Algebraic Principles for Program Correctness Tools in Isabelle/HOL

Victor B. F. Gomes

Supervisor: Georg Struth
Department of Computer Science

University of Sheffield

A thesis submitted for the degree of

Doctor of Philosophy

February 2016

Acknowledgements

I gratefully acknowledge the funding received from the CNPq through the programme Science without Border (CsF) to undertake my PhD.

Foremost, I would like to express my sincere gratitude to my supervisor Prof. Georg Struth for all the support and guidance during these three exciting years. Without his assistance and constant feedback, this thesis would not have been possible. In addition, I would like to thank the rest of my research panel: Prof. Marian Georghe and Dr. Joab Winkler for their insigthful comments and questions.

I would also like to thank my collegue Alasdair Armstrong for his constant assistance while I was learning Isabelle. As well as to all the attendees of the verification reading group for all the in-depth and memorable discussions: Dr. Brijesh Dongol, Dr. James Cranch, Dr. Kirill Bogdanov and Dr. Michael Laurence.

I wish to thank Prof. José Nuno Oliveira, Prof. Peter Jipsen, Dr. Walter Guttmann and Dr. Damien Pous for interesting and insightful conversations during RAMiCS'14. I also thank Prof. Jeremy Gibbons, Dr. Stephan Van Staden, Prof. Roland Backhouse, Prof. Carrol Morgan and Prof. Lindsay Groves for their discussions, suggestions and guidance during MPC'15.

Thanks to all my collegues in the department: Jorge, José, Raluca, Fernando and Umberto. Special thanks to the Brazilian gang: Carol, Tiago, Lúcia, Téo, Yuri, Hegler and Gustavo.

I am especially grateful for my Sheffield friends and flatmates, who I was lucky to spend time with during my PhD: Annemie, Fred, Natalie, Galyia, Jamie, Girija and Theo (thanks for the proofreading).

Agradeço em especial minha família pelo apoio incondicional, força e incentivo. Minha amada mãe, meu pai companheiro e meus queridos irmãos e irmãs: Camilla, Larissa, André, Samuel e Luana.

Abstract

This thesis puts forward a flexible and principled approach to the development of construction and verification tools for imperative programs, in which the control flow and the data level are cleanly separated. The approach is inspired by algebraic principles and benefits from an algebraic semantics layer. It is programmed in the Isabelle/HOL interactive theorem prover and yields simple lightweight mathematical components as well as program construction and verification tools that are themselves correct by construction.

First, a simple tool is implemented using Kleeene algebra with tests (KAT) for the control flow of while-programs, which is the most compact verification formalism for imperative programs, and their standard relational semantics for the data level. A reference formalisation of KAT in Isabelle/HOL is then presented, providing three different formalisations of tests. The structured comprehensive libraries for these algebras include an algebraic account of Hoare logic for partial correctness. Verification condition generation and program construction rules are based on equational reasoning and supported by powerful Isabelle tactics and automated theorem proving.

Second, the tool is expanded to support different programming features and verification methods. A basic program construction tool is developed by adding an operation for the specification statement and one single axiom. To include recursive procedures, KATs are expanded further to quantales with tests, where iteration and the specification statement can be defined explicitly. Additionally, a nondeterministic extension supports the verification of simple concurrent programs.

Finally, the approach is also applied to separation logic, where the control-flow is modelled by power series with convolution as separating conjunction. A generic construction lifts resource monoids to assertion and predicate transformer quantales. The data level is captured by concrete store-heap models. These are linked to the algebra by soundness proofs.

A number of examples shows the tools at work.

I hold the opinion that the construction of computer programs is a mathematical activity like the solution of differential equations, that programs can be derived from their specifications through mathematical insight, calculation, and proof, using algebraic laws as simple and elegant as those of elementary arithmetic.

C. A. R. Hoare

Contents

List of Figures

Chapter 1

Introduction

Program verification is an old subject in computer science, dating at least from von Neumann and Goldstine with assertion boxes in a flow diagram [116], and Alan Turing with possibly the first correctness proof of a program computing factorials [113]. Its importance and applicability are clear and do not need to be reiterated here. However, program verification is still an area of intense research activity and has many open questions. Moreover, it is unfortunately not widely used in the industry. One of the reason for this might be the lack of good, sound and (semi)-automatic tools and methods. Recently, the area has seen a revival, with an explosion of papers due to the increase in computational power and the need for rigorous treatment for concurrent algorithms, where testing-based methods were proven to be inefficient, in particular when considering weak memory models.

For a few years now, abstract algebras have been used to analyse and understand the structure behind programs and verification methods. Kleene algebra with tests [72] (KAT) and their variants, for instance, yield a minimalist and elegant formalism for program verification of imperative while programs by simple equational reasoning. They model the flow of computation algebraically and subsume propositional Hoare logic [73], which means that validity of a Hoare triple can be expressed in the algebra and the inference rules of Hoare logic, except the assignment axiom, can be derived equationally as theorems. Additionally, KAT has been used for verifying program transformations [72], for compiler optimisation [75] and for static analysis [74].

More recently, algebras were used to derive the frame rule of separation logic [101], which is an approach to program verification for mutable resources that is receiving considerable attention over the last decade. Its main feature is the separating conjunction and its key application is the verification of programs with pointers [101, 93, 91], but it has also been used in concurrency verification [90, 63]. Algebraic approaches using power series and convolution to express

separating conjunction in a constructive way [44] can be used as a concise and elegant abstract foundation for separation logic.

The use of abstract algebras thus promises to be a principled sound foundation in the development of correctness tools. They have indeed clear advantages.

Algebraic semantic layer. In the algebraic approach to program verification, there is a clear cut distinction between control flow and data level, where a lightweight middle layer is formed by an algebraic semantics. The control flow can thus be understood and analysed directly and efficiently at the algebraic level. The concrete data level can be integrated via soundness of the algebra with respect to its concrete semantics. A clear example of this is in the proof of Kozen's loop transformation [72], which states that every while program can be rewritten as a program with at most one loop, as long as there are enough free variables available. The proof is clear and simple, and it has been done equationally.

Modularity. This algebraic semantic middle layer offers more modularity and flexibility to the correctness tool when changing the target programming language, its logic or even its concrete semantics. All facts derived at the algebraic level are available to any of its computational models. In addition, the division of concerns between flow and data yields a simple and fast development of verification and refinement tools.

Automation. The algebra deals with the control flow very efficiently. In some cases, inference rules and transformation proofs are fully automatic. For instance the equational theory of KAT and its universal Horn theory has been shown to be decidable [76]. More generally, even when the algebraic structure used is not decidable, the algebra forms a first-order layer in which theorems can usually be proved automatically by state-of-the-art theorem provers. These are highly suitable for first-order theory [64]. Algebras thus have the ideal level of expressivity. On the one hand, they are expressive enough to derive verification, construction and transformation tools. On the other hand, they are still simple enough to be suitable for automatic theorem provers.

Correctness. Finally, soundness of these tools are implied by the algebra, that is, they are correct by construction.

Nevertheless the rôle of algebras in program verification and correctness tools is still not clear; their application has so far been rare [11, 97] and it has never been thoroughly investigated. To the best of the author's knowledge, they have never been fully integrated and used in the development of correctness tools.

The reason may be that these algebras provide limited capabilities for modelling the data level; reasoning at this level may require higher-order logic and therefore other tools and techniques. Advancements in theorem proving technology in the latest years indicates that the formalisation of these algebras within an interactive theorem proving (ITP) environment can benefit from the vast amount of infrastructure for data-level modelling and reasoning provided by these ITPs, delegating the proof obligations of the data level to other mechanisms.

This thesis describes a framework for the analysis of program construction and verification within the Isabelle/HOL theorem proving environment and based on algebraic principles. It unravels the rôle of algebras in the development of correctness tools in an interactive theorem prover.

1.1 Contributions

The main contribution of this thesis lies in the development and implementation of the approach outlined within Isabelle/HOL [89]. We formalise various algebraic components in Isabelle, which are available to the theorem proving community through the Archive of Formal Proofs [9, 55], and derive correctness tools from these algebras. Beyond the proof of concept, this leads to refinement and verification tools for imperative programs correct by construction. The implementation greatly benefits from Isabelle's support for engineering mathematical hierarchies, for linking abstract algebras with concrete models and from its emphasis on proof automation through the integration of state-of-the-art first-order theorem proving and SMT solving technology, which is optimised for equational reasoning. Additionally, we implement a novel algebraic approach to separation logic and derive correctness tools for programs with pointers.

More detailed contributions are as follows.

Algebraic components. Building upon existing formalisation of Kleene algebras [13], various algebraic components are implemented in Isabelle/HOL: three different formalisations of KAT, a novel refinement KAT, demonic refinement algebras (DRA), quantales, and quantales with tests. The one-sorted implementation of KAT is based on an antitest operation, which generates the boolean algebra of tests as its image, and it is a contribution in its own right. In addition, an axiomatisation for *weak* regular algebras given by Conway [34], in which the iteration axioms cannot distinguish between finite and potentially infinite iteration, is also provided. Moreover, soundness of all these algebraic structures is verified with respect to the model of binary relations. In summary, extensive libraries for these algebras are provided which add more than 400 facts. These include

- formalisation of validity of Hoare triples and derivation of the rules of propositional Hoare logic in KAT as well as of additional rules for recursion in the quantale setting;
- derivation of a propositional version of Morgan's basic refinement calculus in refinement KAT;
- proof of program transformations examples, such as transformation theorem for while loops [72] and variants of Back's atomicity refinement theorem for action systems [117, 33];
- derivation of assignment rules for program verification and refinement in the relational model; which yields formal soundness proofs of Hoare logic and Morgan's basic refinement calculus [83].

The algebras are linked with concrete datatypes, data structures and notions of state in a generic fashion by an instantiation mechanism, which exploits the polymorphism of implementations of algebras and models within Isabelle.

Correctness tools. These mathematical components allow us to implement several tools that are correct by construction:

- a KAT-based verification tool for while-program in which classic Hoare logic generates verification conditions;
- a KAT-based basic refinement tool for while-programs obtained by adding an operation for the specification statement and one single axiom to KAT;
- more expressive quantale-based refinement and verification tools for programs with recursive parameterless procedures;
- a quantale-based verification tool for nondeterministic programs, capable of verifying correctness of simple concurrent algorithms by expressing the execution of parallel programs with nondeterministic choice;
- verification tools based on domain algebras.

Finally, we have applied our tools in a series of program construction and verification tasks.

Over the years, several tools have been implemented based on different versions of Hoare logic. Some tools, like the ones presented in this thesis, use a research-based and minimalist programming language, such as the while language. Others focus on real word programming languages and applications, such as C and Java. These include for instance *KeY* [2], *VSE - Verification Support*

Environment [106] and *VeriFast* [104]. They are usually optimised and highly automatic for a specific purpose relying on state-of-the-art decision procedures and SMT solvers at the data level. Soundness of these tools, however, is often not guaranteed and needs to be proven in each version released, since they are not verified relative to a small core, as provided by an LCF-style proof assistant. In addition, different degrees of interactivity are offered by these tools; they focus on complete automation and the user cannot usually prove any remaining verification condition by interaction. This implies that they cannot cope very well with higher-order aspects of data types and store.

In contrast, the tools developed in this thesis are correct by construction. They rely on the correctness of the small core of Isabelle/HOL, satisfying the so called *de Bruijn criterion*, which roughly states that proof objects should be verifiable by a very small and simple program checkable by hand [21]. Additionally, they are semi-automatic, *i.e.*, tactics are used as often as possible to reduce the number of verification conditions. The user has then the whole Isabelle infrastructure, which supports higher-order reasoning, at his disposal to deal with any remaining proof obligation. Moreover, the user can guide the correctness proof, breaking the conditions down in different pieces and calling automatic SMT solvers to discharge them.

Other verification tools have also been developed with the support of interactive proof assistants in mind. For instance, Why3 [49], which is a platform where users can write a program in WhyML, can generate verification conditions and export them to one of the interactive provers supported, such as Coq, PVS or Isabelle/HOL. Moreover, the proof obligations can also be exported to several other automated theorem provers (ATP) and SMT solvers. Nevertheless, Why3 is developed in ML and its own soundness is not guaranteed by the provers; it does not meet the de Bruijn criterion.

Correctness tools developed on top of a proof assistant include Balser et al. [20] in KIV, Ynot [32] in Coq; or SIMPL [102], IMP [86], and Imperative-HOL [28] in Isabelle/HOL. These are generally less automatic, but allow more precise properties to be proved. Nevertheless, they do not share the lightweight middle layer formed by an algebraic semantics such as in our case and none of these tools supports program construction and refinement via Morgan-style reasoning.

Algebraic approach for separation logic. Based on lifting results for power series [44], we implement in Isabelle/HOL a novel algebraic approach for separation logic, in which separating conjunction is expressed as convolution. Contributions include

- the use of power series and convolution in the context of separation logic;

- an entire theory hierarchy formalised in Isabelle in a modular fashion from resource monoids to predicate transformer algebras, which is based on previous work by Proteasa [98]; we formalise conjunctive, disjunctive, monotone and local predicate transformers;
- practical tactics for Isabelle/HOL inspired by *safe* and *auto* and based on [31]; this development benefits from Isabelle's new proof method language Eisbach [80];
- development of a refinement and a *post-hoc* program verification tool for pointed while-programs based on separation logic.

Once again, we have applied these tools to several program construction and verification examples.

Similarly, some tools supporting separation logic try to deal with real world programming languages. These include *Predator* [46], *JStar* [42] and *VeriFast* [104]. Additionally, Why3 also support separation logic reasoning. However, as before, the soundness of these tools is not guaranteed by a theorem prover; and they do not satisfy the *de Bruijin criterion*. Among tools developed on top of proof assistants, *Smallfoot* [22] stands out, which has been implemented within HOL4 and supports concurrent separation logic, an approach based on [30]; and *YNot* [32] in Coq, which provides high level of automation. Formalisations in Isabelle/HOL include that of Kolanski and Klein [69], which is targeted towards a subset of C, and Weber [119], which uses a shallow embedding of a while language similar to the one presented in this thesis, but without allocation and deallocation laws. Once again, none of these tools has a lightweight middle layer formed by an algebraic semantics, providing more modularity and flexibility, and none of these tools supports program refinement.

In summary, the approach advocated in this thesis does indeed yield flexible lightweight tools. Beyond general-purpose formalisation of algebraic structures and their standard models, little Isabelle code is needed when implementing a simple construction or verification tool. The degree of automation in concrete examples is generally high as they rely on Isabelle's excellent libraries for functional data types, such as functional lists, sets and relations, in which proofs, including inductive ones, are often fully automatic. The tools have turned out to be robust and useful at least for educational purposes, where they have been used as support for lectures at the University of Sheffield. However we have so far neither aimed at optimising proof performance nor at competing with state-of-the-art verification tools.

The complete Isabelle code for the mathematical components, tools and algorithmic examples can be obtained online[1]. Reference libraries for variants of

[1] https://github.com/victorgomes/veritas

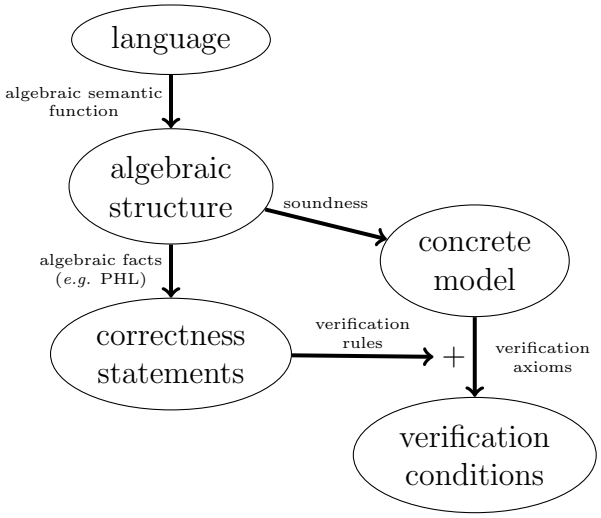

Figure 1.1: Algebraic approach for programm correctness tools.

