method can be encoded in these environment invariants, Stirling is able to prove completeness of his system relative to Owicki-Gries.

Xu et al. [XdRH97] present a systematic approach to the rely-guarantee method. They provide a proof system for partial correctness and demonstrate that this system can be extended to verify properties such as deadlock freedom and convergence. The language used in their paper is essentially the same as in [OG76], being an extension of Dijkstra's guarded command language with the addition of parallel composition and a synchronisation statement. For this language, they present an operational semantics in the style of [Sti88]. Finally, they prove soundness and relative completeness of the proof system w.r.t. their operational semantics. For relative completeness their method is to demonstrate that rely-guarantee is essentially a compositional variant of the Owicki-Gries method, which has been proven complete in e.g. [Apt81]. In this context completeness implies that all valid programs can be proven correct using the proof method.

Nieto [Nie03, Nie01] formalises both the Owicki-Gries method and the Rely-Guarantee method in Isabelle/HOL. Her mechanisation follows the work of Xu et al. by giving an operational semantics for concurrent programs, and then proving both soundness and compositionality (for rely-guarantee). It is worth noting that Nieto's method does not allow nesting of parallel statements. A program consists of multiple sequential component programs which are all placed in parallel with each other at the top level. Her proofs of soundness follow this structure, first proving soundness for atomic programs, then sequential component programs and finally parallel programs. The soundness proof and program semantics given in this thesis do not have this limitation.

Nieto's mechanisation is relatively succinct and elegant, as the operational approach of Xu et al. proves ideal for this purpose. The approach in this thesis differs by giving a denotational semantics for concurrent programs and proving soundness by linking this semantics with an abstract algebra in which all the rules of the rely-guarantee calculus can be derived. Nieto's formalisation consists of around 1000 lines of specification and 2660 lines of proof steps for 184 lemmas including Owicki-Gries, Rely-Guarantee, and verification condition generators for each. By contrast, the formalisation in this thesis encompasses over 500 lemmas and over 5000 lines of proof steps

for just Rely-guarantee alone.

The denotational approach used in this thesis owes a great deal to the work of Brookes [Bro93]. In his paper 'Full abstraction for a shared variable parallel language', Brookes presents *transition traces*—sequences of state pairs of the form $(\sigma, \sigma') \in \Sigma^2$. For example

$$(\sigma_0, \sigma_0)(\sigma_1, \sigma'_1) \ldots (\sigma_{n-1}, \sigma'_{n-1})(\sigma_n, \sigma'_n) \tag{2.1}$$

is a transition trace. The transition (σ_n, σ'_n) would be an example of a *program transition*, while the transitions between them $\sigma'_n)(\sigma_{n+1}$ are *environment transitions*. In Brookes' paper, program transitions (σ, σ') are only allowed to occur within a transition trace provided it is possible for a command C to perform a computation from σ to σ' according to some operational semantics. These sets of transition traces are closed under the operations of stuttering and mumbling. The *mumble language* $w^\dagger$ for a transition trace w is generated inductively: Assume $\sigma_1, \sigma_2, \sigma_3 \in \Sigma$ and u, v, w are transition traces. First, $w \in w^\dagger$. Secondly, if $u(\sigma_1, \sigma_2)(\sigma_2, \sigma_3)v \in w^\dagger$ then $u(\sigma_1, \sigma_3)v \in w^\dagger$.

The *stutter language* $w^\natural$ for a transition trace w is also defined inductively in much the same way: Assume $\sigma_1, \sigma_2 \in \Sigma$ and u, v, w are transition traces. First $w \in w^\natural$. Secondly, if $u(\sigma_1, \sigma_2)v \in w^\natural$ then $u(\sigma_1, \sigma_2)(\sigma_2, \sigma_2)v \in w^\natural$ and $u(\sigma_1, \sigma_1)(\sigma_2, \sigma_2)v \in w^\natural$.

Using these transition traces Brookes is able to prove *full abstraction*. Very roughly speaking, this entails that two sets of transition traces are equivalent iff both sets of traces have the same behaviour w.r.t. the operational semantics. In a trace semantics that is not fully abstract, one might be able to distinguish between programs such as $\mathsf{skip}; \mathsf{skip}$ and skip.

Dingel [Din02] builds upon Brookes' transition traces by giving a refinement calculus that allows for the formal, step-wise development of both shared variable and message passing concurrent programs from abstract specifications. Dingel uses exactly Stirling's [Sti88] notion of environment invariants to provide a notion of context-sensitive refinement, wherein programs are refined within a context representing the actions of the environment.

At this point it is perhaps helpful to show how relies and guarantees occur within transition traces. This is depicted in Figure 2.3. The rely condition imposes constraints on the environment transitions, while the guarantee speaks about the transitions made by the program.

Concurrency for transition traces is achieved via shuffling. This is already a well studied operation in formal language theory and in the context of process algebra [BW90] where concurrency is implemented as shuffle. Properties of this operator have been extensively studied. For example, finding the proper varieties of language closed under shuffle has been studied by Bloom and Ésik [BE96, BE95]. Shuffling over infinite languages has been studied by Mateescu et al. [MMRS97]. Definitions of this operator for both finite and infinite traces are given in Chapter 6 and Chapter 7 respectively.

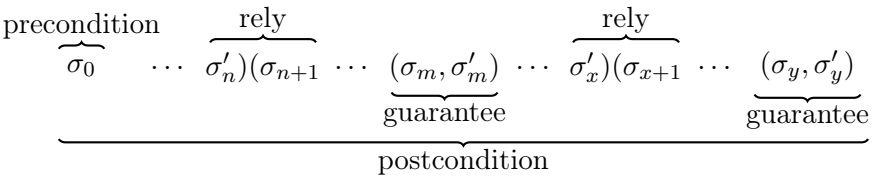


Figure 2.3: Relies and guarantees for a transition trace

In contrast to Brookes' transition traces, one can also consider a denotational approach to rely-guarantee based on so called *Aczel traces*, which originate from an unpublished note by Aczel. Aczel traces are process indexed state pairs. Unlike transition traces, Aczel traces are required to be *connected*, such that in an Aczel trace

$$(\sigma_0, p_1, \sigma'_0)(\sigma_1, p_2, \sigma'_1) \ldots (\sigma_n, p_3, \sigma'_n),$$

it is required that $\sigma'_0 = \sigma_1$. The middle elements of the transitions, p_1,p_2 and p_3 are process labels. These traces are used for example in [dBHdR99]. In contrast to shuffling transition traces, concurrency in this setting has a more intersection-like flavour. To explain; in Aczel traces concurrent composition $X \| Y$ will typically match program transitions from X with environment transitions in Y, and vice versa. Elements of X and Y where this matching cannot be achieved are simply not included in the composition. In this thesis, transition trace style semantics are used exclusively, however it seems reasonable to suggest that the laws presented in succeeding Chapters should hold for either style of trace.

Hayes et al. [HJC13] give a refinement calculus for rely-guarantee which shares many features with the rely-guarantee algebra described in Chapters 6 and 7. These refinement rules for rely-guarantee enable a more general kind of '*rely guarantee thinking*' as opposed to simply verifying programs using the standard rely-guarantee rules. Hayes et al. introduce a command $(\mathbf{guar}\ g \bullet c)$ which behaves as c, but only allows for atomic steps that either stutter. or satisfy g between their pre- and post-state. A step in c which violates this is not a valid step in $(\mathbf{guar}\ g \bullet c)$. Using a Morgan style specification statement $[p, q]$ [Mor88], the statement

$$(\mathbf{guar}\ g \bullet [p, q])$$

implements a program from p to q where every atomic step satisfies g (or stutters).

The second novel command Hayes et al. describe is a rely command $(\mathbf{rely}\ r \bullet c)$ which is intended to reflect the idea of c being implemented in an environment that provides interference of at most r. The stronger the rely condition (the more it states about the environment), the easier this command is to implement.

Using these commands Hayes et al. consider general specifications of the form

$$(\textbf{guar}\ g \bullet (\textbf{rely}\ r \bullet [p, q])) \sqsubseteq c,$$

where c is a command that refines the specification. They note that this is equivalent to the standard Jones quintuple

$$r, g \vdash \{p\}c\{q\}.$$

In [HJC13] their refinement calculus is based of an operational rather than denotational semantics, but it would in theory be very well suited to the denotational approach. The similarities between their work and the algebra and model in Chapter 7 is discussed further in that chapter. Hoare et al. [HMSW11] give a definition of rely-guarantee in concurrent Kleene algebra, defining the Jones quintuple as

$$r, g \vdash \{p\}c\{q\} \iff p(r\|c) \leq q \wedge c \leq g.$$

where r and g are *power invariants*, that is, elements that are invariant under arbitrary powers such that

$$r^\star = r \qquad \text{and} \qquad r^{(\star)} = r.$$

A clear deficiency in this encoding is that c is required to implement g unconditionally, even if its rely condition fails.

This thesis builds upon this concurrent Kleene algebra approach in Chapter 6, and further by incorporating ideas from, and shared with, Hayes et al. [HJC13] in Chapter 7.

2.4 Concurrent Separation Logic

While this thesis is primarily concerned with the rely-guarantee method, concurrent separation logic [O'H07] represents another approach to handling the interference between programs. In concurrent separation logic the parallel rule becomes

$$\frac{\{P_1\}X\{Q_1\} \qquad \{P_2\}Y\{Q_2\}}{\{P_1 * P_1\}X\|Y\{Q_1 * Q_2\}}$$

where $*$ is *separating conjunction*. Separation logic [ORY01, Rey02] in general emphasises the idea of resource separation as a means to controlling the complexity of interaction between processes. Separating conjunction allows reasoning about modifications to one part of some state, while guaranteeing that other parts remain unaffected. This seems like a natural fit for reasoning with concurrent programs, where, as emphasised, one is primarily concerned

with guaranteeing the lack of interference between processes. If concurrent processes are operating on separate parts of some resource, in the sense of separation logic, then they should be able to execute in parallel, which is exactly what the above rule states. This gives a notion of disjoint concurrency, as opposed to the interference based concurrency of rely-guarantee.

Given the algebraic bent of this thesis, it would be remiss not to mention the connection between concurrent separation logic and concurrent Kleene algebra [OPVH15, HMSW11]. Indeed, using similar Isabelle techniques to those presented in this thesis for rely-guarantee (and SKAT) can give a Isabelle based construction and verification tool for concurrent separation logic [DGS15].

There are also attempts to unify the separation logic and rely-guarantee approaches, giving a rely-guarantee calculus equipped with a notion of separating conjunction. Vafeiadis [Vaf08] has performed some exciting research in this direction.

Chapter 3

Algebraic Preliminaries

3.1 Introduction

In this chapter the algebraic foundations for this thesis are described. A hierarchy of algebras is presented starting with idempotent semirings, or dioids, which describe the interaction between sequential composition and choice in programs. These are expanded to Kleene algebras and omega algebras which add notions of finite and strictly infinite iteration respectively. Action algebras add residuals useful for program construction by refinement, and Quantales enhance Kleene algebras with notions from complete lattices, within which concepts from fixpoint theory can be utilised.

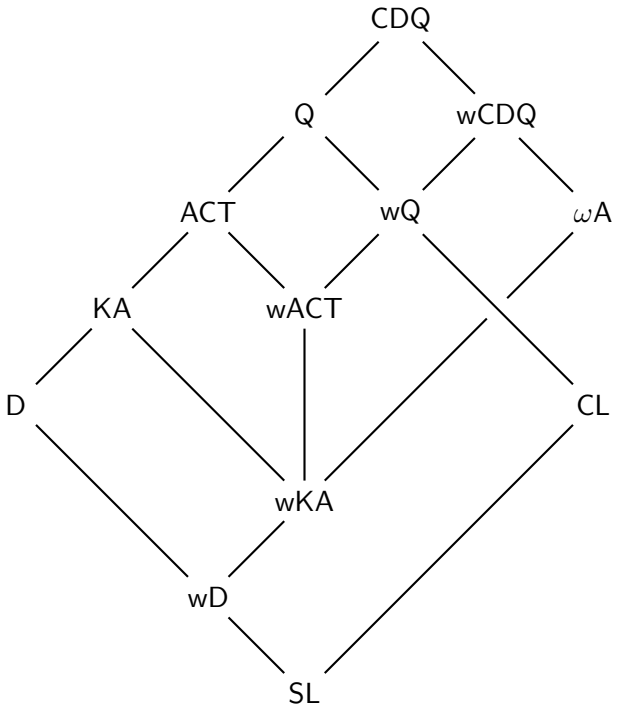


Figure 3.1: Class inclusions for join semilattices (SL), complete lattices (CL), weak dioids (wD), dioids (D), weak Kleene algebras (wKA), Kleene algebras (KA), weak action algebras (wACT), action algebras (ACT), weak quantales (wQ), quantales (Q), ω-algebras (ωA), weak completely distributive quantales (wCDQ) and completely distributive quantales (CDQ).

Figure 3.1 gives the class inclusions for the algebras described in this section. For each algebra with a multiplication operator, there is also a variant in which multiplication is commutative. There are also variations with more than one multiplication operator (with one being commutative), resulting in bi-algebras, for example a bi-Kleene algebra, or bi-dioid (a trioid).

3.2 Dioids

A *semigroup* is an algebraic structure $(S, \cdot)$ where $\cdot$ is an associative binary operation. A *monoid* is an algebraic structure $(M, \cdot, 1)$ where:

- $(M, \cdot)$ is a semigroup,
- 1 is an identity element such that $1 \cdot x = x = x \cdot 1$ holds for all x.

A monoid or semigroup is *commutative* if its binary operator commutes, i.e. $x \cdot y = y \cdot x$. A *semiring* is an algebraic structure $(S, +, \cdot, 0, 1)$ where:

- $(S, +, 0)$ is a commutative monoid,
- $(S, \cdot, 1)$ is a monoid,
- the distributive laws $x \cdot (y + z) = x \cdot y + x \cdot z$ and $(x + y) \cdot z = x \cdot z + y \cdot z$ hold for all x, y, and z,
- the annihilation laws $0 \cdot x = 0$ and $x \cdot 0 = 0$ hold for all x.

A semiring is *weak* if the right annihilation law $x \cdot 0$ does not hold. An idempotent semiring, or *dioid* is a semiring where the addition operator is idempotent, that is $x + x = x$. Typically the multiplication operator in statements such as $x \cdot y$ is omitted and written via juxtaposition as xy.

For the purposes of reasoning about programs, the intuition of the operators in a dioid is usually as follows:

- $x + y$ models non-deterministic choice between two programs x and y,
- $x \cdot y$ models sequential composition between x and y,
- 0 models abort,
- 1 models skip.

In this context it is essential that multiplication is not commutative. Oftentimes standard programming notation $x; y$ will be used for sequential composition, especially for longer examples found in later chapters. However, there is no reason why one should restrict oneself to only considering sequential composition. It is equally possible for multiplication to represent concurrent composition of programs. In this case, we obtain a commutative dioid which models the interaction between concurrent composition and non-deterministic choice. To model situations involving the interaction between both sequential and concurrent composition, we introduce a *trioid* as a structure $(S, +, \cdot, \|, 0, 1)$ such that:

- $(S, +, \cdot, 0, 1)$ is a dioid.
- $(S, +, \|, 0, 1)$ is a commutative dioid.

This definition of a trioid is a special case of a more general definition where the units of sequential and concurrent composition are not shared. For certain models (e.g. multirelations) sequential composition and concurrent composition do not share units in this way. Note that in a trioid there are no axioms governing the interaction between the sequential composition $\cdot$ and the parallel composition $\|$.

3.3 Kleene Algebra

While dioids can capture the interaction between sequential/parallel composition and non-deterministic choice, for programming, notions of iteration are essential. A *Kleene algebra* is a dioid expanded with a star operation which satisfies both the *left unfold axiom*

$$1 + x \cdot x^\star \leq x^\star$$

and *left* and *right induction axioms*

$$z + x \cdot y \leq y \Rightarrow x^\star \cdot z \leq y \qquad \text{and} \qquad z + y \cdot x \leq y \Rightarrow z \cdot x^\star \leq y.$$

In a Kleene algebra the equivalence $1 + x \cdot x^\star = x^\star$ and the right unfold axiom $1 + x^\star \cdot x \leq x^\star$ are derivable as theorems. Thus iteration x^* is modelled as the least fixpoint of the function $\lambda y.1 + x \cdot y$, which is the same as the least fixpoint of $\lambda y.1 + y \cdot x$. A *commutative Kleene algebra* is a Kleene algebra in which multiplication is commutative.

A *bi-Kleene algebra* is a structure $(K, +, \cdot, \|, 0, 1, ^\star, ^{(\star)})$ where

- $(K, +, \cdot, 0, 1, ^\star)$ is a Kleene algebra,
- $(K, +, \|, 0, 1, ^{(\star)})$ is a commutative Kleene algebra.

A *concurrent Kleene algebra* [HMSW11] is a bi-Kleene algebra which satisfies the interchange law

$$(x \| y)(w \| z) \leq (xz) \| (yw).$$

In some contexts, it is also useful to add a meet operation $\sqcap$ to a bi-Kleene algebra, such that $(K, +, \sqcap)$ is a distributive lattice. Such an algebra is called a *bi-Kleene algebra with meet*. This is particularly needed in the context of refinement, where we want to represent specifications as well as programs—while intersecting programs rarely makes sense, intersection of specifications is very useful.

3.4 Models of Kleene Algebra and Dioids

3.4.1 Languages

A *word* is a possibly empty sequence of letters from an alphabet Σ. The set of all (finite) words over Σ is denoted by Σ^*, and the empty word is denoted by ϵ. A *language* L is a set of such words over an alphabet Σ. A language is finite iff all its words are finite. Some common operations over languages are as follows:

- The language product XY of two languages X and Y consists of words of the form st where $s \in X$ and $t \in Y$.
- The Kleene closure X^* is defined as

$$X^* = \bigcup_{i \in \mathbb{N}} X_i$$

  where $X_0 = \{\epsilon\}$, $X_1 = X$, and $X_{i+1} = X_i X$.
- The *shuffle* $\|$ of two finite words is defined inductively as $\epsilon \| s = \{s\}$, $s \| \epsilon = \{s\}$, and $as \| bt = a(s \| bt) \cup b(as \| t)$, which is then lifted to the shuffle product of languages X and Y as

$$X \| Y = \{x \| y : \ x \in X \wedge x \in Y\}.$$

The parallel Kleene closure $X^{(*)}$ can be defined in the same way as the usual Kleene closure, replacing the language product with the shuffle product.

The *regular languages* are defined recursively as follows:

- The empty language $\emptyset$ is regular.
- The singleton language $\{a\}$ is regular for each a in Σ.
- If X and Y are regular, then XY is regular, the union $X \cup Y$ is regular, and the Kleene closure X^* is regular.

It is well known that all finite languages are regular.

It is well known that (regular) languages with language union as $+$, language product as $\cdot$, the empty language as 0, the empty word language $\{\varepsilon\}$ as 1, and the Kleene closure as the star form Kleene algebras. In fact, Kleene algebras are complete with respect to the equational theory of regular languages [Koz94]. This means that any equation that can be proven between regular languages, can also be proven via the axioms of Kleene algebra. This implies that equations in Kleene algebra are decidable. Moreover, commutative Kleene algebras are complete with respect to the equational theory

of regular languages over multisets [Con71], that is languages in which letters within words are allowed to commute arbitrarily. It follows that equations in (commutative) Kleene algebras are decidable. Kozen's completeness proof for Kleene algebras has been mechanised in Coq by Braibant and Pous [BP12]. This proof is highly complex—it involves defining matrices over Kleene Algebras, and proving that those matrices in turn form Kleene algebras. It then uses techniques from automata theory whereby automata are encoded as matrices to prove completeness.

More formally:

Theorem 1. $(\Sigma^*, \cup, \cdot, \emptyset, \{\epsilon\}, ^*)$ *is a Kleene algebra.*

Theorem 2. $(\Sigma^*, \cup, \|, \emptyset, \{\epsilon\}, ^{(*)}$ *is a commutative Kleene algebra.*

Theorem 3. $(\Sigma^*, \cup, \cdot, \|, \emptyset, \{\epsilon\}, ^{(*)})$ *is a concurrent Kleene algebra.*

3.4.2 Binary Relations

Another standard model for Kleene algebra is given by binary relations. A binary relation on a set A is a subset of the Cartesian product $A \times A$. The product of two relations R and S is defined as

$$R \circ S = \{(x, z) \mid \exists y.\ (x, y) \in R \wedge (y, z) \in S\}.$$

The identity relation Id on a set A is defined as

$$\mathsf{Id} = \{(x, x) \mid x \in A\}.$$

The reflexive transitive closure R^* for a relation R is given by $R^* = R^+ \cup \mathsf{Id}$, where the transitive closure R^+ is defined as

$$R^+ = \bigcup_{i \in \mathbb{N}} R^i,$$

where $R^0 = R$ and $R^{i+1} = R^i \circ R$.

Theorem 4. $(A \times A, \cup, \circ, \emptyset, \mathsf{Id}, ^*)$ *is a Kleene algebra for any set A.*

In addition to being complete with respect to the equational theory of regular languages, Kleene algebras are complete with respect to the equational theory of binary relations as well [Koz94].

Binary relations yield a standard semantics for sequential programs, but are unsuitable for concurrent programs due to the lack of any suitable operator for parallel composition.

3.4.3 Further Models

Pomsets, partially ordered multisets, represent a natural generalisation of words (which can be seen as totally ordered multisets, or tomsets) into a concurrent setting [Gis88]. Regular series-parallel pomsets are those which can be constructed via sequential and parallel composition, in a similar fashion to how the regular languages are constructed from regular expressions. Bi-Kleene algebras are sound and complete with respect to the equational theory of regular series-parallel pomset languages, and the equational theory is again decidable [LS13]. Regular series-parallel pomset languages with a suitable notion of pomset subsumption form concurrent Kleene algebras.

3.5 Lattices

A *join semilattice* $(S, \leq)$ is a poset for which binary joins (or suprema) exist, whereas a *complete join semilattice* is a poset for which arbitrary joins exist. Meet semilattices and complete meet semilattices are obtained by duality via the inverse order. Formally, a join or suprema for a subset S of a poset is the greatest element x such that x is greater than or equal to all elements in S. Meets or infima are defined in a dual fashion as the least element x less than or equal to all elements in S.

Equivalently a join semilattice may be defined as an algebraic structure $(S, +)$ where

- $(S, +)$ is a commutative semigroup,
- $x + x = x$,

i.e. $+$ is an associative, commutative and idempotent operator over S. The connection between the order theoretic $(S, \leq)$ and the algebraic $(S, \sqcup)$ is given by the natural partial order $x \leq y \iff x + y = y$. It is well known that every complete join semilattice is also a complete meet semilattice, hence a complete lattice. This is because the presence of arbitrary joins implies the existence of arbitrary meets (and vice versa).

In a lattice, the meet and join operators satisfy the absorption laws

$$x + (x \sqcap y) = x \qquad \text{and} \qquad x \sqcap (x + y) = x.$$

An (algebraic) lattice $(L, +, \sqcap)$ is distributive if for all x, y, and z in L:

$$x \sqcap (y + z) = (x \sqcap y) + (x \sqcap z).$$

The Knaster-Tarski theorem states that every isotone function on a complete lattice L has both a least and greatest fixpoint, or more generally, that the set of fixed points of f in L is itself a complete lattice. Furthermore Kleene's fixed point theorem states that any Scott-continuous function f

has a least fixpoint, defined as the suprema of the ascending Kleene chain of f. A Scott-continuous function is one which preserves all directed suprema, i.e.

$$\sum f(D) = f\left(\sum D\right)$$

where D has the property that for any $a, b \in D$ there exists a $c \in D$ such that $a \leq c$ and $b \leq c$. All isotone functions are Scott-continuous. The ascending Kleene chain of f is

$$\bot \leq f(\bot) \leq f(f(\bot)) \leq f(f(f(\bot))) \leq \ldots$$

In other words, any Scott-continuous function on a complete lattice (or rather a complete partial order) can be iterated from the bottom element to find the least fixed point. One can apply order reversal to obtain a similar result for greatest fixpoints, but Scott-continuous functions in the inverse order must be antitone (in the non-inverse order) which hardly ever holds in practice. Both these theorems have been proven in Isabelle and are useful for reasoning about iteration as fixed points.

3.6 Boolean Algebra

A *Boolean algebra* is a structure $(B, +, \sqcap, \bar{}, 0, 1)$ where:

- $(B, +, \sqcap)$ is a distributive lattice,
- $x + 0 = x$ and $x \sqcap 1 = x$ for all x (identity),
- $x + \overline{x} = 1$ and $x \sqcap \overline{x} = 0$ for all x (complementation).

The simplest non-trivial Boolean algebra is the two-element Boolean algebra. It has two elements, 0 and 1, and its operations are defined by the following rules.

$+$	0	1
0	0	1
0	1	1

$\sqcap$	0	1
0	0	0
0	0	1

x	0	1
$\overline{x}$	1	0

This algebra represents simple Boolean logic, where 0 is *false*, 1 is *true*, $+$ is *or*, $\sqcap$ is *and*, and $\bar{}$ is *not*.

A second common boolean algebra is the *power set Boolean algebra* for a set S. This is a Boolean algebra where the elements are subsets of S. Here, $+$ is set union ($\cup$), $\sqcap$ is set intersection ($\cap$), 0 is the empty set ($\emptyset$), 1 is the set S, and $\bar{}$ is complementation relative to S.

3.7 Galois Connections

A *Galois connection* between two partially ordered sets $(A, \leq)$ and $(B, \leq)$ is a pair of functions $f : A \to B$ and $g : B \to A$ such that

$$f(x) \leq y \iff x \leq g(y).$$

The function f is the *lower adjoint* while g is the *upper adjoint*. Identifying functions as adjoints of Galois connections is highly desirable, as they are known to satisfy many useful properties. In this section, some applications of Galois connections as they relate to our verification tool are considered. For a detailed overview of properties of Galois connections, see [Aar92].

Using Galois connections some notable laws for reasoning with fixed points can be derived, in particular, the least fixpoint fusion law states that if f is a lower adjoint, h and k are isotone functions and $f \circ h = k \circ f$ then $f(\mu h) = \mu k$. Similarly, the greatest fixpoint fusion law states that if g is an upper adjoint, h and k are isotone functions then $g(\nu h) = \nu k$.

These fusion laws have a long history in program refinement and specification. First explicitly named as such in Meijer's *Functional Programming with Bananas, Lenses, Envelopes and Barbed Wire* [MFP91], they go back at least as far as Stoy's *Denotational Semantics* [Sto77]. More recently they have been recognised as generally useful program transformation rules, especially for functional programs.

3.8 Weak Kleene Algebras and Dioids

For applications involving total correctness rather than partial correctness, certain axioms of Kleene algebra and dioids are no longer applicable. In particular the right annihilation axiom $x0 = 0$ should not hold in such situations—clearly if x doesn't terminate, then the 0 will never be able to abort x. A *weak dioid*, or *weak Kleene algebra* is one in which this right annihilation axiom does not hold. In general, for any algebra in this section with the right annihilation law, there exists a weak variant without it.

Furthermore if the right distributivity law $xy + xz \leq x(y + z)$ does not hold for some algebra in addition to right annihilation not holding, then one has a *near* algebra. This is the case in for example, process algebras where the behaviours in Figure 3.2 are different. For the models considered in this thesis, right distributivity always holds, but right annihilation often does not. It is always the case that the opposite of a Kleene algebra, i.e. one where the order of operands for multiplication has been reversed, is a Kleene algebra. However, it is not the case that left Kleene algebras need be closed under opposition [Koz90].

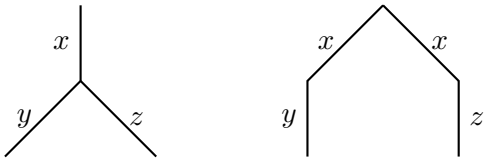


Figure 3.2: Right distributivity

3.9 Kleene Algebra with Tests and Hoare Logic

For modelling actual programs with conditional statements and while loops, Kleene Algebra must be extended with a notion of *tests*. Program tests and assertions are added to Kleene Algebras by embedding a boolean algebra of tests between 0 and 1. Following Kozen [Koz00], a *Kleene algebra with tests* (KAT) is a structure $(K, B, +, \cdot, ^\star, 0, 1, ^-)$ where $(K, +, \cdot, ^*, 0, 1)$ is a Kleene algebra and $(B, +, \cdot, ^-, 0, 1)$ a Boolean subalgebra of K. The operations are overloaded with $+$ as join, $\cdot$ as meet, 0 as the minimal element and 1 the maximal element of B. Complementation $^-$ is only defined on B. We write x, y, z for arbitrary elements of K and a, b, c for tests in B. Conditionals and loops can now be expressed:

$$\text{if } b \text{ } \{ \text{ } x \text{ } \} \text{ else } \{ \text{ } y \text{ } \} \text{ } = \text{ } bx + \overline{b}y, \qquad \text{while } b \text{ do } \{ \text{ } x \text{ } \} \text{ } = \text{ } (bx)^*\overline{b}.$$

Tests play a double role as assertions to encode (the validity of) Hoare triples:

$$\{\!|b|\!\}\, x \,\{\!|c|\!\} \iff bx\overline{c} = 0.$$

Multiplying a program x by a test b at the left or right means restricting its input or output by the condition b. Thus the term $bx\overline{c}$ states that program x is restricted to precondition b in its input and to the negated postcondition c in its output. Accordingly, $bx\overline{c} = 0$ means that x cannot execute from b without establishing c. This faithfully captures the meaning of the Hoare triple $\{\!|b|\!\}\, x \,\{\!|c|\!\}$. It is well known that algebraic relatives of all rules of Hoare logic except assignment can be derived in KAT [Koz00], and that binary relations under union, relational composition, the unit and the empty relation, and the reflexive transitive closure operation form a KAT where the tests are subsets of the identity relation Id.

In the context of concurrency, this approach is no longer appropriate, as tests can no longer be assumed to satisfy the KAT axioms in the presence of interference. The following approach by Tarlecki [Tar85] can be used instead. One can encode validity of a Hoare triple as

$$\vdash \{x\}y\{z\} \iff xy \leq z$$

for arbitrary elements of a Kleene algebra (not necessarily a KAT). Nevertheless all the rules of propositional Hoare logic except the assignment axiom can still be derived [HMSW11].

3.10 Action Algebra

An *action algebra* [Pra90, Koz93] is an structure $(A, +, \cdot, 0, 1, ^\star, \leftarrow, \rightarrow)$, such that $(A, +, \cdot, 0, 1)$ is a dioid, and satisfying

$$xy \leq z \iff^{L} x \leq z \leftarrow y, \quad \text{and} \quad xy \leq z \iff^{R} y \leq x \rightarrow z, \tag{3.1}$$

$$1 + x^\star x^\star + x \leq x^\star \leq x^\star, \tag{3.2}$$

$$1 + yy + x \leq y \implies x^\star \leq y. \tag{3.3}$$

Axiom (3.1) defines the *left* and *right residuals* $\leftarrow$ and $\rightarrow$ as upper adjoints of $- \cdot y$ and $x \cdot -$ respectively. In other words, (3.1L) and (3.1R) describe families of Galois connections indexed by y and x. A commutative action algebra is one in which $xy = yx$. In such an action algebra the left and right residuals coincide. It is straightforward to show that for any action algebra, $(A, +, \cdot, 0, 1, ^\star)$ is a Kleene algebra.

Action algebra is so called as it was intended by Pratt to model *action logic*, wherein the right residual models *preimplication* of actions, had a then b, for example: "Had I bet on that horse I'd be rich". Conversely, *postimplication* models b if ever a, for example: "I'll be rich if that horse ever wins". For a comprehensive list of the properties of action algebras and their most important models formalised in Isabelle see [ASW13a], which includes the language and the relational model.

In [Pra90] Pratt is able to show that action algebra can be equationally axiomatised, i.e. the variety, ACT, of action algebras, is finitely based. The key insight of Pratt is that the star itself can be equationally axiomatised. For the Kleene algebra signature this is not possible, as shown by Redko [Red64]. This set of equivalent axioms to (3.1)–(3.3) is shown below:

$$\begin{aligned} x \rightarrow y \leq x \rightarrow (y + z), \qquad & x(x \rightarrow y) \leq y \leq x \rightarrow xy, \\ y \leftarrow x \leq (y + z) \leq x, \qquad & (y \leftarrow x)x \leq y \leq yx \leq x \\ x^\star \leq (x + y)^\star, \qquad & 1 + x^\star x^\star + x \leq x^\star, \end{aligned}$$

$$(x \rightarrow x)^\star \leq x \rightarrow x.$$

That action algebra can be equationally axiomatised is potentially interesting for automated reasoning as it implies that theorems of action algebra can potentially be solved by theorem provers specialised for equational reasoning, such as Waldmeister [BH96]. One might further hypothesise that automated reasoning tools such as sledgehammer in Isabelle might also be able to reason more effectively with action algebra as opposed to Kleene algebra. However, this does not appear to be the case, at least for sledgehammer [Arm12].