Kleene algebras, and in particular KAT, can be obtained from the Archive of Formal Proofs [13, 9, 55].

1.2 Outline

This thesis exposes the role of algebra and its benefits in program correctness tools in a stepwise fashion. It begins with a very simple computational algebra (KAT) capable of deriving a classic verification logic, the propositional Hoare logic, and of proving important program transformations, such as Kozen's loop theorem. All of these results are independent of any concrete model and therefore generic (Chapters 3 and 4). The algebra is extended with a specification statement and yields a refinement calculus which is derived equationally (Chapter 5). The versatility of the algebraic layer approach for building program correctness tools is demonstrated by adding support to various programming language features, such as recursion (Chapter 6), nondeterminism (Chapter 8) and mutable resources (Chapters 9, 10 and 11). Its modularity is shown in Chapter 7 where the concrete model is changed. Figure 1.1 outlines the algebraic approach for verification and construction tools suggested in this thesis.

More precisely, the thesis is divided into two parts. Part I deals with algebraic structures capable of modelling simple imperative programs and deriving the rules of Hoare logic, while Part II extends the approach to programs with pointers and separation logic.

Here is its outline:

- Chapter 2 exposes the verification techniques foundations, giving an introduction to verification logics and a *recipe* for correctness tools.
- Chapter 3 presents the main algebraic structures for reasoning about imperative programs. It also demonstrates how to use Isabelle/HOL to create various mathematical components and shows our implementation of these components. Examples of program transformation concludes the chapter.
- In Chapter 4, the soundness of KAT with respect to its concrete semantics is proved. A basic verification tool is then developed. The chapter finishes with verification examples.
- Chapter 5 shows the versatility and robustness of the algebraic approach. We extend KAT to its refinement counterpart and derive a refinement tool.
- More powerful algebras are used in Chapter 6 to increase the expressivity of our tools to support recursive programs.
- Chapter 7 shows a different, but equivalent, semantics for imperative programs. It also presents modal Kleene algebra, capable of expressing the weakest predicate transformer of a command.
- In Chapter 8, the tool is extended to nondeterministic programs. A naïve tool for concurrent program based on nondeterminism is developed. Examples show this tool at work.
- Chapter 9 presents the approach to separation logic based on power series, where separating conjunction is expressed by convolution, and derives the inferences rules and refinement laws of separation logic.
- Finally, Chapter 10 presents an implementation of practical tactics for separation logic predicates in Isabelle/HOL, while Chapter 11 shows some construction and verification examples.
- Chapter 12 concludes this thesis.

1.3 Publications

This thesis contains material from the following papers.

- Armstrong, A., Gomes, V. B. F., and Struth, G. Algebras for program correctness in Isabelle/HOL. In Höfner, P., Jipsen, P., Kahl, W., and Müller, M. E., editors, *RAMiCS 2014*, volume 8428 of *LNCS*, pages 49–64. Springer, 2014

- Armstrong, A., Gomes, V. B. F., and Struth, G. Kleene algebra with tests and demonic refinement algebras. *Archive of Formal Proofs*, 2014

- Armstrong, A., Gomes, V. B. F., and Struth, G. Lightweight program construction and verification tools in Isabelle/HOL. In Giannakopoulou, D. and Salaün, G., editors, *SEFM 2014*, volume 8702 of *LNCS*, pages 5–19. Springer, 2014

- Armstrong, A., Gomes, V. B. F., and Struth, G. Algebraic principles for rely-guarantee style concurrency verification tools. In Jones, C. B., Pihlajasaari, P., and Sun, J., editors, *FM 2014*, volume 8442 of *LNCS*, pages 78–93. Springer, 2014

- Dongol, B., Gomes, V. B. F., and Struth, G. A program construction and verification tool for separation logic. In Hinze, R. and Voigtländer, J., editors, *MPC 2015*, volume 9129 of *LNCS*, pages 137–158. Springer, 2015

- Gomes, V. B. F. and Struth, G. Residuated lattices. *Archive of Formal Proofs*, 2015

- Armstrong, A., Gomes, V. B. F., and Struth, G. Building program construction and verification tools from algebraic principles. *Formal Aspects of Computing*, pages 1–29, 2015

Chapter 2

Technical Background

The formal *verification* of programs is a systematic approach for proving the *correctness* of programs with respect to a formal *specification*. The approach used in this thesis is the so-called *inductive assertional method*. It was initiated by Floyd [51] for the correctness of flowcharts and Hoare [61] for **while** programs. It relies on the use of assertions about the state of execution of a program. A specification is usually composed of a *precondition*, an assertion about the state before the execution of the given program, and a *postcondition*, after the execution.

Correctness of programs is usually divided in two parts: *partial correctness* and *termination*. For a program to be partially correct, it is required that whenever the precondition is satisfied in the initial state and *if* the program terminates, then it terminates in a state satisfying its postcondition. Partial means that the program is not guaranteed to terminate. *Total correctness* requires termination.

The fully automatic verification of program properties is in general an undecidable problem. Automating program verification is a topic of intense research. The first tools were created in the later 80's [57], but only recently with the development of powerful machines and efficient theorem provers, such as Isabelle/HOL [89], PVS [95] and Coq [81], that development of these tools has started to boom. Recent tools include: IMP [88], Why3 [49], VeriFast[67], and others.

The recipe for the development of any tool for program correctness contains the following ingredients:

1. a target programming language,
2. a language of expressions (boolean and other supported types),
3. a language of assertions, and
4. a programming logic.

A programming logic contains a set of logical rules for reasoning about the program. This needs to be proven sound with respect to the semantics of the desired programming language. The semantics of assertions is usually classic predicate logic, but it can also be a high-order logic (HOL in Isabelle/HOL) or an intuitionist logic (as in Coq). The remainder of this chapter discusses these ingredients.

2.1 The while Programming Language

The **while** language [61, 5] is a minimalistic imperative programming language. Despite its simplicity, it is proven to be Turing complete [71]. A **while** program is a string of symbols generated by the following BNF grammar:

$$C ::= \textbf{skip} \mid u := t \mid C_1; C_2 \mid \textbf{if } b \textbf{ then } C_1 \textbf{ else } C_2 \textbf{ fi} \mid \textbf{while } b \textbf{ do } C \textbf{ od},$$

where u stands for a program variable, t is an expression of the same type as u, and b is a boolean expression.

As an abbreviation, we define the following command statements:

$$\textbf{if } b \textbf{ then } C \textbf{ fi} \equiv \textbf{if } b \textbf{ then } C \textbf{ else skip fi}$$
$$\textbf{for } i := n \textbf{ to } m \textbf{ do } C \textbf{ od} \equiv i := n; \textbf{while } i \leq m \textbf{ do } C; i := i + 1 \textbf{ od}$$

An informal intuition for each statement of the language is as follows. The statement **skip** does not change the state of the program and just terminates. An *assignment* $u := t$ assigns the value of the expression t to the program variable u. A *sequential composition* $C_1; C_2$ executes C_1 first and, when it terminates, it executes C_2. A *conditional* statement **if** b **then** C_1 **else** C_2 **fi** evaluates b then executes C_1 if b is true or C_2 if b is false. A *loop* **while** b **do** C **od** evaluates b and terminates if b is false, otherwise it executes C and repeats the process.

Nevertheless, to prove properties of a program written in any programming language, a formal meaning or semantics is needed. There are two main approaches in the literature to program semantics, the *operational* approach and the *denotational* one.

Operational Semantics

The operational semantics approach was proposed by Hennessy and Plotkin [60]. It describes a programming language by specifying how it executes on an abstract machine. A *configuration* of this machine is a pair (C, σ) of command and state, in the *big-step* operational semantics, a transition relation $\rightarrow$ is defined between a configuration and its final state. The relation $(C, \sigma) \rightarrow \sigma'$ then represents the complete execution of a command C in state σ terminating in a state σ'.

$$(\textbf{skip}, \sigma) \to \sigma$$

$$(u := t, \sigma) \to \sigma[\sigma(t)/u]$$

$$\frac{(C_1, \sigma) \to \sigma'' \qquad (C_2, \sigma'') \to \sigma'}{(C_1; C_2, \sigma) \to \sigma'}$$

$$\frac{\sigma(b) \qquad (C_1, \sigma) \to \sigma'}{(\textbf{if } b \textbf{ then } C_1 \textbf{ else } C_2 \textbf{ fi}, \sigma) \to \sigma'}$$

$$\frac{\neg\sigma(b) \qquad (C_2, \sigma) \to \sigma'}{\textbf{if } b \textbf{ then } C_1 \textbf{ else } C_2 \textbf{ fi}, \sigma) \to \sigma'}$$

$$\frac{\sigma(b) \qquad (C, \sigma) \to \sigma'' \qquad (\textbf{while } b \textbf{ do } C \textbf{ od}, \sigma'') \to \sigma'}{(\textbf{while } b \textbf{ do } C \textbf{ od}, \sigma) \to \sigma'}$$

$$\frac{\neg\sigma(b)}{(\textbf{while } b \textbf{ do } C \textbf{ od}, \sigma) \to \sigma}$$

Figure 2.1: Big-step operational semantics of the **while** language

Figure 2.1 shows the operational semantics of a **while** language. We write $\sigma(t)$ and $\sigma(b)$ to the evaluation of the expressions t and b on a state σ, this evaluation depends on the language of expressions used and can also be defined by structural induction. We write $\sigma[n/u]$ for the state obtained from σ by substituting its contents in u by n, that is,

$$\sigma[n/u](v) = \begin{cases} n, & \text{if } v = u, \\ \sigma(v), & \text{otherwise.} \end{cases}$$

We say that a command C *diverges* from a state σ if there is no state σ' such that $(C, \sigma) \to \sigma'$, that is, its computation is infinite and the command never terminates.

Let Σ be the set of all possible *proper* states and $\perp$ be an error state. The operational semantics $[\![C]\!]_o$ of a program C is the mapping $[\![C]\!]_o : \Sigma \to 2^{\Sigma \cup \{\perp\}}$

$$[\![C]\!]_o(\sigma) = \{\sigma' \mid (C, \sigma) \to \sigma'\} \cup \{\perp \mid C \text{ diverges from } \sigma\}\,.$$

As a matter of fact, a **while** program is proven to be deterministic [5]; for any program C, there is only one possible outcome σ' for each state σ. The operational semantics can then be strengthen to a simple mapping $[\![C]\!]_o : \Sigma \to \Sigma \cup \{\perp\}$.

The *small-step* operational semantics [96] is an alternative to big-step semantics, where instead of defining the full execution of each command, the transition relation $\rightarrow_1$ only expresses single steps. The semantics of a command is then defined by using the reflexive-transitive closure relation $\rightarrow_1^*$ of $\rightarrow_1$. This can be useful when dealing with fine-grained concurrency [115]. For a sequential **while** program, both semantics have equivalent expressivity [5].

Operational semantics is extremely simple and widely used in the development of correctness tools [88, 85, 102, 111]. This style of semantics is closed to its implementation, but it makes it hard to compare programs and to understand the real meaning of each command. The approach used in this thesis uses an abstract algebraic semantic layer that fits better with denotational semantics.

Denotational Semantics

The idea of the denotational semantics approach is to provide an appropriate semantic domain and define $[\![C]\!]$ by induction on the structure of C. Its mathematical foundation is due to Dana Scott [109]. It uses abstract mathematical concepts and fixed point techniques to deal with loops and recursion [108, 56]. In contrast with operational semantics, the meaning of each command can be explicitly written as a mathematical object.

$$
\begin{aligned}
&[\![\textbf{skip}]\!] = Id_\Sigma \\
&[\![u := t]\!] = \{(\sigma, \sigma[\sigma(t)/u]) \mid \sigma \in \Sigma\} \\
&[\![C_1; C_2]\!] = [\![C_1]\!] \circ [\![C_2]\!] \\
&[\![\textbf{if } b \textbf{ then } C_1 \textbf{ else } C_2 \textbf{ fi}]\!] = \\
&\qquad \{(\sigma, \sigma') \mid \sigma(b) \wedge (\sigma, \sigma') \in [\![C_1]\!]\} \cup \{(\sigma, \sigma') \mid \neg\sigma(b) \wedge (\sigma, \sigma') \in [\![C_2]\!]\} \\
&[\![\textbf{while } b \textbf{ do } C \textbf{ od}]\!] = \mu\Phi \quad \text{where} \\
&\qquad \Phi(\phi) = \{(\sigma, \sigma') \mid \sigma(b) \wedge (\sigma, \sigma') \in \phi \circ [\![C]\!]\} \cup \{(\sigma, \sigma') \mid \neg\sigma(b)\}\,.
\end{aligned}
$$

Figure 2.2: Relational semantics of the **while** language

For a simple sequential imperative **while** language, the default semantic domain used is relation; each command is defined as a relation between states. This is usually called relational semantics. Figure 2.2 shows the semantics for a **while** language. Here, $Id_\Sigma = \{(x, x) \mid x \in \Sigma\}$ is the unit relation,

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

denotes the relational composition and

$$R^* = \bigcup_{i \geq 0} R^i$$

is the reflexive-transitive closure relation, in which R^i is defined recursively as $R^0 = Id_\Sigma$ and $R^{i+1} = R \circ R^i$, for all $i \in \mathbb{N}$. The operator μ states that $\mu\Phi$ is the least fixed point of $\Phi(\phi)$, that is, it is the smallest relation φ such that $\Phi(\varphi) = \varphi$.

Proposition 2.1 ([5])**.** *The relational and big-step semantics for a* **while** *language are equivalent, that is,*

$$(\sigma, \sigma') \in [\![C]\!] \Leftrightarrow (C, \sigma) \to \sigma'.$$

We can lift the evaluation of expressions to the subidentity $[\![b]\!] = \{(\sigma, \sigma) \mid \sigma(b)\}$, where the semantic symbol is overloaded. This simplifies the semantic definition of conditionals and loops.

Proposition 2.2. *The relational semantic of conditionals and loops are equal to the following equations*

$$\textbf{if } b \textbf{ then } C_1 \textbf{ else } C_2 \textbf{ fi} = [\![b]\!] \circ [\![C_1]\!] \cup [\![\neg b]\!] \circ [\![C_2]\!],$$
$$\textbf{while } b \textbf{ do } C \textbf{ od} = ([\![b]\!] \circ [\![C]\!])^* \circ [\![\neg b]\!].$$

Note that, in these definitions, the boolean expression b behaves like a command, filtering out the states where b holds. This interesting twofold nature of boolean expressions, or *tests*, will be exploited in Chapter 3.

Other semantic domains can be used to define the denotational semantics for a programming language, including predicate transformers, languages over an alphabet Σ and Aczel traces.

2.2 Hoare Logic

Hoare logic is a sound and *relatively* complete [4] programming logic for showing the correctness of **while** programs. It is said to be relatively complete, because its completeness depends on the *expressivity* of the assertion language used. The approach was also advocated as a way to give meaning to a programming language, for that reason, it is also known as *axiomatic semantics*. Its central feature is the Hoare triple $\{P\}\ C\ \{Q\}$, which asserts that for all states σ satisfying the precondition P, if the execution of C from σ terminates in a state σ' then σ' satisfies the postcondition Q.