An obvious consequence of the existence of these residuals is that any inequality $x \leq y$ in action algebra can be rewritten as either $1 \leq a \rightarrow b$ or $1 \leq b \leftarrow a$. Interpreting $1 \leq$ as the symbol for theoremhood, $\vdash$, it can

be seen that the theorems of action algebra are its reflexive elements. This gives a Hilbert-style system for action logic.

From the perspective of program verification and refinement using regular algebras, we can rewrite the Tarlecki style triple $px \leq q$ as $x \leq p \to q$. Swapping the order into the refinement ordering $p \to q \sqsupseteq x$ we can see that we end up with a Galois connection

$$p \to q \sqsupseteq x \iff \{p\}x\{q\},$$

which captures the relationship between program refinement and Hoare-style program verification. In general $p \to q$ can be thought of as a specification statement modelling the greatest program from p to q. In Section 3.13 we show that pre- and post-implications are guaranteed to exist in the more general setting of quantales.

3.11 $\star$-continuous Kleene Algebra

Kozen's $\star$-continuous Kleene algebra [Koz94], also called an **N**-algebra by Conway [Con71] is a dioid satisfying the law

$$xy^*z = \sum\{xy^nz \mid n \in \mathbb{N}\}.$$

This law can be seen as the definition of y^* as a suprema of the set of powers of y combined with an infinite left and right distributivity law (see Section 3.13).

One advantage of $\star$-continuous Kleene algebra over Kleene algebra is that the while the completeness proof for Kleene algebra and regular languages is extremely involved, the equivalent completeness proof for $\star$-continuous Kleene algebra can be performed by straightforward induction.

3.12 Weak ω-Algebra and Demonic Refinement Algebra

A weak ω-algebra [LS11] is a left Kleene algebra expanded with an omega operator satisfying

$$xx^\omega = x^\omega,$$
$$y \leq z + x \cdot y \implies y \leq x^\omega + x^\star z.$$

These axioms are called the ω-unfold axiom and the ω-coinduction axiom respectively. In this thesis, the $^\omega$ operator can be considered to represent strictly infinite iteration. Iteration that is either finite or infinite is therefore represented as

$$x^\infty = x^\star + x^\omega.$$

While KAT, and Kleene algebra in general, provide a propositional Hoare logic for partial correctness reasoning, algebraic reasoning about total correctness and non-termination can be done in the context of ω-algebra [HS10, Gut12]. For an alternative treatment of non-termination, von Wright [vW04] introduces a demonic refinement algebra (DRA) as a structure

$$(D, +, \cdot, 0, 1, ^\star, ^\infty)$$

where:

- $(D, \cdot, +, 0, 1, ^\star)$ is a Kleene algebra,
- The strong iteration operator $^\infty$, which represents both finite or infinite iteration, satisfies the unfolding axiom $x^\infty = xx^\infty + 1$ and the induction axiom $z \leq xz + y \implies z \leq x^\infty y$

Von Wright's intended model for DRA is *(positively) conjunctive predicate transformers* over a state space Σ. That is, functions of type

$$(\Sigma \to \{0,1\}) \to (\Sigma \to \{0,1\}).$$

which distribute over arbitrary (non-empty) conjunctions of predicates. Such a model represents (demonically) non-deterministic programs according to a weakest precondition semantics. The *demonic* in demonic non-determinism, and DRA itself, represents that we have no possible influence over the choice operator $+$.

3.13 Quantales

A (unital) *quantale* is a dioid based on a complete lattice where the multiplication distributes over arbitrary suprema. Formally, it is a structure $(S, \leq, \cdot, 1)$ such that $(S, \leq)$ is a complete lattice, $(S, \cdot, 1)$ is a monoid and

$$x\left(\sum Y\right) = \sum_{y \in Y} xy, \qquad \left(\sum X\right) y = \sum_{x \in X} xy.$$

In a quantale $x^\star$ is the sum of all powers x^n, that is

$$x^\star = \sum_{n \in \mathbb{N}} x^n.$$

All quantales are Kleene algebras, and also $\star$-continuous Kleene algebra. Quantales were considered by Conway under the name **S**-algebra, or *standard Kleene algebra* [Con71]. Furthermore, in a quantale one can prove that the star is equal to the following fixed points from Section 3.3, i.e.

$$x^\star = \mu y.1 + xy = \mu y.1 + yx.$$

In a quantale the pre-implication and post-implication operators of action algebra can also be derived as

$$x \to z = \sum\{y \mid xy \leq z\}, \qquad \text{and} \qquad z \leftarrow y = \sum\{x \mid xy \leq z\}.$$

The omega operator from Section 3.12 can be defined in the context of a *completely distributive* quantale [HMSW11] where

$$x \sqcap \sum Y = \sum_{y \in Y} x \sqcap y,$$

i.e. one where meets distribute over arbitrary joins. In such a quantale, x^ω can be defined as a greatest fixpoint

$$x^\omega = \nu y.xy.$$

In such quantales where an omega operator can be defined it is often useful to drop the left infinite distributivity law

$$x\left(\sum Y\right) = \sum_{y \in Y} xy,$$

as this law implies $x0 = 0$ when $Y = \emptyset$. As per Section 3.8 such quantales are called *weak quantales*.

Chapter 4

Isabelle/HOL

4.1 Introduction

Isabelle/HOL [PNW11] is one of the most popular and well established interactive theorem proving (ITP) environments. It is widely used for formalising mathematics and in a multitude of computing applications.

Isabelle supports the engineering of theory hierarchies for algebras and their models. This feature is provided by Isabelle's type classes and locales [Bal10, HW08]. Type classes in Isabelle are similar to those in functional programming languages such as Haskell. Locales provide a more general module mechanism. The two concepts are linked and can often be used interchangeably. Classes usually suffice for simple algebraic specifications and locales for more complex parametric ones. Classes and locales provide a mechanism for theorem propagation: a theorem proved in a certain class is automatically valid in all subclasses; a model that belongs to a class is therefore an element of all superclasses.

Proofs within Isabelle have traditionally been based on reasoning with built-in rewrite-based simplifiers, domain-specific theorem provers, special solvers and tactics. This type of reasoning often requires considerable user expertise and domain-specific knowledge of library functions and lemmas. More recently, external automated theorem proving (ATP) systems and satisfiability modulo theories (SMT) solvers have been integrated via the *Sledgehammer* tool. When Sledgehammer is invoked on a proof goal, it uses a relevance filter to automatically gather hypothesis which are potentially useful for discharging said goal. It then passes both these hypotheses and the goal to the external provers. On success, the external proof output is internally reconstructed by Isabelle to ensure trustworthiness. Isabelle has a small, trustworthy logical Kernel, so proofs verified by Isabelle have an extremely high degree of trustworthiness. The Sledgehammer tool is complemented by additional tools such as *Nitpick* [BN10] and *Quickcheck* [Bul12], which search for counterexamples. For a recent overview of this automation technology see [BBN11].

In the following section, the algebras and concepts introduced in Chapter 3 are used as a vehicle from which the basics of the Isabelle interactive theorem prover are introduced.

4.2 Mechanising Algebras in Isabelle

In Isabelle a dioid can be defined using the typeclass functionality as

class dioid = semiring + ord +
  assumes add-idem[simp]: $x + x = x$
  and less-eq-def: $x \leq y \longleftrightarrow x + y = y$

As can be seen, a dioid is defined by extending the pre-existing semiring class with the additional axiom add-idem. The add-idem law is defined

as a simplification rule by using the simp attribute, so it can be used by Isabelle's built in simplifiers and other tactics. The natural partial order $x \leq y \longleftrightarrow x + y = y$ is also introduced in the dioid typeclass. A simple property of dioids is that addition is (left) isotone. This can be stated as a lemma in Isabelle as:

lemma (**in** dioid) add-iso: $x \leq y \Longrightarrow x + z \leq y + z$

To prove such a lemma in Isabelle, we could manually apply rewriting rules and simplification steps ourselves, for example:

lemma (**in** dioid) add-iso: $x \leq y \Longrightarrow x + z \leq y + z$
  apply (simp only: less-eq-def)
  apply (subst add-assoc[symmetric])
  apply (subst add-assoc)
  apply (subst add-commute) back
  apply (subst add-assoc[symmetric])
  apply (subst add-assoc) back
  apply simp
  done

In the above proof we begin by applying the definition of the natural partial order, which gives us the goal of $(x+z)+(y+z) = y+z$ with the assumption $x+y = y$. The next two steps simply apply associativity rules to rewrite the goal to $x + (z + y) + z = y + z$. The next line applies the add-commute rule to rewrite the goal to $x + (y + z) + z = y + z$. The back statement is used to backtrack between various possible substitutions of add-commute. The following two steps again apply associativity to rewrite the goal back to the form $(x + y) + (z + z) = y + z$. The simplifier can then be used complete the proof by using the assumption and the add-idem axiom.

This is a very long-winded way of completing such a simple proof in Isabelle, but serves to exemplify what one could term the *apply-style* form of proofs in Isabelle. This is the style of reasoning with 'built-in rewrite-based simplifiers, domain-specific theorem provers, special solvers and tactics' that was mentioned in Section 4.1. Isabelle also features a structured proof language called *Isar* [Wen07], which allows for proofs to be written in a human-readable form. Note that in the above proof, it is essentially impossible to follow the proof state without stepping through the solution within the theorem prover. Using Isar, the above proof can be rewritten as below.

lemma (**in** dioid) add-iso: $x \leq y \implies x + z \leq y + z$
proof −
  assume $x \leq y$
  hence [simp]: $x + y \leq y$ **by** (simp add: less-eq-def)

  have $(x + z) + (y + z) = x + (z + y) + z$

by (simp add: add-assoc)
also have ... $= x + (y + z) + z$
by (subst add-commute, simp)
also have ... $= (x + y) + (z + z)$
by (simp add: add-assoc[symmetric] del: add-idem)
also have ... $= y + z$
by simp
finally show ?thesis
by (simp add: less-eq-def)
qed

Now each step of the proof is readily apparent, as the actual steps are not hidden within the internal prover state, as was the case in the apply-style proof.

It would however be rather inconvenient to be forced to write all proofs at such a fine level of granularity. As mentioned, Isabelle provides built in integration with external automated theorem proving tools via the *sledgehammer* tool [BBN11]. Using sledgehammer the add-iso lemma can be proved simply as:

lemma (**in** dioid) add-iso: $x \leq y \implies x + z \leq y + z$
by (metis add-assoc add.left-commute add-idem less-eq-def)

The sledgehammer tool calls external theorem prover tools and SMT solvers which are used as a relevance filters to identify appropriate lemmas for proving the desired result. The proof is then reconstructed in Isabelle using the *metis* theorem prover. Proofs reconstructed via metis are guaranteed to be correct, as metis produces proofs which are verified by Isabelle's logical kernel. As such, none of the external theorem provers nor SMT solvers need be trusted.

The relationship between between dioids and Kleene algebra in Isabelle is established automatically, as Kleene algebras are defined as an expansion of a dioid with a star operation satisfying the above properties, as shown:

class kleene-algebra = dioid + star-op +
assumes star-unfoldl: $1 + x \cdot x^\star \leq x^\star$
and star-inductl: $z + x \cdot y \leq y \Rightarrow x^\star \cdot z \leq y$
and star-inductr: $z + y \cdot x \leq y \Rightarrow z \cdot x^\star \leq y$

The integration of ATP tools in Isabelle via Sledgehammer allows for proofs to be presented at near textbook level granularity. This is shown in Figure 4.1 for a fixpoint fusion law.

```
theorem fixpoint-fusion [simp]:
  fixes k :: 'b::complete-lattice ⇒ 'b
  and h :: 'a::complete-lattice ⇒ 'a
  and f :: 'a ⇒ 'b
  assumes upper-ex: lower-adjoint f
  and hiso: mono h and kiso: mono k
  and comm: f∘h = k∘f
  shows f (μ h) = μ k
proof
  show k (f (μ h)) = f (μ h) using monoD[OF hiso]
    by (metis comm fp-compute o-eq-dest-lhs)
next
  fix y :: 'b assume ky: k y = y
  obtain g where conn: galois-connection f g
    by (metis lower-adjoint-def upper-ex)
  have μ h ≤ g y
  proof (rule fp-induct)
    fix x y :: 'a assume x ≤ y thus h x ≤ h y
      by (rule monoD[OF hiso])
  next
    have f (g y) ≤ y by (metis conn deflation)
    hence f (h (g y)) ≤ y by (metis comm kiso ky monoD o-def)
    thus h (g y) ≤ g y by (metis conn galois-connection-def)
  qed
  thus f (μ h) ≤ y by (metis conn galois-connection-def)
qed
```

Figure 4.1: Fixpoint fusion in Isabelle

4.3 Custom Reasoning Tactics

In this section, the flexibility of Isabelle as a generic ITP system is demonstrated by constructing a tactic for deciding equations in $\star$-continuous Kleene algebra. An advantage of $\star$-continuous Kleene algebra over Kleene algebra is that while the completeness proof for Kleene algebra and regular languages is extremely involved, the equivalent completeness proof for $\star$-continuous Kleene algebra can be performed by straightforward induction [Koz97]. By combining this completeness result with an existing library for deciding regular expression equivalence in Isabelle [KN10], it is possible to construct a tactic in Isabelle to automatically decide equations in $\star$-continuous Kleene algebra.

We start by mapping regular expressions (over natural numbers) to their equivalent expressions in Kleene algebra. This is extremely simple as the operations of regular expressions and Kleene algebra coincide, and indeed, regular expressions are ground terms in the language of Kleene algebra.

primrec rexp-hom :: (nat ⇒ 'a) ⇒ nat rexp ⇒ 'a **where**
  rexp-hom v Zero = 0
| rexp-hom v One = 1
| rexp-hom v (Atom a) = v a
| rexp-hom v (Plus x y) = rexp-hom v x + rexp-hom v y
| rexp-hom v (Times x y) = rexp-hom v x · rexp-hom v y
| rexp-hom v (Star x) = (rexp-hom v x)*

The completeness theorem can then be proven in Isabelle:

lemma completeness: lang x = lang y ⟹ rexp-hom v x = rexp-hom v y

Proving this theorem requires a few hundred lines of Isabelle, so the proof is elided here.

Given the completeness theorem, a tactic can be implemented as follows. First, it rewrites an expression in Kleene algebra to the form expected by the completeness theorem by applying rexp-hom-thm. Second, it applies the completeness theorem for $\star$-continuous Kleene algebra. Third, it applies soundness of the decision procedure for regular languages. Finally, it evaluates the decision procedure for regular languages. This tactic is implemented in Isabelle/ML (the SML dialect used for programming Isabelle) and shown in Figure 4.2.
Using this tactic, equations in $\star$-continuous Kleene algebra can then be proven automatically as shown:

lemma (**in** star-continuous-ka) $1 + x \cdot x^\star + y + z = 1 \cdot (z + y + x^\star + 0)$ **by** ka

lemma (**in** star-continuous-ka) $x + y = y + x \cdot 1$ **by** ka

If one is willing to simply assume completeness of Kleene algebra in Isabelle without mechanising the proof, then this tactic becomes available

```
val ka-tac = Subgoal.FOCUS (fn {ctxt, concl, ...} =>
  let
    val rexp-hom-goal = dummy-all (@{term op ⇒} $
      to-ka (term-of concl) $ term-of concl) |> Syntax.check-term ctxt
    val rexp-hom-thm =
      Goal.prove ctxt [] [] rexp-hom-goal (fn {ctxt,...} => auto-tac ctxt)
    val completeness-thm =
      hd (Locale.get-witnesses ctxt RL
        [@{thm star-continuous-ka.completeness}])
  in
    rtac rexp-hom-thm 1
    THEN rtac completeness-thm 1
    THEN rtac @{thm soundness} 1
    THEN eval-tac ctxt 1
  end)
```

Figure 4.2: A decision procedure for $\star$-continuous Kleene algebra

in Kleene algebra. Furthermore, many useful Kleene algebras are in fact $\star$-continuous, so only having access to this tactic for $\star$-continuous algebras is not a significant drawback.

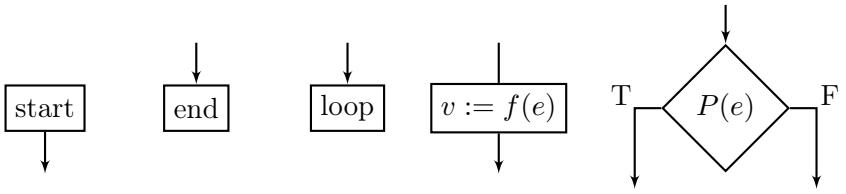
More recent versions of Isabelle than those used for constructing this tactic feature the *Eisbach* tactic language [MWM14]. This tactic language allows tactics like the one above to be constructed by even non-expert users of Isabelle, as they no longer require complex implementations in Isabelle/ML. This tactic language has been used in Chapter 8 to construct refinement and verification tools for rely-guarantee reasoning.

Chapter 5

Program Verification with Schematic Kleene Algebra

5.1 Applying Algebra to Program Verification

This chapter demonstrates how Kleene Algebra can be applied in practice to the verification of simple while programs. It therefore serves as a demonstration of the principles of algebraic tool design described in Chapter 1. We have already seen in Chapter 3 how the rules of propositional Hoare logic can be derived in KAT, but to apply KAT in program development and verification, formal treatment of assignments and program states is required. Axioms for assignments have been added, for instance, in *schematic Kleene algebra with tests* (SKAT) [AK01]. This extension of KAT is targeted at modelling the transformation of flowchart schemes. Flowchart schemes are flowcharts built from the following basic constructs:



A classical reference for flowchart schemes, scheme equivalence, and transformation is Manna's book *Mathematical Theory of Computation* [Man74]. Flowchart schemes represent unevaluated programs, and flowchart equivalence represents program equivalence under all possible interpretations of the function and relation symbols appearing in assignments and tests (f and P in the above flowchart constructs). A formalisation of SKAT in Isabelle is discussed in this section; our formalisation of a complex flowchart equivalence proof [Man74, AK01] is presented in Section 5.4. In this chapter the conceptual development of SKAT together with its formalisation in Isabelle is given. In addition, a verification tool within Isabelle for simple while programs is described, wherein pre- and post-conditions are represented in a novel way using a Kleene module.

This chapter is primarily based upon material appearing in

- A. Armstrong, G. Struth and T. Weber. *Program Analysis and Verification Based on Kleene Algebra in Isabelle/HOL.* In S. Blazy, C. Paulin-Mohring and D. Pichardie (eds.), ITP 2013, LNCS 7998. [ASW13b]

5.2 Schematic KAT and Flowchart Schemes

In this section a development of SKAT within Isabelle is presented. A *ranked alphabet* or *signature* Σ consists of a family of function symbols f, g, ... and relation symbols P, Q, ... together with an arity function mapping symbols to $\mathbb{N}$. There is always a null function symbol with arity 0, it serves the same purpose as null in most programming languages. In Isabelle, ranked

alphabets are implemented as a type class. Variables are represented by natural numbers. Terms over Σ, or Σ-expressions, are defined as a polymorphic Isabelle datatype.

datatype 'a trm = App 'a "'a trm list" | Var nat

Arity checks are omitted to avoid polluting proofs with side conditions. In practice, verifications will fail if arities are violated. Variables and Σ-expressions form assignment statements; together with predicate symbols they form tests in SKAT. Predicate expressions (atomic formulae) are also implemented as a datatype.

datatype 'a pred = Pred 'a "'a trm list"

Evaluation of terms, predicates and tests relies on an interpretation function. It maps function and relation symbols to functions and relations. It is used to define a notion of flowchart equivalence [AK01, Man74] with respect to all interpretations. It is also needed to formalise Hoare logic in Section 5.5 by interpreting Σ-expressions in semantic domains. In Isabelle, it is based on the following pair of functions.

record ('a, 'b) interp =
  interp-fun :: 'a ⇒ 'b list ⇒ 'b
  interp-rel :: 'a ⇒ 'b relation

Σ-expressions are now included into SKAT expressions, which models flowchart schemes.

datatype 'a skat-expr =
    SKAssign nat "'a trm"
  | SKPlus "'a skat-expr" "'a skat-expr" (**infixl** "⊕" 70)
  | SKMult "'a skat-expr" "'a skat-expr" (**infixl** "⊙" 80)
  | SKStar "'a skat-expr"
  | SKBool "'a pred bexpr"
  | SKOne
  | SKZero

In this datatype, SKAssign is the assignment constructor; it takes a variable and a Σ-term as arguments. The other constructors represent the operations of KAT, and thus capture the programming constructs of sequential composition, conditionals and while loops. The type *'a pred bexpr* represents Boolean combinations of predicates, which form the tests in SKAT. The concrete connection between the SKAT syntax and Manna's flowchart schemes is discussed in [AK01], but it is not formalised here. The type *'a skat-expr* is therefore the freely-generated term algebra for the KAT signature, with predicates and assignments as variables.

Having formalised the SKAT syntax a notion of flowchart equivalence can be given using Isabelle's quotient types [KU11, HK13]. First define the

obvious congruence on SKAT terms that includes the KAT axioms and the SKAT assignment axioms

$$\begin{array}{lcll} x := s; y := t & = & y := t[x/s]; x := s & (y \notin FV(s)), \\ x := s; y := t & = & x := s; y := t[x/s] & (x \notin FV(s)), \\ x := s; x := t & = & x := t[x/s], & \\ a[x/t]; x := t & = & x := t; a. & \end{array}$$

In the following inductive definition only the equivalence axioms, a single Kleene algebra axiom and an assignment axiom are shown explicitly. Additional recursive functions for free variables and substitutions support the assignment axioms.

inductive skat-cong :: ('a::ranked-alphabet) skat-expr ⇒ 'a skat-expr ⇒ bool
  (infix ≈ 55) **where**
  refl [intro]: $p \approx p$
| sym [sym]: $p \approx q \implies q \approx p$
| trans [trans]: $p \approx q \implies q \approx r \implies p \approx r$
...
| mult-assoc: $(p \odot q) \odot r \approx p \odot (q \odot r)$
...
| assign1: $[\![x \neq y; y \notin FV s]\!] \implies$
  SKAssign x $s \odot$ SKAssign y $t \approx$ SKAssign y $(t[x/s]) \odot$ SKAssign x s
...

Isabelle's quotient package now allows for formally taking the quotient of SKAT expressions with respect to *skat-cong*. This means that the SKAT axioms are imposed on the SKAT term algebra given by *'a skat-expr*. The SKAT axioms then become available for reasoning about SKAT expressions.

quotient-type 'a skat = ('a::ranked-alphabet) skat-expr / skat-cong

Using this notion of equivalence on SKAT expressions additional syntactic sugar is defined by lifting constructors to SKAT operations, for instance,

lift-definition skat-plus :: ('a::ranked-alphabet) skat ⇒ 'a skat ⇒ 'a skat
  (**infixl** + 70) is SKPlus

We have used Isabelle's transfer tactic to provide nice programming syntax and lift definitions from the congruence. For instance,

lemma skat-assign1:
  $[\![x \neq y; y \notin FV s]\!] \implies (x := s \cdot y := t) = (y := t[x/s] \cdot x := s)$

An interpretation statement formally shows in Isabelle that the algebra thus constructed forms a KAT.

definition tests :: ('a ranked-alphabet) skat ord **where**
  tests = ⦇carrier = test-set, le = (λp q. skat-plus p q = q)⦈

definition free-kat :: ('a ranked-alphabet) skat test-algebra **where**
  free-kat = (|carrier = UNIV, plus = skat-plus, mult = skat-mult,
    one = skat-one, zero = skat-zero, star = skat-star,
    test-algebra.test = tests|)
interpretation skt: kat free-kat

Proving this statement required some work. First, it uses a comprehensive implementation of Kleene algebra with tests defined with explicit carrier sets in Isabelle. This implementation differs from prior implementations in the Archive of Formal Proofs [AGS14b], in that it has the carrier set of the algebra explicitly formalised as a set in Isabelle, rather than as a type. This KAT library contains about 100 lemmas for dealing with the Kleene star and combined reasoning about the interaction between actions and tests. For example, typical properties of the star and tests are $(p + q)^* = p^*(q \cdot p^*)^*$, $(pq)^*p = p(qp)^*$ or $bp = pc \Longleftrightarrow bp!c = !bpc$.

Second, it must be shown that the quotient algebra constructed satisfies the KAT axioms, including the axioms of Boolean algebra for the subalgebra of tests. A principal complication comes from the fact that Boolean complementation is defined as a partial operation, that is, on tests only; thus it cannot be directly lifted from the congruence. It must be defined indirectly using Isabelle's indefinite description operator, which substantially complicates proofs. After this interpretation proof, most statements shown for KAT are automatically available in the quotient algebra. The unfortunate exception is again the partially defined negation symbol, which is not fully captured by the interpretation statement. Here, KAT theorems need to be duplicated by hand for this quotient algebra.

When defining a quotient subtype, Isabelle automatically generates two coercion functions. The *abs-skat* function maps elements of type *'a skat-expr* to elements of the quotient algebra type *'a skat*, while the *rep-skat* function maps in the converse direction. Both these functions are again based on Isabelle's definite description operator, which can be unwieldy. However, for inductively defined types such as those used above, the following equivalent, and computationally more appealing, recursive function instead of *abs-skat* can be used instead. This allows for simple proofs via induction to be performed.

primrec abs :: 'a::ranked-alphabet skat-expr ⇒ 'a skat (⌊-⌋ [111] 110) **where**
  abs (SKAssign x y) = x := y
| abs (SKPlus x y) = abs x + abs y
| abs (SKMult x y) = abs x · abs y
| abs (SKBool p) = test-abs p
| abs SKOne = **1**
| abs SKZero = **0**
| abs (SKStar x) = (abs x)⋆

Mathematically, *abs* (or $\lfloor - \rfloor$) is a homomorphism. It is primarily useful for programming various tactics.

5.3 Formalising a Metatheorem

The following metatheorem due to Angus and Kozen (Lemma 4.4 in [AK01]) has been formalised as an example. This metatheorem that can be instantiated, for instance, to check commutativity conditions, eliminate redundant variables or rename variables in flowchart transformation proofs. This theorem can be instantiated to develop tactics that support proof automation in the flowchart example of the next section.

theorem metatheorem:
  assumes kat-homomorphism f
  and kat-homomorphism g
  and $\bigwedge a.\ a \in \text{atoms}\ p \implies f\ a \cdot q = q \cdot g\ a$
  shows $f\ p \cdot q = q \cdot g\ p$

Proof of this metatheorem proceeds by induction on p, expanding Angus and Kozen's proof. The predicate *kat-homomorphism* in the theorem states that f and g are KAT homomorphisms. This notion is defined for any function f in Isabelle as a locale:

locale kat-homomorphism =
  fixes f :: 'a::ranked-alphabet skat-expr ⇒ 'b::ranked-alphabet skat
  assumes hom-plus: f (x ⊕ y) = f x + f y
  and hom-test-plus: f (SKBool (P :+: Q)) = f (SKBool P) + f (SKBool Q)
  and hom-mult: f (x ⊙ y) = f x · f y
  and hom-test-mult: f (SKBool (P :·: Q)) = f (SKBool P) · f (SKBool Q)
  and hom-star: f (SKStar x) = (f x)⋆
  and hom-one: f SKOne = **1**
  and hom-test-one: f (SKBool BOne) = **1**
  and hom-zero: f SKZero = **0**
  and hom-test-zero: f (SKBool BZero) = **0**
  and hom-not: f (SKBool (BNot P)) = !(f (SKBool P))
  and hom-tests: f (SKBool P) ∈ carrier tests

which formalises that f must preserve all the operations of SKAT. The functions f and g map from SKAT terms into the SKAT quotient algebra defined previously, hence they have the same type as *abs*. The atoms function returns all the atomic subexpressions of a SKAT term, i.e. all the assignments and atomic tests.

Angus and Kozen have observed that if q commutes with all atomic subexpressions of p, then q commutes with p. This is a simple instantiation of the metatheorem. It can be obtained in Isabelle as follows:

lemmas skat-comm = metatheorem[OF abs-hom abs-hom]

This instantiates f and g using the fact that *abs* is a KAT morphism.

Lemma 4.5 in [AK01] states that if a variable x is not read in an expression p, then setting it to null will eliminate it from p.

lemma eliminate-variables
  assumes $x \notin$ reads p
  shows $\lfloor p \rfloor \cdot x := \text{null} = \lfloor \text{eliminate } x\ p \rfloor \cdot x := \text{null}$

In the statement of this lemma, *reads p* is a recursive function that returns all the variables on the right-hand side of all assignments within p, and the function *eliminate x p* removes all assignments to x in p.

As mentioned, the metatheorem and its various instances can be used to develop tactics that check for commutativity and eliminate variables. These tactics take expressions of the quotient algebra and coerce them into the term algebra to perform these syntactic manipulations. All the machinery for these coercions, such as *abs*, is thereby hidden from the user. A simple application example is given by the following lemma.

lemma comm-ex: $(1 := \text{Var } 2;\ 3 := \text{Var } 4) = (3 := \text{Var } 4;\ 1 := \text{Var } 2)$
  by skat-comm

5.4 Verification of Flowchart Equivalence

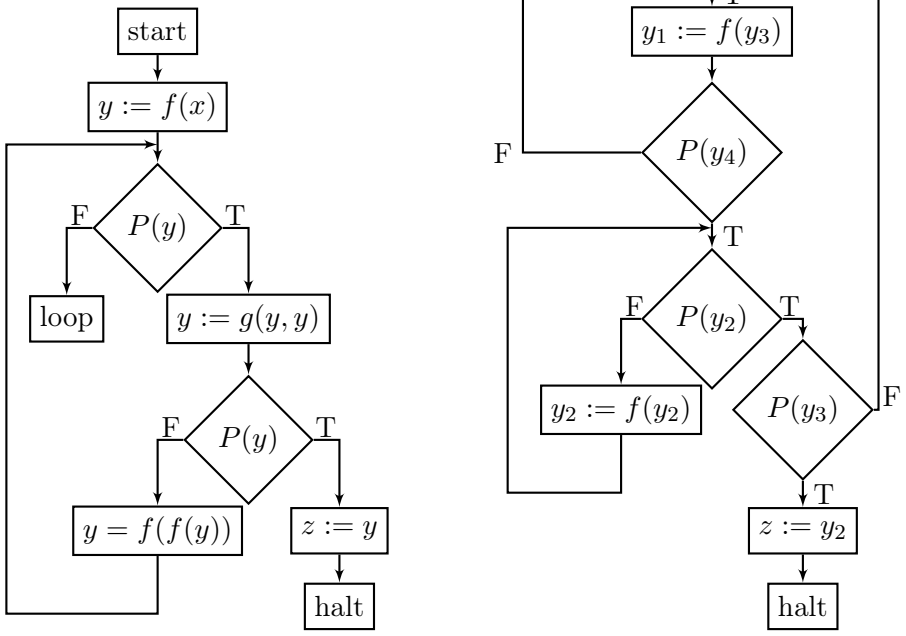
The SKAT implementation from the previous section has been used to verify a well known flowchart equivalence example in Isabelle. It is attributed by Manna to Paterson [Man74]. The flowcharts can be found at page 16f. in Angus and Kozen's paper [AK01] or pages 254 and 258 in Manna's book [Man74]; they are reproduced here in Figure 5.2. Manna's proof essentially uses diagrammatic reasoning, whereas Angus and Kozen's proof is equational. Their algebraic proof can be reconstructed at the same level of granularity in Isabelle. The two flowcharts, translated into SKAT by Angus and Kozen, are shown in Figure 5.1.