Figure 2.3 shows the rules of Hoare logic for partial correctness. Substitution is lifted over the assertion language and usually defined by structural induction.

$$\{P\}\ \textbf{skip}\ \{P\} \tag{skip}$$

$$\{Q[t/u]\}\ u := t\ \{Q\} \tag{assign}$$

$$\frac{P \to P' \qquad \{P'\}\ C\ \{Q'\} \qquad Q' \to Q}{\{P\}\ C\ \{Q\}} \tag{conseq}$$

$$\frac{\{P\}\ C_1\ \{Q\} \qquad \{R\}\ C_2\ \{Q\}}{\{P\}\ C_1; C_2\ \{Q\}} \tag{seq}$$

$$\frac{\{P \wedge b\}\ C_1\ \{Q\} \qquad \{P \wedge \neg b\}\ C_2\ \{Q\}}{\{P\}\ \textbf{if}\ b\ \textbf{then}\ C_1\ \textbf{else}\ C_2\ \textbf{fi}\ \{Q\}} \tag{cond}$$

$$\frac{\{I \wedge b\}\ C\ \{I\}}{\{I\}\ \textbf{while}\ b\ \textbf{do}\ C\ \textbf{od}\ \{I \wedge \neg b\}} \tag{while}$$

Figure 2.3: Rules of Hoare logic

In the while rule, we say that I is a loop *invariant*. We write $\to$ for implication. Note that assertions are mixed with boolean expressions in the conditional and while rule. In matter of fact, the language where the boolean expression is written is usually a subset of the assertion language.

Hoare logic can be strengthened to reason about total correctness [4], where the while rule is replaced with

$$\frac{\{I \wedge b\}\ C\ \{I\} \qquad P \to i \geq 0 \qquad \{I \wedge b \wedge i = n\}\ C\ \{i < n\}}{\{I\}\ \textbf{while}\ b\ \textbf{do}\ C\ \textbf{od}\ \{I \wedge \neg b\}},$$

where a positive number i decreases after each iteration of the loop.

Any correctness tool uses some variant of Hoare logic, which needs to be linked to the programming language semantics. For the tool to be considered safe and sound, a soundness proof needs to be given and it is usually done by structural induction over the language statements. Changes in the programming logic or in the language semantics force the developers to provide a new proof. However, the use of abstract algebras simplifies the approach and gives more flexibility when changing semantics or adding new inference rules [11].

2.3 Weakest Preconditions

The weakest precondition approach was initially advocated by Dijkstra in his seminal monograph [40]. For each programming construct C and postcondition

$$[\textbf{skip}]Q = Q$$

$$[u := t]Q = Q[t/u]$$

$$[C_1; C_2]Q = [C_2][C_1]Q$$

$$[\textbf{if } b \textbf{ then } C_1 \textbf{ else } C_2 \textbf{ fi}]Q = (b \to [C_1]Q) \wedge (\neg b \to [C_2]Q)$$

$$[\textbf{while } b \textbf{ do } C \textbf{ od}]Q = I \wedge (b \wedge I \to [C]I) \wedge (\neg b \wedge I \to Q)$$

Figure 2.4: The weakest liberal precondition operator for a **while** language

Q, the weakest precondition is an assertion that characterizes the set of all initial states such that if C is executed in these states, then it will terminate in a state satisfying the postcondition Q. In this context, assertions are identified as sets of states. It is the weakest due to the fact that it characterises *all* initial states. A weaker concept can also be defined, the weakest liberal precondition, in which termination is not guaranteed.

We write $[C]Q$ for the weakest liberal precondition of a command C leaving the system in a final state satisfying postcondition Q. The notation comes from modal logic and it will be clarified in Chapter 7. The operation is inductively defined over the structure of the programming language; Dijkstra argued that this operator should be strict, isotone, conjunctive and weakly disjunctive, the so called *healthiness conditions*. In other words, it satisfies the following properties:

$$[C]\bot = \bot, \tag{2.1}$$

$$P \to Q \Rightarrow [C]P \to [C]Q, \tag{2.2}$$

$$[C](P \wedge Q) = [C]P \wedge [C]Q, \tag{2.3}$$

$$[C](P \vee Q) \to [C]P \vee [C]Q, \tag{2.4}$$

where $\bot$ represents the assertion that characterizes the empty set of states, $\wedge$ is its conjunction and $\vee$, its disjunction. The last property can be strengthened to an equality on deterministic programs.

Figure 2.4 shows the liberal precondition operator for a **while** language. The weakest precondition can be used to derive Hoare logic, where the validity of the Hoare triple is defined to be

$$\vDash \{P\}\ C\ \{Q\} \Leftrightarrow P \to [C]Q. \tag{2.5}$$

It also yields a semantics for programming languages known as *predicate transformer semantics*, where a command C is seen as a mapping from a set of states satisfying a postcondition Q to a set of states satisfying $[C]Q$.

2.4 Separation Logic

Separation logic is, in its basic format, an extension of Hoare logic for reasoning about pointer manipulation. It was first developed in the early 2000s by Reynolds, O'Hearn, Yang and Ishtiaq [93, 101, 66, 91]. It is based on the logic of bunched implications (BI) [92] and on previous work with pointers by Burstall [29]. Its central feature is the *separating conjunction* $P * Q$, or *spatial conjunction*, which asserts that the predicates P and Q hold for different parts of the memory. The operation provides reasoning about programs with pointers in a modular fashion, isolating the part of a program that is affected by a command from the remainder.

Hoare triples $\{P\}\ C\ \{Q\}$ have a *fault-avoidance* interpretation. Whenever precondition P holds for the initial state, the command C will execute without any fault, such as for instance trying to access a dangling pointer. Additionally, Hoare logic is extended with the *frame rule*:

$$\frac{\{P\}\ C\ \{Q\}}{\{P * R\}\ C\ \{Q * R\}} \qquad \text{(frame)}$$

The name comes from an analogy with animation, where an unchanging frame has dynamic parts of the scene are drawn over it [91]. The rule states that if from an initial state satisfying P a command C safely executes and finishes in a final state where Q holds, then it will also safely execute if the precondition is spatially conjoint with a predicate R and the final state will satisfy $Q * R$. The soundness of this rule is extremely delicate [120], in fact, it usually relies on a relation between the command C and the predicate R. In a concrete model for imperative while programs with pointers, this relation states that the set of modified variables in C and the free variables in R are disjoint. A more abstract condition, called locality, can be used to prove soundness [30].

As a relatively recent development, several other versions of the logic were proposed to deal with different tasks of program verification, such as fractional permissions [26], variables as resource [27], concurrent separation logic [90] and *views* framework [41].

2.5 Refinement Calculus

In 1968, Dijkstra [39] introduced a constructive approach to program verification by stepwise refinement. He proposed that, from the program specification, one should be able to carry out a verified refinement step in such way that it would preserve program correctness. After a finite number of iteration, the final program would then be correct by construction. We write $C_1 \sqsubseteq C_2$ for the refinement of the program C_1 to the program C_2; any program specification satisfied by C_1 will also be satisfied by C_2.

$$w : [Q, Q] \sqsubseteq \textbf{skip} \qquad \text{(skip)}$$

$$u : [Q[t/x], Q] \sqsubseteq u := t \qquad \text{(assign)}$$

$$\text{If } u \neq w \text{ then } u, w : [P, Q] \sqsubseteq w : [P', Q] \qquad \text{(contract)}$$

$$\text{If } Q' \to Q \text{ then If } P \to P \text{ then } w : [P, Q] \sqsubseteq w : [P', Q] \qquad \text{(weak)}$$

$$\text{If } Q' \to Q \text{ then } w : [P, Q] \sqsubseteq w : [P, Q'] \qquad \text{(strengh)}$$

$$w : [P, Q] \sqsubseteq w : [P, M]; w : [M, Q] \qquad \text{(seq)}$$

$$w : [P, Q] \sqsubseteq \textbf{if } b \textbf{ then } w : [P, b \wedge Q] \textbf{ else } w : [P, \bar{b} \wedge Q] \textbf{ fi} \qquad \text{(cond)}$$

$$w : [I, \bar{b} \wedge I] \sqsubseteq \textbf{while } b \textbf{ do } w : [I \wedge b, I] \textbf{ od}. \qquad \text{(while)}$$

Figure 2.5: Morgan's refinement laws

The refinement steps used by Dijkstra were however far from simple; it was very difficult to judge if a step was correct by intuition. A formal refinement calculus was then proposed by Ralph-Johan Back in his PhD thesis [16], in which the correctness of each refinement step was formally proved, providing a rigorous foundation for the stepwise refinement approach.

Later, Morgan [82] proposed a new operation for the refinement calculus, called the *specification statement* $[P, Q]$, which is the most general program starting from a state satisfying predicate P and finishing in a state satisfying Q. Therefore, one would begin the program development by a specification statement and then carry out refinement steps until an executable program. A specification statement has an evident equivalence with a Hoare triple:

$$\{P\}\ C\ \{Q\} \Leftrightarrow [P, Q] \sqsubseteq C.$$

Figure 2.5 shows some of the refinement laws proposed by the Morgan's refinement calculus [83]. The laws use an extended version of the specification statement $w : [P, Q]$, which states that the program satisfying the specification can only change the variables contained in the frame w (a list of program variables). Morgan has also proposed more refinement laws, such as expanding the frame, renaming variables, creating procedure blocks and recursive functions.

Part I

Construction and Verification of *while* Programs

Chapter 3

Algebras for Program Correctness

This chapter presents the main algebraic structures for reasoning about imperative programs and explains their connection with binary relations, which is the standard model for imperative programs used in this thesis. It shows how to use Isabelle/HOL to create various mathematical components and presents our formalisation of variants of KAT and DRA. Finally, examples of program transformation in the end of the chapter illustrate the algebraic approach. The material of this chapter has been published in [8] and [11].

3.1 Kleene Algebras

Programs are usually first-class objects in the algebra: they are formed by composing programs inductively from primitive commands using a small set of operators. The simplest of these program operators are $+$ for (binary) nondeterministic choice and $\cdot$ for sequential composition. The first one is clearly commutative and both operators are associative. Two primitive and essential commands are **abort**, which aborts or annihilates any program execution, and **skip**, which maintains the previous program state and just terminates. These commands are respectively, identity for nondeterministic choice and for sequential composition. We write 0 for **abort** and 1 for **skip**. These facts, together with distributivity and annihilation laws connecting both operators, suggest that any algebra modelling the behaviour of non-probabilistic and sequential programs on a set S should form a *semiring*. Additionally, nondeterministic choice is obviously idempotent, forming a idempotent semiring, or *dioid*.

Formally, a semiring is a structure $(S, +, \cdot, 0, 1)$ over the set S such that

- $(S, +, 0)$ is a commutative monoid, *i.e.*,

$$(x + y) + z = x + (y + z), \tag{3.1}$$
$$x + 0 = x, \tag{3.2}$$
$$0 + x = x, \tag{3.3}$$
$$x + y = y + x; \tag{3.4}$$

  hold for all $x, y, z \in S$.

- $(S, \cdot, 1)$ is a monoid, *i.e.*,

$$(x \cdot y) \cdot z = x \cdot (y \cdot z), \tag{3.5}$$
$$x \cdot 1 = x, \tag{3.6}$$
$$1 \cdot x = x; \tag{3.7}$$

  hold for all $x, y, z \in S$.

- the distributivity laws

$$x \cdot (y + z) = x \cdot y + x \cdot z, \tag{3.8}$$
$$(x + y) \cdot z = x \cdot z + y \cdot z \tag{3.9}$$

- and the annihilation laws

$$0 \cdot x = 0, \tag{3.10}$$
$$x \cdot 0 = 0 \tag{3.11}$$

  hold for all $x, y, z \in S$.

A dioid is a semiring S in which addition is idempotent, that is, $x + x = x$ holds for all $x \in S$. The reduct $(S, +, 0)$ then forms a join-semilattice with least element 0 and with semilattice order defined, as usual, as

$$x \leq y \Leftrightarrow x + y = y. \tag{3.12}$$

Most of the axioms are obvious consequences of the interpretation, for instance, associativity and non-commutativity of sequential composition, or idempotency of choice. Others, such as the annihilation law $x \cdot 0 = 0$ or the left distributivity law $x \cdot (y + z) = x \cdot y + x \cdot z$, require further explanation. They are justified by the soundness proofs with respect to the relational semantics of imperative programs.

A comprehensive list of dioid properties can be found in the Archive of Formal Proofs entry for Kleene algebras [14]. Here we only mention isotonicity of addition and multiplication, that is,

$$x \leq y \Rightarrow x + z \leq y + z, \tag{3.13}$$
$$x \leq y \Rightarrow xz \leq yz, \tag{3.14}$$
$$x \leq y \Rightarrow zx \leq zy. \tag{3.15}$$

Here and henceforth we drop the multiplication symbol.

To model **while** programs in a partial correctness setting, a notion of finite iteration is needed; potentially infinite loops need not be considered. This is obtained by expanding dioids to Kleene algebras (KA).

Formally a *Kleene algebra* $(K, +, \cdot, ^*, 0, 1)$ is a dioid expanded by a star operator * which satisfies for all $x, y, z \in K$, the *unfold* and *induction axioms*

$$1 + x^*x \leq x^*, \tag{3.16}$$
$$1 + xx^* \leq x^*, \tag{3.17}$$
$$z + yx \leq y \Rightarrow zx^* \leq y, \tag{3.18}$$
$$z + xy \leq y \Rightarrow x^*z \leq y. \tag{3.19}$$

This axiomatises finite iteration as a least fixpoint as usual. Formally, x_0 is a least pre-fixpoint of a function f if and only if $f(x_0) \leq x_0$ and $f(y) \leq y \Rightarrow x_0 \leq y$ hold for all x and y. It follows from the axioms that x^* is the least pre-fixpoint of the function $\lambda\alpha.\ 1 + x \cdot \alpha$. It is also straightforward to show that $1 + x \cdot x^* = x^*$; hence x^* is in fact a fixpoint of $\lambda\alpha.\ 1 + x \cdot \alpha$. Similarly, it is also a (pre-)fixpoint of $\lambda\alpha.\ 1 + \alpha \cdot x$. The two induction axioms are perhaps stronger than expected. They impose that x^*z is also a (pre-)fixpoint of $\lambda\alpha.\ z + x \cdot \alpha$ and that zx^* is a (pre-)fixpoint of $\lambda\alpha.\ z + \alpha \cdot x$. From a mathematical point of view, the two induction axioms amalgamate a fixpoint property and a continuity property in an intricate way. From a computational point of view, they link iteration with tail recursion. This is explained further in Chapter 6. In addition, the power of the induction laws is mandatory for applications.

Kleene algebra can be understood as the algebra of regular expressions under the regular operations $+$, $\cdot$ and *. In fact, the regular expressions over an alphabet Σ form the ground terms over Σ in the language of Kleene algebras. By standard results of formal language theory, the regular languages over the alphabet Σ arise as the images of the regular expressions under the interpretation homomorphism that maps regular expressions into languages. In particular, the regular languages over Σ are generated freely by Σ in the variety of Kleene algebras. This implies that an equation is derivable from the axioms of Kleene algebra if and only if its terms are interpreted as the same regular language. The equational theory of

Kleene algebra is therefore decidable by automata-theoretic tools—it is PSPACE-complete—and the well known identities from formal language theory hold in Kleene algebras, for instance,

$$x^*x^* = x^*, \tag{3.20}$$
$$x^{**} = x^*, \tag{3.21}$$
$$1 + xx^* = x^*, \tag{3.22}$$
$$1 + x^*x = x^*, \tag{3.23}$$
$$x(yx)^* = (xy)^*x, \tag{3.24}$$
$$(x + y)^* = x^*(yx^*)^*. \tag{3.25}$$

For verifying imperative programs, however, binary relations are the semantics of choice. Hence consider the set $A \times A$ of all binary relations over a set A under set union $\cup$, relational composition $\circ$, the empty relation $\emptyset$, the unit relation Id_A and the reflexive transitive closure relation *.

Proposition 3.1. *Let A be a set. Then the structure $(2^{A\times A}, \cup, \circ, ^*, \emptyset, Id_A)$ forms a Kleene algebra.*

This structure is called the *full relational Kleene algebra* over A. Furthermore it can be shown that every subalgebra of a full relational Kleene algebra forms a Kleene algebra—called *relational Kleene algebra*. For an comprehensive overview of variants of Kleene algebras, their laws and their models, see [14].

Interestingly for verification purposes, it has been shown that the equational theory of regular languages is isomorphic to the equational theory of binary relations under the operations of Kleene algebras. This means that the equational theory of relational Kleene algebras is decidable as well. A decision procedure for this theory has already been implemented in Isabelle [77] and Coq [97].

3.2 Kleene Algebras with Tests

For modelling tests in conditionals and loops, or assertions of programs, additional structure is needed. For this purpose, Kleene algebra must be enriched with a notion of *test*, which, in a concrete model, would correspond to predicates over the program state.

Formally a *test dioid* is a dioid with a boolean algebra embedded into the subalgebra of elements between 0 and 1, that is, a structure $(S, B, +, \cdot, ^-, 0, 1)$ such that

- $(S, +, \cdot, 0, 1)$ is a dioid and

- $(B, +, \cdot, \bar{}, 0, 1)$ with $B \subseteq S$ is a boolean algebra.

The elements 0 and 1 of the dioid correspond respectively to the least and greatest elements of the boolean algebra. Similarly, $+$ corresponds to join and $\cdot$ to meet. Complementation $\bar{}$ has no counterpart in the dioid, it is a partial operation that is defined on B, but not on $S - B$.

$$\begin{aligned} \textbf{abort} &= 0, \\ \textbf{skip} &= 1, \\ x; y &= xy, \\ \textbf{if } b \textbf{ then } x \textbf{ else } y \textbf{ fi} &= bx + \bar{b}y, \\ \textbf{while } b \textbf{ do } x \textbf{ od} &= (bx)^*\bar{b}. \end{aligned}$$

Figure 3.1: Algebraic semantics for a **while** language.

A *Kleene algebra with tests* (KAT) [72] is a Kleene algebra that is also a test dioid. An algebraic semantics for conditional and while loops can now be given. Figure 3.1 shows the full algebraic semantics for a **while** language. Here and henceforth we write x, y, z for general elements and b, c for tests. Multiplying a program x with a test b from the left means restricting the input of the program to those states where the test holds; multiplying from the right means output restriction. This behaviour is clear in the relational model, in which the composition of a subidentity P by a relation R filters out the elements in the domain of R that are not in P. Similar reasoning is applied when composing a relation R with a subidentity Q.

In the case of KAT the canonical model is given by the so-called guarded regular languages, which is essentially a refined regular language model. It can be shown that the guarded regular languages over an alphabet Σ are generated freely by Σ in the variety of KAT. Therefore an equation is derivable from the axioms of KAT if and only if its terms are interpreted as the same guarded regular language [76]. The equational theory of KAT is again decidable by automata-theoretic tools—and still PSPACE-complete. This decision procedure has been implemented in Coq [97].

More importantly for our purposes, the standard relational semantics of imperative programs forms a model of KAT, as indicated by the following soundness result.

Proposition 3.2. *Let A be a set. Then $(2^{A\times A}, B, \cup, \circ, ^*, \bar{}, \emptyset, Id_A)$ forms a Kleene algebra with tests, where*

- $B = \{(x, y) \mid (x, y) \in Id_A\}$ *is the set of all subidentities in* $2^{A \times A}$ *and*
- *complementation is defined as* $\bar{P} = \{(x, y) \mid (x, y) \in Id_A \wedge (x, y) \notin P\}$.

This model is called the *full relational* KAT over A and once again every subalgebra forms a relational KAT. The equational theories of relational and guarded regular language KAT are again isomorphic [76], hence the equational theory of relational KAT is decidable in PSPACE as well. Interestingly, decidability can be extended into the realm of universal Horn clauses with all equations in the antecedent being of the form $t = 0$. Using a procedure known as *hypothesis elimination*, such quasi-identities can be reduced to equivalent identities [76]. It turns out that the inference rules of propositional Hoare logic fall into that class.

In sum, KAT provides a minimalist algebraic system that captures the standard relational semantics for the control flow of while programs and it is even complete with respect to the equational theory of those programs under the relational semantics. The power of KAT for reasoning under assumptions, which is essential for program construction and verification, has been demonstrated in various applications outlined in the introduction.

3.3 Isabelle/HOL Formalisation

We have formalised Kleene algebra with tests in the Isabelle/HOL theorem proving environment. Some basic features of the tool are introduced while discussing our formalisation. For further information on Isabelle/HOL we refer to its excellent standard documentation [89]. Our full formalisation of KAT is available from the Archive of Formal Proofs [9] and can be read in parallel.

Isabelle is an interactive proof assistant with embedded first-order automatic theorem provers, SMT-solvers and counterexample generators, apart from provers, solvers and simplifiers for higher-order logic. As an LCF-style framework it is based on a small logical core to guarantee correctness. All algebras and models implemented are consistent with respect to this core and all theorems proved are correct relative to it. In particular, all proof outputs produced by external theorem provers must be internally reconstructed in order to be accepted. Isabelle has been used to formalise a wide range of mathematical theories and applied in numerous computing applications, including program correctness and verification. Isabelle/HOL, in particular, is based on a typed higher-order logic which supports reasoning with sets, polymorphic data types, inductive definitions and recursive functions.

Algebraic hierarchies like the one presented in the previous section are typically formalised within Isabelle's axiomatic type class infrastructure. As an exam-

ple, the following type class formalises dioids using the existing class for semirings and expanding it with the idempotency axiom for addition.

class *dioid* = *semiring* +
  assumes *add-idem*: $x + x = x$

The operation of addition used by Isabelle is polymorphic; it has type $\alpha \Rightarrow \alpha \Rightarrow \alpha$. The type class mechanism supports theory expansion and the formalisation of subclass relationships. Theorems proved for reducts or superclasses thus become automatically available in expansions or subclasses. The fact that dioids form a join-semilattice can, for instance, be captured by the following statement.

subclass *(***in** *dioid) join-semilattice*
  by *unfold-locales (auto simp add: add.commute add.left-commute)*

The first line of the statement indicates the proof obligation, which is to prove that the join semilattices form a subclass of dioids. When calling the tactic *unfold-locales*, Isabelle generates all the subgoals necessary to prove this claim. Because a join-semilattice is a set equipped with an associative, commutative and idempotent join operation, these are as follows.

1. $\forall\ x\ y\ z.\ (x + y) + z = x + (y + z)$
2. $\forall\ x\ y.\ x + y = y + x$
3. $\forall\ x.\ x + x = x$

These subgoals are automatically discharged by calling Isabelle's internal theorem prover *auto* with lemmas named *add.commute* and *add.left-commute* as parameters. These lemmas have been proved in the context of semirings and are available in the type class of *dioid* due its definition by expansion.

Isabelle contains a range of built-in tactics, provers and simplifiers in addition to *auto*. In particular, its internal Sledgehammer tool is able to invoke external automated theorem provers and SMT solvers and reconstruct their output internally in order to guarantee trustworthiness. Isabelle also offers different modes of interactive reasoning, notably the proof scripting language Isar which supports human-readable proofs. The example in Figure 3.2 proves a *co*-simulation law for the Kleene star in the context of Kleene algebras. The proof is split into simple human-readable steps, which are proved automatically by Sledgehammer and internally verified by the prover metis. A fully automatic proof by Isabelle is possible as well. The lemma is named *star-cosim* and can be used in future proofs. *metis* is the name of Isabelle's internal theorem prover which reconstructs proofs given by the external provers called by Sledgehammer. Lemmas such as

lemma *star-cosim:* $z{\cdot}x \leq y{\cdot}z \longrightarrow z{\cdot}x^\star \leq y^\star{\cdot}z$
proof
  assume $z{\cdot}x \leq y{\cdot}z$
  hence $y^\star{\cdot}z{\cdot}x \leq y^\star{\cdot}y{\cdot}z$
    by *(metis mult-isol mult-assoc)*
  also have $\ldots \leq y^\star{\cdot}z$
    by *(metis mult-isor star-1r)*
  finally have $z + y^\star{\cdot}z{\cdot}x \leq y{\cdot}z$
    by *(metis add-lub-var mult-1-left mult-isor star-ref)*
  thus *?thesis*
    by *(metis star-inductr)*
qed

Figure 3.2: Example of proof in Isar, *co*-simulation law for Kleene star.

mult-isol or *star-ref*, which are used by metis, are not provided by the user, but selected by the Sledgehammer tool according to syntactic criteria and by using machine learning.

An important Isabelle feature is that the mathematical structures formalised are all polymorphic—their elements can have various types. Within this infrastructure, abstract algebras can be linked formally with their models by instantiation or interpretation statements. This, for instance, allows us to verify Proposition 3.1 with Isabelle, thus formalising soundness of Kleene algebras with respect to binary relations.

interpretation *rel-kleene-algebra: kleene-algebra (op* $\cup$*) (op O) Id* $\{\}$
  (op $\subseteq$*) (op* $\subset$*) rtrancl*
  ⟨ *proof* ⟩

Once more, Isabelle generates a series of proof obligations that must be discharged by calling its simplifiers and provers. By formalising such soundness results in Isabelle, theorems are automatically propagated across classes and models, as supported by polymorphism. Those proved for Kleene algebra become available automatically for relations, and in particular for relations over concrete detailed store models.

Our formalisation of Kleene algebra with tests integrates into the existing Kleene algebra hierarchy in Isabelle [14]. More precisely, we have formalised KAT as an expansion of Kleene algebras. Due to the nature of tests, we have formalised three different approaches and developed comprehensive libraries of facts for these. The first formalisation is one-sorted. It implements functions for tests and antitests (boolean test complements) that generate the boolean algebra

of tests as their image. The second, a two-sorted algebra, follows the standard approach of embedding a boolean algebra of tests into a dioid. Both approaches are purely axiomatic; they do not mention an underlying carrier set. While such axiomatic versions often suffice for verification applications, a third formalisation with explicit carrier sets is provided as a basis for mathematical investigations.

The first implementation is based on an unpublished manuscript by Peter Jipsen and Georg Struth. It is inspired by a previous axiomatisation of domain semirings in [38]. The main idea is to add a function t to a semiring or dioid S and axiomatise it in such a way that the image $t(S)$ forms a boolean subalgebra of tests. The function t is assumed to be a retraction, that is, it satisfies $t \circ t = t$. It then follows from general properties of retractions that $p \in t(S)$ if and only if $t(p) = p$, where $t(S)$ denotes the image of S under t. We can use this fixpoint property for typing tests and verifying closure conditions. For encoding test complementation, however, it is more suitable to axiomatise an antitest function n which satisfies $t = n \circ n$.

class *dioid-tests-zerol* = *dioid-one-zerol* + *comp-op* +
  assumes *test-one*: $n\ n\ 1 = 1$
  and *test-mult*: $n\ n\ (n\ n\ x \cdot n\ n\ y) = n\ n\ y \cdot n\ n\ x$
  and *test-mult-comp*: $n\ x \cdot n\ n\ x = 0$
  and *test-de-morgan*: $n\ x + n\ y = n\ (n\ n\ x \cdot n\ n\ y)$

abbreviation *test-operator* :: $'a \Rightarrow 'a$ **where** $t\ x \equiv n\ (n\ x)$

As a matter of fact, if these axioms are added to an arbitrary semiring, idempotence is enforced. It is straightforward to verify that tests satisfy the boolean algebra axioms, but unfortunately this fact cannot be expressed explicitly in Isabelle by a subclass or sublocale statement, simply because the carrier set S is not explicit in a type class. Thus we cannot formally integrate Isabelle's library for boolean algebra and had to build up our own with the most important boolean theorems for tests. The expansion of test dioids to Kleene algebras with tests is straightforward.

class *kat* = *kleene-algebra* + *dioid-tests*

We have also verified that our test axioms are independent, using Isabelle's counterexample generator nitpick to find counterexamples when trying to prove each individual axiom from the remaining ones. Despite its limitations, this formalisation is simple and yields a high degree of automation. Overall, 122 theorems about Kleene algebras with tests and boolean algebra were proved, all of them fully automatically.

The second, a two-sorted formalisation of test dioids, integrates Isabelle's boolean algebra type class more directly. Its implementation is due to Alasdair Armstrong. As it is two sorted, it requires Isabelle's locale mechanism instead of type classes. For our purpose, however, the precise differences between type classes and locales are irrelevant.

locale *dioid-tests-zerol* =
  fixes *test* :: $'a$::*boolean-algebra* ⇒ $'b$::*dioid-one-zerol*
  and *not* :: $'b$::*dioid-one-zerol* ⇒ $'b$::*dioid-one-zerol*
  assumes *test-sup*: *test* (*sup p q*) = ‘*p* + *q*‘
  and *test-inf*: *test* (*inf p q*) = ‘*p* · *q*‘
  and *test-top*: *test top* = *1*
  and *test-bot*: *test bot* = *0*
  and *test-not*: *test* (−*p*) = ‘−*p*‘
  and *test-inj*: ‘*p* = *q*‘ ⟶ *p* = *q*

In this formalisation, the function *test* embeds the boolean algebra into the dioid as usual. From a mathematical point of view it is an injective boolean algebra homomorphism. The class of dioids with tests is expanded by the operation *not*, which corresponds to boolean complementation at the level of embedded tests. The function ‘ − ‘ acts on the syntax tree of expressions putting the string *test* in front of each leaf labelled with one of the variables commonly used for tests. This replaces expressions using the function *test* with the typical KAT notation, where the embedding is implicit. Hence with this syntax translation, one can write ‘$p + q$‘ for the join of two tests.

In this two-sorted approach, Isabelle's libraries for boolean algebras become once more automatically available. From an automation point of view, however, we noted little difference between the two approaches. Here we do not explicitly show the expansion of test dioids to Kleene algebras with tests.

The third and last implementation of test dioids provides explicit carrier sets. It follows the general Isabelle recipe for setting up such algebras. This formalisation expands carrier-based formalisations of dioids and boolean algebras. In this setting, algebraic signatures are specified in records. The axioms yield a dioid where the carrier set of tests is a subset of the main carrier and the operations are embedded as usual.

This approach is supported by Armstrong's carrier-based background theories, implemented from scratch with more than 250 theorems about lattices, dioids and Kleene algebras. Because of the additional constraints, Sledgehammer may struggle to automate simple proofs. Hence there is a trade-off between mathematical precision and automation.

record *'a test-dioid-structure* = *'a dioid* + *test* :: *'a ord*

abbreviation *tests A* ≡ *carrier* (*test A*)

locale *dioid-tests-zerol* =
  fixes *A* :: *'a test-dioid-structure* (**structure**)
  assumes *is-dioid*: *dioid-zerol A*
  and *test-subset*: *tests A* ⊆ *carrier A*
  and *test-le*: *le* (*test A*) = *dioid.nat-order A*
  and *test-ba*: *boolean-algebra* (*test A*)
  and *test-one*: *top* (*test A*) = *1*
  and *test-zero*: *bot* (*test A*) = *0*
  and *test-join*: ⟦*x*, *y* ∈ *tests A*⟧ ⟹ *join* (*test A*) *x y* = *x* + *y*
  and *test-meet*: ⟦*x*, *y* ∈ *tests A*⟧ ⟹ *meet* (*test A*) *x y* = *x* · *y*

In sum, the three formalisations all have their advantages and disadvantages. The one-sorted and two-sorted implementation offer comparable proof automation and might be superior for program verification applications. The carrier-based implementation leads to less automatic proofs, but for investigations in universal algebra, for instance, this price needs to be paid.

3.4 Propositional Hoare Logic

Propositional Hoare logic (PHL) denotes Hoare logic without the assignment rule. Hence it contains precisely the rules of Hoare logic which concern the control flow. It is well known that this fragment can be derived within KAT [73], hence KAT ⊆ PHL, and it is easy to see that it is even a strict subclass. Hoare logic essentially offers one inference rule per programming construct for simple **while** programs and its soundness and relative completeness with respect to the standard relational semantics makes it a powerful tool for verification condition generation: it can be applied in nondeterministic fashion to the syntax tree of any simple while-program.

This section briefly reports on the derivation of PHL in KAT with Isabelle. The central feature of Hoare logic is the Hoare triple, the validity of which can be encoded in KAT as

$$\{p\}\ x\ \{q\} \ \Leftrightarrow\ px\bar{q} = 0. \tag{3.26}$$

The right-hand side states that there are no successful terminating executions of program x from states where assertion p holds into states where assertion q fails. In other words, if x is executed from precondition p and if it terminates, then postcondition q must hold after its execution. Here and henceforth we write p, q and r for assertions; although in practice assertions are tests, we differentiate

tests in conditionals or loops and assertions about the program by notation. Interestingly, this encoding leads to an expression of the form $t = 0$, which is amenable to hypothesis elimination.