In the code, lists delimited by brackets indicate blocks of sequential code; loop expressions indicate the star of a block of code that follows. The *seq* function converts a block of code given as a list of SKAT expressions into a SKAT expression. The *halt* command sets all non output variables used in the scheme to *null*. To make algebraic reasoning more efficient, Angus and Kozen introduce definitions that abbreviate atomic commands, in particular assignments, and tests. Their example is followed here (see Figure 5.3). The flowchart equivalence problem can then be expressed more succinctly and abstractly in KAT, as all assignment statements, which are dealt with by SKAT, have been abstracted.
The proof that rewrites these KAT expressions, however, needs to descend to SKAT in order to derive commutativity conditions between expressions that

```
definition scheme1 ≡ seq
  [ 1 := vx, 4 := f (Var 1), 1 := f (Var 1)
  , 2 := g (Var 1) (Var 4), 3 := g (Var 1) (Var 1)
  , loop
    [ !(P (Var 1)), 1 := f (Var 1)
    , 2 := g (Var 1) (Var 4), 3 := g (Var 1) (Var 1)
    ]
  , P (Var 1), 1 := f (Var 3)
  , loop
    [ !(P (Var 4)) + seq
      [ P (Var 4)
      , (!(P (Var 2)) · 2 := f (Var 2))⋆
      , P (Var 2), ! (P (Var 3))
      , 4 := f (Var 1), 1 := f (Var 1)
      ]
    , 2 := g (Var 1) (Var 4), 3 := g (Var 1) (Var 1)
    , loop
      [ !(P (Var 1)), 1 := f (Var 1)
      , 2 := g (Var 1) (Var 4), 3 := g (Var 1) (Var 1)
      ]
    , P (Var 1)
    , 1 := f (Var 3)
    ]
  , P (Var 4)
  , (!(P (Var 2)) · 2 := f (Var 2))⋆
  , P (Var 2), P (Var 3), 0 := Var 2
  , halt
  ]

definition scheme2 ≡
  [ 2 := f vx, P (Var 2)
  , 2 := g (Var 2) (Var 2)
  , skat-star (seq
    [ !(P (Var 2))
    , 2 := f (f (Var 2))
    , P (Var 2)
    , 2 := g (Var 2) (Var 2)
    ])
  , P (Var 2), 0 := Var 2, halt
  ]
```

Figure 5.1: Equivalent flowchart schemes formalised in Isabelle



Scheme S_{6E} [Man74, p. 258] Scheme S_{6A} [Man74, p. 254]

Figure 5.2: Two equivalent flowchart schemes

```
seq [x1,p41,p11,q214,q311,loop [!a1,p11,q214,q311],a1,p13
    ,loop [!a4 + seq [a4,(!a2·p22)⋆,a2,!a3,p41,p11]
          ,q214,q311,loop [!a1,p11,q214,q311],a1,p13]
    ,a4,(!a2·p22)⋆,a2,a3,z2,halt]
=
seq [s2,a2,q222,(seq [!a2,r22,a2,q222])⋆,a2,z2,halt]
```

Figure 5.3: Abbreviated forms of scheme1 and scheme2

depend on variables and Σ-terms. These conditions are then lifted to KAT. The condition expressed in Lemma *comm-ex* from Section 5.3, for instance, reduces to the KAT identity $pq = qp$ when abbreviating $1 := Var\ 2$ as p, and $3 := Var\ 4$ as q. In our proof these commutativity conditions are inferred in a lazy fashion. This follows Angus and Kozen's proof essentially line by line.

The proof heavily depends on the underlying KAT library. Additionally, the tactics mentioned in the previous section have to be further refined to be able to efficiently manipulate the large SKAT expressions that occur in the proof. Most of these implement commutations in lists of expressions modulo commutativity conditions on atomic expressions which are inferred from SKAT terms on the fly.

The size of our proof as a LaTeX document is about 12 pages, twice as many as in Angus and Kozen's manual proof, but this is essentially due to aligning their horizontal equational proofs in a vertical way due to Isabelle's syntax. A previous proof in a special-purpose SKAT prover required 41 pages [AHK06]. This impressively demonstrates the power of Isabelle's proof automation. Previous experience in theorem proving with algebra shows that the level of proof automation in algebra is often very high [HS07, GSW11, FS12]. In this regard, the proof experience here is perhaps slightly underwhelming, as custom tactics and low-level proof techniques were needed for our step-by-step proof reconstruction. A higher degree of automation seems difficult to achieve, and a complete automation of the scheme equivalence proof is currently out of reach. The main reason is that the flowchart terms in KAT are much longer, and combinatorially more complex, than those in typical textbook proofs.

Decision procedures for variants of Kleene algebras, can help overcome these difficulties, as can be seen in Pous' formalisation of SKAT in Coq [Pou13a], which was developed concurrently with the Isabelle implementation presented in this Chapter. Pous gives a decision procedure for not only Kleene Algebra, but also mechanises a decision procedure for KAT based on guarded strings due to Kozen. By using this decision procedure Pous is able to give a significantly shorted proof of the above theorem—this is because he can automate all the reasoning steps that only rely on axioms of KAT, and focus

on the core complexity of the proof. The decision procedure in Chapter 4 for $\star$-continuous Kleene algebra cannot be used here as it is unable to deal with the tests found in KAT.

5.5 Hoare Logic with Kleene Modules

It is well known that Hoare logic—except the assignment rule—can be encoded in KAT as well as in other variants of Kleene algebra such as modal Kleene algebras [MS06] and Kleene modules [EMS03]. The latter are algebraic relatives of propositional dynamic logic. This chapter represents the first combination of these algebras with the assignment rule and their application in program verification.

Here a novel approach has been used in which SKAT and Kleene modules are combined. This allows for the separation of tests conceptually from the pre- and post-conditions of programs.

A *Kleene module* [Lei06] is a structure $(K, L, :)$ where K is a Kleene algebra, L a join-semilattice with least element $\bot$ and : a mapping of type $L \times K \to L$ where

$$\begin{aligned} P : (x + y) &= P : x \sqcup P : y, & (P \sqcup Q) : x &= P : x \sqcup Q : x, \\ P : (x \cdot y) &= P : x : y, & (P \sqcup Q) : x \leq Q &\longrightarrow P : x^* \leq Q, \\ P : 0 &= \bot, & P : 1 &= P. \end{aligned}$$

In this context, L models the state space, propositions or assertions of a program, K its actions, and the scalar product maps a proposition and an action to a new proposition. We henceforth assume that L is a Boolean algebra with maximal element $\top$ and use a KAT instead of a Kleene algebra as the first component of the module. The interaction between assertions, as modelled by the Boolean algebra L, and tests, as modelled by the Boolean algebra B, is captured by the new axiom

$$P : p = P \sqcap (\top : p).$$

The scalar product $\top : p$ coerces the test p into an assertion ($\top$ does not restrict it); the scalar product $P : p$ is therefore equal to a conjunction between the assertion P and the test p.

Isabelle's locales are used to implement modules over KAT. Hoare triples can then be defined as usual.

definition hoare-triple :: 'b ⇒ 'a ⇒ 'b ⇒ bool (⦃-⦄-⦃-⦄ [54,54,54] 53) **where**
⦃P⦄x⦃Q⦄ ≡ P :: x ⊑$_L$ Q

As : is a reserved symbol in Isabelle, :: is used for the scalar product. The index L refers to the Boolean algebra of assertions and the order $\sqsubseteq_L$ is the order on this Boolean algebra. The Hoare rules excluding assignment can now be derived as theorems in these modules [EMS03]. In Isabelle this is

achieved almost automatically via sledgehammer. Applying the resulting Hoare-style calculus—which operates by equational reasoning—for program verification requires the provision of more fine-grained syntax for assertions and refinement statements in addition to adding some form of assignment axiom.

This first-order syntax is obtained once more by specialising KAT to SKAT, and by interpreting the SKAT expressions in the Boolean algebra of propositions or states. As is common, program states are represented as functions from variables to values. Assertions correspond to sets of states. Hence the Boolean algebra L is instantiated as a powerset algebra over states. Similar implementations are already available in theorem provers such as Isabelle, HOL and Coq [Nip98, Sch06, CG10, NMS+08], but they have not been implemented as simple instantiations of more general algebraic structures. Assignment statements are translated in Gordon style [CG10] into forward predicate transformers which map assertions (preconditions) to assertions (postconditions).

This is, of course, compatible with the module-based approach. To implement the scalar product within the KAT module, we begin by writing an evaluation function which, given an interpretation and a SKAT expression, returns the forward predicate transformer for that expression.

fun eval-skat-expr ::
  ('a::ranked-alphabet, 'b) interp ⇒ 'a skat-expr ⇒ 'b mems ⇒ 'b mems
  where
  eval-skat-expr D (SKAssign x t) Δ = assigns D x t Δ
| eval-skat-expr D (SKBool a) Δ = filter-set (eval-bexpr D a) Δ
| eval-skat-expr D ($p \odot q$) Δ = eval-skat-expr D q (eval-skat-expr D p Δ)
| eval-skat-expr D ($p \oplus q$) Δ = eval-skat-expr D p Δ ∪ eval-skat-expr D q Δ
| eval-skat-expr D (SKStar p) Δ = ($\bigcup n$. iter n (eval-skat-expr D p) Δ)
| eval-skat-expr D SKOne Δ = Δ
| eval-skat-expr D SKZero Δ =

It can now be proven that if two SKAT expressions are equivalent according to the congruence defined in Section 5.2, then they represent the same predicate transformer. The proof is by induction. This property allows the lifting of the *eval-skat-expr* function to the quotient algebra.

theorem skat-cong-eval:
  skat-cong p q $\Longrightarrow$ $\forall\Delta$. eval-skat-expr D p Δ = eval-skat-expr D q Δ

lift-definition eval ::
  ('a::ranked-alphabet, 'b) interp ⇒ 'a skat ⇒ 'b mems ⇒ 'b mems
  is eval-skat-expr

Using this lifting, one can reason algebraically in instances of SKAT that have been generated by the evaluation function. This enables the derivation

of an assignment rule for forward reasoning in Hoare logic from the SKAT axioms.

lemma hoare-assignment: $P[x/t] \subseteq Q \implies \{\!|P|\!\}\, x := t \,\{\!|Q|\!\}$

We could equally derive a forward assignment rule $P\,\{\!|x := s|\!\}\,P[x/s]$, but this seems less useful in practice.

To facilitate automated reasoning a notion of loop invariants is added as syntactic sugar on while loops. Invariants are assertions used by the tactic that generates verification conditions.

$$\text{while } b \text{ invariant } i \text{ do } \{\ p\ \} = (bp)^*\overline{b}.$$

A refined while rule which uses the loop invariant is as follows.

lemma hoare-while-inv:
assumes b-test: $b \in$ carrier tests
and $Pi : P \subseteq i$ **and** iQ: $i \cap (\text{UNIV} ::! b) \subseteq Q$
and inv-loop: $\{\!|i \cap (\text{UNIV} :: b)|\!\}\, p \,\{\!|i|\!\}$
shows $\{\!|P|\!\}$ while b invariant i do $\{\ p\ \}\,\{\!|Q|\!\}$

Note that this particular rule has been instantiated to the powerset algebra of states, but it could as well have been defined abstractly.

Isabelle already provides a package for Hoare logic [Sch06]. Since there is one Hoare rule per programming construct, it uses a tactic to blast away the control structure of programs. A similar tactic for our SKAT-based implementation, called *hoare-auto* can easily be implemented.

5.6 Verification Examples

This variant of Hoare logic has been applied to prove the partial correctness of some simple algorithms. Instead of applying each rule manually, the tactic *hoare-auto* is used to make their verification with sledgehammer almost fully automatic. Note that more complex examples would certainly require more user interaction or perhaps even more sophisticated tactics to discharge the generated proof obligations. In addition, libraries for more complex data types would be required. These simple examples are given in Figure 5.4

Finally, the algebraic approach described in the previous section is expressive enough for deriving further program transformation or refinement rules, which would only be admissible (not derivable) in Hoare logic. As an example consider the two proofs of two simple Hoare-style inference rules given in Figure 5.5. Program refinement or transformation rules could be derived in a similar way given suitable constructs in the algebra, as per Section 3.10.
Only the derivation of the first rule is not fully automatic. The side condition $P :: p = (\top :: p) \sqcap P$ expresses the fact that if assertion P holds before

lemma euclids-algorithm:
$\{\!|mem.\ mem\ 0 = x \wedge mem\ 1 = y|\!\}$
WHILE !(pred (*EQ* (Var 1) (*NAT* 0)))
INVARIANT $\{mem.$ gcd $(mem0)\ (mem1) =$ gcd $x\ y\}$
DO
2 := Var 1;
1 := *MOD* (*Var*0) (*Var*1);
0 := Var 2
WEND
$\{\!|\{mem.\ mem\ 0 =$ gcd $x\ y\}|\!\}$
by hoare-auto (metis gcd-red-nat)

lemma factorial:
$\{\!|\{mem.\ mem\ 0 = x\}|\!\}$
1 = NAT 1;
(*WHILE* !(pred (*EQ* (Var 0) (*NAT* 0)))
INVARIANT $\{mem.$ fact $x = mem\ 1 *$ fact $(mem\ 0)\}$
DO
1 := *MULT* (Var 1) (Var 0); 0 := *MINUS* (Var 0) (*NAT* 1)
WEND)
$\{\!|\{mem.\ mem\ 1 =$ fact $x\}|\!\}$
by hoare-auto (metis fact-reduce-nat)

Figure 5.4: Simple verification examples

lemma derived-rule1:
assumes $P1, P2, Q1, Q2 \subseteq$ carrier A **and** $p \in$ carrier K
and $\{\!|P1|\!\}p\{\!|Q1|\!\}$ **and** $\{\!|P2|\!\}p\{\!|Q2|\!\}$
shows $\{\!|P1 \sqcap P2|\!\}p\{\!|Q1 \sqcap Q2|\!\}$
using assms
apply (auto simp add: hoare-triple-def assms, subst A.bin-glb-var)
by (metis A.absorb1 A.bin-lub-var A.meet-closed A.meet-comm mod-closed mod-join)+

lemma derived-rule2:
assumes $P, Q, R \subseteq$ carrier A **and** $p \in$ carrier K **and** $P :: p = (\top :: p) \sqcap P$
and $\{\!|Q|\!\}p\{\!|R|\!\}$
shows $\{\!|P \sqcap Q|\!\}p\{\!|P \sqcap R|\!\}$
by (insert assms) (smt derived-rule1 derived-rule2 insert-subset)

Figure 5.5: Derived Hoare logic rules

execution of program p, which is the left-hand side of the equation, then it also holds after p is executed. The expression $\top :: p$ represents the assertion that holds after p is executed without any input restriction.

These examples demonstrate the benefits of the algebraic approach in defining syntax, deriving domain-specific inference rules and linking with more refined models and semantics of programs with exceptional ease. While, in the context of verification, these tasks belong more or less to the meta-level, they are part of actual correctness proofs in program construction, transformation or refinement. This could likely be an important domain for future applications.

5.7 Conclusion

In this Chapter schematic Kleene algebra with tests has been implemented in Isabelle/HOL. This represents the first application of KAT/SKAT within a verification tool built with a standard interactive theorem prover. Pous' implementation of SKAT in Coq [Pou13a] was done concurrently and independently. The only prior mechanisation of SKAT was in Aboul-Hosn and Kozen's KAT-ML [AHK06], an extremely specialised system, and therefore lacking a great deal of the utility provided by a general interactive theorem prover. In particular, it does not allow the verification of programs as in Section 5.5 and 5.6.

The implementation of schematic Kleene algebra in this chapter has been used to formalise a complex flowchart equivalence proof by Angus and Kozen. The proof is significantly shorter than a previous formalisation in a custom theorem prover for Kleene algebra with tests. The proof follows Angus and Kozen's manual proof almost exactly and translates it essentially line-by-line into Isabelle, despite some weaknesses in proof automation which sometimes forced reasoning at quite a low level.

SKAT has been extended to support the verification of simple algorithms in a Hoare-logic style. This approach provides a seamless bridge between our abstract algebraic structures and concrete programs as per the approach outlined in Section 1.2. We have tested our approach on a few simple verification examples. Beyond that, additional Hoare-style rules and tactics for proof automation have been derived abstractly in the algebraic setting. These can be instantiated to different semantics and application domains. In the context of verification the main applications of algebra seem to be at this meta-level. The situation is different when developing programs from specifications or proving program equivalence, as the flowchart scheme tranformation shows.

This Chapter serves as an example of the benefits algebra can bring in program development and verification. In future chapters, similar techniques based on the rely-guarantee method will be used to investigate algebras for concurrent programs.

Chapter 6

Algebra for Rely-Guarantee part 1

6.1 Introduction

To make the rely-guarantee method applicable to concrete program development and verification tasks, its integration into tools is essential. To capture the flexibility of the method, a number of features appear desirable. First, solid, denotational or operational, mathematical models for fine-grained program behaviour must be implemented. Second, one would like an abstract layer at which inference rules and refinement laws can be derived easily. Third, a high degree of proof automation is mandatory for the analysis of complex, concrete programs.

This Chapter presents an approach for providing such a tool integration in the interactive theorem proving environment Isabelle/HOL, following the general principles outlined in Section 1.2. At the most abstract level, algebras are used to reason about the control flow of programs as well as for deriving inference rules and refinement laws. In the context of rely-guarantee these axioms give a concise account of what the rely-guarantee method actually entails, and provide insights into the conceptual and operational role of interference constraints. Such structural insights are a main contribution of this approach. At the most concrete level, detailed models of program stores can support fine-grained reasoning about program data flow and interference. These models are then linked with the algebras via soundness proofs. Isabelle allows these layers to be implemented in a modular way and relate them formally with one another. This provides a high degree of confidence in the correctness of this development, and also supports the construction of custom proof tactics and procedures for program verification and refinement tasks. By virtue of being implemented in Isabelle, the entire implementation is guaranteed to be correct by construction.

In this Chapter algebraic principles for rely-guarantee style reasoning are examined. Starting from [HMSW11], a basic minimal set of axioms for rely and guarantee conditions, which suffice to derive the standard rely-guarantee inference rules, is derived. However, algebra by its nature is inherently compositional (see Section 6.3 for details), so it turns out that naïve forms of these axioms do not fully capture the semantics of interference in execution traces. It is therefore necessary to explore how the compositionality of these axioms can be broken in the right way, so as to capture the intended trace semantics.

Second, a simple trace based semantics (which so far is restricted to finite executions and disregards termination and synchronisation) is linked to these algebras by formal soundness proof. Despite the simplicity of this model, a prototypical verification tool within Isabelle can be demonstrated by verifying a simple example from the literature. Beyond that, this approach provides a coherent framework from which more complex and detailed models and algebras can be implemented in future chapters, especially Chapter 7.

Third, the usual inference rules of the rely-guarantee method with the

exception of assignment axioms, are derived directly from the algebra, and the assignment axioms are derived from the models. This formalisation in Isabelle allows one to reason seamlessly across these layers, which capture both the control flow and the data flow of concurrent programs respectively.

This chapter is primarily based upon material appearing in

- A. Armstrong, V. B. F. Gomes and G. Struth. *Algebraic Principles for Rely-Guarantee Style Concurrency Verification Tools.* In C. Jones, P. Pihlajasaari and J. Sun (eds.), FM 2014, LNCS 8442. [AGS14a]

6.2 A Rely-Guarantee Algebra

First it is shown that bi-Kleene algebras (Section 3.3) can be expanded into a simple algebra that supports the derivation of rely-guarantee style inference rules. This development does *not* use the interchange law of concurrent Kleene algebra for several reasons. First, this law fails for fair parallel composition $x \parallel_f y$ in models with possibly infinite, or non-terminating programs. In this model, $x \cdot y \not\leq x \parallel_f y$ whenever x is non-terminating. Secondly, and perhaps most relevantly, it is not needed for deriving the usual rules of rely-guarantee.

Definition 1. A rely-guarantee algebra is a structure

$$(K, I, +, \sqcap, \cdot, \|, ^\star, 0, 1),$$

where

- $(K, +, \sqcap)$ is a distributive lattice,
- $(K, +, \cdot, \|, 0, 1)$ is a trioid,
- $(K, +, \cdot, 0, 1, ^\star)$ is a Kleene algebra.
- I is a distinguished subset of rely and guarantee conditions or *interference constraints* that satisfy the following axioms:

$$r\|r \leq r, \tag{6.1}$$
$$r \leq r\|r', \tag{6.2}$$
$$r\|(x \cdot y) = (r\|x) \cdot (r\|y), \tag{6.3}$$
$$r\|x^+ \leq (r\|x)^+. \tag{6.4}$$

By convention, r and g refer to elements of I, depending on whether they are used as relies or guarantees, and x, y, z for arbitrary elements of K. The operations $\|$ and $\sqcap$ must be closed with respect to I.

The general idea here is to constrain a program by a rely condition by executing the two in parallel. Axiom (6.1) states that interference from a

constraint being run twice in parallel is no different from just the interference from that constraint begin run once in parallel. Axiom (6.2) states that interference from a single constraint is less than interference from itself and another interference constraint. Axiom (6.3) allows an interference constraint to be split across sequential programs. Axiom (6.4) is similar to Axiom (6.3) in intent, except it deals with finite iteration.

Some elementary consequences of these rules are

$$1 \leq r, \qquad r^\star = r \cdot r = r = r \| r, \qquad r \| x^+ = (r \| x)^+.$$

Theorem 5. *Axioms (6.1), (6.2) and (6.3) are independent.*

Proof. Isabelle's *Nitpick* [BN10] counterexample generator can be used to construct models which violate each particular axiom while satisfying all others. □

Theorem 6. *Axiom (6.3) implies (6.4) in a quantale where* $\|$ *distributes over arbitrary suprema.*

Proof. In a quantale, x^+ can be defined as a sum of powers $x^+ = \sum_{i \geq 1} x^i$ where $x^1 = x$ and $x^{i+1} = x \cdot x^i$. By induction on i we get $r \| x^i = (r \| x)^i$, hence

$$r \| x^+ = r \| \sum_{i \geq 1} x^i = \sum_{i \geq 1} r \| x^i = \sum_{i \geq 1} (r \| x)^i = (r \| x)^+.$$

□

In first-order Kleene algebras (6.3) and (6.4) are independent, but it is impossible to find a counterexample with Nitpick because it generates only finite counterexamples, and all finite Kleene algebras are a fortiori quantales.

Jones quintuples can be encoded in this setting as

$$r, g \vdash \{p\} x \{q\} \iff p \cdot (r \| x) \leq q \wedge x \leq g. \tag{6.5}$$

This means that program x when constrained by a rely r, and executed after p, behaves as q. Moreover, all behaviours of x are included in its guarantee g. This encoding is due to [HMSW11]. The key difference is that the interference constraints are only required to satisfy axioms (6.1)–(6.4), rather than be power invariants (see Section 2.3). Axioms (6.1)–(6.4) are all implied by the stronger requirements for power invariants [HMSW11]. Note that this encoding is stronger than in traditional rely-guarantee, as x is required to unconditionally implement g. This limitation of the encoding is circumvented later, in Chapter 7. Although, note that the algebra here could easily be extended with some additional operator f such that $f(r, x) \leq q$ would encode that x implements q only under interference of at most r. For more complex examples than those presented in Section 6.6 such an encoding proves useful.

Theorem 7. *The standard rely-guarantee inference rules shown in Figure 6.1 can be derived with the above encoding (6.5).*

Proof. This statement has been formalised in Isabelle, so a fully detailed proof is not given here. However, to demonstrate the algebraic style of reasoning involved, the derivation of the parallel rule is given.

For the parallel rule, we have the assumptions

$$p_1(r_1 \| x) \leq q_1, \quad p_2(r_2 \| x) \leq q_2, \quad x \leq g_1, \quad y \leq g_2 \quad g_1 \leq r_2, \quad \text{and} \quad g_2 \leq r_1.$$

It must be shown that

$$(p_1 \sqcap p_2)((r_1 \sqcap r_2) \| x \| y) \leq q_1 \sqcap q_2, \qquad \text{and} \qquad x \| y \leq g_1 \| g_2.$$

Showing $x \| y \leq g_1 \| g_2$ is trivial due to isotonicity. Next we show that:

$$\begin{aligned}(p_1 \sqcap p_2)((r_1 \sqcap r_2) \| x \| y) &\leq p_1(r_1 \| x \| y) \\ &\leq p_1(r_1 \| x \| g_2) \\ &\leq p_1(r_1 \| x \| r_1) \\ &\leq p_1(r_1 \| x) \\ &\leq q_1\end{aligned}$$

The same reasoning can be applied to show that

$$(p_1 \sqcap p_2)((r_1 \sqcap r_2) \| x \| y) \leq q_2,$$

which completes the proof of the parallel rule. □

Based on Theorem 5 and Theorem 6, (6.1) to (6.4), which are all necessary to derive these rules, represent a minimal set of axioms from which these inference rules can be derived.

If residuals are added to the rely-guarantee algebra quintuples can be encoded in the following way, which is equivalent to the encoding in Equation (6.5).

$$r, g \vdash \{p\}x\{q\} \iff x \leq r/(p \to q) \sqcap g. \tag{6.6}$$

which can be shown via properties of residuals as follows:

$$p(r \| x) \leq q \wedge x \leq g \iff (r \| x \leq p \to q) \wedge x \leq g \iff x \leq r/(p \to q) \sqcap g.$$

This encoding allows for thinking in terms of program refinement, as in [HJC13], since $r/(p \to q) \sqcap g$ defines the weakest program that when placed in parallel with interference from r, and guaranteeing interference at most g, goes from p to q—essentially a generic specification for a concurrent program.

$$\frac{p \cdot r \leq p}{r, g \vdash \{p\}1\{p\}}\ \text{Skip}$$

$$\frac{r' \leq r \qquad g \leq g' \qquad p \leq p' \qquad r', g' \vdash \{p'\}x\{q'\} \qquad q' \leq q}{r, g \vdash \{p\}x\{q\}}\ \text{Weakening}$$

$$\frac{r, g \vdash \{p\}x\{q\} \qquad r, g \vdash \{q\}y\{s\}}{r, g \vdash \{q\}x \cdot y\{s\}}\ \text{Sequential}$$

$$\frac{r_1, g_1 \vdash \{p_1\}x\{q_1\} \qquad g_1 \leq r_2 \qquad r_2, g_2 \vdash \{p_2\}y\{q_2\} \qquad g_2 \leq r_1}{r_1 \sqcap r_2, g_1 \| g_2 \vdash \{p_1 \sqcap p_2\}x \| y\{q_1 \sqcap q_2\}}\ \text{Parallel}$$

$$\frac{r, g \vdash \{p\}x\{q\} \qquad r, g \vdash \{p\}y\{q\}}{r, g \vdash \{p\}x + y\{q\}}\ \text{Choice}$$

$$\frac{p \cdot r \leq p \qquad r, g \vdash \{p\}x\{p\}}{r, g \vdash \{p\}x^\star\{p\}}\ \text{Star}$$

Figure 6.1: Rely-guarantee inference rules

6.3 Breaking Compositionality

While the algebra in the previous section is adequate for deriving the standard inference rules, its equality is too strong to capture many interesting statements about concurrent programs. Consider the following congruence rule for parallel composition, which is inherent in the algebraic approach in Section 6.2:

$$x = y \implies r \| x = r \| y.$$

This can be read as follows; if x and y are equal, then they must be equal under all possible interference from any arbitrary r. In general, all the operators in any algebra must satisfy such congruence laws. At first, this might seem to preclude any fine-grained reasoning about interference using purely algebraic approaches. This is not the case, but breaking this inherent compositionality in just the right way to capture interesting properties of interference requires extra work. Consider the case where x and y behave the same under no interference; we need a way to state that they are equal in this case without requiring that they are equivalent under arbitrary (or any) interference. More concretely, x and y might perform the same task, yet x performs it atomically, whereas y does not. In such a case, y should be much more vulnerable to interference from its environment (r above).

A way of achieving this is to expand the rely-guarantee algebra from Section 6.2 with an additional function $\pi : K \to K$ and redefining our quintuples as,

$$r, g \vdash \{p\}x\{q\} \iff p \cdot (r \| c) \leq_\pi q \wedge x \leq g. \tag{6.7}$$

Where $x \leq_\pi y$ is $\pi(x) \leq \pi(y)$. Since for any operator $\bullet$ it is not required that

$$\pi(x) = \pi(y) \implies \pi(x \bullet z) = \pi(y \bullet z),$$

compositionality can be broken in just the right way, provided that appropriate properties for π are chosen. These properties are extracted from properties of the trace model, which will be explained in detail in the coming sections. Many of these properties however, can be derived from the simple fact that, in the trace model, $\pi = \lambda x.\ x \sqcap c$, where c is healthiness condition filtering out ill-defined traces. The most important such properties are:

$$\pi(x) \leq x, \tag{6.8}$$

$$\pi(\pi(x)) = \pi(x), \tag{6.9}$$

$$x \leq y \implies x \leq_\pi y. \tag{6.10}$$

In addition to these properties, π must satisfy the following:

$$x^\star \leq_\pi \pi(x)^\star, \tag{6.11}$$

$$x \cdot y \leq_\pi \pi(x) \cdot \pi(y), \tag{6.12}$$

$$z + x \cdot y \leq_\pi y \implies x^\star \cdot z \leq_\pi y, \tag{6.13}$$

$$z + y \cdot x \leq_\pi y \implies z \cdot x^\star \leq_\pi y. \tag{6.14}$$

For any operator $\bullet$, the operator $x \bullet_\pi y$ is the defined as $\pi(x \bullet y)$, and x^π is defined as $\pi(x^\star)$.

Theorem 8. *$(\pi(K), +_\pi, \cdot_\pi, ^\pi, 0, 1)$ is a Kleene algebra.*

Proof. The operator π is a retraction (6.9), that is, $\pi^2 = \pi$. Therefore, $x \in \pi(K)$ iff $\pi(x) = x$. This condition can then be used to check the closure conditions for all operations. This property has been formalised in Isabelle. □

Definition 2. We re-define the rely-guarantee algebra as a structure

$$(K, I, +, \sqcap, \cdot, \|, ^\star, \pi, 0, 1)$$

which, in addition to the rules in Section 6.2, satisfies (6.8) to (6.14) for π.

Theorem 9. *All rules in Figure 6.1 can be derived in this algebra. For each rule, the encoding in (6.7) is used for the quintuple, rather than the encoding in (6.5).*

Proof. Their proofs remain the same as in Theorem 7, mutatis mutandis. As an example, the main part of the derivation of the parallel rule from Theorem 7 becomes:

$$\begin{aligned}(p_1 \sqcap p_2)((r_1 \sqcap r_2)\|x\|y) &\leq_\pi p_1(r_1\|x\|y)\\ &\leq_\pi p_1(r_1\|x\|g_2)\\ &\leq_\pi p_1(r_1\|x\|r_1)\\ &\leq_\pi p_1(r_1\|x)\\ &\leq_\pi q_1\end{aligned}$$

□

6.4 Finite Language Model

Now a finite language model is constructed satisfying the axioms in Section 6.2 and 6.3. Restricting attention to finite languages means issues such termination side-conditions need not be considered, nor must additional restrictions on parallel composition, such as fairness. However, as will be seen in Chapter 7, all the results in this section can be adapted to potentially infinite languages, and this Isabelle/HOL formalisation includes general definitions by using coinductively defined lazy lists to represent words, and having a weakly-fair shuffle operator for such infinite languages.

Consider languages where the alphabet contains letters which are state pairs of the form $(\sigma_1, \sigma_2) \in \Sigma^2$. A word in such a language is *connected* if every such pair in a word has the same first state as the previous transition's second state. For example, $(\sigma_1, \sigma_2)(\sigma_2, \sigma_3)$ is connected, while $(\sigma_1, \sigma_2)(\sigma_3, \sigma_3)$ is connected only if $\sigma_2 = \sigma_3$. Sets of connected words are essentially Aczel traces [dBHdR99] lacking the usual process labels. Denote the set of all connected words by C. The function π from the previous section is defined as $\lambda X.\ X \cap C$.

Sequential composition in this model is language product, as per usual. Concurrent composition is the shuffle product defined in Section 3.2. As mentioned, the shuffle product is associative, commutative, and distributes over arbitrary joins. Both products share the same unit, $\{\epsilon\}$ and zero, $\emptyset$. In Isabelle proving properties of shuffle is surprisingly tricky (especially if one considers infinite words as will be seen in Chapter 7). For a in-depth treatment of the shuffle product over infinite words see Subsection 7.3.1 or [MMRS97].

Theorem 10. $(\mathcal{P}((\Sigma^2)^\star), \cup, \cdot, \|, \emptyset, \{\epsilon\})$ *forms a trioid.*

The interference constraints in this model are sets containing all the words which can be built from some set of state pairs in Σ^2. The function $\langle R \rangle$ lifts a relation R to a language containing words of length one for each pair in

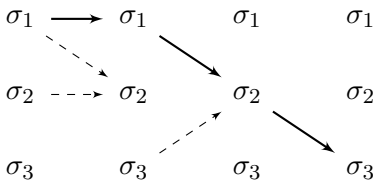
R. The set of interference constraints I is then defined as $\{r.\ \exists R.r = \langle R\rangle^\star\}$. This definition ensures that elements of I are power invariants in the sense of [HMSW11], which as noted in Section 6.2 is enough to prove (6.1)–(6.4).