Lemma 3.1. *In every* KAT*,*

$$px\bar{q} = 0 \Leftrightarrow px \leq xq. \tag{3.27}$$

Proof. Suppose $px\bar{q} = 0$. Then $px = px(q + \bar{q}) = pxq + px\bar{q} = pxq \leq xq$. Conversely suppose $px \leq xq$, then $px\bar{q} \leq xq\bar{q} = 0$. □

We show this proof in order to illustrate the simplicity of equational reasoning in KAT. It therefore comes as no surprise that such proofs can be discharged automatically in Isabelle, via its Sledgehammer tool.

The equivalence in Lemma 3.1 justifies $px \leq xq$ as an alternative definition of the validity of Hoare triples. In fact it is often preferable in proofs. Intuitively it indicates that all successful executions of x with inputs restricted by p are contained in the set of executions of x with output restricted by q.

Proposition 3.3 ([73])**.** KAT $\subseteq$ PHL*, that is, the following inference rules are derivable in* KAT*.*

$$\{p\}\ \mathbf{skip}\ \{p\}, \tag{3.28}$$

$$p \leq p' \wedge q' \leq q \wedge \{p'\}\ x\ \{q'\} \Rightarrow \{p\}\ x\ \{q\}, \tag{3.29}$$

$$\{p\}\ x\ \{r\} \wedge \{r\}\ y\ \{q\} \Rightarrow \{p\}\ x; y\ \{q\}, \tag{3.30}$$

$$\{pb\}\ x\ \{q\} \wedge \{p\bar{b}\}\ y\ \{q\} \Rightarrow \{p\}\ \mathbf{if}\ b\ \mathbf{then}\ x\ \mathbf{else}\ y\ \mathbf{fi}\ \{q\}, \tag{3.31}$$

$$\{pb\}\ x\ \{p\} \Rightarrow \{p\}\ \mathbf{while}\ b\ \mathbf{do}\ x\ \mathbf{od}\ \{\bar{b}p\}. \tag{3.32}$$

Note that $p \leq q$ in KAT *corresponds to implication $p \to q$ in Hoare logic.*

Proof. Each derivation requires essentially one line of equational reasoning. The skip rule is straightforward. In the consequence rule, assuming $p \leq p'$, $q' \leq q$ and $p'x \leq xq'$, then by monotonicity we derive $px \leq p'x \leq q'x \leq qx$. The sequential rule is proved by monotonicity and associativity of multiplication, that is, assuming $px \leq xr$ and $ry \leq yq$, then $pxy \leq xry \leq xyq$. In the conditional case, if $pbx \leq xq$ and $p\bar{b}y \leq yq$, then

$$p(bx + \bar{b}y) = pbx + p\bar{b}y = b(pbx) + \bar{b}(p\bar{b}y) \leq bxq + \bar{b}yq = (bx + \bar{b}y)q.$$

Finally, the while rule assumes $pbx \leq xp$, thus

$$pbx \leq bxp \Rightarrow p(bx)^* \leq (bx)^*p \Rightarrow p(bx)^*\bar{b} \leq (bx)^*p\bar{b} = (bx)^*\bar{b}p.$$

This last derivation uses the co-simulation lemma proved in §3.3. □

The manual proof of the while rule can be translated more or less directly into an Isabelle proof.

lemma *while-rule:* $[\![test\ p;\ test\ b;\ \{p{\cdot}b\}x\{p\}]\!] \Longrightarrow \{p\}(b{\cdot}x)^\star{\cdot}!b\{p{\cdot}!b\}$
proof *(unfold hoare-triple-def, auto)*
  assume *assms:* $test\ p$ **and** $test\ b$ **and** $p{\cdot}b{\cdot}x \leq x{\cdot}p$
  hence $b{\cdot}p{\cdot}b{\cdot}x \leq b{\cdot}x{\cdot}p$
    by *(metis mult.assoc mult-isol)*
  thus $p{\cdot}((b{\cdot}x)^\star{\cdot}!b) \leq (b{\cdot}x)^\star{\cdot}!b{\cdot}(p{\cdot}!b)$
    by *(metis assms test-mult-comm-var star-sim2 mult-isor kat-eq3 mult.assoc test2)*
qed *(metis test-comp-closed-var test-mult-closed)*

Once more this illustrates the simplicity and concision of reasoning about programs in KAT. It is reflected by the degree of automation in Isabelle proofs. Although these proofs leave nothing to desire, it is worth pointing out that the validity of all inference rules of PHL could have been decided, since all identities in antecedents are of the form $t = 0$. Obviously, formulas like $p \leq p'$ can be rewritten as $p\bar{p'} = 0$ in the consequence rule.

3.5 Demonic Refinement Algebras

Total program semantics require another variant of Kleene algebra [117, 118]. This section presents *demonic refinement algebra* (DRA), a Kleene algebra in which the right annihilation axiom $x0 = 0$ is absent and which is expanded by an operation for possibly infinite iterations $^\infty$ which satisfy the axioms

$$x^\infty = x^* + x^\infty 0, \tag{3.33}$$
$$1 + xx^\infty = x^\infty, \tag{3.34}$$
$$y \leq xy + z \Rightarrow y \leq x^\infty z. \tag{3.35}$$

They are respectively known as the isolation axiom, unfold axiom and the coinduction axiom.

This captures total correctness, since an agent has no control over termination; $(bx)^\infty\bar{b}$, for instance, models a while loop which may not terminate. For similar reasons, $x0 = 0$ is invalid due to potentially infinite processes. In the isolation axiom, x^*0 annihilates if all processes in x are finite whereas $x^\infty 0$ projects on the strictly infinite processes of x^∞. A strict infinite iteration operator $^\omega$ is sometimes used, where $x^\omega = x^\infty 0$.

The refinement community's notation unfortunately deviates from the regular algebra notation. Their refinement order $\sqsubseteq$ is the converse of $\leq$; the symbols $\top$, $\sqcap$, ; and $^\omega$ are used instead of 0, +, $\cdot$ and $^\infty$. Finally tests are known as *guards*.

We have formalised demonic refinement algebras in Isabelle/HOL, integrating into the existing Kleene algebra hierarchy [14]. More precisely, we have formalised demonic refinement algebra as an expansion of Kleene algebra with a left annihilator, adding simply the unfold, coinduction and isolation axiom for $^\infty$. By this expansion, all facts proved for this variant of Kleene algebra become automatically available in demonic refinement algebra.

class *dra* = *kleene-algebra-zerol* + *strong-iteration-op* +
  assumes *iteration-unfoldl*: $1 + x \cdot x^\infty = x^\infty$
  and *coinduction*: $y \leq z + x \cdot y \longrightarrow y \leq x^\infty \cdot z$
  and *isolation*: $x^\infty = x^\star + x^\infty \cdot 0$

We have developed a comprehensive library of theorems of demonic refinement algebra from the literature. Many equational algebraic theorems can be proved fully automatically. The semantics of choice, in this case, is a predicate transformer algebra, which we discuss in detail in Chapter 7.

Demonic refinement algebras can also be expanded by tests in the obvious way. All axiomatisations of the algebraic structures from the previous section have been also given for dioids without the right annihilation law $x0 = 0$. This makes all three formalisations compatible with demonic refinement algebra. The one-sorted formalisation of tests, for instance, is

class *dra-tests* = *dioid-tests-zerol* + *dra*

An expansion to proper test dioids is, of course, given in our Isabelle theory files.

The addition of tests or guards makes demonic refinement algebra suitable for program development applications. Additionally, we have formalised the dual notion of *assertion*. Assertions are used as context information for weakest precondition reasoning [117, 118] in guarded command languages. We have formalised assertions as $p^o = \bar{p}\top + 1$. The constant $\top$ denotes the greatest element of the demonic refinement algebra, which exists in this class and is equal to 1^∞. Intuitively, an assertion p^o aborts when p is false and skips when p is true. We have verified that guards and assertions are adjoints of Galois connections,

$$px \leq y \Leftrightarrow x \leq p^o y, \tag{3.36}$$

$$xp^o \leq y \Leftrightarrow x \leq yp, \tag{3.37}$$

as well as further properties from the literature.

Atomicity Refinement Theorem

Demonic refinement algebra is also interesting for modelling concurrency in Back's *action system* framework [18]. As a complex example we have verified three algebraic versions of Back's *atomicity refinement theorem* [17, 117, 118, 33, 65]. For an explanation we refer to these articles. Here we only discuss algebraic aspects and proof automation. Von Wright's variant states that the identity

$$x(y+z+v+w)^\infty p \leq x(yz^\infty p+v+w)^\infty$$

can be derived from the 12 assumptions

$$p \leq 1, \quad x = xp, \quad y = py, \quad pz = 0, \quad z^\infty = z^*, \quad v^\infty = v^*,$$
$$vz \leq zv, vw \leq wv, \quad vp \leq pv, \quad yw \leq wy, \quad zw \leq wz, \quad pw \leq wp.$$

Note that $z^\infty = z^*$ and $w^\infty = w^*$ express that z and w are finite. Von Wright's original proof covers about 3 pages. Our Isabelle proof essentially translates this proof at this level of granularity; a more coarse grained automation seems difficult for metis. The main reason is that the terms appearing in this proof are quite long and many rules can match. This combinatorics is difficult to handle in particular for metis, which is inferior to Sledgehammer's external provers. In fact, a more general proof of this theorem with Prover9 [65] was much more coarse grained but required excessive running times. Theorems like this provide interesting benchmarks for Sledgehammer in particular and automated theorem provers in general. This general version can also be found in our Isabelle files.

We have also verified Cohen's simplified version of the atomicity refinement theorem [33] which derives the equation

$$(x+y+z)^\infty = (pz)^\infty(x+\bar{p}z+y\bar{p})^\infty(yp)^\infty$$

from the assumptions $t\ p = p$, $x0 = 0$, $y0 = 0$, $py\bar{p} = 0$, $pz\bar{p} = 0$, $ypx \leq xy$, $xpz \leq zx$ and $ypz \leq zy$. Cohen assumes partial correctness, so we must explicitly express that x and y must terminate: $x0 = 0$ and $y0 = 0$. Our proof requires 10 particular steps with Isar.

As a last example, Figure 3.3 shows the proof of the version due to Höfner, Struth and Sutcliffe [65].

The results in this section show that libraries that support program refinement can be developed quite easily at the algebraic level with Isabelle. Demonic refinement algebra is part of more powerful calculi which have been described, for instance, in the book of Back and von Wright [19]. Their approach is based on lattice and fixpoint theory. It can easily be obtained by theory expansion from our formalisation of demonic refinement algebra.

theorem *atom-ref-back-struth*:
assumes $s \leq s{\cdot}q\ a \leq q{\cdot}a\ q{\cdot}b = 0$
$r{\cdot}b \leq b{\cdot}r\ r{\cdot}q \leq q{\cdot}r$
$(a + r + b){\cdot}l \leq l{\cdot}(a + r + b)\ q{\cdot}l \leq l{\cdot}q$
$r^{\infty} = r^{\star}\ q \leq 1$
shows $s{\cdot}(a + b + r + l)^{\infty}{\cdot}q \leq s{\cdot}(a{\cdot}b^{\infty}{\cdot}q + r + l)^{\infty}$
proof −
have $s{\cdot}(a + b + r + l)^{\infty}{\cdot}q = s{\cdot}l^{\infty}{\cdot}(a + b + r)^{\infty}{\cdot}q$
by (*metis add.commute assms*(6) *iteration-sep mult.assoc*)
also have ... $= s{\cdot}l^{\infty}{\cdot}(b + r)^{\infty}{\cdot}(a{\cdot}(b + r)^{\infty})^{\infty}{\cdot}q$
by (*metis add-assoc′ iteration-denest add.commute mult.assoc*)
also have ... $= s{\cdot}l^{\infty}{\cdot}b^{\infty}{\cdot}r^{\infty}{\cdot}(a{\cdot}b^{\infty}{\cdot}r^{\infty})^{\infty}{\cdot}q$
by (*metis assms*(4) *iteration-sep mult.assoc*)
also have ... $\leq s{\cdot}l^{\infty}{\cdot}b^{\infty}{\cdot}r^{\infty}{\cdot}(q{\cdot}a{\cdot}b^{\infty}{\cdot}r^{\infty})^{\infty}{\cdot}q$
by (*metis assms*(2) *iteration-iso mult-isol-var eq-refl order-refl*)
also have ... $= s{\cdot}l^{\infty}{\cdot}b^{\infty}{\cdot}r^{\infty}{\cdot}q{\cdot}(a{\cdot}b^{\infty}{\cdot}r^{\infty}{\cdot}q)^{\infty}$
by (*metis iteration-slide mult.assoc*)
also have ... $\leq s{\cdot}q{\cdot}l^{\infty}{\cdot}b^{\infty}{\cdot}r^{\infty}{\cdot}q{\cdot}(a{\cdot}b^{\infty}{\cdot}r^{\infty}{\cdot}q)^{\infty}$
by (*metis assms*(1) *mult-isor*)
also have ... $\leq s{\cdot}l^{\infty}{\cdot}q{\cdot}b^{\infty}{\cdot}r^{\infty}{\cdot}q{\cdot}(a{\cdot}b^{\infty}{\cdot}r^{\infty}{\cdot}q)^{\infty}$
by (*metis assms*(7) *iteration-sim mult.assoc mult-double-iso*)
also have ... $\leq s{\cdot}l^{\infty}{\cdot}q{\cdot}r^{\infty}{\cdot}q{\cdot}(a{\cdot}b^{\infty}{\cdot}r^{\infty}{\cdot}q)^{\infty}$
by (*metis assms*(3) *iteration-idep mult.assoc order-refl*)
also have ... $\leq s{\cdot}l^{\infty}{\cdot}q{\cdot}r^{\infty}{\cdot}q{\cdot}(a{\cdot}b^{\infty}{\cdot}r^{\star}{\cdot}q)^{\infty}$
by (*metis assms*(8) *eq-refl*)
also have ... $\leq s{\cdot}l^{\infty}{\cdot}q{\cdot}r^{\infty}{\cdot}q{\cdot}(a{\cdot}b^{\infty}{\cdot}q{\cdot}r^{\star})^{\infty}$
by (*metis assms*(5) *iteration-iso mult.assoc mult-isol star-sim1*)
also have ... $= s{\cdot}l^{\infty}{\cdot}q{\cdot}r^{\infty}{\cdot}q{\cdot}(a{\cdot}b^{\infty}{\cdot}q{\cdot}r^{\infty})^{\infty}$
by (*metis assms*(8))
also have ... $\leq s{\cdot}l^{\infty}{\cdot}r^{\infty}{\cdot}q{\cdot}(a{\cdot}b^{\infty}{\cdot}q{\cdot}r^{\infty})^{\infty}$
by (*metis assms*(9) *mult-1-right mult-double-iso mult-isor*)
also have ... $\leq s{\cdot}l^{\infty}{\cdot}r^{\infty}{\cdot}(a{\cdot}b^{\infty}{\cdot}q{\cdot}r^{\infty})^{\infty}$
by (*metis assms*(9) *mult-1-right mult-double-iso*)
also have ... $= s{\cdot}l^{\infty}{\cdot}(a{\cdot}b^{\infty}{\cdot}q + r)^{\infty}$
by (*metis add.commute mult.assoc iteration-denest*)
also have ... $\leq s{\cdot}(a{\cdot}b^{\infty}{\cdot}q + r + l)^{\infty}$
by (*metis add.commute iteration-subdenest mult.assoc mult-isol*)
finally show *?thesis* **.**
qed

Figure 3.3: Proof of Back's atomicity refinement theorem in Isabelle/HOL. This version is due to Höfner, Struth and Sutcliffe [65].

3.6 Loop Transformation Example

We now consider a classical program transformation example which has first been considered in the partial correctness setting of Kleene algebra with tests. We formalise Kozen's loop transformation theorem:

Proposition 3.4 ([72]). *Every sequential while program, appropriately augmented with subprograms of the form $z(bc + \bar{b}\bar{c})$, can be viewed as a while program with at most one loop under certain* preservation *assumptions.*

Hence any while program, suitably augmented with finitely many new *dummy* subprograms, is equivalent to a simple while program of the form

$$x;\ \textbf{while}\ b\ \textbf{do}\ y\ \textbf{od},$$

where x and y do not contain any nested loops.

A key ingredient of Kozen's approach are commutativity conditions of the form $bx = xb$. We use preservation conditions instead, which are of the form $bx = bxb$ and $\bar{b}x = \bar{b}x\bar{b}$. These conditions state that x preserves b, that is, the validity of b is not changed by the execution of x. In Kleene algebra with tests, these two conditions are equivalent. However we prove the transformation theorem in the weaker setting of *pre-Conway algebras*, where the former imply the latter, but not vice versa. Pre-Conway algebras are defined as

class *pre-conway* = *pre-dioid-one-zerol* + *dagger-op* +
  assumes *dagger-denest*: $(x + y)^\dagger = (x^\dagger \cdot y)^\dagger \cdot x^\dagger$
  and *dagger-prod-unfold*: $(x \cdot y)^\dagger = 1 + x \cdot (y \cdot x)^\dagger \cdot y$
  and *dagger-simr*: $z \cdot x \leq y \cdot z \longrightarrow z \cdot x^\dagger \leq y^\dagger \cdot z$

As the first line states, they are based on *pre-dioids* with only a left-annihilating zero [14]. In these structures, the left distributivity law $x(y + z) = xy + xz$ is weakened to sub-distributivity $xy+xz \leq x(y+z)$ which is equivalent to isotonicity $x \leq y \Rightarrow zx \leq zy$. Furthermore, the right annihilation law $x0 = 0$ is absent. To avoid confusion we use the operator † instead of *. The denest and product-unfold axioms are part of Conway's *classical axioms* for regular algebra [34], but several other axioms, including the idempotency axiom $x^{\dagger\dagger} = x^\dagger$, are absent. In particular, Conway's classical axioms are based on a full dioid. In fact, the dioid-based version plus dagger idempotence is equivalent to the axioms of right Kleene algebra; and complete with respect to the equational theory of regular expressions (see [52] for an overview).

In preparation to the proof of the loop transformation theorem we have verified a number of laws about the dagger in pre-Conway algebra, for instance isotonicity

and unfold laws for the dagger, and a slide rule:

$$x \leq y \Rightarrow x^\dagger \leq y^\dagger, \tag{3.38}$$

$$x^\dagger = 1 + xx^\dagger, \tag{3.39}$$

$$x^\dagger = 1 + x^\dagger x, \tag{3.40}$$

$$x(yx)^\dagger = (xy)^\dagger x; \tag{3.41}$$

along with some preservation properties,

$$bxb = bx \Rightarrow bx^\dagger = (bx)^\dagger b, \tag{3.42}$$

$$bxb = bx \Rightarrow b(bx + \bar{b}y)^\dagger = (bx)^\dagger b. \tag{3.43}$$

The proof itself is by structural induction on **while** programs. This can be formalised in Isabelle by defining a grammar for programs and imposing the quotient of pre-Conway algebra identities, using Isabelle's quotient package. We only discuss the individual cases of this inductive argument. For each program construct, an inner loop is moved to the outside of a program and these program transformations are verified in pre-Conway algebra with tests. Programs can be augmented by dummy subprograms under preservation assumptions. We follow Kozen's case analysis, but proofs for individual cases are different due to our more general assumptions and the weaker axioms of pre-Conway algebras. Following Kozen, we take the sequential composition operator to be of lower precedence than the other program constructs. We need to prove three cases: for conditionals, loops and sequential composition.

For conditionals, Kozen shows that the following programs

$z; bc + \bar{b}\bar{c};$
if b
  then $(x_1;$ **while** d_1 **do** y_1 **od**$)$
  else $(x_2;$ **while** d_2 **do** y_2 **od**$)$
fi

and

$z; bc + \bar{b}\bar{c};$
if c **then** x_1 **else** x_2 **fi**;
while $cd_1 + \bar{c}d_2$ **do** (**if** c **then** y_1 **else** y_2 **fi**) **od**

are equivalent.

Translated into pre-Conway algebra we must prove that

$$z(bc + \bar{b}\bar{c})(bx_1(d_1y_1)^\dagger\bar{d_1} + \bar{b}x_2(d_2y_2)^\dagger\bar{b_2}) = \\ z(bc + \bar{b}\bar{c})(cx_1 + \bar{c}x_2)((cd_1 + \bar{c}d_2)(cy_1 + \bar{c}y_2))^\dagger\overline{cd_1 + \bar{c}d_2}.$$

This consists of two phases. First, the two terms are simplified by right distributivity, yielding two subterms each. Second, we proved this by verifying the following two equations between these subterms, using preservation:

$$bcx_1(d_1y_1)^\dagger\bar{d_1} = bcx_1(cd_1y_1 + \bar{c}d_2y_2)^\dagger(c\bar{d_1} + \bar{c}\bar{d_2})$$
$$b\bar{c}x_2(d_2y_2)^\dagger\bar{d_2} = b\bar{c}x_2(cd_1y_1 + \bar{c}d_2y_2)^\dagger(c\bar{d_1} + \bar{c}\bar{d_2})$$

For nested loops, Kozen proves the following two programs equivalent:

while b **do** (x; **while** c **do** y **od**) **od**
if b **then** x; **while** $(b + c)$ **do** (**if** c **then** y **else** x **fi**) **od fi**

The corresponding proof in pre-Conway algebra was fully automatic. No preservation assumptions are needed for the proof.

$$(bx(cy)^\dagger\bar{c})^\dagger\bar{b} = bx((b + c)(cy + \bar{c}x))^\dagger(\overline{b + c}) + \bar{b}$$

The case of sequential composition has two subcases. The first one—called *postcomputation*—composes a while loop with a loop-free y program:

(**while** b **do** x **od**); y
if $\bar{b}$ **then** y **else** (**while** b **do** (x; **if** $\bar{b}$ **then** y **fi**) **od**) **fi**

The corresponding identity in Conway algebra is

$$(bx)^\dagger\bar{b}y = \bar{b}y + b(bx(\bar{b}y + b))^\dagger\bar{b}.$$

Due to the weaker setting, our proof differs from Kozen's.

$$\begin{aligned} b(bx(\bar{b}y + b))^\dagger\bar{b} &= b\bar{b} + bbx((\bar{b}y + b)bx)^\dagger(\bar{b}y + b)\bar{b} \\ &= bx(\bar{b}ybx + bx)^\dagger\bar{b}y\bar{b} \\ &= bx(\bar{b}y0 + bx)^\dagger\bar{b}y \\ &= bx(bx)^\dagger(\bar{b}y0)^\dagger\bar{b}y \\ &= bx(bx)^\dagger\bar{b}y(0\bar{b}y)^\dagger \\ &= bx(bx)^\dagger\bar{b}y. \end{aligned}$$

The first step uses the product unfold law. The second step uses right distributivity and boolean algebra. The third step uses the preservation assumption $\bar{b}y = \bar{b}y\bar{b}$. The forth step uses denesting and right annihilation. The fifth step uses the sliding rule. The last step uses right annihilation and the rule $0^\dagger = 1$, which can be derived from the left unfold law. Finally, adding the term $\bar{b}y$ to both sides and applying unfold yields the desired identity.

The second subcase is the composition of two while loops, which leads to the equivalence of

while b **do** x **od**;
while c **do** y **od**

and

if $\bar{b}$
  then while c **do** y **od**
  else while b **do** $(x;$ **if** $\bar{b}$ **then while** c **do** y **od fi)** **od**
fi.

The identity to be proven is

$$(bx)^\dagger\bar{b}(cy)^\dagger\bar{c} = \bar{b}(cy)^\dagger\bar{c} + b(bx(\bar{b}(cy)^\dagger\bar{c} + b))^\dagger\bar{b}.$$

Its proof has two steps. We first prove that $(cy)^\dagger\bar{c}$ preserves b, that is, $b(cy)^\dagger\bar{c} = b(cy)^\dagger\bar{c}b$ and $\bar{c}(cy)^\dagger\bar{c} = \bar{b}(cy)^\dagger\bar{c}\bar{b}$. Then we prove the identity by applying the previous subcase. This finishes the case analysis.

We have formally shown that every Kleene algebra with tests is a pre-Conway algebra where we interpret † as *.

sublocale *kat* $\subseteq$ *pre-conway star* ⟨*proof*⟩

Thus our proof generalises Kozen's result; and Isabelle makes our theorem automatically available in Kleene algebra with tests. We have also shown that every demonic refinement algebra is a pre-Conway algebra when interpreting † as $^\infty$.

sublocale *dra-tests* $\subseteq$ *pre-conway strong-iteration* ⟨*proof*⟩

Hence our result holds in demonic refinement algebra as well; our proof generalises a previous result by Solin [105].

Finally, Rabehaja and Sanders [99] have further generalised the loop refinement theorem to a probabilistic demonic refinement algebra in which the star

and the isolation axiom are absent and the left distributivity axiom is weakened to general left sub-distributivity and to a special left distributivity axiom $b(x + y) = bx + by$ for tests b. We have adapted our proof so that it covers all three cases. We do not display this most generic result here since probabilistic variants are not the subject of this article. Our Isabelle file contains all relevant details [9]. Note that left distributivity does not hold in pre-Conway algebras and that the product unfold axiom and simulation axiom cannot be derived from Rabehaja and Sanders' axioms. The decision whether the Conway-style axiom set is appropriate for probabilistic reasoning depends on the choice of probabilistic semantics.

In pre-Conway algebras, the dagger axioms are too weak to distinguish between finite and potentially infinite iteration. Conway's axiom $x^{\dagger\dagger} = x^\dagger$, which we have dropped, holds of *, but not of $^\infty$, since $x^{\infty\infty} = \top$. Conway has analysed the relevance of this axiom for regular algebras and remarked that it is equivalent to $1^\dagger = 1$. In demonic refinement algebra, however, $1^\infty = \top$.

3.7 Conclusions

This chapter extends a reference formalisation for variants of Kleene algebras in Isabelle/HOL [13] by two algebras that are important for program verification and correctness applications: Kleene algebras with tests and demonic refinement algebras. The library provides more than 10 algebraic structures and hundreds of theorems. The applicability of the algebraic approach was demonstrated by two program transformation examples: the atomicity refinement theorem and Kozen's while loop transformation theorem. The examples are independent of a concrete model and make evident the advantages of an abstract middle layer algebraic semantics of programs.

A coherent integration of algebraic methods into program analysis tools has thereby been achieved. The associated Isabelle theories in the Archive of Formal Proofs [9] serve as a reference for extensions and further applications.

Chapter 4

A Basic Verification Tool

This chapter shows how to develop a quite simple, but powerful and expressive *post-hoc* verification tool from Kleene algebras with tests and its variants. It explains the design principle of building tools that it will be used throughout this thesis. It also shows how to integrate the algebra with its data level in Isabelle/HOL. The approach has been successfully used in different contexts [8, 10, 7, 43, 11]. It presents the material published on [10, 11].

4.1 Principles of Tool Design

Isabelle offers axiomatic type classes and locales mechanisms capable of formalising algebraic hierarchies. These algebras can be linked with their models via soundness proofs, or in Isabelle jargon, via *instantiation* or *interpretation*. These mechanisms for designing modular mathematical components were explained in more detail in §3.3. Algebras axiomatised in this way are polymorphic, *i.e.*, their type is parametric and can be instantiated. For instance, Kleene algebras can be instantiated to binary relations over the same signature, and then further to binary relations over program stores with arbitrary data domain. Theorems are propagated across the hierarchy: theorems proved in an algebraic class are automatically available in all models and subclasses; models established for an algebraic class are automatically recognised as models of all superclasses.

It is therefore possible to design program correctness tools along two modular principles. The first characterises the transition from an algebraic semantics to an intermediate model and further to a concrete set-theoretical program semantics, including a notion of state as well as data types and data structures. This is shown in Figure 4.1. Computational algebras are used to specify the control flow of computation of the target programming language and to derive equationally program transformation, verification conditions or refinement laws. For example,

algebra	intermediate semantics	concrete semantics
control flow	abstract data flow	concrete data flow
verification conditions	-	**verification tools**
transformation laws	-	**construction tools**

Figure 4.1: Principles of tool design

as shown in §3.4, KAT is expressive enough to derive the rules of propositional Hoare logic (ignoring the assignment rule), thus generating verification conditions for simple **while** programs. In addition, its refinement extension rKAT (*cf.* §5.1) allows the derivation of control flow laws of Morgan's refinement calculus.

The intermediate semantics correspond to a set-theoretical computational model without a concrete notion of state, such as store and heap. Here, we restrict our attention to binary relations, which is the standard semantics of sequential imperative programs; other models include traces, languages and matrices. Chapter 7 demonstrates the use of the same approach for predicate transformer semantics. This layer can be useful to generate more expressive verification conditions, such as Kleymann's consequence inference rule. However, it cannot derive assignments axioms, since there is no notion of variables.

Finally, in the concrete semantics, states are instantiated to a concrete type. In Isabelle, states can be typically implemented as a record of program variables, in order to handle variables of differing types [103]. The update function of the record is used to interpret assignment statements of the programming language. In this layer, fully fledged Hoare logics and/or refinement calculi can now be derived as theorems.

The second principle is characterised by the use of domain-specific algebras within a subclass hierarchy for particular features of the programming language. For instance, tests in a command statement can be captured by boolean algebras, whereas the control flow of a simple **while** language is captured by KAT, which embeds the boolean algebra of tests. In addition, assertions about a program, which are more expressive than tests, can be capture by complete boolean algebras, which can again be embedded in *quantales with tests*. The latter can then express a notion of recursion and more powerful transformation laws can be derived. While these algebras focus on program construction and verification in a partial correctness setting based on a relational semantics, as seen in the previous chapter, algebras for total correctness can also be obtained by dropping some of

the Kleene algebra axioms and have been implemented in Isabelle [8, 15].

These principles with respect to algebraic subclasses and computational model refinement allow a clear and modular implementation of algebraic and set-theoretic components in Isabelle that can be combined and extended for the development of correctness tools. Ultimately, verification of concrete program are performed in the most concrete model by a *shallow* embedding of the programming language. The algorithmic verification and construction examples show in this thesis indicate that users can work under the assumption that they manipulate concrete programming language syntax, intuitive algebraic laws of programming or concrete semantic objects in a seamless way.

In addition, the use of domain-specific mathematical components splits the verification task in different concerns. When dealing with the control flow, automation is increased, since algebraic reasoning is well supported by Isabelle's integrated first-order theorem provers and SMT solvers. For set-theoretic models or inductive reasoning with data types, which is often higher-order, Isabelle offers different provers and simplification procedures. Moreover, the new proof method language Eisbach [80] available in Isabelle 2015 allows a rapid prototyping of special-purpose tactic, such as a verification condition generator. This modularity of components and separation of concerns makes program verification proofs simple, concise and highly automatic.

The elaboration of our approach with concrete models and algebras is the subject of the remainder of this thesis. Here, we restrict ourselves to the verification and refinement of imperative programs in a partial correctness setting. Similar algebras for a total correctness setting have been discussed elsewhere [8]. In this chapter, we use the relational model as the concrete model and Kleene algebra with tests as the abstract algebra. Tools for separation logic [43] (discussed in the second part of this thesis) and for concurrency verification [7] demonstrate the versatility of the approach.

4.2 Data Flow Integration

This section sets up the concrete relational semantics of imperative programs by supplying notions of program state and state updates, and uses these notions to derive Hoare's assignment axiom. This gives us a full Hoare logic as an extension of PHL based on KAT. The formalism is used for verification condition generation; it dispenses with the entire control flow in program verification applications.