Theorem 11. $(\mathcal{P}((\Sigma^2)^\star), I, \cup, \cdot, \|, ^\star, \pi, \emptyset, \{\epsilon\})$ *is a rely-guarantee algebra as defined in Definition 2.*

Since $\langle R\rangle$ is atomic, it satisfies several useful properties, such as,

$$\langle R\rangle^\star \| \langle S\rangle = \langle R\rangle^\star; \langle S\rangle; \langle R\rangle^\star, \qquad \langle R\rangle^\star \| \langle S\rangle^\star = (\langle R\rangle^\star; \langle S\rangle^\star)^\star.$$

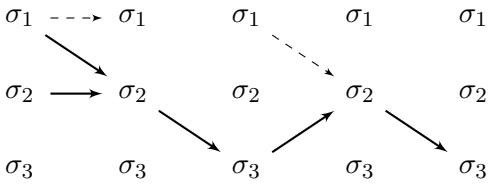
To demonstrate how this model works, consider the graphical representation of a language shown below.



The language contains the following six words

$$(\sigma_1,\sigma_1)(\sigma_1,\sigma_2)(\sigma_2,\sigma_3), \qquad (\sigma_1,\sigma_2)(\sigma_1,\sigma_2)(\sigma_2,\sigma_3),$$
$$(\sigma_2,\sigma_2)(\sigma_1,\sigma_2)(\sigma_2,\sigma_3), \qquad (\sigma_1,\sigma_1)(\sigma_3,\sigma_2)(\sigma_2,\sigma_3),$$
$$(\sigma_1,\sigma_2)(\sigma_3,\sigma_2)(\sigma_2,\sigma_3), \qquad (\sigma_2,\sigma_2)(\sigma_3,\sigma_2)(\sigma_2,\sigma_3),$$

where only the first, $(\sigma_1,\sigma_1)(\sigma_1,\sigma_2)(\sigma_2,\sigma_3)$ is consistent. This word is highlighted with solid arrows in the diagram above. Now if one shuffles the single state pair (σ_2,σ_3) into the above language, one ends up with a language containing all the words (and more not shown) represented in the diagram below:



By performing this shuffle action, there is no longer a connected word from σ_1 to σ_3, but instead there is a connected word from σ_2 to σ_3 and σ_1 to σ_3. These new connected words were constructed from previously unconnected words—the shuffle operator can generate many connected words from two unconnected words. If we only considered connected words, à la Aczel traces, it would be impossible to define such a shuffle operator directly on the traces themselves, and instead one would have to rely on some operational semantics to generate traces in the concurrent composition, or perform concurrent composition in some other way (see Section 2.3).

6.5 Enriching the Model

To model and verify programs, additional concepts such as tests and assignment axioms are required. A *test* is any language P where $P \leq \langle \mathsf{Id} \rangle$. Such a language is a restriction of the identity relation to P and is denoted by Id_P. The statement $\mathsf{test}(P)$ denotes $\langle \mathsf{Id}_P \rangle$. In Kleene algebra the sequential composition of two tests should be equal to their intersection. However, the traces $\mathsf{test}(P); \mathsf{test}(Q)$ and $\mathsf{test}(P \cap Q)$ are incomparable, as all words in the former have length two, while all the words in the latter have length one. This means that these tests at first seem rather different to those of Kozen's KAT. To overcome this problem, one can use the concepts of *stuttering* and *mumbling*, following [Bro93] and [Din02]. As introduced in Chapter 2 we inductively generate the *mumble language* $w^\dagger$ for a word w in a language over Σ^2 as follows: Assume $\sigma_1, \sigma_2, \sigma_3 \in \Sigma$ and $u, v, w \in (\Sigma^2)^\star$. First, $w \in w^\dagger$. Secondly, if $u(\sigma_1, \sigma_2)(\sigma_2, \sigma_3)v \in w^\dagger$ then $u(\sigma_1, \sigma_3)v \in w^\dagger$. This operation is lifted to languages in the obvious way as

$$X^\dagger = \bigcup \{x^\dagger .\ x \in X\}.$$

Stuttering is represented as a rely condition $\langle \mathsf{Id} \rangle^\star$ where Id is the identity relation. Two languages X and Y are equal under stuttering if

$$\langle \mathsf{Id} \rangle^\star \| X =_\pi \langle \mathsf{Id} \rangle^\star \| Y.$$

Assuming mumbling is applied to both sides of the following equation, it is the case that

$$\mathsf{test}(P \cap Q) \leq_\pi \mathsf{test}(P); \mathsf{test}(Q)$$

as the longer words in $\mathsf{test}(P); \mathsf{test}(Q)$ can be mumbled down into the shorter words of $\mathsf{test}(P \cap Q)$, whereas stuttering gives the opposite direction,

$$\langle \mathsf{Id} \rangle^\star \| (\mathsf{test}(P); \mathsf{test}(Q)) \leq_\pi \langle \mathsf{Id} \rangle^\star \| \mathsf{test}(P \cap Q).$$

For the remainder of this Chapter assume that all languages are implicitly mumble closed.

Using tests, if statements and while loops are encoded as

$$\begin{aligned} \mathsf{if}\ \ P\ \{\ X\ \}\ \mathsf{else}\ \{\ Y\ \} &= \mathsf{test}(P); X + \mathsf{test}(-P); Y, \\ \mathsf{while}\ \ P\ \{\ X\ \} &= (\mathsf{test}(P); X)^\star; \mathsf{test}(-P). \end{aligned}$$

Next, the operator $\mathsf{end}(P)$ is defined to contain all the words which end in a state satisfying P. Some useful properties of end include

$$\begin{aligned} \mathsf{end}(P); \mathsf{test}Q \leq_\pi \mathsf{end}(P \cap Q), \qquad \mathsf{test}(P) \leq \mathsf{end}(P), \\ \mathrm{range}(\mathsf{Id}_P \circ R) \leq P \implies \mathsf{end}(P); \langle R \rangle^\star \leq_\pi \mathsf{end}(P). \end{aligned}$$

In this model, assignment is defined as

$$x := e \;=\; \bigcup v.\, \mathsf{test}\{\sigma.\, \mathsf{eval}(\sigma, e) = v\} \cdot x \leftarrow v$$

where $x \leftarrow v$ denotes the atomic command which assigns the value v to x. This is very different to how assignment is encoded in SKAT, as this is purely defined within the model, whereas SKAT has syntactic axioms for handling assignment.

The eval function atomically evaluates an expression e in the state σ. Using this definition, the assignment rule

$$\begin{aligned}&\mathsf{unchanged}(\mathsf{vars}(e)) \cap \mathsf{preserves}(P) \cap \mathsf{preserves}(P[x/e]),\\ &\mathsf{unchanged}(-\{x\})\\ &\vdash \{\mathsf{end}(P)\}\ x := e\ \{\mathsf{end}(P[x/e])\}.\end{aligned}$$

can be derived.

The rely condition states the following: First, the environment is not allowed to modify any of the variables used when evaluating e, i.e. those variables must remain unchanged. Second, the environment must preserve the precondition. Third, the postcondition of the assignment statement is also preserved. In turn, the assignment statement itself guarantees at minimum that it leaves every variable other than x unchanged. In truth, depending on e, many more complex statements can be guaranteed based on the exact nature of the assignment. For example the assignment $x := x - 2$ can also guarantee that x will always decrease. The predicates $\mathsf{preserves}$ and $\mathsf{unchanged}$ are defined as

$$\begin{aligned}\mathsf{preserves}(P) &= \langle\{(\sigma, \sigma').\ P(\sigma) \implies P(\sigma')\}\rangle^\star,\\ \mathsf{unchanged}(X) &= \langle\{(\sigma, \sigma').\ \forall v \in X.\ \sigma(v) = \sigma'(v)\}\rangle^\star.\end{aligned}$$

Further predicates useful for defining interference constraints can be defined in a similar fashion. For example, one could define $\mathsf{increasing}$ and $\mathsf{decreasing}$, defined much as unchanged except they only requiring that variables increase or decrease, rather than stay the same.

6.6 Examples

To demonstrate how the parallel rule behaves, consider the following simple statement, which simply assigns two variables in parallel:

$$\begin{aligned}&\langle \mathsf{Id}\rangle^\star, \langle\top\rangle^\star \vdash \{\mathsf{end}(x = 2 \wedge y = 2 \wedge z = 5)\}\\ &\qquad x := x + 2 \parallel y := z\\ &\{\mathsf{end}(x = 4 \wedge y = 5 \wedge z = 5)\}.\end{aligned}$$

The environment $\langle \mathsf{Id} \rangle^\star$ is only giving us stuttering interference. Since this program is being considered in isolation, no guarantees need to be made about how it affects the environment. To apply the parallel rule from Figure 6.1, the interference constraints and pre/postcondition are weakened or strengthened as needed to fit the form of the parallel rule.

First, the rely condition is weakened to

$$\mathsf{unchanged}\{x\} \sqcap \mathsf{unchanged}\{y, z\}.$$

Second, the guarantee condition is strengthened to

$$\mathsf{unchanged}\{y, z\} \parallel \mathsf{unchanged}\{x\}.$$

When the parallel rule is applied, each assignment's rely will become the other assignment's guarantee. Finally, the precondition and postcondition is split into $\mathsf{end}(x = 2) \sqcap \mathsf{end}(y = 2 \wedge z = 5)$ and $\mathsf{end}(x = 4) \sqcap \mathsf{end}(y = 5 \wedge z = 5)$ respectively. Upon applying the parallel rule, two trivial goals are obtained:

$\langle \mathsf{unchanged}\{x\} \rangle^\star, \langle \mathsf{unchanged}\{y, z\} \rangle^\star \vdash \{\mathsf{end}(x = 2)\}\ x := x + 2\ \{\mathsf{end}(x = 4)\}$,
$\langle \mathsf{unchanged}\{y, z\} \rangle^\star, \langle \mathsf{unchanged}\{x\} \rangle^\star \vdash \{\mathsf{end}(y = 2 \wedge z = 5)\}$
$\quad y := z$
$\{\mathsf{end}(y = 5 \wedge z = 5)\}$.

These goals can easily be tackled with the assignment rule from Section 6.5.

Figure 6.2 shows the FINDP program, which has been used by numerous authors e.g. [Owi75, Jon81, dRdBH$^+$01a, HJC13]. The program finds the least element of an array satisfying a predicate P. The index of the first element satisfying p is placed in the variable *fmin*. If no element of the array satisfies P, then f will be set to the length of the array. The program has two subprograms, A and B, running in parallel, one of which searches the even indices while the other searches the odd indices. A speedup over a sequential implementation is achieved as A will terminate when B finds an element of the array satisfying P which is less than i_A.

This FINDP program can be verified using just the algebra and model described previously in this chapter. More details are included in the paper upon which this chapter is based [AGS14a]. A proof of this fact is not included here as a complete (refinement) proof in Isabelle of the correctness of this algorithm is given in Chapter 8, albeit using the refined algebra from Chapter 7.

6.7 Conclusion

In this Chapter variants of semirings and Kleene algebras intended to model rely-guarantee and interference based reasoning have been introduced. A

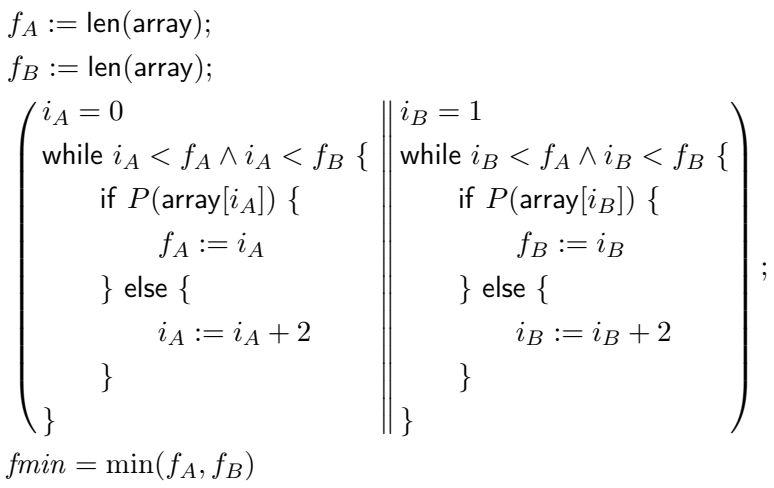


Figure 6.2: FINDP Program

simple, finite, interleaving model for these algebras using familiar concepts from traces and language theory has been given. This theory has been implemented in Isabelle/HOL, providing a solid mathematical basis on which to build a prototype tool for mechanised refinement and verification tasks. This implementation serves as a basis from which further interesting aspects of concurrent programs, such as refinement based proofs, and non-termination can be explored in future chapters.

Algebra clearly plays a very important role in this development. First, it allowed us the inference rules of rely-guarantee to be derived very rapidly and with little proof effort. The algebra provides simple and minimal laws from which a rely-guarantee calculus can be derived using the quintuple encoding found in [HMSW11]. Second, it yielded an abstract layer at which many properties that would be difficult to prove in concrete models can be verified with relative ease by simple equational reasoning. Third, as pointed out in Chapter 3, some fragments of the algebras considered are even decidable. Therefore, decision procedures for some aspects of rely-guarantee reasoning can be implemented in Isabelle. For example, the $\star$-continuous Kleene algebra decision procedure from Section 3.11 can be used, as the language model forms a quantale and therefore a $\star$-continuous Kleene algebra. However, due to the presence of parallel composition and additional constructs such as π it is doubtful that decision procedures would be very useful.

In the coming Chapters, the algebraic approach presented here will be refined to allow much more fine grained reasoning about interference, in contrast to the rather all or nothing approach given by the π operator.

Chapter 7

Algebra for Rely-Guarantee part 2

7.1 Introduction

In Chapter 6 a simple algebra was considered whereby an operator $\pi(x)$ was utilised to enable the algebra to encode non-trivial statements about concurrent programs. In the finite language model considered previously, this corresponded to restricting sets of execution traces to only the consistent traces; those where the environment always skips (i.e. does nothing). In this chapter, this approach is enriched by an operator that lets us describe the behaviour of a program's environment in a more fine grained way. This operator allows the definition of a concurrency rule that in some sense captures the true essence of the rely-guarantee method more faithfully, specifically that the guarantee condition holding should depend on the rely condition holding. This link between the rely and the guarantee is absent in Chapter 6. In addition, the finite language model is replaced by a detailed infinite language based model. The key part of the soundness proof for the rely-guarantee algebra is given in full detail, despite having also been mechanised in Isabelle.

7.2 A Refined Algebra for Rely-Guarantee

It is no longer assumed that the interference constraints r,g and the processes x,y are elements of the same algebra—in this richer setting, attempting to conflate both interference constraints and programs into a one-sorted algebra becomes unwieldy. In the model, we would like interference constraints to be relations and for programs to be traces. Interference constraints are taken to be elements of a complete lattice $(I, \sqsubseteq)$, whereas programs/processes are considered to be elements of the following algebra:

Definition 3. The program algebra is an algebra

$$(K, +, \sqcap, \cdot, \rightarrow, \|, ^\star, ^\omega, 0, 1),$$

where:

- $(K, +, \sqcap)$ is a distributive lattice (Section 3.5),
- $(K, +, \cdot, \|, 0, 1)$ is trioid (Section 3.2),
- $(K, +, \cdot, 0, 1, ^\star, ^\omega)$ is a weak ω-algebra (Section 3.12),
- The preimplication operator $\rightarrow$ satisfies the Galois connection

$$xy \leq z \iff y \leq x \rightarrow z. \tag{7.1}$$

Consider two operators $\rhd$ and : of type $I \to K \to K$, with the following intuitive meanings: The process $r \rhd x$ behaves as x as long as its environment satisfies the invariant r. The process $r : x$ is a restriction of the process x to just those executions where all environment steps satisfy the invariant r. The operator $\pi(x)$ from Chapter 6 is therefore intended to be equivalent to $\bot : x$ where $\bot$ is the least interference constraint (no interference). Naïvely, one might expect that these operators should be adjoints of a Galois connection

$$r : x \leq y \iff x \leq r \rhd y. \tag{7.2}$$

However this is not necessarily the case, as there exist models in which this Galois connection does not hold, including the model considered later in this Chapter (Section 7.3). Note however, that for this model (7.2) does in fact hold when considering both stutter-mumble and prefix closed traces (Subsection 7.3.3). In models where (7.2) does not hold, one typically has only the left implication

$$x \leq r \rhd y \implies r : x \leq y. \tag{7.3}$$

A *closure*, or *exterior* operator is an isotone idempotent function f such that $x \leq f(x)$ (extensiveness). Conversely, a *co-closure*, or *interior* operator f is an isotone and idempotent function satisfying the co-extensiveness property $f(x) \leq x$. It is required that $r \rhd -$ is a closure operator. The upper adjoint in every Galois connection is a closure operator, and indeed, every closure operator arises in this way from a suitable Galois connection. This implies that there exists a subset P of K where (7.2) holds provided $y \in P$. For the traces model later in this chapter we will see that P corresponds to the prefix and stutter-mumble closed traces.

It turns out that the : operator is unnecessary as we can state all the laws we require as a property of $\rhd$. Furthermore the Jones quintuple can be stated purely in terms of $\rhd$. Additionally the $\rhd$ operator leads to a natural refinement calculus for rely-guarantee.

While it is no longer the case that programs and interference constraints are both elements of the same algebra, the notion that interference constraints correspond to a subset of programs is preserved with the operator $\langle - \rangle : I \to K$ which lifts interference constraints to programs. Intuitively, $\langle g \rangle$ is the greatest program satisfying the constraint g.

Definition 4. An interference algebra can be now defined as a two-sorted structure

$$(K, I, \rhd, \langle - \rangle),$$

where:

- K is the algebra from Definition 3.

- I is a complete lattice $(I, \sqsubseteq)$. Join and meet in this lattice are denoted by $\sqcup$ and $\sqcap$ respectively. The top and bottom elements are $\top$ and $\bot$ respectively.
- $\rhd$ is a closure operator
- The $\rhd$ and $\langle - \rangle$ operators satisfy:

$$(r \rhd x)(r \rhd y) \leq r \rhd xy, \tag{7.4}$$
$$(r \rhd x)^\star \leq r \rhd x^\star, \tag{7.5}$$
$$r \leq g \implies g \rhd x \leq r \rhd x, \tag{7.6}$$
$$\top \rhd x = x, \tag{7.7}$$
$$\langle g_1 \rangle \| \langle g_2 \rangle \leq \langle g_1 \sqcup g_2 \rangle, \tag{7.8}$$
$$\langle g_1 \rangle \langle g_2 \rangle \leq \langle g_1 \sqcup g_2 \rangle, \tag{7.9}$$
$$\langle g \rangle^\star = \langle g \rangle, \tag{7.10}$$
$$(x \sqcap \langle g \rangle)(y \sqcap \langle g \rangle) = xy \sqcap \langle g \rangle, \tag{7.11}$$
$$(r \sqcup g_2 \rhd x_1 \sqcap \langle g_1 \rangle) \parallel (r \sqcup g_1 \rhd x_2 \sqcap \langle g_2 \rangle) \leq r \rhd (x_1 \sqcap \langle g_1 \rangle) \| (x_2 \sqcap \langle g_2 \rangle). \tag{7.12}$$

Axiom (7.4) and (7.5) give the behaviour of $\rhd$ w.r.t. sequential composition; namely, $\rhd$ is required to sub-distribute over sequential composition, as in the rely-guarantee algebra from Chapter 6. This axiom is visualised graphically in Figure 7.1. Such diagrams are intended to provide a intuitive feel for how the axioms work over just the definitions themselves—for the seqential composition rules this may be unnecessary, but it serves to introduce the notation for the more interesting parallel case. Note that in a quantale (7.5) could be derived from (7.4), but this is not possible here. Axiom (7.6) states that $\rhd$ is antitone in its first argument. Intuitively, $\bot \rhd x$ must be greater than $\top \rhd x$ as $\bot \rhd x$ is a program that only behaves as x when its environment does nothing (stutters), otherwise it can do anything. On the other hand $\top \rhd x$ is always equal to x (7.7)—it behaves as x, no matter how much interference it receives from the environment. Axioms (7.8) to (7.11) describe how $\langle - \rangle$ interacts with the program composition operators. Essentially they state the following: that the guarantee of both sequential and parallel composition is the combination of the guarantee for both subprograms, that guarantee is invariant under the star, and that intersection by a guarantee distributes over sequential composition. Axiom (7.11) lets us distribute guarantees over sequential composition, similarly to how $\rhd$ sub-distributes as per (7.4). This axiom is shown in Figure 7.2.

The final axiom (7.12) describes how concurrent composition interacts with the $\rhd$ operator. This axiom is considerably more complex than those shown previously, and requires a more in depth explanation. Breaking the

$$(r \rhd x)(r \rhd y) \leq r \rhd xy$$

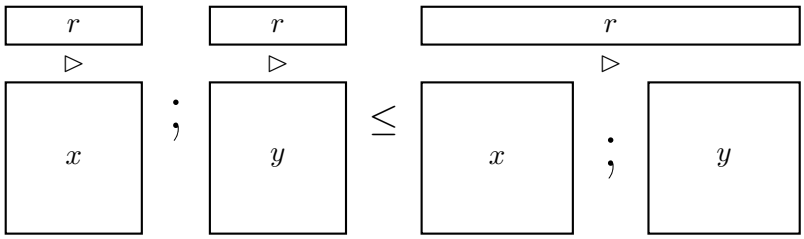


Figure 7.1: Diagrammatic depiction of axiom (7.4)

$$(x \sqcap \langle g \rangle)(y \sqcap \langle g \rangle) = xy \sqcap \langle g \rangle$$

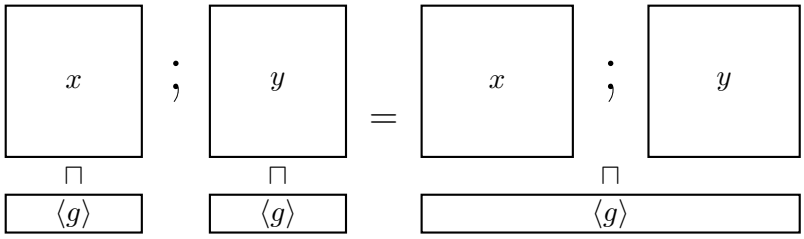


Figure 7.2: Diagrammatic depiction of axiom (7.11)

$$(r \sqcup g_2 \rhd x \sqcap \langle g_1 \rangle) \| (r \sqcup g_1 \rhd y \sqcap \langle g_2 \rangle) \leq r \rhd (x \sqcap \langle g_1 \rangle) \| (y \sqcap \langle g_2 \rangle)$$

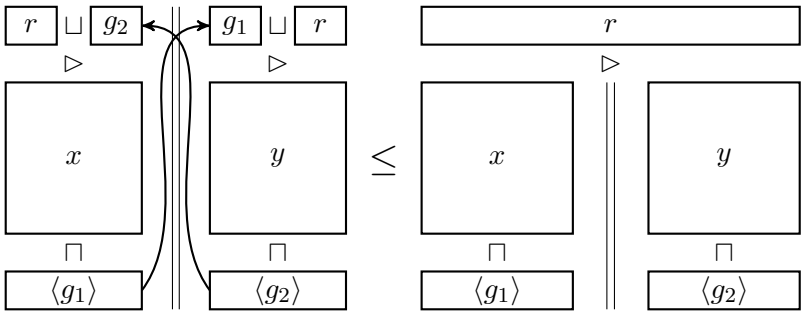


Figure 7.3: Diagrammatic depiction of axiom (7.12)

above down, $x_1 \sqcap \langle g_1 \rangle$ is describing a program x_1 in which every program transition satisfies the guarantee g_1. However, this only holds true as long as each environment transition satisfies either r or g_2. The guarantee g_2 is coming from the other side of the parallel composition, where we have a program that behaves as $x_2 \sqcap \langle g_2 \rangle$ as long as its environment behaves as r or g_1. The entire parallel composition can be seen as being run in an external environment which guarantees that r holds. This rule is intended to capture the circularity inherent in the rely-guarantee method—each of the two programs is relying on the other program, which in turn is relying on the first program, and so on. Figure 7.3 gives a diagrammatic presentation of the above—as can be seen the guarantees for each program becomes part of the rely for the other, and the global rely condition is pushed within both sides of the parallel composition.

In this setting the definition of the rely-guarantee quintuple is

$$r, g \vdash \{p\}x\{q\} \iff x \leq p \to r \rhd q \sqcap \langle g \rangle. \tag{7.13}$$

Note that precedences are such that $p \to r \rhd x \sqcap y$ is $p \to (r \rhd (x \sqcap y))$. The right-hand side of the inequality, $p \to r \rhd q \sqcap \langle g \rangle$ describes all processes starting from a state p which when run in an environment satisfying the rely r, must implement the program q, which in turn guarantees that it does not execute steps outside its guarantee g. Note the difference between $(r \rhd q) \sqcap \langle g \rangle$ and $r \rhd q \sqcap \langle g \rangle$. The first guarantees g irrespective of the environment r, while the second guarantees g only if the environment satisfies r.

Compare this with the specification statement from [HJC13] discussed in Section 2.3

$$(\textbf{guar}\ g \bullet (\textbf{rely}\ r \bullet [p, q])) \sqsubseteq x.$$

In the notation used here, this would roughly be $x \leq \langle g \rangle \sqcap (r \rhd (p \to q))$. While $r \rhd x$ has essentially the same intuitive meaning as $(\textbf{rely}\ r \bullet x)$, as does $(\textbf{guar}\ g \bullet x)$ and $x \sqcap \langle g \rangle$, the specification statement here must be in a completely different order, as above in 7.13.

The order used by Hayes et al. does not make sense as a specification statement for a generic concurrent program in the interference algebra from Definition 4 (obviously this is not to say that it is incorrect within Hayes paper). Reasons for this are (partly) as follows: First, as previously mentioned, it is not necessarily the case that $r \rhd (x \sqcap \langle g \rangle)$ is the same as $(r \rhd x) \sqcap \langle g \rangle$. Second $p \to (r \rhd q)$ is distinct from $r \rhd (p \to q)$ in the following sense: $r \rhd (p \to q)$ ignores the possible environment transition between the precondition p and the program that tries to implement q, while $p \to (r \rhd q)$ does not. The specification statement/quintuple (6.6) in Chapter 6 is closer to that of Hayes et al.

7.2.1 Tests in the Algebra

To keep the algebraic side as simple as possible, no particular axioms are assumed for tests. In general, when we need a property we expect to hold for tests in some model, it is simply assumed it holds as an assumption when deriving proof rules dependent on those properties. As an example, when deriving the parallel rule for rely-guarantee, we need the following property of tests: $(p \to x) \| (p \to y) \leq p \to x \| y$ (where p is a test), but we simply take that as a premise of that particular rely-guarantee rule in the algebra. Note also that while p is used in the above rule much like an atomic test in KAT, postconditions q are oft intended to represent programs ending in states satisfying q—this is similar in intent to the Tarlecki triples [Tar85] $px \leq q$ discussed in Chapter 3.

Theorem 12. *Assuming the tests p_1, p_2, q_1 and q_2 satisfy*

$$(p_1 \to x) \| (p_2 \to y) \leq (p_1 \sqcap p_2) \to (x \| y), \quad \textit{and} \quad q_1 \| q_2 \leq r \rhd q_1 \sqcap q_2,$$

the standard rely-guarantee concurrency rule

$$\frac{r \sqcup g_1 \leq r_2 \quad r \sqcup g_2 \leq r_1 \quad g_1 \sqcup g_2 \leq g \quad r_1, g_1 \vdash \{p_1\} x \{q_1\} \quad r_2, g_2 \vdash \{p_2\} y \{q_2\}}{r, g \vdash \{p_1 \sqcap p_2\} x \| y \{q_1 \sqcap q_2\}}$$

can be derived in this algebra.

Proof. Translated via (7.13) the assumptions

$$r_1, g_1 \vdash \{p_1\} x \{q_1\} \qquad \text{and} \qquad r_2, g_2 \vdash \{p_2\} y \{q_2\}$$

become

$$x \leq p_1 \to r_1 \rhd \langle g_1 \rangle \sqcap q_1 \qquad \text{and} \qquad y \leq p_2 \to r_2 \rhd \langle g_2 \rangle \sqcap q_2$$

respectively. Along with Axiom (7.12) these can be used to derive the consequence of the rely-guarantee parallel rule via straightforward inequational reasoning.

$$\begin{aligned}
(x \| y) &\leq (p_1 \to r_1 \rhd q_1 \sqcap \langle g_1 \rangle) \| (p_2 \to r_2 \rhd q_2 \sqcap \langle g_2 \rangle) && \text{(assumptions)} \\
&\leq p_1 \sqcap p_2 \to (r_1 \rhd q_1 \sqcap \langle g_1 \rangle) \| (r_2 \rhd q_2 \sqcap \langle g_2 \rangle) && \text{(properties of tests)} \\
&\leq p_1 \sqcap p_2 \to (r \sqcup g_2 \rhd q_1 \sqcap \langle g_1 \rangle) \| (r \sqcup g_1 \rhd q_2 \sqcap \langle g_2 \rangle) && \text{(assumptions)} \\
&\leq p_1 \sqcap p_2 \to (q_1 \sqcap \langle g_1 \rangle) \| (q_2 \sqcap \langle g_2 \rangle) && \text{(7.12)} \\
&\leq p_1 \sqcap p_2 \to r \rhd (q_1 \| q_2) \sqcap (\langle g_1 \rangle \| \langle g2 \rangle) && \text{(properties of } \sqcap) \\
&\leq p_1 \sqcap p_2 \to r \rhd (q_1 \sqcap q_2) \sqcap \langle g_1 \sqcup g_2 \rangle && \text{(7.10, properties of tests)}
\end{aligned}$$

□

As can be seen, the derivation of the standard parallel composition rule from this axiom in the algebra is straightforward—this shouldn't be too surprising, as Axiom (7.12) is intended to capture the key complexity inherent in this rule, minus any properties of tests. The difficulty of course lies in proving that Axiom (7.12) is valid for any particular model. In the following Section a model for this algebra is given based on infinite traces, and Axiom (7.12) is derived within it.

Theorem 13. *All the other rules from Figure 7.4 can be derived in the interference algebra.*

Proof. All the other rules are straightforward to derive using simple inequational reasoning. For full details see Appendix A, or the Isabelle implementation. □

$$\frac{}{r, g \vdash \{p\}1\{p\}}\ \text{Skip}$$

$$\frac{r' \leq r \qquad g \leq g' \qquad p \leq p' \qquad r', g' \vdash \{p'\}x\{q'\} \qquad q' \leq q}{r, g \vdash \{p\}x\{q\}}\ \text{Weakening}$$

$$\frac{r, g \vdash \{p\}x\{q\} \qquad r, g \vdash \{q\}y\{s\}}{r, g \vdash \{p\}x \cdot y\{s\}}\ \text{Sequential}$$

$$\frac{r_1, g_1 \vdash \{p_1\}x\{q_1\} \qquad g_1 \leq r_2 \qquad r_2, g_2 \vdash \{p_2\}y\{q_2\} \qquad g_2 \leq r_1}{r_1 \sqcap r_2, g_1 \sqcup g_2 \vdash \{p_1 \sqcap p_2\}x \| y\{q_1 \sqcap q_2\}}\ \text{Parallel}$$

$$\frac{r, g \vdash \{p\}x\{q\} \qquad r, g \vdash \{p\}y\{q\}}{r, g \vdash \{p\}x + y\{q\}}\ \text{Choice}$$

$$\frac{r, g \vdash \{p\}x\{p\}}{r, g \vdash \{p\}x^\star\{p\}}\ \text{Star}$$

$$\frac{r, g \vdash \{p\}x\{p\} \qquad r, g \vdash \{p\}x^\omega\{x^\star\}}{r, g \vdash \{p\}x^\omega\{p\}}\ \text{Omega}$$

Figure 7.4: Updated rely-guarantee inference rules

7.2.2 Refinement

The definition of the rely-guarantee quintuple in (7.13) lends itself naturally to thinking in terms of *refinement*, whereby programs are derived

$$p \to r \rhd p \sqcap \langle g \rangle \sqsubseteq 1 \qquad \text{(Skip)}$$

$$p \to r \rhd q \sqcap \langle g \rangle \sqsubseteq (p \to r \rhd s \sqcap \langle g \rangle)(s \to r \rhd q \sqcap \langle g \rangle) \qquad \text{(Sequential)}$$

$$\begin{array}{c} (p_1 \sqcap p_2) \to r \rhd (q_1 \sqcap q_2) \sqcap \langle g_1 \sqcup g_2 \rangle \\ \sqsubseteq \\ (p_1 \to r \sqcup g_2 \rhd q_1 \sqcap \langle g_1 \rangle) \| (p_2 \to r \sqcup g_1 \rhd q_2 \sqcap \langle g_2 \rangle) \end{array} \qquad \text{(Parallel)}$$

$$p \to r \rhd p \sqcap \langle g \rangle \sqsubseteq (p \to r \rhd p \sqcap \langle g \rangle)^\star \qquad \text{(Star)}$$

Figure 7.5: Rely-guarantee refinement rules

incrementally from a specification. The right-hand side of the quintuple $p \to (r \rhd q \sqcap \langle g \rangle)$ can be seen as the greatest program from p to q, that guarantees g in an environment r. This is therefore a suitable specification statement for an arbitrary concurrent program. When thinking in terms of refinement, the order $\leq$ is usually flipped as $x \sqsubseteq y \iff y \leq x$. This order naturally allows reasoning from the specification to the program, as the specification will always be greater than the program. The order $\sqsubseteq$ is called the *refinement order*. The quintuple $r, g \vdash \{p\}x\{q\}$ therefore holds iff x is a refinement of $p \to r \rhd q \sqcap \langle g \rangle$.

Much as in the action algebra from Section 3.10, each of the inference rules in Figure 7.4 can be translated to an equivalent rely-guarantee refinement rule. For example, the sequential rule would become

$$p \to r \rhd q \sqcap \langle g \rangle \sqsubseteq (p \to r \rhd s \sqcap \langle g \rangle)(s \to r \rhd q \sqcap \langle g \rangle).$$

Other refinement rules are given in Figure 7.5.

In the following section, an infinite language model for the above algebra is given. It is perhaps easy to imagine a great variety of language/trace based models (with an interleaving parallel composition operator) applicable to the above algebra. However it remains to be seen if this algebra is more broadly applicable beyond those models traditionally considered in the rely-guarantee literature.

7.3 Infinite Language Model

In this section a model for rely-guarantee style reasoning based on (potentially) infinite languages is presented. Such a model is considerably more complicated than the finite language model presented in Chapter 6. In particular, many operations that can be defined inductively for finite traces must instead be defined co-inductively for infinite traces. Furthermore, issues such as *fairness* must now be considered for the parallel composition operator.

Traces are potentially infinite sequences of state transitions, for example,

$$(\sigma_0, \sigma_0')(\sigma_1, \sigma_1')(\sigma_2, \sigma_2')(\sigma_3, \sigma_3') \ldots$$

As before the transitions (σ_0, σ_0'), (σ_1, σ_1') etc are called program transitions, while transitions between program transitions such as $\sigma_0')(\sigma_1$ are called environment transitions. Potentially infinite languages are therefore sets containing such potentially infinite words, or equivalently they taken to be subsets of $(\Sigma^2)^\infty$.

Sequential combination of languages is defined as

$$XY = \{xy \mid x \in X \wedge y \in Y\} \cup \{x \mid x \in X \wedge \neg \mathsf{finite}(x)\}.$$

Preimplication is defined as in a quantale (Section 3.13), by

$$X \to Y = \bigcup \{Z \mid XZ \subseteq Y\}.$$

The iteration operators can then be defined as fixpoints

$$\begin{aligned} X^\omega &= \nu Y.XY, \\ X^\star &= \mu Y.\{\epsilon\} \cup XY, \\ X^\infty &= X^\omega \cup X^\star. \end{aligned}$$

Lemma 14. $(K, \cup, \cdot, ^\star, 0, 1)$ *is a weak Kleene algebra (Section 3.8).*

Lemma 15. $(K, \cup, \cdot, ^\star, ^\omega, 0, 1)$ *is a weak ω-algebra (Section 3.12).*

Denote the set of all finite traces as $\mathcal{F}$. For sequential composition we have infinite distributivity from the right, but only for finite languages from the left

$$(\bigcup \mathfrak{X})Y = \bigcup \{XY \mid X \in \mathfrak{X}\}, \qquad X \subseteq \mathcal{F} \implies X(\bigcup \mathfrak{Y}) = \bigcup \{XY \mid Y \in \mathfrak{Y}\}.$$

Lemma 16. $(K, \subseteq, \cdot)$ *is a weak quantale*

A language X is prefix closed iff $xy \in X$ implies $x \in X$. Unlike many trace based models for concurrent programs, here it is not assumed that sets of traces are prefix closed. This is primarily to keep the model as simple as possible.

For implementing this model in Isabelle, extensive use is made of Andreas Lochbihler et al's coinductive library [Loc10, BHL+14]. This library provides a type of coinductive lazy lists, 'a llist, which function much like the lists found in lazy functional programming languages such as Haskell. Since Isabelle features strict evaluation, defining functions over such lists is not as simple as it would be in Haskell, as will be seen in Subsection 7.3.1. All the properties and definitions in this section have been formalised in Isabelle, barring Lemma 34 in Subsection 7.3.4 and certain properties of stutter-mumble closure (see Section 7.3.2)

7.3.1 The Shuffle Operation

Define $x \sqcup\!\sqcup_t y$ as the *shuffle* of words x and y along a *trajectory* t. A trajectory is a potentially infinite word over the alphabet $\{l, r\}$. Figure 7.6 demonstrates how this works for two simple words x and y. The trajectory t determines in which orders the state pairs are selected from x and y in the final shuffle $x \sqcup\!\sqcup_t y$.

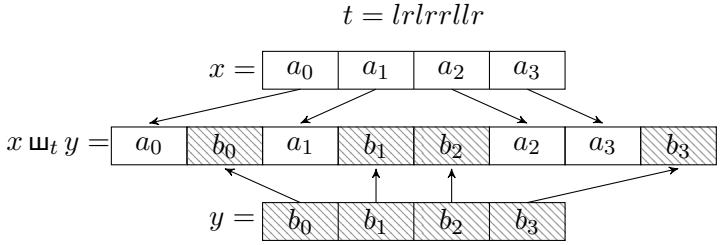


Figure 7.6: An example of a shuffle.

In Figure 7.6 the notation a_n and b_n is used to represent the state pairs (σ_n, σ'_n) and (τ_n, τ'_n) respectively. More formally, the shuffle of two languages can be defined as

$$
\begin{aligned}
x \sqcup\!\sqcup_t \epsilon &= x, \\
\epsilon \sqcup\!\sqcup_t y &= y, \\
x{:}x' \sqcup\!\sqcup_{t:t'} y{:}y' &= \begin{cases} x{:}(x' \sqcup\!\sqcup_{t'} y{:}y') & \text{if } t = l \\ y{:}(x{:}x' \sqcup\!\sqcup_{t'} y') & \text{if } t = r. \end{cases}
\end{aligned}
$$

The shuffle of two potentially infinite words cannot be defined so straightforwardly in Isabelle. As mentioned, Isabelle is a strictly-evaluated, rather than lazily-evaluated language, so the above definition would fail to terminate when both arguments are infinite. This function therefore needs to be defined coinductively in Isabelle as an unfold.

definition op ⧢ :: 'a llist → (unit + unit) llist → 'b llist → ('a + 'b) llist
  where $x \sqcup\!\sqcup_t y \equiv$ unfold-llist

$(\lambda(t,\ x,\ y).\ t = \text{LNil})$
$(\lambda(t,\ x,\ y).\ \text{case (lhd } t\text{) of Inl () } \to \text{ Inl (lhd } x\text{) | Inr () } \to \text{ Inr (lhd } y\text{))}$
$(\lambda(t,\ x,\ y).\ \text{case (lhd } t\text{) of Inl () } \to \text{ (ltl } t\text{, ltl } x\text{, } y\text{) | Inr () } \to \text{ (ltl } t\text{, } x\text{, ltl } y\text{))}$
$(t,\ x,\ y)$

The above definition constructs the shuffle in the following way. The first function $\lambda(t, x, y).t = \text{LNil}$ ensures that the function terminates when the trajectory is empty. The second function takes either the first element of x or y depending on whether the head of the trajectory is l or r respectively. The final function generates the next seed value for the unfold. Finally the initial seed value for the unfold is given to unfold-llist as $(t,\ x,\ y)$

Typically functions over lazy lists in Isabelle are prefixed with the letter 'l' to differentiate them from those over finite lists, so the hd function would be lhd, and so on. Note that the shuffle operation $x \sqcup\!\sqcup_t y$ is only well defined when the number of l elements in t is equal to the length of x, and similarly for the r elements of t and the length of y. A trajectory is said to be valid w.r.t. a pair of words if this is the case. In Isabelle, this is encoded by the predicate

definition Valid :: 'a llist $\to$ (unit + unit) llist $\to$ 'b llist $\to$ bool **where**
Valid $x\ t\ y \equiv$ llength $(\mathcal{L}\ t) =$ llength $x \wedge$ llength $(\mathcal{R}\ t) =$ llength y.

Valid $x\ t\ y$ enforces a very weak notion of fairness on shuffles, whereby any event in x or y is eventually guaranteed to occur in $x \sqcup\!\sqcup_t y$.

Another thing to note is the type of the $\sqcup\!\sqcup$ operation in Isabelle. It takes two words of types 'a llist and 'b llist respectively, and produces a word of type ('a + 'b) llist. Obviously when 'a and 'b are equal, the sum type can be stripped away to end up with a word of type 'a llist. For the most part this is done implicitly when it is clear, but the function

$$\begin{aligned} &\langle -, - \rangle - :: (\text{'a} \to \text{'c}) \to (\text{'b} \to \text{'c}) \to (\text{'a} + \text{'b}) \to \text{'c}, \\ &\langle f, g \rangle(\text{Inl } a) = f\ a, \\ &\langle f, g \rangle(\text{Inr } b) = g\ b, \end{aligned}$$

is often used in Isabelle to eliminate the sum type, in particular when instantiated as $\langle \text{id}, \text{id} \rangle$. Furthermore we have the functions

$$\mathcal{L} :: (\text{'a} + \text{'b})\ \text{llist} \to \text{'a llist} \qquad \text{and} \qquad \mathcal{R} :: (\text{'a} + \text{'b})\ \text{llist} \to \text{'b llist}$$

which return words containing the left and right parts of a word over 'a + 'b respectively.

We now define the operator $x \sqcup\!\sqcup y$ (notice the lack of a trajectory). Unlike $x \sqcup\!\sqcup_t y$, which returned a single interleaving of x and y, the operation $x \sqcup\!\sqcup y$ returns all interleavings of x and y for all valid trajectories. This is used later to derive (7.12) in the model. It is defined straightforwardly as

$$x \sqcup\!\sqcup y = \{x \sqcup\!\sqcup_t y.\ \exists t.\ \text{Valid } x\ t\ y\}. \tag{7.14}$$

Lemma 17. *(7.14) is equivalent to* $x \sqcup\!\sqcup y = \{z.\ \mathcal{L}\ z = x \wedge \mathcal{R}\ z = y\}$.

The equivalent definition of $x \sqcup\!\sqcup y$ in Lemma 17 is typically much simpler to work with than when $x \sqcup\!\sqcup y$ is defined in terms of $x \sqcup\!\sqcup_t y$. Figure 7.7 shows the shuffling of two short finite words along all valid trajectories. As in Figure 7.6 each letter a_n or b_n represents a state pair (σ_n, σ'_n) or (τ_n, τ'_n) respectively. As can been seen, even for such short words, the resulting shuffle language contains many potential interleavings. In general, shuffling causes a large 'combinatorial explosion' in the number of words in a language. This can make verification tasks quite tricky, assuming one is not able to abstract away from analysing individual interleaving sequences.

Finally, the shuffle operation is lifted to languages in the obvious manner as

$$X \| Y = \{x \sqcup\!\sqcup y.\ x \in X \wedge y \in Y\}.$$

Lemma 18. $(K, \cup, \|, 0, 1)$ *is a commutative dioid.*

Lemma 19. $(K, \cup, \cdot, \|, 0, 1)$ *is a trioid.*

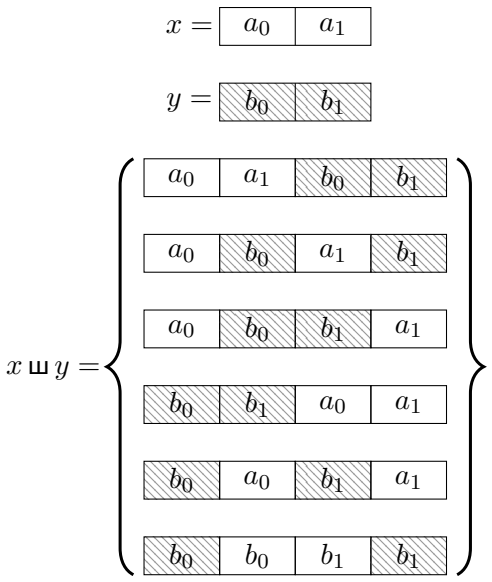


Figure 7.7: An example of shuffling with all possible valid trajectories.

A canonical inhabitant of the set $x \sqcup\!\sqcup y$ is given by $\mathsf{alternate}(x, y)$, which, as the name suggests, simply alternates between picking elements from x and y. For example,

$$\mathsf{alternate}(a_0a_1, b_0b_1b_2) = a_0b_0a_1b_1b_2.$$

alternate satisfies all the properties one would expect, namely,

$$\mathsf{alternate}(x, y) \in x \sqcup\!\sqcup y, \quad \mathcal{L}\ \mathsf{alternate}(x, y) = x, \quad \text{and} \quad \mathcal{R}\ \mathsf{alternate}(x, y) = y.$$

7.3.2 Stuttering and Mumbling Closure

Following [Bro93] and [Din02] and reiterating the definitions in Section 2.3 the *mumble language* $w^\dagger$ for a word $w \in K$ is generated inductively: Assume $\sigma_1, \sigma_2, \sigma_3 \in \Sigma$ and $u, v, w \in K$. First, $w \in w^\dagger$. Secondly, if $u(\sigma_1, \sigma_2)(\sigma_2, \sigma_3)v \in w^\dagger$ then $u(\sigma_1, \sigma_3)v \in w^\dagger$.

The *stutter language* $w^\downarrow$ for a trace w is also defined inductively in much the same way: Assume $\sigma_1, \sigma_2 \in \Sigma$ and $u, v, w \in K$. First $w \in w^\downarrow$. Secondly, if $u(\sigma_1, \sigma_2)v \in w^\downarrow$ then $u(\sigma_1, \sigma_2)(\sigma_2, \sigma_2)v \in w^\downarrow$ and $u(\sigma_1, \sigma_1)(\sigma_2, \sigma_2)v \in w^\downarrow$. Both these operations are lifted to languages in the obvious way as

$$X^\dagger = \bigcup\{x^\dagger.\ x \in X\} \qquad \text{and} \qquad X^\downarrow = \bigcup\{x^\downarrow.\ x \in X\}.$$

Lemma 20. *$-^\dagger$ and $-^\downarrow$ are closure operators.*

Lemma 21. *Stuttering and mumbling commute: $X^{\dagger\downarrow} = X^{\downarrow\dagger}$*

Lemma 22. *Stuttering and mumbling preserve prefix closure.*

Define the *stutter-mumble closure* of X as $X^\ddagger$ where $X^\ddagger = X^{\dagger\downarrow}$.

Lemma 23. $X^\ddagger \cup Y^\ddagger = (X \cup Y)^\ddagger$

Lemma 24. $X^\ddagger Y^\ddagger = (XY)^\ddagger$

Lemma 25. $X^{\ddagger\star} = X^{\star\ddagger}$ *and* $X^{\ddagger\omega} = X^{\omega\ddagger}$

In general, and in contrast to Chapter 6, it is not assumed that any arbitrary language X is stutter-mumble closed unless explicitly denoted as such by $X^\ddagger$. Furthermore, stutter-mumble closure is avoided in order to keep proofs as straightforward as possible, nor is stutter-mumble closure required for the example in Chapter 8. In [Bro93] Brookes uses stutter-mumble closure to ensure full-abstraction (see Section 2.3), however we are able to prove what we need to prove without requiring such a strong property for our trace semantics.

7.3.3 Relies and Guarantees

Having defined traces in the model, we now turn our attention to the relies and guarantees. As discussed in Section 7.2 and in contrast to Chapter 6 the algebra we are now considering is a two sorted algebra with different objects representing programs and interference constraints.

Interference constraints are considered to be reflexive and transitively closed relations, such that $R = R^\star$. Reflexive and transitive closure is required because one cannot in general assume anything about the exact amount of interference received between any pair of program transitions. The join of two interference constraints, $R \sqcup G$, is therefore simply $(R \cup G)^\star$, similarly, the meet of interference constraints, $R \sqcap G$, is defined as $(R \cap G)^\star$.

Lemma 26. *Interference constraints form a complete lattice.*

$\langle G \rangle$ defines the greatest set of traces such that each program transition comes from the interference constraint G.

The : operator from Section 7.2 is defined as

$$R : X = \{x \mid \mathsf{env}(R, x) \wedge x \in X\}.$$

The predicate $\mathsf{env}(R, x)$ holds for an interference constraint R and a trace x provided every environment transition in x is contained within R. It is defined as

$$\mathsf{env}(R, x) \iff \forall x'\ \sigma\ \sigma'\ \tau\ \tau'\ x''.\ x \neq x'(\sigma, \sigma')(\tau, \tau')x'' \vee (\sigma', \tau) \in R.$$

The operator $R : X$ therefore filters out all the traces from X where the environment performs an action not permitted by R. If every environment transition in x satisfies R, then x will be in $R : X$.

The $\rhd$ operator from Section 7.2 is defined as

$$R \rhd X = \{y \mid \exists x \in X.\ x \simeq_R y\},$$

$$\begin{aligned} x \simeq_R y \iff (\exists z\ x'\ \sigma\ \sigma'\ \tau\ \tau'\ \tau''\ y'.\ & \\ x = z(\sigma, \sigma')(\tau, \tau')x' & \\ \wedge\ y = z(\sigma, \sigma')(\tau, \tau'')y' & \\ \wedge\ (\sigma', \tau) \notin R & \\ \wedge\ \mathsf{env}(R, z(\sigma, \sigma'))) & \\ \wedge\ \mathsf{finite}(x) = \mathsf{finite}(y) & \\ \vee\ (x = y \wedge \mathsf{env}(R, x)). & \end{aligned}$$

For a pair of words x and y, $x \simeq_R y$ if both words are equal and their environment transitions never violate R, or if they are equal up until the point where an environment transition occurs which violates R. Therefore $R \rhd X$ defines the set of words such that every $x \in R \rhd X$ corresponds to a word in X up until R is violated by the environment, after which point x may perform any possible combination of program and environment steps.

Note that if x is finite then y must also be finite—this is required for the sequential composition axioms (7.4). This is justifiable as it is reasonable to assume that programs containing no loops cannot be made infinite by the environment. All processes containing loops will still contain infinite divergences after an environment transition fails. However, all the other axioms can still be derived without this condition being present within $\simeq_R$, and this has been verified in Isabelle. Given a more in depth treatment of termination perhaps this condition could be removed. One could also imagine reducing this equivalence to an implication, with the obvious caveat that $\simeq_R$ would no longer be an equivalence relation, but this has not been investigated thoroughly.

Lemma 27. *$R : -$ is a closure operator and $R \rhd -$ is a co-closure operator*

Lemma 28. *$\simeq_R$ is an equivalence relation.*

The following lemma states that $\rhd$ and : are well behaved w.r.t. stutter-mumble closure.

Lemma 29. *$R \rhd X^\ddagger = (R \rhd X)^\ddagger$ and $R : X^\ddagger = (R : X)^\ddagger$*

A note on prefix closure

As mentioned above it is not assumed that sets of traces are prefix closed. This is primarily to keep the model as simple as possible. We can prove all the properties we require from Section 7.2 without it, so why would we want it? Well, in concert with stutter-mumble closure, it does allow for several nice properties to be proven, such as the Galois connection between : and $\rhd$ discussed in Section 7.2.

However to prove this, we must slightly redefine the : operator for prefix and stutter-mumble closed traces. A trace x is in $R \mathbin{\hat{:}} X$ if $\mathsf{env}(R, x)$ holds or if it is of the form

$$x = x'(\sigma, \sigma')(\tau, \tau) \dots (\tau, \tau)$$

where $(\sigma', \tau) \notin R$ and $\mathsf{env}(R, x'(\sigma, \sigma'))$. In other words, a trace in $R \mathbin{\hat{:}} X$ may only stutter finitely after an environment step occurs which is not in R.

Theorem 30. *If X and Y are prefix and stutter-mumble closed, the Galois connection*

$$R \mathbin{\hat{:}} X \leq Y \iff X \leq R \rhd Y$$

from Section 7.2 holds in the infinite traces model

Proof. For the left to right implication assume $R \mathbin{\hat{:}} X \leq Y$ holds. It must be shown for all x that $x \in X$ implies that $x \simeq_R y$ for some $y \in Y$. There are two possibilities for x, either $\mathsf{env}(R, x)$ holds, or it does not. The case where $\mathsf{env}(R, x)$ holds is trivial, as this straightforwardly implies $x \in R \mathbin{\hat{:}} X$, and thus $x \in Y$ and $x \simeq_R x$.

The more complex case is when $\mathsf{env}(R, x)$ does not hold. In this case there must exist a minimal point in x where the environment step fails to satisfy R (see Lemma 31), formally:

$$\begin{aligned}
&\exists x'\ \sigma\ \sigma'\ \tau\ \tau'\ x''.\\
&\quad x = x'(\sigma, \sigma')(\tau, \tau')x''\\
&\quad \wedge\ \mathsf{env}(R, x(\sigma, \sigma'))\\
&\quad \wedge\ (\sigma', \tau) \notin R\\
&\quad \wedge\ \mathsf{finite}(x')
\end{aligned}$$

We know from prefix closure that $x'(\sigma, \sigma')(\tau, \tau') \in X$. As X is stutter-mumble closed, $x'(\sigma, \sigma')(\tau, \tau)(\tau, \tau')$ must also be in X. Due to prefix closure, this implies that $x'(\sigma, \sigma')(\tau, \tau) \in X$. From the definition of $\hat{:}$ we can conclude that $x'(\sigma, \sigma')(\tau, \tau) \in R \mathbin{\hat{:}} X$, and therefore $x'(\sigma, \sigma')(\tau, \tau) \in Y$. It is clear that $x \simeq_R x'(\sigma, \sigma')(\tau, \tau)$ and this completes this half of the proof.

For the right to left implication we assume the right hand side of the implication and that $x \in R \mathbin{\hat{:}} X$ and show $x \in Y$. It is either the case that $\mathsf{env}(R, x)$ and $x \in X$, or

$$\begin{aligned} x \in \{ & x(\sigma, \sigma')(\tau, \tau) \dots (\tau, \tau) \mid \exists \tau'\ x'. \\ & \mathsf{env}(R, x(\sigma, \sigma')) \\ & \wedge\ x(\sigma, \sigma')(\tau, \tau')x' \in X \\ & \wedge\ (\sigma', \tau) \notin R \\ & \wedge\ \mathsf{finite}(x)\}, \end{aligned}$$

which is the case when there is an environment transition in x violating R. This can be followed be finite stuttering of τ. The case where $\mathsf{env}(R, x)$ and $x \in X$ holds is is trivial. The left hand side of the implication translates to

$$\forall x \in X.\ \exists y \in Y.\ y \simeq_R x.$$

From $x \in X$ we therefore obtain a $y \in Y$ such that $x \simeq_R y$, however since $\mathsf{env}(R, x)$, it must be the case that $x = y$, and therefore $x \in Y$.

For the other case we know that x has the form $x'(\sigma, \sigma')(\tau, \tau) \dots (\tau, \tau)$ for some x', σ and τ, such that $(\sigma', \tau) \notin R$ and that there must exist an x'' where $x'(\sigma, \sigma')(\tau, \tau) \dots (\tau, \tau)(\tau, \tau')x''$ is in $X^\ddagger$ and therefore X. Due to prefix closure $x'(\sigma, \sigma')(\tau, \tau)$ must also be in X. We can therefore obtain a $y \in Y$ such that $x'(\sigma, \sigma')(\tau, \tau) \simeq_R y$. From the definition of $\simeq_R$ we can see that $x'(\sigma, \sigma')(\tau, \tau)$ must be a prefix of y (modulo stuttering), and therefore $x'(\sigma, \sigma')(\tau, \tau) \in Y$. This completes the proof. □

It is interesting to note that this property appears to rely quite heavily on properties provided by prefix closure, as well as stutter and mumble closure. The Galois connection in Theorem 30 cannot be proven in a model without them, as one cannot show that $x \in X$ if $x \in R \mathbin{\hat{:}} X$ without such closure conditions. Not requiring this Galois connection in the algebra in Section 7.2 therefore allows a wider array of potential models without such notions.

In the Isabelle theory backing this Chapter, prefix closure is not formalised.

7.3.4 Properties of Shuffle and Traces

In this section, several properties of the shuffle operator and traces are given. All of these properties are used for deriving the concurrency rule (7.12) from Section 7.2 in the model, as shown in Subsection 7.3.5.

Lemma 31. *The property* $\neg\mathsf{env}(R, x)$ *implies there exists* x_p, σ, σ', τ, τ', *and* x_t, *where*

1. $x = x_p(\sigma, \sigma')(\tau, \tau')x_t$
2. $(\sigma', \tau) \notin R$
3. $\mathsf{finite}(x_p)$
4. $\mathsf{env}(R, x_p(\sigma, \sigma'))$

Proof. Conditions *1-3* come directly from the definition of env, which fails when x contains an environment step not in R. We can show inductively that if this is not the first such failure, then there must be one before it, and so on, until we find the first. This gives us a way to split x into $x_p(\sigma, \sigma')(\tau, \tau')x_t$ in such a way that condition *4* holds. □

Lemma 31 allows for a variant of $x \simeq_R y$ which does not rely on env. First define

$$\begin{aligned} x \sim_R y \iff \exists z\ x'\ \sigma\ \sigma'\ \tau\ \tau'\ \tau''\ y'.& \\ x = z(\sigma, \sigma')(\tau, \tau')x'& \\ \wedge\ y = z(\sigma, \sigma')(\tau, \tau'')y'& \\ \wedge\ (\sigma', \tau) \notin R& \\ \wedge\ \mathsf{finite}(x) = \mathsf{finite}(y)& \\ \wedge\ \mathsf{env}(R, z(\sigma, \sigma')).& \end{aligned}$$

Clearly $x \simeq_R y = (x \sim_R y \vee (x = y \wedge \mathsf{env}(R, x))$, now define

$$\begin{aligned} x \hat{\sim}_R y \iff \exists z\ x'\ \sigma\ \sigma'\ \tau\ \tau'\ \tau''\ y'.& \\ x = z(\sigma, \sigma')(\tau, \tau')x'& \\ \wedge\ y = z(\sigma, \sigma')(\tau, \tau'')y'& \\ \wedge\ (\sigma', \tau) \notin R,& \\ \wedge\ \mathsf{finite}(x) = \mathsf{finite}(y)& \end{aligned}$$

which is $\sim_R$ without any requirement that $\mathsf{env}(R, z(\sigma, \sigma'))$.

Lemma 32. $x \hat{\sim}_R y \iff x \sim_R y$

Proof. The direction $x \sim_R y \implies x \hat{\sim}_R y$ is trivial. For the other direction, look at the shared prefix of x and y, which is $z(\sigma, \sigma')$, if $\mathsf{env}(R, z(\sigma, \sigma'))$ then $x \sim_R y$ trivially holds. If $\neg\mathsf{env}(R, z(\sigma, \sigma'))$ we apply Lemma 31 to find a shared prefix for which $x \sim_R y$ holds. □

This lemma is very useful, as it lets one avoid having to reason specifically about the first environment transition failure when proving $x \simeq_R y$ (which would often require induction), and instead reason about any such failure. However when $x \simeq_R y$ is assumed in a proof one may always assume the shared prefix before the environment failure contains only valid environment steps.

Lemma 33. *For a trace $(\sigma, \sigma')x(\tau, \tau')$, if $\mathsf{finite}(x)$, $x \in \langle G \rangle$ and $(\sigma', \tau) \notin R \sqcup G$ there exists w, γ, γ', and φ, such that*

$$\forall \tau'.\ \exists \varphi'\ w'.\ (\sigma, \sigma')x(\tau, \tau') = w(\gamma, \gamma')(\varphi, \varphi')w' \wedge (\gamma', \varphi) \notin R \wedge \mathsf{env}(R, w(\gamma, \gamma'))$$

Proof. By induction on x. □

Intuitively Lemma 33 states that if there exists some environment failure from an environment $R \sqcup G$ between σ' and τ, which are separated by some program transitions in x that all satisfy G, then the environment failure must have been caused by the global environment R, and there must be an environment transition in $(\sigma, \sigma')x(\tau, \tau')$ where this occurs.

Lemma 34. *The shuffle*

$$Z = x_p(\sigma_0, \sigma_0')(\sigma_1, \sigma_1')x_t \sqcup\!\sqcup y_p(\tau_0, \tau_0')(\tau_1, \tau_1')y_t$$

is equal to

$$\begin{aligned}
&\bigcup\{(x_p \sqcup\!\sqcup y_p')\{(\sigma_0, \sigma_0')y_p''(\tau_0, \tau_0')(\tau_1, \tau_1')y_t'(\sigma_1, \sigma_1')\}(x_t \sqcup\!\sqcup y_t'') \mid y_p = y_p'y_p'' \wedge y_t = y_t'y_t''\} \\
\cup &\bigcup\{(x_p \sqcup\!\sqcup y_p')\{(\sigma_0, \sigma_0')y_p''(\tau_0, \tau_0')(\sigma_1, \sigma_1')x_t'(\tau_1, \tau_1')\}(x_t'' \sqcup\!\sqcup y_t) \mid y_p = y_p'y_p'' \wedge x_t = x_t'x_t''\} \\
\cup &\bigcup\{(x_p \sqcup\!\sqcup y_p')\{(\sigma_0, \sigma_0')y_p''(\sigma_1, \sigma_1')\}(x_t \sqcup\!\sqcup y_p'''(\tau_0, \tau_0')(\tau_1, \tau_1')y_t) \mid y_p = y_p'y_p''y_p'''\} \\
\cup &\bigcup\{(x_p' \sqcup\!\sqcup y_p)\{(\tau_0, \tau_0')x_p''(\sigma_0, \sigma_0')(\sigma_1, \sigma_1')x_t'(\tau_1, \tau_1')\}(x_t'' \sqcup\!\sqcup y_t) \mid x_p = x_p'x_p'' \wedge x_t = x_t'x_t''\} \\
\cup &\bigcup\{(x_p' \sqcup\!\sqcup y_p)\{(\tau_0, \tau_0')x_p''(\sigma_0, \sigma_0')(\tau_1, \tau_1')y_t'(\sigma_1, \sigma_1')\}(x_t \sqcup\!\sqcup y_t'') \mid x_p = x_p'x_p'' \wedge y_t = y_t'y_t''\} \\
\cup &\bigcup\{(x_p' \sqcup\!\sqcup y_p)\{(\tau_0, \tau_0')y_p''(\tau_1, \tau_1')\}(x_p'''(\sigma_0, \sigma_0')(\sigma_1, \sigma_1')x_t \sqcup\!\sqcup y_t) \mid x_p = x_p'x_p''x_p'''\}
\end{aligned}$$

Note that for the conditions $y_p = y_p'y_p''$ and similar there is an implicit assumption that y_p' is finite, which is omitted for brevity.