Soundness of PHL

As shown in previous sections, Kleene algebra with tests is expressive enough to derive propositional Hoare logic. In order to prove that our derivation of Hoare logic is sound, it is enough to prove that the concrete semantics used form a KAT. The semantics of choice for an imperative sequential language is the relational model, in which the semantics of each command of the **while** language, except assignment, can be defined algebraically and it is shown in Figure 3.1.

An interpretation proof in Isabelle links the algebra with its model. The actual proof is straightforward, as explained in §3.3, it relies on the fact that relations form a Kleene algebra and sets form a boolean algebra. The only axioms to be proved are the ones about the test operator, which can be dispatched by the tactical *auto*.

interpretation *rel-kat: kat* (*op* ∪) (*op O*) *Id* {} (*op* ⊆) (*op* ⊂) *rtrancl* (λ*x*. *Id* ∩ (- *x*))
by *default auto*

Lifting Operators

The implementation of KAT used in this chapter is the singled sorted version, *i.e.*, all the elements in the algebra are of the same *sort*. In the concrete model, the sort of choice are relations $R \subseteq \Sigma \times \Sigma$ between states. We write Σ as the set of all possible states. Nevertheless, as it is usual, tests and assertions need to be modelled as set of states. For the relational model we need then to inject a test/assertion $P \subseteq \Sigma$ into relational subidentities.

$$\lfloor B \rfloor = \{(\sigma, \sigma) \mid \sigma \in B\}$$

Additionally, atomic commands, such as assignments, are usually defined as a state transformer $f : \Sigma \to \Sigma$. The operator $\langle f \rangle$, so-called *graph* of function f, lifts state transformers to relations.

$$\langle f \rangle = \{(\sigma, f(\sigma)) \mid \sigma \in \Sigma\}$$

Following Nipkow [87] and others [102]. The last operator becomes essentially a new command in our **while** language. This command is usually called *basic* in the literature. A new inference rule in Hoare logic is needed to deal with it.

Lemma 4.1. *The following inference rule is derivable in relational* KAT.

$$P \subseteq \{\sigma \mid f(\sigma) \in Q\} \Rightarrow \{\lfloor P \rfloor\} \ \langle f \rangle \ \{\lfloor Q \rfloor\}.$$

The Isabelle proof is simple and automatic. This rule, which cannot be derived from the algebra, is the only one, proved in the model, needed to implement our basic verification tool.

Assignment Statement

As mentioned before, states are defined in Isabelle as a record of program variables. For each variable, Isabelle provides a *retrieve* and an *update* function, which support variables of any Isabelle type. Isabelle's built-in list data type and its built-in list libraries can thus be used, for instance, for reasoning about list-based programs. Of course, due to polymorphism, other Isabelle data types may be used as well, and in combination.

We implement assignment statements as relations of type $\Sigma \times \Sigma$ such as

$$(`x := e) \equiv \langle \lambda\sigma.\ x\_update\ (e\ \sigma)\ \sigma \rangle\,,$$

where $`x$ is a program variable, $x\_update$ is its update function provided by Isabelle's record package, and $e$ an expression of the same type as $`x$. Here and henceforth, all program variables are prefixed by `.

An assignment axiom can be derived as corollary of the inference rule for state transformers in Lemma 4.1, completing hence the implementation of Hoare logic in Isabelle/HOL.

Corollary 4.1. *Hoare's assignment axiom is derivable in relational* KAT*.*

$$P \subseteq Q[e/`x] \Rightarrow\ \{\lfloor P \rfloor\}\ (`x := e)\ \{\lfloor Q \rfloor\},$$

where $Q[e/`x]$ denotes substitution of variable $`x$ by evaluated expression e in Q.

This yields the following result.

Theorem 4.1. *The rules of Hoare logic are derivable in relational* KAT*.*

Annotated Commands

Find a suitable invariant for a while loop is in general an undecidable problem. Nevertheless, it is a subject of intense research; machine learning, reducing the scope of variables, abstract interpretation and other advance techniques are promising approaches to handle invariants.

In this basic verification tool, we do not handle invariants. A while loop needs to be correctly annotated. To enhance verification condition generation, we have verified the following additional inference rule

$$p \leq i \wedge \overline{p}i \leq q \wedge \{ib\}\ x\ \{i\} \Rightarrow \{p\}\ \textbf{while}\ b\ \textbf{inv}\ i\ \textbf{do}\ x\ \textbf{od}\ \{q\}.$$

This result is easily derived by the consequence and the while rule of Hoare logic.

Although it is not strictly necessary, we have added the possibility of annotate the program in any point, giving a specific precondition and/or postcondition to a command, that is, we have verified the rule

$$p \leq p' \wedge \{p'\}\ x\ \{q'\} \wedge q' \leq q \Rightarrow \{p\}\ \{p'\}x\{q'\}\ \{q\}.$$

These are not new commands, but rewriting of previous commands with extra information that can be used by a tactic, such as a verification condition generator. For that reason, the following forgetful equalities hold:

$$\textbf{while}\ b\ \textbf{inv}\ i\ \textbf{do}\ x\ \textbf{od} = \textbf{while}\ b\ \textbf{do}\ x\ \textbf{od},$$
$$\{p'\}x\{q'\} = x.$$

Note the abuse of notation for an annotated program and a Hoare triple. This is common practice in the literature and it will be clear by the context.

Verification Condition Generation

Isabelle 2015 offers the new proof method language Eisbach [80], in which a proof tactic *hoare* can be easily developed using the rules of Hoare logic. The tactic generates verification conditions automatically and tries to blast away the entire control structure, thus generating verification conditions at the data level.

The full implementation of *hoare* in Isabelle/HOL is shown in Figure 4.2. The rules of Hoare logic are grouped under a list of theorems named *hl-rules*. A tactic is created by the keyword **method**, which might include lists of lemmas as arguments declared after **uses**. The syntax used to define a method is essentially the same one when proving a lemma in Isabelle. The operations are similar to the ones for regular expressions, where '|' is choice, ',' is sequential operation, and '+' and '?' are quantifiers indicating *one or more* and *zero or one* respectively. The *hoare* tactic essentially applies each rule of Hoare logic by programming construct, decomposing thereby the program control structure. It chooses the largest possible set of states whenever it encounters a schematic variable.

Through this integration of the data flow level, we have now a simple tool for program verification at our disposition. Relative to the underlying general purpose theories for boolean algebras, Kleene algebras and KAT and for binary relations in Isabelle, the implementation effort required for building these tools was minor. Very little code was needed for implementing the tool and the proofs required for its verification were almost entirely automatic. This makes our tool particularly lightweight; the separation between data and control and the use of algebra at the control flow level has certainly paid off. The remaining section shows the tool at work.

named-theorems *hl-rules*

method *hoare-init* **uses** *simp* =
((*subst simp* | *subst fst-conv* | *subst snd-conv*)+)*?*

method *hoare-step* **uses** *simp hl* =
(*hoare-init simp*: *simp*, (*assumption* | *rule subset-refl* | *rule mono-rules* | *rule hl hl-rules* | *rule allI* | *rule ballI*))

method *hoare-ind* **uses** *simp hl* =
(*hoare-step simp*: *simp hl*: *hl*; (*hoare-ind simp*: *simp hl*: *hl*)*?*)+

method *hoare* **uses** *simp hl* =
(*hoare-init simp*: *simp*; (*hoare-ind simp*: *simp hl*: *hl*)*?*)

Figure 4.2: The *hoare* tactic.

4.3 Euclid's Algorithms

Figure 4.3 shows our first example. It is Euclid's classic algorithm for computing the greatest common divisor of two numbers. Before writing the algorithm, we need to specify the program state as a record; in this example, the state is simply a triplet of natural numbers named x, y and z.

Note that we have used a slightly different notation for assertions, this is essentially syntactic sugar. When interpreted by Isabelle, for instance, the precondition of Euclid's algorithm is

$$\{\!|\ `x = xo\ \wedge\ `y = yo\ |\!\} = \{\lfloor\{\sigma \mid x(\sigma) = xo\ \wedge\ y(\sigma) = yo\}\rfloor\}.$$

The algorithm is annotated with a precondition, a postcondition and a loop invariant. After applying the tactic *hoare*, three verification conditions need to be proven: initialisation of the invariant, its maintenance and the establishment of the postcondition. In this particular example, the conditions are extremely simple; after applying *auto*, the only remaining proof obligation is the following

$$\forall s.\ 0 < ys \Rightarrow gcd\ (x\ s)\ (y\ s)\ =\ gcd\ (y\ s)\ ((x\ s)\ mod\ (y\ s)),$$

which can be discharged by *Sledgehammer* using a desired property about greatest common divisors in Isabelle's library.

The next example, the integer division algorithm, in Figure 4.4, shows how the state can be extended to accommodate more program variables. The record (store) *div_state* is extended with the variables from the previous record *state*

```
record state =
    x :: nat
    y :: nat
    z :: nat

lemma euclids:
  ⊢ {‘x = xo ∧ ‘y = yo}
    while ‘y ≠ 0
    inv gcd ‘x ‘y = gcd xo yo
    do
      ‘z := ‘y;
      ‘y := ‘x mod ‘y;
      ‘x := ‘z
    od
    {‘x = gcd xo yo}
by (hoare, auto) (metis gcd-red-nat)
```

Figure 4.3: Euclid's algorithm.

```
record div-state = state +
  q :: nat
  r :: nat

lemma div:
  ⊢ { ‘x ≥ 0 }
    ‘q := 0; ‘r := ‘x;
    while ‘y ≤ ‘r
    inv ‘x = ‘q * ‘y + ‘r ∧ ‘r ≥ 0
    do
      ‘q := ‘q + 1;
      ‘r := ‘r − ‘y
    od
  { ‘x = ‘q * ‘y + ‘r ∧ ‘r ≥ 0 ∧ ‘r < ‘y }
by hoare auto
```

Figure 4.4: Integer division algorithm.

plus the variables q and r. The verification in this case is completely automatic. After calling the verification condition generator *hoare*, the tactic *auto* proves all remaining proof obligations.

4.4 Arrays and for Loops

Imperative arrays are usually modelled as functional lists in theorem provers [102], because recursive datatypes are easier to reason about in functional languages. Nevertheless, following Apt *et al.* [5], we model arrays as functions **nat** $\to \alpha$ from natural numbers to a polymorphic target type, which is a more natural way of defining arrays. Since **nat** is a recursive datatype, it is possible to reason about arrays by induction on their indexes. This is clearly an advantage, because properties about arrays are usually defined in terms of them.

In many algorithms, a index -1 is used for initialization, to indicate error or simply to establish a desired invariant where the assertion is vacuously true in that index. Unfortunately, **nat** does not contain this value, integers could be used instead of natural numbers, but their representation is slightly more complex than **nat** in Isabelle. Therefore, opposite to the norm, arrays start on index 1 and the index 0 represents an outbound.

The i-th element of an array a is obtained by $a(i)$. An assignment to an element of the array is written as $a(i) := e$, which is syntactic sugar for

$$(a(i) := e) \equiv (a := a(i := e)).$$

In other words, it assigns the updated array $a(i := e)$ to the variable a. The notation $a(i := e)$ in Isabelle denotes an updated function, that is, $a(i := e)(x)$ returns e if $x = i$ and $a(x)$ otherwise.

A **for** loop is usually expressed by using a while loop, thence in order to generate the necessary verification conditions for its Hoare rule, it would be enough to simply unfold its definition and apply the corresponding rule for a **while** loop. Nevertheless, when writing a **for** loop, one usually has a very simple invariant in mind, and doing so, it does not benefit from the extra information that this kind of loop may offer. The following rule is added to the tactic *hoare*:

$$\begin{aligned} & P \subseteq Q[n/m] \cap \{s.\ n\ s \leq m\ s\} \\ \wedge\ & \{Q[m/i] \wedge (i < m)\}\ x\ \{Q[(i+1)/m] \wedge (i+1 \leq m)\} \\ \Rightarrow\ & \{P\}\ \textbf{for}\ i := n\ \textbf{to}\ m\ \textbf{do}\ x\ \textbf{od}\ \{Q\}, \end{aligned}$$

which states that $Q[m/i]$ is the invariant of the underlying **while** loop. The first condition simply initialises the invariant, whereas the second condition maintains

```
record power-state =
  b:: nat
  i :: nat
  n :: nat

lemma power:
  ⊢ {| True |}
    'i := 1;
    'b := 1;
    while 'i < 'n
    inv 'b = a ^ 'i ∧ 'i ≤ 'n
    do
      'b := 'b * a;
      'i := 'i + 1
    od
  {| 'b = a ^ 'n |}
  by hoare auto

lemma power':
  ⊢ {| True |}
    'b := 1;
    for 'i := 1 to 'n do
      'b := 'b * a
    od
    {| 'b = a ^ 'n |}
    by hoare auto
```

Figure 4.5: Power algorithm

the invariant. The establishment of the postcondition is obvious, since in the end of the loop i is equal to m, then $Q[m/i] = Q$.

Figures 4.5 shows how succinct and practical this notation can be. This example simply calculates the exponentiation $a^{`n}$ and stores it in the variable *'b*. Two versions of the program are presented, one written using a while loop and then second one using a for loop. Both generate similar proof obligations. The while version generates three, whereas the for version generates only two conditions. The third one is the trivial establishment of the postcondition. All the conditions are simple enough to be directly discharged by Isabelle's prover *auto*.

Another example using an array is presented in Figure 4.6. This is the classical linear search algorithm for arrays. Once again, we show a version using a

while loop and another one using a for loop. That demonstrates the simplicity when using the latter. The postcondition states that whenever the value m can be found in the array, then it is located in the position $`j$. Otherwise $`j$ may contain any value. The proof obligations are trivial and can be proved by *auto* and Sledgehammer. A better variant of this algorithm would set $`j$ to 0 in the beginning and then have the postcondition

if $(\forall k.\ 1 \leq k \wedge k < `n \longrightarrow a(k) \neq m)$ **then** $(`j = 0)$ **else** $(a(`j) = m)$.

This can be used to indicate if the element m has been found in the array. This variant is also available in our Isabelle files.

4.5 Procedures, Local Variables and Function Calls

This section discusses how additional features of programming languages can be added to our tools with ease. We focus on procedures, local variables and function calls.

We use Isabelle's definition mechanism for creating new HOL objects to add support for procedures and procedures calls to our verification tools. A procedure MAX', for instance, that takes two global variables $`x$ and $`y$, compares them and stores their maximum in variable $`y$, can be defined as

definition *MAX'* $\equiv$
  if $`x \geq `y$ **then**
    $`y := `x$
  fi

Isabelle considers MAX' as a relation on the state space which has been fixed a priori by declaring a record of variables for a program. The procedure MAX' can thus be used by any program on that state space. Simple procedure calls can be written, for instance, as

$$`x := 0;\ `y := 1;\ MAX'.$$

Reasoning about such programs usually requires unfolding procedure definitions. These are created automatically by Isabelle—for instance *MAX'_def* unfolds the definition of MAX'.

Without encapsulation and parameters, however, procedures like MAX' are of dubious value. We have therefore added support for local variables as well.

```
record ls-state =
  i :: nat
  j :: nat
  n :: nat

lemma linear-search:
  ⊢ {| 'n ≥ 1 |}
    'i := 1;
    while 'i < 'n
    inv (∀k. 1 ≤ k ∧ k < 'i ⟶ a(k) ≠ m) ∨ (a('j) = m) ∧ ('i ≤ 'n)
    do
      if a('i) = m then
        'j := 'i
      fi;
      'i := 'i + 1
    od
  {| (∀k. 1 ≤ k ∧ k < 'n ⟶ a(k) ≠ m) ∨ (a('j) = m) |}
  apply (hoare, auto)
  using less-SucE by blast

lemma linear-search':
  ⊢ {| 'n ≥ 1 |}
    for 'i := 1 to 'n do
      if a('i) = m then
        'j := 'i
      fi
    od
  {| (∀k. 1 ≤ k ∧ k < 'n ⟶ a(k) ≠ m) ∨ (a('j) = m) |}
  apply (hoare, auto)
  using less-SucE by blast
```

Figure 4.6: Linear search algorithm

Apt and al. [5] propose a block statement

$$\textbf{local } `x \ := \ t \textbf{ in } R \textbf{ end}$$

to distinguish between local and global variables and provide explicit scope for them. Executing a local block can change the value of a local variable $`x$ within its scope, but not the values of variables outside of its scope—including those named $`x$.