Proof. Consider the six possible ways that (σ_0, σ_0'), (σ_1, σ_1'), (τ_0, τ_0') and (τ_1, τ_1') could be arranged in any interleaving in Z:

1. $(\sigma_0, \sigma_0')(\tau_0, \tau_0')(\tau_1, \tau_1')(\sigma_1, \sigma_1')$
2. $(\sigma_0, \sigma_0')(\tau_0, \tau_0')(\sigma_1, \sigma_1')(\tau_1, \tau_1')$
3. $(\sigma_0, \sigma_0')(\sigma_1, \sigma_1')(\tau_0, \tau_0')(\tau_1, \tau_1')$

4. $(\tau_0, \tau_0')(\sigma_0, \sigma_0')(\sigma_1, \sigma_1')(\tau_1, \tau_1')$
5. $(\tau_0, \tau_0')(\sigma_0, \sigma_0')(\tau_1, \tau_1')(\sigma_1, \sigma_1')$
6. $(\tau_0, \tau_0')(\tau_1, \tau_1')(\sigma_0, \sigma_0')(\sigma_1, \sigma_1')$

The above statement formalises this fact, using equational reasoning to expand the possible interleavings of the shuffle. It splits x_p, x_t, y_p, and y_t as appropriate between the interesting pairs listed above. □

7.3.5 Concurrency Rule

Theorem 35. *The following concurrency axiom from Section 7.2 holds in the potentially infinite language model:*

$$(R \sqcup G_2 \rhd X \sqcap \langle G_1 \rangle) \| (R \sqcup G_1 \rhd Y \sqcap \langle G_2 \rangle) \leq R \rhd (X \sqcap \langle G_1 \rangle) \| (Y \sqcap \langle G_2 \rangle)$$

Proof. We begin by performing the following simplifications using the definition of $\rhd$ to the left hand side of the inequality:

$$\begin{aligned}
&(R \sqcup G_2 \rhd X \sqcap \langle G_1 \rangle) \| (R \sqcup G_1 \rhd Y \sqcap \langle G_2 \rangle) \\
&= \{x \mid \exists x' \in X \sqcap \langle G_1 \rangle.\ x' \simeq_{R \sqcup G_2} x\} \| \{y \mid \exists y' \in Y \sqcap \langle G_2 \rangle.\ y' \simeq_{R \sqcup G_1} y\} \\
&= \bigcup \{x \uplus y \mid (\exists x' \in X \sqcap \langle G_1 \rangle.\ x' \simeq_{R \sqcup G_2} x) \wedge (\exists y' \in Y \sqcap \langle G_2 \rangle.\ y' \simeq_{R \sqcup G_1} y)\}.
\end{aligned}$$

This reduces the problem of solving the inequality into showing

$$x \uplus y \in R \rhd (X \sqcap \langle G_1 \rangle) \| (Y \sqcap \langle G_2 \rangle)$$

from the assumptions

$$x' \in X, \tag{7.15}$$
$$x' \in \langle G_1 \rangle, \tag{7.16}$$
$$y' \in Y, \tag{7.17}$$
$$y' \in \langle G_2 \rangle, \tag{7.18}$$
$$x' \simeq_{R \sqcup G_2} x, \tag{7.19}$$
$$y' \simeq_{R \sqcup G_1} y. \tag{7.20}$$

Obviously it suffices to prove that

$$x \uplus y \in R \rhd x' \uplus y'$$

which in turn, becomes

$$\forall z \in x \uplus y.\ \exists z' \in x' \uplus y'.\ z' \simeq_R z. \tag{7.21}$$

In other words, for any particular interleaving z of x and y, a trace z' in $x' \sqcup\!\sqcup y'$ must be constructed such that $z' \simeq_R z$. We start by expanding the definitions of $\simeq_R$ in the assumptions.

$$x' \simeq_{R \sqcup G_2} x \iff (x' \sim_{R \sqcup G_2} x) \vee (x' = x \wedge \mathsf{env}(R \sqcup G_2, x))$$
$$y' \simeq_{R \sqcup G_1} y \iff (y' \sim_{R \sqcup G_1} y) \vee (y' = y \wedge \mathsf{env}(R \sqcup G_1, y))$$

The case where $x' = x$ and $y' = y$ is obviously trivial as this implies $z' = z$.

For the case where neither $x' = x$ nor $y' = y$, we obtain x_1, x_2, x_3, σ_0, σ_0', σ_1, σ_1', σ_1'', y_1, y_2, y_3 τ_0, τ_0', τ_1, τ_1', and τ_1'' from $x' \sim_{R \sqcup G_2} x$ and $y' \sim_{R \sqcup G_1} y$ where

$$x' = x_1(\sigma_0, \sigma_0')(\sigma_1, \sigma_1')x_2 \tag{7.22}$$
$$x = x_1(\sigma_0, \sigma_0')(\sigma_1, \sigma_1'')x_3 \tag{7.23}$$
$$y' = y_1(\tau_0, \tau_0')(\tau_1, \tau_1')y_2 \tag{7.24}$$
$$y = y_1(\tau_0, \tau_0')(\tau_1, \tau_1'')y_3 \tag{7.25}$$
$$\sigma_0')(\sigma_1 \notin R \sqcup G_2 \tag{7.26}$$
$$\tau_0')(\tau_1 \notin R \sqcup G_1 \tag{7.27}$$
$$\mathsf{env}(R \sqcup G_2, x_1(\sigma_0, \sigma_0')) \tag{7.28}$$
$$\mathsf{env}(R \sqcup G_1, y_1(\tau_0, \tau_0'))) \tag{7.29}$$
$$\mathsf{finite}(y_2) = \mathsf{finite}(y_3) \tag{7.30}$$
$$\mathsf{finite}(x_2) = \mathsf{finite}(x_3) \tag{7.31}$$

To prove (7.21), consider the interleavings of

$$z \in x_1(\sigma_x, \sigma_x')(\tau_x, \tau_x')x_3 \;\sqcup\!\sqcup\; y_1(\sigma_y, \sigma_y')(\tau_y, \tau_y')y_3.$$

which are enumerated by Lemma 34, as the following cases:

1. For this case there exists a y_1' and y_1'' where $y_1 = y_1'y_1''$ and there exists a y_3' and y_3'' where $y_3 = y_3'y_3''$ such that

   $$z \in (x_1 \sqcup\!\sqcup y_1')\{(\sigma_0, \sigma_0')y_1''(\tau_0, \tau_0')(\tau_1, \tau_1'')y_3'(\sigma_1, \sigma_1'')\}(x_3 \sqcup\!\sqcup y_3'').$$

   We know that y_1' and y_3' are finite by Lemma 34. This implies that $x_1 \sqcup\!\sqcup y_1'$ is also finite, which ensures that the subtrace where R is violated by the environment actually exists.

   From the definition of $\sqcup\!\sqcup$ we know that there must exist valid trajectories t_1 and t_2 such that

   $$z = (x_1 \sqcup\!\sqcup_{t_1} y_1')(\sigma_0, \sigma_0')y_1''(\tau_0, \tau_0')(\tau_1, \tau_1'')y_3'(\sigma_1, \sigma_1'')(x_3 \sqcup\!\sqcup_{t_2} y_3'').$$

   We now construct the trace

   $$z' = (x_1 \sqcup\!\sqcup_{t_1} y_1')(\sigma_0, \sigma_0')y_1''(\tau_0, \tau_0')(\tau_1, \tau_1')(\mathsf{alternate}((\sigma_1, \sigma_1')x_2, y_2)),$$

(which is clearly in $x' \sqcup y'$) and prove that $z' \simeq_R z$. Since $z' \neq z$, by Lemma 32 we are only required to prove that $z' \hat{\sim}_R z$. This is done by finding an environment step that is not in R, and for which both z' and z are equal to up until that environment transition. In this case $t'_0)(t_1$ is not in R by (7.27). We must also show that the finiteness of $x_3 \sqcup_{t_2} y''_3$ is the same as $\mathsf{alternate}((\sigma_1, \sigma'_1)x_2, y_2)$—this is trivial due to (7.30) and (7.31). Such finiteness goals are always trivial throughout the proof, so they will be omitted from future cases.

2. For this case there exists a y'_1 and y''_1 where $y_1 = y'_1 y''_1$ and there exists a x'_3 and x''_3 where $x_3 = x'_3 x''_3$ such that

$$z \in (x_1 \sqcup y'_1)\{(\sigma_0, \sigma'_0)y''_1(\tau_0, \tau'_0)(\sigma_1, \sigma''_1)x'_3(\tau_1, \tau''_1)\}(x''_3 \sqcup y_3).$$

Again y'_1 and x'_3 are finite by Lemma 34.

Similarly to case 1, we acquire valid trajectories t_1 and t_2 such that

$$z = (x_1 \sqcup_{t_1} y'_1)(\sigma_0, \sigma'_0)y''_1(\tau_0, \tau'_0)(\sigma_1, \sigma''_1)x'_3(\tau_1, \tau''_1)(x''_3 \sqcup_{t_2} y_3),$$

and construct z' as

$$z' = (x_1 \sqcup_{t_1} y'_1)(\sigma_0, \sigma'_0)y''_1(\tau_0, \tau'_0)(\sigma_1, \sigma'_1)(\mathsf{alternate}(x_2, (\tau_1, \tau'_1)y_2)).$$

For this case proving $z' \hat{\sim} z$ is trickier than in case 1. This is because neither (7.26) nor (7.27) occurs in z' or z. Instead, we inspect the subsequence $(\sigma_0, \sigma'_0)y''_1(\tau_0, \tau'_0)(\sigma_1, \sigma''_1)$. We can prove y''_1 is finite, as y_1 is itself finite. Furthermore $\sigma'_0)(\sigma_1 \notin R \sqcup G_2$ (7.26), and $y''_1(\tau_0, \tau'_0) \in \langle G_2 \rangle$ by (7.18). This allows Lemma 33 to be applied, obtaining w, γ, γ' and φ such that

$$\begin{aligned} \forall \sigma'''_1.\ \exists \varphi'\ w'.\ (\sigma_0, \sigma'_0)y''_1(\tau_0, \tau'_0)(\sigma_1, \sigma'''_1) = w(\gamma, \gamma')(\varphi, \varphi')w'. \\ \wedge\ \gamma')(\varphi \notin R \\ \wedge\ \mathsf{env}(R, w(\gamma, \gamma')). \end{aligned} \tag{7.32}$$

Instantiating σ'''_1 as σ''_1 and σ'_1 respectively, we obtain

$$\exists \varphi'\ w'.\ z = (x_1 \sqcup_{t_1} y'_1)w(\gamma, \gamma')(\varphi, \varphi')w'x'_3(\tau_1, \tau''_1)(x''_3 \sqcup_{t_2} y_3),$$

and

$$\exists \varphi'\ w'.\ z' = (x_1 \sqcup_{t_1} y'_1)w(\gamma, \gamma')(\varphi, \varphi')w'(\mathsf{alternate}(x_2, (\tau_1, \tau'_1)y_2)).$$

Now z' and z share a common finite prefix up until a point where R is violated by the environment, which is what we need to prove $z' \hat{\sim}_R z$, and therefore $z' \simeq_R z$.

3. For this case there exists a y_1', y_1'' and y_1''' where $y_1 = y_1' y_1'' y_1'''$ such that

$$z \in (x_1 \sqcup y_1')\{(\sigma_0, \sigma_0')y_1''(\sigma_1, \sigma_1')\}(x_3 \sqcup y_1'''(\tau_0, \tau_0')(\tau_1, \tau_1')y_3).$$

Again with y_1' and y_1'' being finite by Lemma 34.

We again obtain trajectories t_1 and t_2 such that

$$z = (x_1 \sqcup\!\!\sqcup_{t_1} y_1')(\sigma_0, \sigma_0')y_1''(\sigma_1, \sigma_1')(x_3 \sqcup\!\!\sqcup_{t_2} y_1'''(\tau_0, \tau_0')(\tau_1, \tau_1')y_3),$$

and construct z' as

$$z' = (x_1 \sqcup\!\!\sqcup_{t_1} y_1')(\sigma_0, \sigma_0')y_1''(\sigma_1, \sigma_1')(\mathsf{alternate}(x_2, (y_1'''(\tau_0, \tau_0')(\tau_1, \tau_1')y_3))).$$

Like in case 2, Lemma 33 is applied to obtain an environment transition that violates R.

$$\begin{aligned} \forall \sigma_1'''.\ \exists \varphi'\ w'.\ (\sigma_0, \sigma_0')y_1''(\sigma_1, \sigma_1''') = w(\gamma, \gamma')(\varphi, \varphi')w'. \\ \wedge \gamma')(\varphi \notin R \\ \wedge \mathsf{env}(R, w(\gamma, \gamma')). \end{aligned} \tag{7.33}$$

Instantiating σ_1''' as σ_1'' and σ_1' respectively, we obtain

$$\exists \varphi'\ w'.\ z = (x_1 \sqcup\!\!\sqcup_{t_1} y_1')w(\gamma, \gamma')(\varphi, \varphi')w'(x_3 \sqcup\!\!\sqcup_{t_2} y_1'''(\tau_0, \tau_0')(\tau_1, \tau_1')y_3),$$

and

$$\exists \varphi'\ w'.\ z' = (x_1 \sqcup\!\!\sqcup_{t_1} y_1')w(\gamma, \gamma')(\varphi, \varphi')w'(\mathsf{alternate}(x_2, (y_1'''(\tau_0, \tau_0')(\tau_1, \tau_1')y_3))).$$

As in case 2, z' and z share a common finite prefix up until a point where R is violated by the environment, which is what we need to prove $z' \hat{\sim}_R z$, and therefore $z' \simeq_R z$.

Obviously there are 3 more cases demanded by Lemma 34, but cases 4-6 are symmetrical to cases 1-3 w.r.t. x, σ and y, τ respectively.

Finally we consider the case where either $x' = x$ or $y' = y$, but not both. In this instance the proof essentially reduces to the same situations as case 3 and case 6 from when both equalities were false.

□

This property is quite tricky to prove in Isabelle, partly because it is difficult to get Isabelle to recognise the similarities between all 9 cases involved in the proof. It ends up being easier to ignore the proof engineering involved in extracting the common aspects of these cases, and simply copy and paste proof segments where appropriate, mutatis mutandis. Furthermore, certain lemmas required by this property, such as Lemma 34 are very tedious to prove mechanically, requiring a great deal of equational reasoning within large set comprehensions. Overall the Isabelle proof of this property amounts to several thousand lines of proof script. The other axioms from Section 7.2 are much more straightforward, so their proofs are elided here, given that they have been formalised in Isabelle.

7.3.6 Interchange Laws

The exchange law below from [HHM+11] does not hold in this model, except in the case where X and Y are finite. Given the presence of the preimplication operator, a similar exchange law exists for it, again with the same constraint on X and Y.

$$X \subseteq \mathcal{F} \wedge Y \subseteq \mathcal{F} \implies (X \| Y)(Z \| W) \subseteq XW \| YZ,$$
$$X \subseteq \mathcal{F} \wedge Y \subseteq \mathcal{F} \implies (X \to W) \| (Y \to Z) \subseteq (X \| Y) \to (Z \| W).$$

These properties are quite useful in practice, as certain elements of the model are guaranteed to be finite, such as tests.

7.3.7 Soundness

Theorem 36. $((\Sigma^2)^\infty, \{R \mid R = R^*\}, \rhd, \langle - \rangle)$ *is an interference algebra.*

Proof. All the rules for an interference algebra in Definition 4 have been proven in Isabelle. The most complex rule, (7.12), has been also been proven above in Subsection 7.3.5. □

In this Chapter it has been shown that all the rules of (propositional) rely-guarantee can be derived in the interference algebra from Section 7.2, and furthermore that the (potentially) infinite trace model in this Section satisfies the rules of the rely/guarantee algebra. This constitutes a proof that rely/guarantee is sound w.r.t the infinite language semantics presented in this Section.

7.4 Conclusion

In this Chapter, a refined algebra for rely-guarantee reasoning has been developed and presented. A key axiom was given in (7.12), and it was proven in Subsection 7.3.5, in addition to being proven within Isabelle. This axiom constitutes the core complexity of the soundness proof for the rely-guarantee algebra. The vast majority of the mechanisation effort for the infinite language model went into verifying this particular axiom. In general, properties involving shuffling tend to be quite hard to verify. This is partly due to the large combinatorial blowup in language size caused by shuffling. One advantage pomset based models might have over interleaving semantics is that this explosion in language sizes would be avoided.

The algebra and model in this Chapter significantly improves on the algebra and model in Chapter 6. In particular, this algebra better captures the nature of interference between concurrent processes, and does so in a more general fashion. Unlike the encoding of the Jones quintuple in Chapter 6, based on [HMSW11], the encoding in this Chapter more faithfully captures

the intended relation between the rely and guarantee. In essence, the encoding in [HMSW11] and Chapter 6 is an over-simplification, and fails to capture all the intended properties of interference. This improved encoding has the benefit of greatly simplifying verification and refinement efforts using this algebra and model.

In the next chapter this algebra and model is used to verify actual programs, and it is explained how tests and assignment statements can be encoded in the model.

Chapter 8

Examples

8.1 Introduction

In this chapter, the model presented in Section 7.3 is expanded with notions of tests, as well as assignment statements. The addition of such constructs allows for the verification and refinement of actual concurrent programs within Isabelle. It will be shown how the common FINDP example, oft used with the rely-guarantee literature, can be derived via refinement within Isabelle.

8.2 Tests in the Model

A test, denoted $\mathsf{test}(P)$, contains sequences with a single transition in which both the first and last states of that transition satisfy P. In other words, tests are singleton stuttering steps satisfying P.

$$\mathsf{test}(P) = \{(\sigma, \sigma) \mid P(\sigma)\}.$$

The program $\mathcal{F}; \mathsf{test}(P)$ therefore denotes any terminating program which finishes (by stuttering) in a state satisfying P. In general, rather than allowing arbitrary stuttering, statements are defined with the minimum amount of stuttering required[1]. In fact, the only stuttering in this model is a single explicit step after each assignment statement. For succinctness, $\mathcal{F}; \mathsf{test}(P)$ is abbreviated as $\mathsf{end}(P)$. Note that this is inconsistent with the definition of end in Chapter 6. For proving correctness, the postcondition q from the quintuple (7.13) is assumed to be an element of the form $\mathsf{end}(Q)$. This gives a total correctness interpretation of the quintuple, wherein programs are required to terminate in states ending in Q. If $\mathsf{end}(Q)$ is defined as $\top; \mathsf{test}(Q)$ then one obtains a partial correctness setting wherein programs may either terminate in states satisfying Q or diverge. The proof rules presented in this section hold for either definition of end unless specified otherwise.

An interference constraint that preserves the predicate P, $\mathsf{preserve}(P)$, can be defined as

$$\mathsf{preserve}(P) = \{(\sigma, \sigma') \mid P(\sigma) \implies P(\sigma')\}^\star.$$

For the sake of concision $\mathsf{preserve}(P)$ is sometimes denoted $\vec{P}$.

In KAT, the conjunction of tests p and q is simply pq. Here, however, things are rather more tricky. In general, for a program $\mathsf{end}(P \wedge Q)$ there are two ways to break it up into a program $\mathsf{end}(P)$ that establishes P and another $\mathsf{end}(Q)$ that establishes Q. First, we can establish P, and then sequentially establish Q guaranteeing that both the environment and the program implementing $\mathsf{end}(Q)$ will preserve P. This is described by the rule

$$\vec{P} \rhd \mathsf{end}(P \wedge Q) \sqsubseteq \mathsf{end}(P); (\mathsf{test}(P) \to \vec{P} \rhd \mathsf{end}(Q) \cap \langle \vec{P} \rangle).$$

[1] In general, allowing more stuttering (and mumbling) gives nicer algebraic properties, especially for tests. However, this comes at the expense of making proofs significantly more complicated.

Second, we can establish P and Q in parallel provided that the environment preserves P and Q, and if the program implementing $\mathsf{end}(P)$ can guarantee that it preserves Q, and vice versa for $\mathsf{end}(Q)$ and P. This is described by the rule

$$\vec{P} \sqcap \vec{Q} \rhd \mathsf{end}(P \wedge Q) \sqsubseteq \mathsf{end}(P) \cap \langle \vec{Q} \rangle \parallel \mathsf{end}(Q) \cap \langle \vec{P} \rangle.$$

Fortunately for disjunction things are simpler, as we have

$$\mathsf{test}(P \vee Q) = \mathsf{test}(P) + \mathsf{test}(Q).$$

Technically, a dual rule also exists for conjunction as

$$\mathsf{test}(P \wedge Q) = \mathsf{test}(P) \cap \mathsf{test}(Q),$$

but this is less useful as $\cap$, unlike $+$, is not an implementable programming construct. As mentioned, $\mathsf{test}(P); \mathsf{test}(Q)$ is not in general the conjunction of P and Q. However, $\mathsf{test}(P); \mathsf{test}(Q)$ can usually be refined to $\mathsf{test}(P \wedge Q)$ when used as a precondition:

$$\mathsf{test}(P); \mathsf{test}(B) \to R \rhd \mathsf{end}(Q) \cap \langle G \rangle \sqsubseteq \mathsf{test}(P \wedge B) \to R \rhd \mathsf{end}(Q) \cap \langle G \rangle.$$

This property holds whenever $R \leq \vec{P}$, and whenever either $\mathsf{test}(Q) \neq \emptyset$ or $\mathsf{test}(P \wedge B) \neq \emptyset$. In other words, it must be the case that R preserves the precondition P, and either the post state must be reachable, or the conjunction of P and B cannot be empty. These side-conditions appear to be a consequence of total correctness—in a partial correctness setting (where $\mathsf{end}(Q) = \top; \mathsf{test}(Q)$) they are unnecessary.

A similar rule allows redundant tests to be dropped:

$$\mathsf{test}(P); \mathsf{test}(B) \to R \rhd \mathsf{end}(Q) \cap \langle G \rangle \sqsubseteq \mathsf{test}(P) \to R \rhd \mathsf{end}(Q) \cap \langle G \rangle.$$

Here it is only required that $R \leq \vec{P}$. These two rules allow for the following inference rule, allowing tests to be pushed into preconditions:

$$R \leq \vec{P} \implies R, G \vdash \{P \wedge B\} X \{Q\} \implies R, B \vdash \{P\} \mathsf{test}(B); X \{Q\}.$$

Finally we have that

$$R \rhd \mathsf{end}(P \wedge Q) \cap \langle G \rangle \sqsubseteq \mathsf{test}(P); \mathsf{test}(Q)$$

whenever $R \leq \vec{P}$, and $\mathsf{test}(P \wedge Q) \neq \emptyset$. This statement allows a specification implementing P and Q to be refined to a pair of tests for P and Q. This relies on the environment preserving P between $\mathsf{test}(P)$ and $\mathsf{test}(Q)$, so $\mathsf{test}(Q)$ is then guaranteed to start in a state satisfying P (and Q). For this to work, there must exists at least one state where P and Q both hold, hence $\mathsf{test}(P \wedge Q) \neq \emptyset$.

These rules are particularly useful as they allow one to define inference rules for the if statement and while loop, defined exactly as in KAT by

$$\text{if } B \text{ } \{ \text{ } X \text{ } \} \text{ else } \{ \text{ } Y \text{ } \} = \mathsf{test}(B); X \cup \mathsf{test}(\neg B); Y,$$
$$\text{while } B \text{ do } \{ \text{ } X \text{ } \} = (\mathsf{test}(B); X)^\star; \mathsf{test}(\neg B).$$

Of course this definition of while using the star is only suitable for partial correctness. For total correctness while would be defined as

$$\text{while } B \text{ do } \{ \text{ } X \text{ } \} = (\mathsf{test}(B); X)^\infty; \mathsf{test}(\neg B),$$

where the loop can be split into finite and infinite parts

$$(\mathsf{test}(B); X)^\star; \mathsf{test}(\neg B) \cup (\mathsf{test}(B); X)^\omega.$$

In general the approach here is to give a termination proof that $(\mathsf{test}(B); X)^\omega$ reduces to 0 under the influence of the rely condition, then allowing us to treat the loop as in the partial correctness world where termination is assumed. The inference rules for the if statement and (partial correctness) while loop are shown in Figure 8.1.

$$\frac{R \leq \vec{B} \qquad R, G \vdash \{P \wedge B\} X \{Q\} \qquad R, G \vdash \{P \wedge \neg B\} Y \{Q\}}{R, G \vdash \{P\} \text{if } B \text{ } \{ \text{ } X \text{ } \} \text{ else } \{ \text{ } Y \text{ } \} \{Q\}} \text{ If Statement}$$

$$\frac{R \leq \vec{I} \qquad R, G \vdash \{I \wedge P\} X \{I\}}{R, G \vdash \{I\} \text{while } P \text{ do } \{ \text{ } X \text{ } \} \{\neg P \wedge I\}} \text{ While (Partial)}$$

Figure 8.1: Inference rules for if and partial correctness while

8.3 Assignment Statements

The assignment statement is defined here as

$$v := E = \{(\sigma, \sigma')(\sigma', \sigma') \mid \sigma' = (v \leftarrow E)\, \sigma\}.$$

For a state σ the expression $(v \leftarrow E)\, \sigma$ modifies the value of v in σ such that it equals E. Here the exact nature of the state or store being used is left unspecified—in general it will change depending on the program being verified. This encoding differs from the encoding used in Chapter 6, as there the expression E was evaluated before the assignment statement as a test. This encoding is considerably more straightforward.

Common properties of program stores/heaps have been considered in the context of separation logic via *separation algebras* [COY07, DGS15] and also

algebraic separation logic [DHM11]. Such separation algebras have been mechanised in Isabelle by Klein et al. [KKB12a, KKB12b] and Dongol et al [DGS15]. In theory it should be possible to cleanly integrate them here, but for the purposes of simplicity it is typically assumed in the upcoming examples that the program store can be represented as a simple record of variables.

The assignment inference rule for rely/guarantee is

$$\mathsf{preserve}(Q[E/v]), G \vdash \{Q[E/v]\}\ v := E\ \{Q\}.$$

where

$$G = \{(\sigma, (v \leftarrow E)\,\sigma) \mid Q[E/v](\sigma)\}$$

The pre- and post-conditions are the same as in Hoare logic. The validity of the assignment rule depends on the environment not falsifying the precondition $Q[E/v]$ before the assignment executes. This gives the rely condition $\mathsf{preserve}(Q[E/v])$. The guarantee condition G requires slightly more explanation. The guarantee therefore guarantees that the program will start in a state satisfying $Q[E/v]$, and finish in that same state where v has been updated. It can guarantee that it starts in this state precisely because it relies on the fact that the environment will preserve the validity of $Q[E/v]$.

Note that this assignment rule is the 'correct way round', where the substitution occurs in the precondition as per Hoare logic, rather than the (confusingly) Floyd-style rule in Chapter 6. The assignment inference rule is equivalent to the following refinement rule derived as follows:

Theorem 37.

$$\mathsf{test}(Q[E/v]) \to \mathsf{preserve}(Q[E/v]) \rhd \mathsf{end}(Q) \cap \langle G\rangle \sqsubseteq v := E.$$

where $G = \{(\sigma, (v \leftarrow E)\,\sigma) \mid \sigma.\ Q[E/v](\sigma)\}$.

Proof. Assume $w \in v := E$. From the definition of assignment obtain σ and σ' where

$$w = (\sigma, \sigma')(\sigma', \sigma') \qquad \text{and} \qquad \sigma' = (v \leftarrow E)\,\sigma.$$

It must be shown that $w \in \mathsf{test}(Q[E/v]) \to \mathsf{preserve}(Q[E/v]) \rhd \mathsf{end}(Q) \cap \langle G\rangle$. Applying the Galois connection for preimplication (7.1) and the definition of w, this becomes

$$(\tau, \tau)(\sigma, \sigma')(\sigma', \sigma') \in \mathsf{preserve}(Q[E/v]) \rhd \mathsf{end}(Q) \cap \langle G\rangle$$

where τ is a state satisfying $Q[E/v]$. Given $Q[E/v]$ is inhabited, it can be shown that there must exist a (τ', τ') in $\mathsf{test}(Q)$. Now consider the two cases where $(\tau, \sigma) \in \mathsf{preserve}(Q[E/v])$ and $(\tau, \sigma) \notin \mathsf{preserve}(Q[E/v])$.

The first case is trivial as it can be shown that σ satisfies $Q[E/v]$ and therefore σ' satisfies Q. As such $(\tau,\tau)(\sigma,\sigma')(\sigma',\sigma')$ is in $\mathsf{end}(Q) \cap \langle G\rangle$—showing that (σ,σ') satisfies the guarantee is straightforward given how it is defined.

For the second case it can be shown that

$$(\tau,\tau)(\sigma,\sigma')(\sigma',\sigma') \simeq_{\mathsf{preserve}(Q[E/v])} (\tau,\tau)(\sigma,\sigma)(\sigma',\sigma')(\tau',\tau').$$

$(\tau,\tau)(\sigma,\sigma)(\sigma',\sigma')(\tau',\tau')$ is in $\mathsf{end}(Q) \cap \langle G\rangle$ as every step is an identity transition (and therefore in any guarantee), while (τ',τ') is in $\mathsf{test}(Q)$.

□

Using these rules for tests and assignments, it is possible to mechanically verify and refine concurrent programs within Isabelle. Examples of this are given in the following section.

8.4 Example: Find P

First, the state of the program (given previously in Figure 6.2) is defined as a record containing all the variables used in the program, as well as the array to be searched.

record store =
  s-f$_A$ :: nat
  s-f$_B$ :: nat
  s-i$_A$ :: nat
  s-i$_B$ :: nat
  s-fmin :: nat
  s-A :: nat list

A program is then a language of words containing store × store pairs.

type-synonym program = (store × store) llist set

The var datatype represents the set of possible variables. The array, s-A, is immutable so it is not a valid variable. Furthermore, the assignment axiom above has no notion of array indexing and therefore does not handle assignment to arrays.

datatype var = f$_A$ | f$_B$ | i$_A$ | i$_B$ | fmin

Rather than just giving definitions in Isabelle (which can be rather cryptic for the uninitiated), some of the definitions in this Chapter are given as Z specifications—although some of the quirks of Z notation are ignored in favour of more standard mathematical notation as used throughout this thesis. A Z specification for the above store is given in Figure 8.2. Such specifications are used throughout this chapter where the Isabelle notation is unwieldy or unclear.

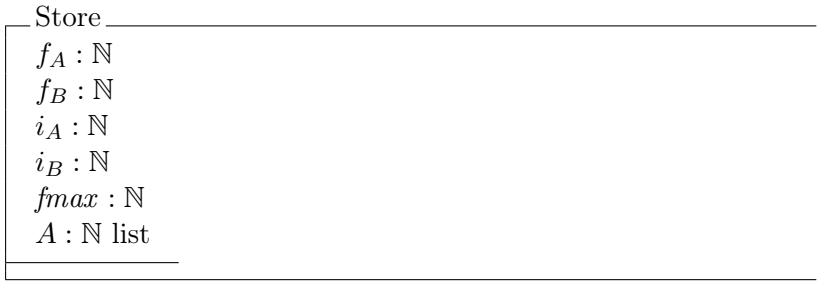


Figure 8.2: Z specification for the store

The function deref looks up the values of variables in the store

```
primrec deref :: var ⇒ store ⇒ nat where
  deref f_A s = s-f_A s
| deref f_B s = s-f_B s
| deref i_A s = s-i_A s
| deref i_A s = s-i_B s
| deref fmax s = s-fmax s
```

To implement the FINDP program, a datatype representing program expressions is needed

```
datatype expr =
    Const nat
  | Var var
  | ALen
  | Lookup expr
  | BinOp nat ⇒ nat ⇒ nat expr expr
  | Fun nat ⇒ nat expr
```

An expression can either be a constant value, a variable, the length of the array, a statement which looks up the result of an expression in the array, a binary operation applied to two expressions, or a unary operator applied to an expression. The following Isabelle function defines how expressions are evaluated.

```
primrec eval-expr :: expr ⇒ store ⇒ nat where
  eval-expr (Const c) s = c
| eval-expr (Var v) s = deref v s
| eval-expr ALen s = length (s-A s)
| eval-expr (Lookup x) s = s-A s ! eval-expr x s
| eval-expr (BinOp f x y) s = f (eval-expr x s) (eval-expr y s)
| eval-expr (Fun f x) s = f (eval-expr x s)
```

The expr datatype only gives expressions over natural numbers. For if statements and while loops boolean expressions are also needed. These are given by the bool-expr datatype.

datatype bool-expr =
    BTrue (**1**)
  | BFalse (**0**)
  | BDisj bool-expr bool-expr (**infixl** $\oplus$ 65)
  | BConj bool-expr bool-expr (**infixl** $\otimes$ 70)
  | BImpl bool-expr bool-expr
  | BIff bool-expr bool-expr
  | BNot bool-expr
  | BExpr1 nat $\Rightarrow$ bool expr
  | BExpr2 nat $\Rightarrow$ nat $\Rightarrow$ bool expr expr
  | BArray nat $\Rightarrow$ nat list $\Rightarrow$ bool expr

Boolean expressions can either be BTrue (true) or BFalse (false). They can be combined using the standard logical operations of disjunction (BDisj denoted $\oplus$), conjunction (BConj denoted $\otimes$), implication (BImpl) and if and only if (BIff). They can also be negated (BNot). Finally, the generators of a boolean expression are either unary predicates over expressions (BExpr1), binary predicates over expressions (BExpr2), or a predicate comparing an expression to the contents of the array. Given that bool-expr is essentially the absolutely free (or ground term) boolean algebra for the above generators, a much more minimalistic definition could be given, as many of the above operations can be defined in terms of one another. However, here simplicity is valued over concision, to ensure that everything works smoothly within Isabelle. Boolean expressions can be evaluated with the following function:

primrec eval-bool-expr :: bool-expr $\Rightarrow$ store $\Rightarrow$ bool **where**
  eval-bool-expr BTrue s = True
| eval-bool-expr BFalse s = False
| eval-bool-expr (BDisj x y) s = (eval-bool-expr x s $\vee$ eval-bool-expr y s)
| eval-bool-expr (BConj x y) s = (eval-bool-expr x s $\wedge$ eval-bool-expr y s)
| eval-bool-expr (BImpl x y) s = (eval-bool-expr x s $\rightarrow$ eval-bool-expr y s)
| eval-bool-expr (BIff x y) s = (eval-bool-expr x s $\leftrightarrow$ eval-bool-expr y s)
| eval-bool-expr (BNot x) s = ($\neg$ eval-bool-expr x s)
| eval-bool-expr (BExpr1 f x) s = f (eval-expr x s)
| eval-bool-expr (BExpr2 f x y) s = f (eval-expr x s) (eval-expr y s)
| eval-bool-expr (BArray p x) s = p (eval-expr x s) (s-A s)

The function which updates the store is defined in the obvious fashion as

primrec assign :: var $\Rightarrow$ expr $\Rightarrow$ store $\Rightarrow$ store (infix $\leftarrow$ 67) **where**
  ($f_A \leftarrow$ e) s = store.make (eval-expr e s) (s-f_B s) (s-i_A s) (s-i_B s) (s-fmax s) (s-A s)
| ($f_B \leftarrow$ e) s = store.make (s-f_A s) (eval-expr e s) (s-i_A s) (s-i_B s) (s-fmax s) (s-A s)
| ($i_A \leftarrow$ e) s = store.make (s-f_A s) (s-f_B s) (eval-expr e s) (s-i_B s) (s-fmax s) (s-A s)
| ($i_B \leftarrow$ e) s = store.make (s-f_A s) (s-f_B s) (s-i_A s) (eval-expr e s) (s-fmax s) (s-A s)
| (fmax $\leftarrow$ e) s = store.make (s-f_A s) (s-f_B s) (s-i_A s) (s-i_B s) (eval-expr e s) (s-A s)

The reason to define expressions in such a way, rather than simply using arbitrary expressions within Isabelle, is to enable easily computable substitution operations to be defined over them. For expressions the substitution function e-subst is defined as:

```
primrec e-subst :: expr ⇒ expr ⇒ var ⇒ expr where
  e-subst (Const c) e v = Const c
| e-subst (Var v') e v = (if v = v' then e else Var v')
| e-subst ALen e v = ALen
| e-subst (Lookup x) e v = Lookup (e-subst x e v)
| e-subst (BinOp f x y) e v = BinOp f (e-subst x e v) (e-subst y e v)
| e-subst (Fun f x) e v = Fun f (e-subst x e v)
```

The definition of substitution for boolean expressions is similar, except the nicer syntax of $x[e/v]$ is introduced for the substitution of e for the variable v in a boolean expression e.

```
primrec be-subst :: bool-expr ⇒ expr ⇒ var ⇒ bool-expr (-[-/-]) where
  be-subst BTrue e v = BTrue
| be-subst BFalse e v = BFalse
| be-subst (BDisj x y) e v = BDisj (x[e/v]) (y[e/v])
| be-subst (BConj x y) e v = BConj (x[e/v]) (y[e/v])
| be-subst (BImpl x y) e v = BImpl (x[e/v]) (y[e/v])
| be-subst (BIff x y) e v = BIff (x[e/v]) (y[e/v])
| be-subst (BNot x) e v = BNot (x[e/v])
| be-subst (BExpr1 f x) e v = BExpr1 f (e-subst x e v)
| be-subst (BExpr2 f x y) e v = BExpr2 f (e-subst x e v) (e-subst y e v)
| be-subst (BArray p x) e v = BArray p (e-subst x e v)
```

The following two lemmas are used to prove that the assignment rule holds, by relating the assign function with substitution.

lemma [simp]: eval-expr e ((x_v ← E) s) = eval-expr (e-subst e E x_v) s

lemma assign-eval-bool:
  eval-bool-expr (Q[E/x_v]) s ↔ eval-bool-expr Q ((x_v ← E) s)

8.5 The FINDP program

Now that the basic setup has been performed in defining expressions, substitution, and the store over which FINDP executes, the actual FINDP program can be introduced. The Isabelle code for the FINDP program from Figure 6.2 in Section 6.6 can be seen in Figure 8.3. Some additional syntactic sugar for ‖ has been defined as

$$\text{parallel } \{X\} \text{ meanwhile } \{Y\} = X \| Y,$$

so that parallel compositions may be laid across multiple lines in a more readable manner. In the FINDP program, the first parallel subprogram is called subprogram A, and the second is subprogram B.

```
definition FINDP :: (nat → bool) → program where
FINDP P ≡
f_A := ALen;
f_B := ALen;
parallel {
  i_A := Const 0;
  while (i_A less than f_A) ⊗ (i_A less than f_B) do {
    if BExpr1 P (Lookup (Var i_A)) {
      f_A := Var i_A
    } else {
      i_A := Var i_A ADD Const 2
    }
  }
} meanwhile {
  i_B := Const 1;
  while (i_B less than f_A) ⊗ (i_B less than f_B) do {
    if BExpr1 P (Lookup (Var i_B)) {
      f_B := Var i_B
    } else {
      i_B := Var i_B ADD Const 2
    }
  }
};
fmin := MIN (Var f_A) (Var f_B)
```

Figure 8.3: The FINDP program in Isabelle

Before diving into the refinement proof, the loop invariants for both the while loops within the parallel composition are given below:

```
definition invariant-A P ≡
  BArray (λi arr. ∀i'. i' < i ∧ even i' → ¬ P (arr ! i')) (Var i_A)
  ⊗ BExpr1 even (Var i_A)
  ⊗ BArray (λi arr. i < length arr → P (arr ! i)) (Var f_A)
  ⊗ BExpr2 (op <) (Var f_A) (BinOp (op +) ALen (Const 1))
```

definition invariant-B P ≡
BArray (λi arr. ∀i'. i' < i ∧ odd i' → ¬ P (arr ! i')) (Var i_B)
⊗ BExpr1 odd (Var i_B)
⊗ BArray (λi arr. i < length arr → P (arr ! i)) (Var f_B)
⊗ BExpr2 (op <) (Var f_B) (BinOp (op +) ALen (Const 1))

Written more readably as a Z specification, invariant-B becomes:

invariant-B[P]

Store

$\forall i.\ i < i_B \wedge \mathsf{odd}(i) \implies \neg P(A\ !\ i)$
$\mathsf{odd}(i_B)$
$f_B < \mathsf{length}(A) \implies P(A\ !\ f_B)$
$f_B < \mathsf{length}(A) + 1$

In words, it states that the loop in subprogram B preserves the fact that all odd indices of the array less than the current index do not satisfy P, that the index is always odd, that f_B is never greater than the length of the array, and that if it is less than the length of the array, then the value of the array at index f_B satisfies P. The invariant for subprogram A is symmetric, so no Z specification for it is provided.

After the loops both terminate, and therefore the parallel composition, the following test will hold:

definition loop-post P ≡
BExpr2 (op <) (BinOp min (Var f_A) (Var f_B)) (BinOp (op +) ALen (Const 1))
⊗ BArray (λfmax arr. ∀i. i < fmax → ¬ P (arr ! i)) (BinOp min (Var f_A) (Var f_B))
⊗ BArray (λfmax arr. fmax < length arr → P (arr ! fmax))
(BinOp min (Var f_A) (Var f_B))

as a Z specification:

loop-post[P]

Store

$\mathsf{min}(f_A, f_B) < \mathsf{length}(A) + 1$
$\forall i.i < \mathsf{min}(f_A, f_B) \implies \neg P(A\ !\ i)$
$\mathsf{min}(f_A, f_B) < \mathsf{length}(A) \implies P(A\ !\ \mathsf{min}(f_A, f_B))$

In words, this states that after the loop executes, the minimum of f_A and f_B, $\mathsf{min}(f_A, f_B)$, will be either equal to the length of the array or less than it. There can be no indices into the array less than this value for which P holds. If it is less than the length of the array, then it is an index into the array at a point where P holds.

The final postcondition for FINDP is as follows:

definition post P ≡
BExpr2 (op <) (Var fmin) (BinOp (op +) ALen (Const 1))
⊗ BArray (λfmin arr. ∀i. i < fmin → ¬ P (arr ! i)) (Var fmin)
⊗ BArray (λfmin arr. fmin < length arr → P (arr ! fmin)) (Var fmin)

It is the same as loop-post P except the variable fmin replaces the minimum value of f_A and f_B.

8.6 FINDP refinement proof

The statement of the FINDP refinement proof in Isabelle is as follows:

theorem findp-refine: $\mathcal{T}\ \mathbf{0} \rightarrow (\{\} \rhd \mathcal{F} \cdot \mathcal{T}$ (post P) ∩ guar ⊤) ⊑ FINDP P
(is ?specification ⊑ -)
proof −

Recall that $\mathcal{T}\ P \rightarrow (R \rhd \mathcal{F} \cdot \mathcal{T}\ Q \cap$ guar $G)$ is the specification for a program that begins in a state P and terminates in a state Q guaranteeing G given an environment satisfying R. The syntax here is slightly different from previously due to Isabelle's syntax. $\mathcal{T}\ P$ is the same as $\mathsf{test}(P)$ and $\mathcal{F} \cdot \mathcal{T}\ Q$ is $\mathsf{end}(Q)$. This theorem states that we need to prove that Figure 8.3 is a refinement of the specification that implements post P from any precondition under no interference, and guarantees nothing (i.e. it might produce arbitrary interference).

The proof begins by using the refinement rule for sequential composition followed by the assignment rule.

have ?specification ⊑
$\mathcal{T}\ \mathbf{0} \rightarrow (\{\} \rhd \mathcal{F} \cdot \mathcal{T}$ (Var f_A = ALen) ∩ guar ⊤);
$\mathcal{T}$ (Var f_A = ALen) → ({} ▷ $\mathcal{F} \cdot \mathcal{T}$ (post P) ∩ guar ⊤)
by refine
also have ... ⊑
f_A := ALen;
$\mathcal{T}$ (Var f_A = ALen) → ({} ▷ $\mathcal{F} \cdot \mathcal{T}$ (post P) ∩ guar ⊤)
by verify

The refine and verify tactics are simple tactics written in Isabelle's *Eisbach* tactic language [MWM14]. The refine tactic simply applies refinement rules wherever it can, whereas verify is used to refine specifications into assignments and attempts to discharge the preconditions of the assignment rule. This first step introduces the variable f_A initialised to the length of the array. This is then repeated for the variable f_B.

also have ... ⊑
f_A := ALen;
$\mathcal{T}$ (Var f_A = ALen) → ({} ▷ $\mathcal{F} \cdot \mathcal{T}$ (Var f_A = ALen ⊗ Var f_B = ALen) ∩ guar ⊤);
$\mathcal{T}$ (Var f_A = ALen ⊗ Var f_B = ALen) → ({} ▷ $\mathcal{F} \cdot \mathcal{T}$ (post P) ∩ guar ⊤)
by refine

also have ... $\sqsubseteq$
  f_A := ALen;
  f_B := ALen;
  $\mathcal{T}$ (Var f_A = ALen $\otimes$ Var f_B = ALen) $\rightarrow$ ({} $\rhd$ $\mathcal{F} \cdot \mathcal{T}$ (post P) $\cap$ guar $\top$)
  by verify

Next, the final assignment statement fmax := MIN (Var f_A) (Var f_B) is implemented by showing that it goes from the postcondition of the loop, loop-post P, to the final postcondition post P. This leaves the specification

$$\mathcal{T}\ (\text{Var } \mathrm{f}_A = \text{ALen} \otimes \text{Var } \mathrm{f}_B = \text{ALen}) \rightarrow (\{\} \rhd \mathcal{F} \cdot \mathcal{T}\ (\text{loop-post P}) \cap \text{guar } \top)$$

for the parallel part of the FINDP program.

also have ... $\sqsubseteq$
  f_A := ALen; f_B := ALen;
  $\mathcal{T}$ (Var f_A = ALen $\otimes$ Var f_B = ALen) $\rightarrow$ ({} $\rhd$ $\mathcal{F} \cdot \mathcal{T}$ (loop-post P) $\cap$ guar $\top$);
  $\mathcal{T}$ (loop-post P) $\rightarrow$ ({} $\rhd$ $\mathcal{F} \cdot \mathcal{T}$ (post P) $\cap$ guar $\top$)
  by refine
also have ... $\sqsubseteq$
  f_A := ALen;
  f_B := ALen;
  $\mathcal{T}$ (Var f_A = ALen $\otimes$ Var f_B = ALen) $\rightarrow$ ({} $\rhd$ $\mathcal{F} \cdot \mathcal{T}$ (loop-post P) $\cap$ guar $\top$);
  fmax := MIN (Var f_A) (Var f_B)
  by (verify add: loop-post-def post-def)

The next step is to introduce the parallel parts of the FINDP program. The lemma findp-parallel-post states that the overall postcondition, loop-post P, of the parallel composition can be derived from the individual postconditions of the parallel subprograms (which can be seen in the code below). The specifications for both parallel subprograms are defined as schematic variables ?specification-A and ?specification-B respectively. The rely and guarantee conditions rely-A[P] and rely-B[P] are also introduced. The rely condition for one parallel subprogram is the guarantee of the other. A Z specification describing rely-A is given in Figure 8.4. Given that the two subprograms are symmetric, the specification for rely-B would be similar, just replacing i_A with i_B, odd with even and so on. In Isabelle it was checked that these specifications were somehow minimal by removing parts of the specification and attempting to run the proof script without them.

In words, each subprogram depends on its environment not modifying its own internal variables—for subprogram A this amounts is f_A and i_A being unchanged. The array A is immutable, so the environment cannot change it. The environment then guarantees that either it remains in a state where f_B is equal to the length of A (when it has not found anything satisfying P), or it transitions to a state such that $A' \,!\, f'_B$ satisfies P and for every odd index v less than f'_B, $A' \,!\, v$ does not satisfy P. In this case f'_B must be less than f_B and f'_B is set to i_B (which itself remains unchanged). In Isabelle syntax rely-A P is rely-A[P], and similarly for rely-B[P].

rely-A$_1[P]$

ΔStore

$A' = A$
$f'_A = f_A$
$i'_A = i_A$
$\forall v < f'_B.\ \mathsf{odd}(v) \implies \neg P(A' \,!\, v)$
$f'_B < f_B$
$f'_B = i_B$
$i'_B = i_B$

rely-A$_2[P]$

ΔStore

$A' = A$
$f'_A = f_A$
$i'_A = i_A$
$i'_B < i_B$
$f_B = \mathsf{length}(A) \implies f'_B = \mathsf{length}(A')$

$$\text{rely-A}[P] \quad \widehat{=} \quad (\text{rely-A}_1[P] \ \vee \ \text{rely-A}_2[P])^*$$

Figure 8.4: Z specification for rely-A

also have ... $\sqsubseteq$
  f_A := ALen;
  f_B := ALen;
  parallel {
    $\mathcal{T}$ (Var f_A = ALen) $\rightarrow$
    (rely-A P $\rhd$ $\mathcal{F} \cdot \mathcal{T}$ (BNot ((i_A < f_A) $\otimes$ (i_A < f_B)) $\otimes$ inv-A P) $\cap$ guar (rely-B P))
  } meanwhile {
    $\mathcal{T}$ (Var f_B = ALen) $\rightarrow$
    (rely-B P $\rhd$ $\mathcal{F} \cdot \mathcal{T}$ (BNot ((i_B < f_A) $\otimes$ (i_B < f_B)) $\otimes$ inv-B P) $\cap$ guar (rely-A P))
  };
  fmax := MIN (Var f_A) (Var f_B)
  (is -$\sqsubseteq$ -; -; parallel { ?specification-A } meanwhile { ?specification-B }; -)
  by (refine add: findp-parallel-post)

Now it is shown that ?specification-A can be refined into the concrete parallel subprogram par-A P.

also have ... $\sqsubseteq$
  f_A := ALen;
  f_B := ALen;
  parallel {
    par-A P
  } meanwhile {
    $\mathcal{T}$ (Var f_B = ALen) $\rightarrow$
    (rely-B P $\rhd$ $\mathcal{F} \cdot \mathcal{T}$ (BNot (($i_B < f_A$) $\otimes$ ($i_B < f_B$)) $\otimes$ inv-B P) $\cap$ guar (rely-A P))
  };
  fmax := MIN (Var f_A) (Var f_B)

This is done as a separate subproof. First the isotonicity tactic is applied to focus on the relevant part of the program that needs to be refined. The schematic variable ?if-pre defines a weakened precondition used in this refinement proof. A more readable Z specification of ?if-pre is given in Figure 8.5.

proof isotonicity
  let ?if-pre =
      BArray (λi arr. $\forall$i'. i' < i $\wedge$ even i' $\longrightarrow$ $\neg$ P (arr ! i')) (Var i_A) $\otimes$
      BExpr1 even (Var i_A) $\otimes$
      BArray (λi arr. i < length arr $\longrightarrow$ P (arr ! i)) (Var f_A) $\otimes$
      BExpr2 op < (Var f_A) (BinOp op + ALen (Const (Suc 0))) $\otimes$
      BExpr2 op < (Var i_A) (Var f_A)

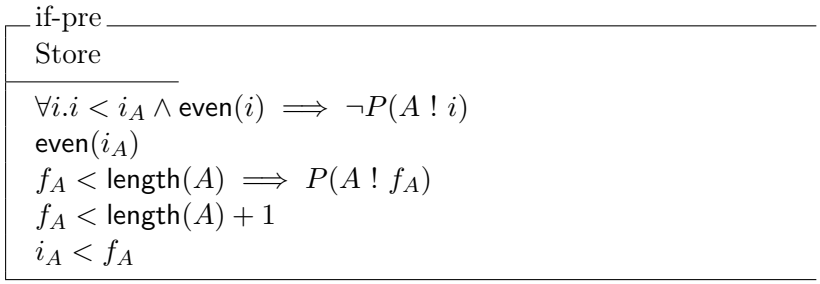


Figure 8.5: Z specification for ?if-pre for A

The first part of this subproof is to refine the assignment statement from Figure 8.3 that introduces and initialises the loop index variable i_A to zero.

  have ?specification-A $\sqsubseteq$
    $\mathcal{T}$ (Var f_A = ALen) $\rightarrow$
      (rely-A P $\rhd$ $\mathcal{F} \cdot \mathcal{T}$ (Var f_A = ALen $\otimes$ Var i_A = Const 0) $\cap$ guar (rely-B P));
    $\mathcal{T}$ (Var f_A = ALen $\otimes$ Var i_A = Const 0) $\rightarrow$
      (rely-A P $\rhd$ $\mathcal{F} \cdot \mathcal{T}$ (BNot ($i_A < f_A \otimes i_A < f_B$) $\otimes$ inv-A P) $\cap$ guar (rely-B P))
    by refine

```
also have ... ⊑
  i_A := Const 0;
  𝒯 (Var f_A = ALen ⊗ Var i_A = Const 0) →
    (rely-A P ▷ ℱ · 𝒯 (BNot (i_A < f_A ⊗ i_A < f_B) ⊗ inv-A P) ∩ guar (rely-B P))
  by (verify add: subset-iff preserve-def rely-A-def
                  store.defs rely-B-def test-def Language.test-def)
```

Notice that now the verify tactic requires more arguments than previously when it was called with no arguments at all—this is because we are now refining in the context of a parallel subprogram where the rely and guarantee condition must be taken into consideration. The next step is to introduce the while loop. This requires a more involved proof, as the preconditions for the while rule must be discharged.

```
also have ... ⊑
  i_A := Const 0;
  while (i_A < f_A ⊗ i_A < f_B) do {
    𝒯 (inv-A P ⊗ (i_A < f_A ⊗ i_A < f_B)) →
      (rely-A P ▷ ℱ · 𝒯 (inv-A P) ∩ guar (rely-B P))
  }
  apply refine
  apply (simp add: inv-A-def)
  apply (rule rtrancl-mono)
  apply (simp add: preserve-def inv-A-def subset-iff rely-A-def)
  apply (simp-all only: state-existence)
  apply (simp add: lessthan-def inv-A-def)
  apply (rule-tac x = store.make 0 0 0 0 0 [] in exI)
  apply (simp add: store.defs)
  apply (simp add: lessthan-def inv-A-def)
  apply (rule-tac x = store.make 1 1 0 0 0 [1] in exI)
  by (simp add: store.defs)
```

The precondition of the loop body specification is now weakened to ?if-pre before the if statement within the while loop is introduced. For this the custom Eisbach weakening tactic is used.

```
also have ... ⊑
  i_A := Const 0;
  while (i_A < f_A ⊗ i_A < f_B) do {
    𝒯 ?if-pre → (rely-A P ▷ ℱ · 𝒯 (inv-A P) ∩ guar (rely-B P))
  }
  by weakening (simp add: inv-A-def lessthan-def)
```

Now the body of the loop can be refined to the if statement. This step also requires a slightly more involved proof.

```
also have ... ⊑
  i_A := Const 0;
  while (i_A < f_A ⊗ i_A < f_B) do {
    if BExpr1 P (Lookup (Var i_A)) {
      𝒯 (?if-pre ⊗ BExpr1 P (Lookup (Var i_A))) →
```

```
        (rely-A P ▷ 𝓕 · 𝓣 (inv-A P) ∩ guar (rely-B P))
    } else {
      𝓣 (?if-pre ⊗ BNot (BExpr1 P (Lookup (Var i_A)))) →
        (rely-A P ▷ 𝓕 · 𝓣 (inv-A P) ∩ guar (rely-B P))
    }
  }
  apply refine
  apply (rule rtrancl-mono)
  apply (simp add: rely-A-def preserve-def subset-iff)
  apply (simp add: state-existence inv-A-def)
  apply (rule-tac x = store.make 0 0 0 0 0 0 [] in exI)
  by (simp add: store.defs)
```

The postcondition for the first branch of the if statement is then strengthened slightly to allow it to be refined into the required assignment statement. Again the weakening tactic is used—despite the name it can be used to weaken preconditions or strengthen postconditions.

```
also have ... ⊑
  i_A := Const 0;
  while (i_A < f_A ⊗ i_A < f_B) do {
    if BExpr1 P (Lookup (Var i_A)) {
      𝓣 (?if-pre ⊗ BExpr1 P (Lookup (Var i_A))) →
        (rely-A P ▷ 𝓕 · 𝓣 (inv-A P ⊗ BExpr1 P (Lookup (Var i_A))) ∩ guar (rely-B P))
    } else {
      𝓣 (?if-pre ⊗ BNot (BExpr1 P (Lookup (Var i_A)))) →
        (rely-A P ▷ 𝓕 · 𝓣 (inv-A P) ∩ guar (rely-B P))
    }
  }
  by weakening simp
```

The first branch of the if statement is now refined into the assignment $f_A := \text{Var } i_A$.

```
also have ... ⊑
  i_A := Const 0;
  while (i_A < f_A ⊗ i_A < f_B) do {
    if BExpr1 P (Lookup (Var i_A)) {
      f_A := Var i_A
    } else {
      𝓣 (?if-pre ⊗ BNot (BExpr1 P (Lookup (Var i_A)))) →
        (rely-A P ▷ 𝓕 · 𝓣 (inv-A P) ∩ guar (rely-B P))
    }
  }
  apply (verify add: rely-A-def preserve-def subset-iff inv-A-def
                     test-def Language.test-def rely-B-def store.defs)
  by auto
```

The second branch of the if statement can then be refined into the assignment statement $i_A := \text{Var } i_A \text{ ADD Const } 2$.

```
  also have ... ⊑
    i_A := Const 0;
    while (i_A < f_A ⊗ i_A < f_B) do {
      if BExpr1 P (Lookup (Var i_A)) {
        f_A := Var i_A
      } else {
        i_A := Var i_A ADD Const 2
      }
    }
    apply (verify add: rely-A-def preserve-def subset-iff inv-A-def
                       test-def Language.test-def rely-B-def store.defs)
    using less-Suc-eq by auto
```

This completes the subproof for subprogram A.

```
  also have ... ⊑ par-A P
    by (simp add: par-A-def)
  finally show ?specification-A ⊑ par-A P .
qed
```

The same proof can then be repeated for subprogram B. Since the proof is effectively the same as above, for the purposes of this refinement it has been taken out and proven in the separate lemma par-B-proof.

```
also have ... ⊑
  f_A := ALen;
  f_B := ALen;
  parallel {
    par-A P
  } meanwhile {
    par-B P
  };
  fmax := MIN (Var f_A) (Var f_B)
  by isotonicity (auto intro: verify par-B-proof)
```

Expanding the definitions of par-A P and par-B P one arrives at the final form of the FINDP program from Figure 8.3.

```
also have ... ⊑
  f_A := ALen;
  f_B := ALen;
  parallel {
    i_A := Const 0;
    while (i_A < f_A) ⊗ (i_A < f_B) do {
      if BExpr1 P (Lookup (Var i_A)) {
        f_A := Var i_A
      } else {
        i_A := Var i_A ADD Const 2
      }
    }
  } meanwhile {
    i_B := Const 1;
```

```
    while (i_B < f_A) ⊗ (i_B < f_B) do {
      if BExpr1 P (Lookup (Var i_B)) {
        f_B := Var i_B
      } else {
        i_B := Var i_B ADD Const 2
      }
    }
  };
  fmin := MIN (Var f_A) (Var f_B)
  by (simp add: par-A-def par-B-def)
also have ... ⊑ findp P
  by (simp add: findp-def)
```

Finally, all the previous steps are collated into the overall goal of the theorem.

```
finally show ?specification ⊑ FINDP P .
qed
```

which completes the proof. □

8.7 Conclusion

In this chapter the model given in Section 7.3 was expanded with the requisite notions of tests and assignments for refining concrete programs. The model is easily powerful and extensive enough, that many further program constructs could be defined and use. In this section, only the minimum amount was formalised so as to refine the FINDP program. The experience of using Isabelle in the above refinement proof is overall very positive. Custom tactics allow much of the reasoning to be abstracted away, and many of the proof steps can be discharged via simple calls to the refine or verify tactics. These tactics are not hard to implement—compared to the development of tactics in Isabelle/ML, as seen in Section 3.11, Eisbach makes the development of such tactics available to even the novice Isabelle user. While verification condition generator tactics, such as those in Chapter 5, could previously take several days to build and debug, the tactics used above were developed over the span of less than an hour. Custom tactics can therefore easily be created to support every new program refinement task.

In addition to custom tactics, Isabelle takes care of handling the state of the proof, preventing any mistakes from being made. Once the core complexity of the refinement task has been achieved, which is mostly identifying the correct loop invariants and interference constraints, the refinement can proceed straightforwardly. Even the more complex apply-style steps in the above refinement rarely amount to calling anything other than either the simplifier/auto, or using sledgehammer.

Chapter 9

Conclusion

9.1 General Contribution

This thesis elaborates on a general approach using algebra in Isabelle for developing program verification and construction tools within Isabelle. First, SKAT has been mechanised as a case study in the use of algebra in the verification of simple while programs. The increased generality over Hoare logic was demonstrated via the formalisation of a complex flowchart equivalence proof. Such a proof would not be possible using only Hoare logic. Furthermore it was shown that the use of algebra allows for the derivation of new programming constructs and inference rules for them—a useful feature in practical applications.

Second, this approach was used to conceive a rely-guarantee algebra with a minimalistic set of axioms for a rely-guarantee calculus. Based on these axioms a prototypical tool was constructed based on a simple finite language model. Note that due to a rather limited treatment of interference, this algebra does not quite capture the full expressivity of the rely-guarantee method. Nevertheless, it remains suitable for simple verification tasks.

Third, constructs were added to the rely-guarantee algebra to support fine grained reasoning about interference in concurrent programs. Again, a minimalistic set of axioms was investigated for these new operations, and it was shown that a complete set of inference rules for rely-guarantee could be derived from them.

Fourth, a detailed model based on infinite languages was presented. Proving that this model satisfies the laws of the rely-guarantee algebra constitutes a soundness proof for the rely-guarantee calculus arising from the axioms of the rely-guarantee algebra. Mechanising this model within Isabelle proved to be quite tricky. Properties of the shuffle operator alone such as associativity are rather tricky to prove. Combining this with interference results in quite difficult proofs, such as in Chapter 7. This complexity is demonstrated by the proof of the main parallelism rule for the algebra in Chapter 7. Note the level of detail required by the interactive theorem prover tends to make such proofs much harder to mechanically verify than to prove them by hand. Section 9.3 contains information on the overall scope and size of the Isabelle formalisation, as well as conclusions on the overall use of Isabelle and interactive theorem proving in general.

Finally, the rely-guarantee algebra and the infinite language model was applied to a well known example from the literature: the FINDP program. This program was constructed via refinement within Isabelle. Overall, the derivation of this program within Isabelle was quite smooth, and greatly assisted by Isabelle's new Eisbach tactic language—a feature not used elsewhere in this thesis due to its recent addition in Isabelle 2015. While the verification tools given throughout this thesis are perhaps prototypical in nature, I believe they offer some insight into the use of interactive theorem proving tools in program verification and construction tasks.

9.2 Future Work

In this section some possible directions for future work are listed:

- Despite having a powerful infinite trace based model mechanised in Isabelle, which fully supports reasoning about non-termination, relatively little attention has been given to this issue in this thesis. This is partly because the example given in Chapter 8 is clearly terminating, and therefore suitable for a partial correctness approach.
- There are some limitations inherent in the idempotent semiring based algebraic approach for program verification. A notable example of this is that it is hard to reason about local state. Typical Morgan style refinement-calculi usually have some notion of a *frame*, allowing for reasoning about local state. Integrating separation logic style approaches could perhaps be an interesting way to work around this problem, and is a topic of current research [Vaf08]. Furthermore constructs such as procedure calls are not straightforward to encode within a Kleene-algebra based approach.
- It would be very interesting to investigate non-traditional models for rely-guarantee. The approach in Section 7.3 is very much inspired by Brookes' transition traces. It would nicely validate the algebraic approach, if the axioms given in Chapter 7 were shown to be applicable for such models. Those models, could, for instance, be based on partially ordered multisets (pomsets), automata, or even more esoteric models.
- While FINDP is a commonly used example, and therefore useful for comparison purposes, it is not a particularly complicated example. More complicated examples could be formalised within Isabelle. Doing so would likely suggest areas for improvement in terms of automation for verification and refinement tasks within Isabelle.

9.3 Experience with Isabelle

Overall the construction of both the infinite traces model and FINDP example in Chapters 6, Chapter 7 and Chapter 8 constitutes just over 10,000 lines of Isabelle proof script. As of the time of writing this constitutes around 933 separate theorems and lemmas and 110 unique definitions. As mentioned in Section 2.3 this is substantially longer than previous implementations of the rely-guarantee method in Isabelle [Nie01]. Mostly this is due to the difficulty of implementing a denotational semantics as opposed to an operational one. However, this approach has many advantages. First, adding a new construct to an operational semantics would require most proofs to be updated

with additional cases to handle the new construct. By contrast, new constructs can be seamlessly added to the algebra/language in this thesis, and no existing properties in the model would need to change. In this sense, the approach here is more extensible and modular. Second, the approach in this thesis offers arbitrary nested parallelism and supports proof by refinement, neither of which were present in previous mechanisations.

The formalisation of SKAT with Angus and Kozen's algebraic proof of Manna's flowchart scheme equivalence constitutes around 8800 lines of Isabelle proof script. Part of this length comes from the custom Kleene Algebra and Boolean algebra with explicit carrier sets libraries. These libraries are more expressive (but harder to work with) than those in the AFP, and were developed as part of this formalisation. A large amount of the length of this formalisation effort comes from the flowchart equivalence proof, which constitutes around 3300 lines. This could probably be shortened with better automation, as mechanisations in Coq have shown [Pou13b].

Developed along with various co-authors, libraries for variants of idempotent semirings, Kleene algebra, and Kleene algebra with tests were developed as part of the work involved in this thesis [ASW13a, AGS14b]. These represent in total around 7900 lines of Isabelle proof script.

Overall, the experience of using Isabelle as part of this thesis has been overwhelmingly positive. While mechanising certain tricky properties can at times be extremely difficult compared to paper proofs, the confidence Isabelle provides is certainly worthwhile. Even when mechanising such hard properties, using Isabelle to guide one's proof search to ensure no missteps are made is often worthwhile, despite the difficulties. There are in fact many areas where the use of an automated theorem proving tool makes one's life considerably easier than using pen and paper. For example, in the case of the FINDP example, Isabelle can keep track of all the assumptions and state inherent in the proof; and automation allows tedious and repetitive parts of the refinement to be blasted away. This is perhaps where interactive theorem proving tools shine brightest—when proofs are simple, but otherwise repetitive and long winded. This is certainly the case for a lot of program verification tasks, but tends to be less the case for concurrent programs. However, once the core difficulty of the concurrent interaction/interference involved has been identified and formalised, proofs of concurrent programs can largely occur straightforwardly, as in the sequential case. For example, in the FINDP proof, once the correct invariants and interference constraints are identified, the proof is straightforward and largely automatic.

9.4 Final Conclusion

Overall this thesis uses algebras based on idempotent semirings to develop tools for the verification and refinement of concurrent programs within Is-

abelle/HOL. This goes further than previous work, both within and without interactive theorem proving tools. It does this by providing a complete mechanised denotational semantics for rely-guarantee, backed with a convenient algebraic layer for the derivation of inference and refinement rules. It has been shown that this approach and mechanisation is applicable for concrete refinement tasks. Despite being somewhat prototypical in nature, I believe that this approach offers many advantages for the development of concurrent algorithms.

Appendix A

Derivation of the Rely-Guarantee Inference Rules

Theorem 38. *The sequential rule*

$$\frac{r, g \vdash \{p\}x\{q\} \qquad r, g \vdash \{q\}y\{s\}}{r, g \vdash \{q\}xy\{s\}}$$

can be derived in the algebra.

Proof. Translated via (7.13) the assumptions

$$r, g \vdash \{p\}x\{q\} \qquad \text{and} \qquad r, g \vdash \{q\}y\{s\}$$

become

$$x \leq p \to r \rhd \langle g \rangle \sqcap q \qquad \text{and} \qquad y \leq q \to r \rhd \langle g \rangle \sqcap q$$

To prove the rule, it must be shown that

$$xy \leq (p \to r \rhd \langle g \rangle \sqcap q)(q \to r \rhd \langle g \rangle \sqcap q) \leq (p \to r \rhd \langle g \rangle \sqcap s)$$

The first inequality is trivial, for the second, the Galois connection for preimplication is used, and then the proof can be achieved via straightforward equational reasoning as follows:

$$\begin{aligned}
& p(p \to r \rhd \langle g \rangle \sqcap q)(q \to r \rhd \langle g \rangle \sqcap q) \\
\leq\ & (r \rhd \langle g \rangle \sqcap q)(q \to r \rhd \langle g \rangle \sqcap q) \\
\leq\ & r \rhd (\langle g \rangle \sqcap q)(q \to r \rhd \langle g \rangle \sqcap q) \\
\leq\ & r \rhd q(q \to r \rhd \langle g \rangle \sqcap q) \\
\leq\ & r \rhd r \rhd \langle g \rangle \sqcap q \\
\leq\ & r \rhd r \rhd \langle g \rangle \sqcap q
\end{aligned}$$

□

Theorem 39. *The choice rule*

$$\frac{r, g \vdash \{p\}x\{q\} \qquad r, g \vdash \{p\}y\{q\}}{r, g \vdash \{p\}x + y\{q\}}$$

can be derived in the algebra.

Proof. This rule is trivial as

$$x + y \leq (p \to r \rhd \langle g \rangle \sqcap q) + (p \to r \rhd \langle g \rangle \sqcap q) \leq (p \to r \rhd \langle g \rangle \sqcap q).e$$

□

Theorem 40. *The skip rule*

$$\frac{p \leq \langle g \rangle}{r, g \vdash \{p\}1\{p\}}$$

can be derived in the algebra.

Proof. This rule is also trivial as

$$1 \leq p \to r \rhd \langle g \rangle \sqcap p \iff p \leq r \rhd \langle g \rangle \sqcap p \leq r \rhd p.$$

□

Theorem 41. *The star rule*

$$\frac{r, g \vdash \{p\} x \{p\}}{r, g \vdash \{p\} x^\star \{p\}}$$

can be derived in the algebra.

Proof. We must show that

$$x^\star \leq p \to r \rhd \langle g \rangle \sqcap p.$$

To do so we use the special case of the Kleene algebra induction axioms

$$xy \leq y \implies x^\star \leq y,$$

We must then prove

$$\begin{aligned} & x(p \to r \rhd \langle g \rangle \sqcap p) \\ \leq\ & (p \to r \rhd \langle g \rangle \sqcap p)(p \to r \rhd \langle g \rangle \sqcap p) \\ \leq\ & (p \to r \rhd \langle g \rangle \sqcap p). \end{aligned}$$

The first inequality is just applying the assumption and the second is a special case of the sequential rule where p, q and s are equal. □

Theorem 42. *Assuming the test p satisfies*

$$p \cdot p \to r \rhd p = p \to r \rhd p$$

The omega rule

$$\frac{r, g \vdash \{p\} x \{p\} \qquad r, g \vdash \{p\} x^\omega \{x^\star\}}{r, g \vdash \{p\} x^\omega \{p\}}$$

can be derived in the algebra.

Proof. We must show that

$$\begin{aligned} x^\omega &\leq p \to r \rhd \langle g \rangle \sqcap x^\star \\ &\leq p \to r \rhd \langle g \rangle \sqcap (p \to r \rhd \langle g \rangle \sqcap p) \\ &\leq p \to r \rhd (p \to r \rhd \langle g \rangle \sqcap p) \\ &\leq p \to p \to r \rhd r \rhd \langle g \rangle \sqcap p \\ &\leq pp \to r \rhd \langle g \rangle \sqcap p \\ &\leq p \to r \rhd \langle g \rangle \sqcap p \end{aligned}$$

□

Appendix B

List of Algebraic Structures

Definition 5. A *semigroup* is an algebraic structure $(S, \cdot)$ where

$$(x \cdot y) \cdot z = x \cdot (y \cdot z).$$

Definition 6. A *monoid* is an algebraic structure $(M, \cdot, 1)$ where:

- $(M, \cdot)$ is a semigroup,
- 1 is an identity element such that $1 \cdot x = x = x \cdot 1$ holds for all x.

Definition 7. A *commutative monoid* is a monoid $(M, \cdot)$ where $x \cdot y = y \cdot x$.

Definition 8. A *semiring* is an algebraic structure $(S, +, \cdot, 0, 1)$ where:

- $(S, +, 0)$ is a commutative monoid,
- $(S, \cdot, 1)$ is a monoid,
- the distributive laws $x \cdot (y + z) = x \cdot y + x \cdot z$ and $(x + y) \cdot z = x \cdot z + y \cdot z$ hold for all x, y, and z,
- the annihilation laws $0 \cdot x = 0$ and $x \cdot 0 = 0$ hold for all x.

Definition 9. A *weak semiring* is a semiring in which is the right annihilation law $x \cdot 0$ does not hold.

Definition 10. An *idempotent semiring* or *dioid* is a semring where addition is idempotent, i.e.

$$x + x = x.$$

Definition 11. A *commutative dioid* is a dioid $(S, +, \cdot, 0, 1)$ where $x \cdot y = y \cdot x$.

Definition 12. A *Kleene algebra* $(S, +, \cdot, 0, 1, ^\star)$ is a dioid expanded with a star operation satisfying both the *left unfold axiom*

$$1 + x \cdot x^\star \leq x^\star$$

and *left* and *right induction axioms*

$$z + x \cdot y \leq y \Rightarrow x^\star \cdot z \leq y \qquad \text{and} \qquad z + y \cdot x \leq y \Rightarrow z \cdot x^\star \leq y.$$

Definition 13. A *bi-Kleene algebra* is a structure $(K, +, \cdot, \|, 0, 1, ^\star, ^{(\star)})$ where

- $(K, +, \cdot, 0, 1, ^\star)$ is a Kleene algebra,
- $(K, +, \|, 0, 1, ^{(\star)})$ is a commutative Kleene algebra.

Definition 14. A *concurrent Kleene algebra* [HMSW11] is a bi-Kleene algebra which satisfies the interchange law

$$(x \| y)(w \| z) \leq (xz) \| (yw).$$

Definition 15. A *(join/meet) semilattice* is a structure $(S, +)$ where

- $(S, +)$ is a commutative semigroup,
- $x + x = x$.

Definition 16. A *lattice* is a structure $(S, +, \sqcap)$ where:

- $(S, +)$ is a join semilattice,
- $(S, \sqcap)$ is a meet semilattice,
- $x + (x \sqcap y) = x$ and $x \sqcap (x + y) = x$.

Definition 17. A *distributive lattice* is a lattice $(S, +, \sqcap)$ where

$$x \sqcap (y + z) = (x \sqcap y) + (x \sqcap z).$$

Definition 18. A *Boolean algebra* is a structure $(B, +, \sqcap, \overline{\ }, 0, 1)$ where:

- $(B, +, \sqcap)$ is a distributive lattice,
- $x + 0 = x$ and $x \sqcap 1 = x$ for all x (identity),
- $x + \overline{x} = 1$ and $x \sqcap \overline{x} = 0$ for all x (complementation).

Definition 19. A *Galois connection* between two partially ordered sets $(A, \leq)$ and $(B, \leq)$ is a pair of functions $f : A \to B$ and $g : B \to A$ such that

$$f(x) \leq y \iff x \leq g(y).$$

Definition 20. A *Kleene algebra with tests* (KAT) [Koz00] is a structure

$$(K, B, +, \cdot, ^\star, 0, 1, \overline{\ })$$

where $(K, +, \cdot, ^*, 0, 1)$ is a Kleene algebra and $(B, +, \cdot, \overline{\ }, 0, 1)$ a Boolean subalgebra of K, that is, $B \subseteq K$. Complementation is only defined on the Boolean subalgebra.

Definition 21. An *action algebra* [Pra90] is an structure

$$(A, +, \cdot, 0, 1, ^\star, \leftarrow, \rightarrow),$$

such that $(A, +, \cdot, 0, 1)$ is a dioid, and satisfying

$$xy \leq z \overset{L}{\iff} x \leq z \leftarrow y, \quad \text{and} \quad xy \leq z \overset{R}{\iff} y \leq x \rightarrow z,$$
$$1 + x^\star x^\star + x \leq x^\star \leq x^\star,$$
$$1 + yy + x \leq y \implies x^\star \leq y.$$

Definition 22. A *$\star$-continuous Kleene algebra* [Koz94], also called an **N**-algebra by Conway [Con71] is a dioid satisfying the law

$$xy^*z = \sum\{xy^nz \mid n \in \mathbb{N}\}.$$

Definition 23. A *weak ω-algebra* [LS11] is a left Kleene algebra expanded with an omega operator satisfying

$$\begin{aligned} xx^\omega &= x^\omega, \\ y \leq z + x \cdot y \implies y &\leq x^\omega + x^\star z. \end{aligned}$$

Definition 24. A *demonic refinement algebra* (DRA) [vW04] is a structure

$$(D, +, \cdot, 0, 1, ^\star, ^\infty)$$

where:

- $(D, \cdot, +, 0, 1, ^\star)$ is a Kleene algebra,
- The strong iteration operator $^\infty$, which represents both finite or infinite iteration, satisfies the unfolding axiom $x^\infty = xx^\infty + 1$ and the induction axiom $z \leq xz + y \implies z \leq x^\infty y$

Definition 25. A *quantale* is a structure $(S, \leq, \cdot, 1)$ such that $(S, \leq)$ is a complete lattice, $(S, \cdot, 1)$ is a monoid and

$$x\left(\sum Y\right) = \sum_{y \in Y} xy, \qquad \left(\sum X\right) y = \sum_{x \in X} xy.$$

Definition 26. A *rely-guarantee algebra* is a structure

$$(K, I, +, \sqcap, \cdot, \|, ^\star, 0, 1),$$

where

- $(K, +, \sqcap)$ is a distributive lattice,
- $(K, +, \cdot, \|, 0, 1)$ is a trioid,
- $(K, +, \cdot, 0, 1, ^\star)$ is a Kleene algebra.
- I is a distinguished subset of rely and guarantee conditions or *interference constraints* that satisfy the following axioms:

$$\begin{aligned} r\|r &\leq r, \\ r &\leq r\|r', \\ r\|(x \cdot y) &= (r\|x) \cdot (r\|y), \\ r\|x^+ &\leq (r\|x)^+. \end{aligned}$$

Definition 27. An *interference algebra* is a two sorted algebra

$$(K, I, \rhd, \langle - \rangle),$$

where:

- K is an algebra $(K, +, \sqcap, \cdot, \to, \|, ^\star, ^\omega, 0, 1)$ where $(K, +, \sqcap)$ is a distributive lattice, $(K, +, \cdot, \|, 0, 1)$ is trioid, and $(K, +, \cdot, 0, 1, ^\star, ^\omega)$ is a weak ω-algebra. Additionally the preimplication operator $\to$ satisfies

$$xy \leq z \iff y \leq x \to z.$$

- I is a complete lattice $(I, \sqsubseteq)$. Join and meet in this lattice are denoted by $\sqcup$ and $\sqcap$ respectively.

- $\rhd$ is a closure operator

- The $\rhd$ and $\langle - \rangle$ operators satisfy:

$$\begin{aligned}
(r \rhd x)(r \rhd y) &\leq r \rhd xy, \\
(r \rhd x)^\star &\leq r \rhd x^\star, \\
r \leq g \implies g \rhd x &\leq r \rhd x, \\
\top \rhd x &= x, \\
\langle g_1 \rangle \| \langle g_2 \rangle &\leq \langle g_1 \sqcup g_2 \rangle, \\
\langle g \rangle \langle g \rangle &= \langle g \rangle, \\
\langle g \rangle^\star &= \langle g \rangle, \\
(x \sqcap \langle g \rangle)(y \sqcap \langle g \rangle) &= xy \sqcap \langle g \rangle, \\
(r \sqcup g_2 \rhd x \sqcap \langle g_1 \rangle) \parallel (r \sqcup g_1 \rhd y \sqcap \langle g_2 \rangle) &\leq r \rhd (x \sqcap \langle g_1 \rangle) \| (y \sqcap \langle g_2 \rangle).
\end{aligned}$$

Bibliography

[Aar92] C.J. Aarts. Galois connections presented calculationally, 1992.

[ABFGS14] A. Armstrong, V. B. F. Gomes, and G. Struth. Algebras for program correctness in Isabelle/HOL. In P. Jipsen and W. Kahl, editors, *RAMiCS 2014*, 2014.

[AGS14a] A. Armstrong, V. B. F. Gomes, and G. Struth. Algebraic principles for rely-guarantee style concurrency verification tools. In C. B. Jones, P. Pihlajasaari, and J. Sun, editors, *FM 2014*, volume 8442 of *LNCS*, pages 78–93. Springer, 2014.

[AGS14b] A. Armstrong, V. B. F. Gomes, and G. Struth. Kleene algebras with tests and demonic refinement algebras. *Archive of Formal Proofs*, 2014. `http://afp.sourceforge.net/entries/KAT_and_DRA.shtml`.

[AGS14c] A. Armstrong, V. B. F. Gomes, and G. Struth. Lightweight program construction and verification tools in Isabelle/HOL. In D. Giannakopoulou and G. Salaün, editors, *SEFM 2014*, volume 8702 of *LNCS*, pages 5–19. Springer, 2014.

[AHK06] K. Aboul-Hosn and D. Kozen. KAT-ML: an interactive theorem prover for Kleene algebra with tests. *Journal of Applied Non-Classical Logics*, 16(1–2):9–34, 2006.

[AK01] A. Angus and D. Kozen. Kleene algebra with tests and program schematology. Technical Report TR2001-1844, Computer Science Department, Cornell University, July 2001.

[Apt81] K. R. Apt. Recursive assertions and parallel programs. *Acta Informatica*, 15(3):219–232, 1981.

[Arm12] A. Armstrong. An evaluation of automated theorem proving in regular algebra, 2012. Extended abstract presented at RAMiCs 13.

[AS12] A. Armstrong and G Struth. Automated reasoning in higher-order regular algebra. In T. Griffin and W. Kahl, editors, *RAMiCs 2012*, volume 7560 of *LNCS*. Springer, 2012.

[ASW13a] A. Armstrong, G. Struth, and T. Weber. Kleene algebra. *Archive of Formal Proofs*, 2013. `http://afp.sourceforge.net/entries/Kleene_Algebra.shtml`.

[ASW13b] A. Armstrong, G. Struth, and T. Weber. Program analysis and verification based on Kleene algebra in Isabelle/HOL. In S. Blazy, C. Paulin-Mohring, and D. Pichardie, editors, *ITP 2013*, number 7998 in LNCS, pages 197–212. Springer, 2013.

[ASW14] A. Armstrong, G. Struth, and T. Weber. Programming and automating mathematics in the Tarski-Kleene hierarchy. *Journal of Logical and Algebraic Methods in Programming*, 83(2):87–102, March 2014.

[Bal10] C. Ballarin. Tutorial to locales and locale interpretation. In *Contribuciones Científicas en honor de Mirian Andrés*. Servicio de Publicaciones de la Universidad de La Rioja, Spain, 2010.

[BBN11] J. C. Blanchette, L. Bulwahn, and T. Nipkow. Automatic proof and disproof in Isabelle/HOL. In C. Tinelli and V. Sofronie-Stokkermans, editors, *FroCos 2011*, volume 6989 of *LNCS*, pages 12–27. Springer, 2011.

[BE95] S. L. Bloom and Z. Ésik. Nonfinite axiomatizability of shuffle inequalities. In P. D. Mosses, M. Nielsen, and M. I. Schwartzbach, editors, *Proceedings of the 6th International Joint Conference CAAP/FASE on Theory and Practice of Software Development*, TAPSOFT '95, pages 318–333. Springer, 1995.

[BE96] S. L. Bloom and Z. Ésik. Free shuffle algebras in language varieties. *Theor. Comput. Sci.*, 163(1&2):55–98, 1996.

[BH96] A. Buch and T. Hillenbrand. Waldmeister: High performance equational theorem proving. In J. Calmet and C. Limongelli, editors, *Proceedings of the 4th International Symposium on Design and Implementation of Symbolic Computation Systems*, volume 1128 of *LNCS*, pages 63–64. Springer, 1996.

[BHL+14] J. C. Blanchette, J. Hölzl, A. Lochbihler, P. Lorenz, A. Popescu, and D. Traytel. Truly modular (co)datatypes for Isabelle/HOL. In G. Klein and R. Gamboa, editors, *ITP 2014*, volume 8558 of *LNCS*, pages 93–110. Springer, 2014.

[BN10] J. C. Blanchette and T. Nipkow. Nitpick: A counterexample generator for higher-order logic based on a relational model finder. In M. Kaufmann and L. C. Paulson, editors, *ITP 2010*, number 6172 in LNCS, pages 131–146. Springer, 2010.

[BP12] T. Braibant and D. Pous. Deciding kleene algebras in coq. *Logical Methods in Computer Science*, 8(1), 2012.

[Bro93] S. Brookes. Full abstraction for a shared variable parallel language. In M. Okada and P Panangaden, editors, *LICS 93*, page 98–109, 1993.

[Bro04] S. D. Brookes. A semantics for concurrent separation logic. In P. Gardner and N. Yoshida, editors, *CONCUR 2004*, volume 3170 of *LNCS*, pages 16–34. Springer, 2004.

[BS10] R. Berghammer and G. Struth. On automated program construction and verification. In C. Bolduc, J. Desharnais, and B. Ktari, editors, *MPC 2010*, number 6120 in LNCS, pages 22–41. Springer, 2010.

[Bul12] L. Bulwahn. The new quickcheck for isabelle - random, exhaustive and symbolic testing under one roof. In C. Hawblitzel and D. Miller, editors, *CPP 2012*, volume 7679 of *LNCS*, pages 92–108. Springer, 2012.

[BW90] J. C. M. Baeten and W. P. Weijland. *Process Algebra*. Number 18 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1990.

[CG10] H. Collavizza and M. Gordon. Forward with Hoare. In A. W. Roscoe, C. B. Jones, and K. R. Wood, editors, *Reflections on the Work of C. A. R. Hoare*, pages 101–121. Springer, 2010.

[Coh00] E. Cohen. Separation and reduction. In R. C. Backhouse and J. Nuno Oliveira, editors, *MPC 2000*, volume 1837 of *LNCS*, pages 45–59. Springer, 2000.

[Con71] J. H. Conway. *Regular algebra and finite machines*. Chapman and Hall, 1971.

[COY07] C. Calcagno, P. W. O'Hearn, and H. Yang. Local action and abstract separation logic. In L. Ong, editor, *Proceedings of the 22nd Annual IEEE Symposium on Logic in Computer Science*, LICS '07, pages 366–378, Washington, DC, USA, 2007. IEEE Computer Society.

[dBHdR99] F. S. de Boer, U. Hannemann, and W.-P. de Roever. Formal justification of the rely-guarantee paradigm for shared-variable concurrency: a semantic approach. In J. M. Wing, J. Woodcock, and J. Davies, editors, *FM'99 — Formal Methods*, number 1709 in LNCS, pages 1245–1265. Springer, 1999.

[DGS15] B. Dongol, V. B. F. Gomes, and G. Struth. A program construction and verification tool for separation logic. In R. Hinze and J. Voigtländer, editors, *MPC 2015*, volume 9129 of *LNCS*, pages 137–158. Springer, 2015.

[DHM11] H.-H. Dang, P. Höfner, and B. Möller. Algebraic separation logic. *J. Log. Algebr. Program.*, 80(6):221–247, 2011.

[Din02] J. Dingel. A refinement calculus for shared-variable parallel and distributed programming. *Formal Aspects of Computing*, 14(2):123–197, 2002.

[DMS06] J. Desharnais, B. Möller, and G. Struth. Kleene algebra with domain. *ACM Trans. Comput. Log.*, 7(4):798–833, 2006.

[DMS11] J. Desharnais, B. Möller, and G. Struth. Algebraic notions of termination. *Logical Methods in Computer Science*, 7(1), February 2011.

[dRdBH+01a] W.-P. de Roever, F. de Boer, U. Hanneman, J. Hooman, Y. Lakhnech, M. Poel, and J. Zwiers. *Concurrency verification: an introduction to state-based methods.* Cambridge University Press, Cambridge, 2001.

[dRdBH+01b] W. P. de Roever, F. S. de Boer, U. Hannemann, J. Hooman, Y. Lakhnech, M. Poel, and J. Zwiers. *Concurrency Verification: Introduction to Compositional and Noncompositional Methods*, volume 54 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 2001.

[EMS03] T. Ehm, B. Möller, and G. Struth. Kleene modules. In R. Berghammer, B. Möller, and G. Struth, editors, *RelMiCS*, volume 3051 of *LNCS*, pages 112–124. Springer, 2003.

[FD07] T. Fernandes and J. Desharnais. Describing data flow analysis techniques with kleene algebra. *Science of Computer Programming*, 65(2):173–194, 2007.

[FS12] S. Foster and G. Struth. Automated analysis of regular algebra. In B. Gramlich, D. Miller, and U. Sattler, editors, *IJCAR 2012*, volume 7364 of *LNCS*, pages 271–285. Springer, 2012.

[Gis88] J. L. Gischer. The equational theory of pomsets. *Theoretical Computer Science*, 61(2–3):199–224, 1988.

[GSW11] W. Guttmann, G. Struth, and T. Weber. Automating algebraic methods in Isabelle. In S. Quin and Z. Qiu, editors, *ICFEM 2011*, volume 6991 of *LNCS*, pages 617–632. Springer, 2011.

[Gut12] W. Guttmann. Algebras for iteration and infinite computations. *Acta Inf.*, 49(5):343–359, 2012.

[HHM+11] C. A. R. Hoare, A. Hussain, B. Möller, P. W. O'Hearn, R. L. Petersen, and G. Struth. On locality and the exchange law for concurrent processes. In J.-P. Katoen and B. König, editors, *CONCUR 2011*, number 6901 in LNCS, pages 250–264. Springer, 2011.

[HJC13] I. J. Hayes, C. B. Jones, and R. J. Colvin. Refining rely-guarantee thinking. Technical report, University of Newcastle, 2013.

[HK13] B. Huffman and O. Kuncar. Lifting and transfer: A modular design for quotients in isabelle/hol. In G. Gonthier and M. Norrish, editors, *CPP2013*, volume 8307 of *LNCS*, pages 131–146. Springer, 2013.

[HMSW11] C. A. R. Hoare, B. Möller, G. Struth, and I. Wehrman. Concurrent Kleene algebra and its foundations. *J. Log. Algebr. Program.*, 80(6):266–296, 2011.

[HS07] P. Höfner and G. Struth. Automated reasoning in Kleene algebra. In F. Pfenning, editor, *CADE*, volume 4603 of *LNCS*, pages 279–294. Springer, 2007.

[HS10] P. Höfner and G. Struth. Algebraic notions of nontermination: Omega and divergence in idempotent semirings. *J. Log. Algebr. Program.*, 79(8):794–811, 2010.

[HW08] F. Haftmann and M. Wenzel. Local theory specifications in Isabelle/Isar. In S. Berardi, F. Damiani, and U. de'Liguoro, editors, *TYPES 2008*, volume 5497 of *LNCS*, pages 153–168. Springer, 2008.

[Jon81] C. B. Jones. *Development methods for computer programs including a notion of interference*. PhD thesis, Oxford University, 1981.

[Jon83] C. B. Jones. Specification and design of (parallel) programs. In R. E. A. Mason, editor, *IFIP Congress*, pages 321–332, 1983.

[KKB12a] G. Klein, R. Kolanski, and A. Boyton. Mechanised separation algebra. In L. Beringer and A. P. Felty, editors, *ITP 2012*, volume 7406 of *LNCS*, pages 332–337. Springer, 2012.

[KKB12b] G. Klein, R. Kolanski, and A. Boyton. Separation algebra. *Archive of Formal Proofs*, 2012, 2012.

[KN10] A. Krauss and T. Nipkow. Regular sets and expressions. *Archive of Formal Proofs*, May 2010. `http://afp.sf.net/entries/Regular-Sets.shtml`, Formal proof development.

[Koz90] D. Kozen. On kleene algebras and closed semirings. In B. Rovan, editor, *MFCS'90*, volume 452 of *LNCS*, pages 26–47. Springer, 1990.

[Koz93] Dexter Kozen. On action algebras. In *Logic and Information Flow*, pages 78–88. MIT Press, 1993.

[Koz94] D. Kozen. A completeness theorem for Kleene algebras and the algebra of regular events. *Information and Computation*, 110(2):366–390, 1994.

[Koz97] D. Kozen. *Automata and computability*. Undergraduate texts in computer science. Springer, 1997.

[Koz00] D. Kozen. On Hoare logic and Kleene algebra with tests. *ACM TOCL*, 1(1):60–76, 2000.

[KP00] D. Kozen and M.-C. Patron. Certification of compiler optimizations using Kleene algebra with tests. In J. Lloyd, V. Dahl, U. Furbach, M. Kerber, K.-K. Lau, C. Palamidessi, L. Moniz Pereira, Y. Sagiv, and P. J. Stuckey, editors, *Computational Logic*, volume 1861 of *LNCS*, page 568–582. Springer, 2000.

[KU11] C. Kaliszyk and C. Urban. Quotients revisited for Isabelle/HOL. In W. C. Chu, W. E. Wong, M. J. Palakal, and C-C. Hung, editors, *SAC*, pages 1639–1644. ACM, 2011.

[Lei06] H. Leiß. Kleene modules and linear languages. *J. Log. Algebr. Program.*, 66(2):185–194, 2006.

[Loc10] Andreas Lochbihler. Coinductive. *Archive of Formal Proofs*, February 2010. `http://afp.sf.net/entries/Coinductive.shtml`, Formal proof development.

[LS11] M. R. Laurence and G. Struth. Omega algebras and regular equations. In *RAMICS 2011*, volume 6663 of *LNCS*, pages 248–263. Springer, 2011.

[LS13] M. R. Laurence and G. Struth. Completeness results for bi-Kleene algebras and regular pomset languages, 2013. (submitted).

[Man74] Z. Manna. *Mathematical theory of computation.* McGraw-Hill, 1974.

[MFP91] Erik Meijer, Maarten Fokkinga, and Ross Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In J. Hughes, editor, *Proc. 5th ACM Conf. Functional Programming Languages and Computer Architecture*, volume 523 of *LNCS*, pages 124–144. Springer, 1991.

[MMRS97] A. Mateescu, G. D. Mateescu, G. Rozenberg, and A. Salomaa. Shuffle-like operations on ω-words. In G. Păun and A. Salomaa, editors, *New Trends in Formal Languages*, number 1218 in LNCS, pages 395–411. Springer, 1997.

[Mor88] C. Morgan. The specification statement. *ACM Trans. Program. Lang. Syst.*, 10(3):403–419, 1988.

[MPdS13] N. Moreira, D. Pereira, and S. M. de Sousa. On the mechanisation of rely-guarantee in Coq. Technical report, Faculdade de Ciências, Universidade do Porto, 2013.

[MS06] B. Möller and G. Struth. Algebras of modal operators and partial correctness. *Theor. Comput. Sci.*, 351(2):221–239, 2006.

[MWM14] D. Matichuk, M. Wenzel, and T. C. Murray. An isabelle proof method language. In G. Klein and R. Gamboa, editors, *ITP 2014*, LNCS, pages 390–405. Springer, 2014.

[Nie01] L. P. Nieto. *Verifcation of Parallel Programs with the Owicki-Gries and Rely-Guarantee Methods in Isabelle/HOL.* PhD thesis, Technische Universität München, October 2001.

[Nie03] L. P. Nieto. The rely-guarantee method in Isabelle/HOL. In P. Degano, editor, *Programming Languages and Systems*, number 2618 in LNCS, pages 348–362. Springer, 2003.

[Nip98] T. Nipkow. Winskel is (almost) right: Towards a mechanized semantics. *Formal Asp. Comput.*, 10(2):171–186, 1998.

[NMS+08] A. Nanevski, G. Morrisett, A. Shinnar, P. Govereau, and L. Birkedal. Ynot: Dependent types for imperative programs. In J. Hook and P. Thiemann, editors, *ICFP*, pages 229–240. ACM, 2008.

[NPW02] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[OG76] S. S. Owicki and D. Gries. An axiomatic proof technique for parallel programs I. *Acta Inf.*, 6:319–340, 1976.

[O'H07] P. W. O'Hearn. Resources, concurrency, and local reasoning. *Theor. Comput. Sci.*, 375(1-3):271–307, 2007.

[OPVH15] P. W. O'Hearn, R. L. Petersen, J. Villard, and A. Hussain. On the relation between concurrent separation logic and concurrent kleene algebra. *J. Log. Algebr. Meth. Program.*, 84(3):285–302, 2015.

[ORY01] P. W. O'Hearn, J. C. Reynolds, and H. Yang. Local reasoning about programs that alter data structures. In L. Fribourg, editor, *CSL 2001*, volume 2142 of *LNCS*, pages 1–19. Springer, 2001.

[Owi75] S. Owicki. *Axiomatic Proof Techniques for Parallel Programs*. PhD thesis, Cornell University, 1975.

[PNW11] L. Paulson, T. Nipkow, and M. Wenzel. Isabelle. `http://www.cl.cam.ac.uk/research/hvg/Isabelle/index.html`, 2011.

[Pou13a] D. Pous. Kleene algebra with tests and Coq tools for while programs. In S. Blazy, C. Paulin-Mohring, and D. Pichardie, editors, *ITP 2013*, number 7998 in LNCS, pages 180–196. Springer B, 2013.

[Pou13b] D. Pous. Kleene algebra with tests and coq tools for while programs. In S. Blazy, C. Paulin-Mohring, and D. Pichardie, editors, *ITP 2013*, volume 7998 of *LNCS*, pages 180–196. Springer, 2013.

[Pra90] V. R. Pratt. Action logic and pure induction. In J. van Eijck, editor, *JELIA '90*, volume 478 of *LNCS*, page 97–120. Springer, 1990.

[Red64] V. N. Redko. On defining relations for the algebra of regular events. *Ukraïn. Mat. Z.*, 16:120–126, 1964. In Russian.

[Rey02] J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS 2002*, pages 55–74. IEEE Computer Society, 2002.

[Sch06] N. Schirmer. *Verification of sequential imperative programs in Isabelle-HOL*. PhD thesis, Technische Universität München, 2006.

[Sti88] C. Stirling. A generalization of Owicki-Gries's hoare logic for a concurrent while language. *Theoretical Computer Science*, 58(1âĂŞ3):347 – 359, 1988.

[Sto77] J. E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*. MIT Press, Cambridge, MA, USA, 1977.

[Tar85] A. Tarlecki. A language of specified programs. *Science of Computer Programming*, 5:59–81, 1985.

[Vaf08] V. Vafeiadis. *Modular fine-grained concurrency verification*. PhD thesis, University of Cambridge, 2008.

[vW04] J. von Wright. Towards a refinement algebra. In B Möller and E. Boiten, editors, *MPC 2002*, volume 51, pages 23 – 45, 2004.

[Wen07] M. Wenzel. Isabelle/Isar – a generic framework for human-readable proof documents. In R. Matuszewski and A. Zalewska, editors, *From Insight to Proof — Festschrift in Honour of Andrzej Trybulec, Studies in Logic, Grammar, and Rhetoric*. University of Białystok, 2007.

[XdRH97] Q. Xu, W-P. de Roever, and J. He. The rely-guarantee method for verifying shared variable concurrent programs. *Formal Aspects of Computing*, 9(2):149–174, 1997.