From a mathematical point of view, a block is essentially a function that receives a state and returns a command, which is a relation on the state space. Following [102], we model these functions $C : \Sigma \to \Sigma \times \Sigma$ as *dynamic commands*. They are lifted to commands by applying

$$dyn\ C = \{(\sigma, \sigma') \mid (\sigma, \sigma') \in (C\ \sigma)\}.$$

Deriving the following characteristic property of Hoare triples is then straightforward.

$$\forall \sigma \in P.\ \{P\}\ (C\ \sigma)\ \{Q\} \Rightarrow \{P\}\ (dyn\ C)\ \{Q\}.$$

The semantics of a local block can be programmed in Isabelle as

$$\textbf{local } `x \ := \ `t \textbf{ in } R \textbf{ end} = dyn(\lambda\sigma.\ `x := t; R; `x := `x(\sigma)).$$

The following proof shows how easy it is to establish the characteristic property of local blocks by automatic reasoning, namely that the value of the variable $`x$ is the same before and after executing the block **local** $`x \ := \ t$ **in** R **end**.

lemma $\vdash \{\!|\ `x = u\ |\!\}$ **local** $`x := t$ **in** R **end** $\{\!|\ `x = u\ |\!\}$
by *hoare auto*

Modelling state spaces as records has the advantage of being polymorphic and allowing the integration of arbitrary data types. This, however, forces us to declare all local variables "globally" prior to writing and analysing algorithms. For a general overview of alternative state spaces representations in Isabelle see [103].

The following little verification example shows a program with a local variable that changes the value of a global variable inside a block statement.

lemma $\vdash \{\!|\ `x = n\ |\!\}$
local $`x := `x + 1$ **in**
$`x := 2;$
$`y := `x + 1$
end
$\{\!|\ `x = n \wedge `y = 3\ |\!\}$

```
definition MAX xo yo ≡ begin
  local ‘x := ‘xo in
    ‘y := ‘yo;
    if ‘x ≥ ‘y then
      ‘y := ‘x
    fi
  end
  return ‘y
end
```

Figure 4.7: Maximum function of two natural numbers

```
by (hoare first: hl-split) auto
```

The tactic *hoare* has been annotated with *first: hl_split* to force that the post-condition be split before applying *hoare* to the resulting subgoals. This rule can be proved algebraically.

$$\{p\}\ x\ \{q_1\} \wedge \{p\}\ x\ \{q_2\} \Rightarrow \{p\}\ x\ \{q_1 q_2\}$$

Figure 4.7 shows how the procedure MAX' can be refined to a more convincing procedure MAX that uses local variables and parameters. The HOL object created this time is a function from the two parameters to a pair consisting of a relation in the state space and a variable, which holds the value, which the procedure returns. The next verification example shows that MAX, when called within a Hoare triple, satisfies the characteristic block property.

```
lemma ⊢ {| ‘x = xo ∧ ‘y = yo |}
    ‘z := call (MAX xo yo)
  {| (‘x = xo ∧ ‘y = yo) ∧ ‘z = max xo yo |}
  by (hoare simp: MAX-def first: hl-split) auto
```

The command **call** strips the pair, executes function MAX and sets the value of variable $‘z$. The tactic *hoare* is again annotated with *first: hl_split*. In addition it now expands the definition of MAX by applying *simp: MAX_def*. The variable $‘y$ in the definition of MAX is local to MAX; hence MAX must not be able to change the value of any variable $‘y$ outside of its scope. This is precisely the characteristic property proved in the lemma above.

4.6 Conclusions

We have used Kleene algebra with tests for developing a simple program verification tool based on Hoare logic in Isabelle. The verification tool has a lightweight flexible middle layer for formal methods which can easily be adapted and extended. Programs are analysed directly on their relational semantics, but most relational manipulations are captured algebraically. The approach can therefore be integrated directly into any formal method which uses similar semantics. Moreover, it can be further enhanced and adapted flexibly to other analysis tasks. Using the algebra in combination with Isabelle's integrated automated theorem provers and SMT solvers made the development simple and effective. Finally, case studies showed the tool at work.

A possible extension of the tool is the development of a decision procedure for KAT within Isabelle, since universal Horn formulas of the form $r_1 = 0 \wedge ... \wedge r_n = 0 \Rightarrow s = t$ are decidable in PSPACE via a technique called hypothesis elimination. Note that the inferences rules of Hoare logic fall into this form. However, formally verified decision for KAT are only available in Coq [97]; the development in Isabelle would certainly enhance the performance of the tool presented. A further extension would replace KAT by algebras for total correctness reasoning, such as DRA, and by rely-guarantee style algebras for shared variable concurrency. A verification tool for the latter has already been developed in [7], which includes a semantics of finite transition traces, and has been further extended to infinite traces in Armstrong PhD thesis [6].

Chapter 5

A Refinement Tool

This chapter shows how to easily derive a construction tool from refinement algebras. Kleene algebras are extended to refinement algebras in which Morgan's specification statement operator [83] can be expressed algebraically. The algebra is then linked to the concrete relational model derived in the previous chapter. This yields a construction/refinement tool which is correct by construction. The approach has been published in [10] and [11].

5.1 The Specification Statement

KAT can be extended to a Morgan-style refinement calculus by adding one single axiom. We keep the partial correctness setting, which suffices for practical program construction tasks. Extending it to a total correctness setting with termination variants seems straightforward, and existing Isabelle infrastructure for proving program termination could be used.

Our approach follows Morgan's classic book on *Programming from Specifications* [83]. We think of specifications as programs that need not be executable. Morgan starts from the largest program which relates a given precondition p to a given postcondition q—the *specification statement*—and uses refinement laws to transform it incrementally and compositionally into an executable program which is correct by construction. In KAT, the axiomatisation of Morgan's specification statement is very simple.

A *refinement Kleene algebra with tests* (rKAT) is a KAT expanded by an operation $[\_,\_] : B \times B \to K$ which satisfies

$$\{p\}\ x\ \{q\} \Leftrightarrow x \leq [p,q]. \tag{5.1}$$

It is easy to show that (5.1) implies the characteristic properties

$$\{p\}\ [p,q]\ \{q\}, \qquad \{p\}\ x\ \{q\} \Rightarrow x \leq [p,q]$$

of the specification statement. First of all, program $[p, q]$ relates precondition p with postcondition q whenever it terminates. Second, it is the largest program with that property.

Morgan's basic refinement calculus provides one refinement law per program construct. Once more we ignore assignments at this stage and focus on the control flow. Deriving Morgan's laws in rKAT is strikingly easy. We use the standard refinement order $\sqsubseteq$, which is the converse of $\leq$.

Proposition 5.1. *The following refinement laws are derivable in* rKAT*:*

$$\begin{aligned}
p \leq q &\Rightarrow [p, q] \sqsubseteq \mathbf{skip}, && \text{(skip law)}\\
p' \leq p \wedge q \leq q' &\Rightarrow [p, q] \sqsubseteq [p', q'], && \text{(consequence law)}\\
[0, 1] &\sqsubseteq x, && \text{(abort law)}\\
x &\sqsubseteq [1, 0], && \text{(magic law)}\\
[p, q] &\sqsubseteq [p, r]; [r, q], && \text{(sequential composition law)}\\
[p, q] &\sqsubseteq \mathbf{if}\ b\ \mathbf{then}\ [bp, q]\ \mathbf{else}\ [\bar{b}p, q]\ \mathbf{fi}, && \text{(conditional law)}\\
[p, \bar{b}p] &\sqsubseteq \mathbf{while}\ b\ \mathbf{do}\ [bp, p]\ \mathbf{od}. && \text{(while law)}
\end{aligned}$$

Proof. The laws are usually derived in set theory. This proof illustrates the simplicity of using rKAT instead.

- With $[p, q] \sqsubseteq x$ being equivalent to $px \leq xq$ the proofs of the skip, abort and magic laws become trivial.

- For the consequence law, remember that $\{p\}\ [p, q]\ \{q\}$ is one of the characteristic properties of the specification statement. Hence assuming $p \leq p'$ and $q' \leq q$, we calculate

$$p[p', q'] \leq p'[p', q'] \leq [p', q']q' \leq [p', q']q \Rightarrow [p, q] \sqsubseteq [p', q'].$$

- For the sequential composition law we use the sequential composition rule of Hoare logic:

$$\{p\}\ [p, r]\ \{r\} \wedge \{r\}\ [r, q]\ \{q\} \Rightarrow \{p\}\ [p, r]; [r, q]\ \{q\} \Leftrightarrow [p, q] \sqsubseteq [p, r]; [r, q].$$

- For the conditional law we use the conditional rule of Hoare logic:

$$\begin{aligned}
\{pb\}\ [pb, q]\ \{q\} \wedge \{p\bar{b}\}\ [p\bar{b}, q]\ \{q\} &\Rightarrow \{p\}\ \mathbf{if}\ b\ \mathbf{then}\ [bp, q]\ \mathbf{else}\ [\bar{b}p, q]\ \mathbf{fi}\ \{q\}\\
&\Leftrightarrow [p, q] \sqsubseteq \mathbf{if}\ b\ \mathbf{then}\ [bp, q]\ \mathbf{else}\ [\bar{b}p, q]\ \mathbf{fi}.
\end{aligned}$$

- For the while law we use the while rule of Hoare logic:

$$\{pb\}\ [pb, p]\ \{p\} \Rightarrow \{p\}\ \textbf{while}\ b\ \textbf{do}\ [bp, p]\ \textbf{od}\ \{\bar{b}p\}$$
$$\Leftrightarrow [p, \bar{b}p] \sqsubseteq \textbf{while}\ b\ \textbf{do}\ [bp, p]\ \textbf{od}. \qquad \square$$

The proofs of these refinement laws are automatic in Isabelle, as the following examples show.

lemma *strengthen-post*: $[\![test\ p;\ test\ q;\ test\ q']\!] \Longrightarrow q' \leq q \Longrightarrow [p\ q] \sqsubseteq [p\ q']$
by (*meson consequence-rule spec-char1 spec-char2 ref-order-def*)

lemma *weaken-pre*: $[\![test\ p;\ test\ p';\ test\ q]\!] \Longrightarrow p \leq p' \Longrightarrow [p\ q] \sqsubseteq [p'\ q]$
by (*meson consequence-rule order-refl spec-char1 spec-char2 ref-order-def*)

lemma *while-ref*: $[\![test\ b;\ test\ p]\!] \Longrightarrow [p\ p{\cdot}!b] \sqsubseteq (b{\cdot}[p{\cdot}b\ p])^{\star}{\cdot}!b$
by (*metis ht-def-var spec-char1 spec-def test-mult-closed while-rule ref-order-def*)

The specification statements $[0, 1]$ and $[1, 0]$ represent extreme programs. The first one is usually called the **abort** statement; it is a pathological program that can be refined by any other. The second one, also known as the **magic** statement, is an impossible program that can always be refined to any other one; it magically satisfies any statement.

In the algebraic context, the consequence law is essentially an isotonicity property—the same is true for the corresponding Hoare rule. It indicates that if we can weaken the precondition p to p' and/or strengthen the postcondition q to q', then we can refine the specification statement $[p, q]$ to $[p', q']$. In our Isabelle implementation we have therefore split this law into two, as the above proofs show.

The other refinement laws operate as expected. They allow us to introduce control structure—a sequential composition, a conditional, a while-loop—into a specification statement.

5.2 Data Flow Integration

Chapter 4 has set up a concrete relational semantics of imperative programs by supplying notions of program store and store update and used them to derive Hoare's assignment axiom. In this section, we show how to use the same semantics to derive three refinement laws for assignment. This gives us a Morgan-style refinement calculus based on rKAT.

First, we verify soundness of rKAT with respect to the relational model.

Proposition 5.2. *Let A be a set and define, for all $P, Q \subseteq Id_A$, the specification statement as*

$$[P, Q] = \bigcup \{R \subseteq A \times A \mid \{P\}\ R\ \{Q\}\}.$$

Then $(2^{A\times A}, B, \cup, \circ, [\_, \_], ^*, ^{-}, \emptyset, id)$ forms a rKAT.

We call this algebra the *full relational* rKAT over A. Once more, every subalgebra is a relational rKAT. Due to its higher-order nature, this fact is proved in Isabelle by calling the internal theorem provers *safe* and *force*, supplying appropriates lemmas as parameters. The proof, which is relative to soundness of KAT, is nevertheless very simple.

interpretation *rel-rkat: rkat* (*op* $\cup$) (*op* O) *Id* {} (*op* $\subseteq$) (*op* $\subset$) *rtrancl test-not spec*
  apply *(default, safe)*
  apply *(force simp: spec-def intro: Sup-upper)*
  apply *(force intro: specD)*
done

Next, we use our definition of assignments to derive three of Morgan's refinement laws for that command.

Lemma 5.1. *The following refinement laws are derivable in relational* rKAT.

$$P \subseteq Q[e/`x] \Rightarrow [\lfloor P \rfloor, \lfloor Q \rfloor] \sqsubseteq (`x := e), \tag{5.2}$$
$$Q' \subseteq Q[e/`x] \Rightarrow [\lfloor P \rfloor, \lfloor Q \rfloor] \sqsubseteq [\lfloor P \rfloor, \lfloor Q' \rfloor];\ (`x := e), \tag{5.3}$$
$$P' \subseteq P[e/`x] \Rightarrow [\lfloor P \rfloor, \lfloor Q \rfloor] \sqsubseteq (`x := e);\ [\lfloor P' \rfloor, \lfloor Q \rfloor]. \tag{5.4}$$

Law (5.3) and (5.4) are called the *following* and *leading* refinement law for assignments [83]. They are particularly useful for program construction, where they allow one to insert an assignment statement before or after a block of code. As in the case of verification, we have programmed a tactic called *morgan* using Eisbach, which automatically tries to apply the rules of the basic refinement calculus.

Theorem 5.1. *The laws of Morgan's basic refinement calculus are derivable in relational* rKAT.

Although a Morgan style refinement calculus has been derived, the traditional one has the concept of frame, which allows users to declare explicitly which program variables can be changed. Our simple refinement calculus does not implement this feature. We are using Isabelle's built-in records to model program stores (cf. §4.2), and these are not dynamic, that is, they cannot change structure according to a higher-order function.

Through this integration of the data flow level, we have now two tools for program construction and program verification at our disposition. Relative to the underlying general purpose theories for boolean algebras, Kleene algebras and KAT and for binary relations in Isabelle, the implementation effort required for building these tools was minor. The program construction tools could be built literally in one afternoon from the verification tool. Very little code was needed for implementing these tools and the proofs required for their verification were almost entirely automatic. This makes our tools particularly lightweight; the separation between data and control and the use of algebra at the control flow level has certainly paid off. The following two sections of this chapter show the tools at work.

5.3 Sum of Even Fibonacci Numbers

We now present our first construction example. We verify a program that computes the sum of even Fibonacci numbers below a given threshold[1]. Its input is the threshold $m \in \mathbb{N}$; its return value is stored in a variable $`sum$. Because of typing there is no specific precondition, *i.e.*, $m \geq 0$. The postcondition is defined by the following predicate

$$\mathit{is\text{-}sum\text{-}efib}\ `\mathit{sum}\ m = \exists n.\ `\mathit{sum} = \mathit{sum\text{-}efib}\ m\ n \wedge \mathit{fib}\ n > m,$$

which states that it exists a n such that m is smaller than the n-th Fibonacci number and $`sum$ is equal to the sum of all previous even numbers. The predicate is specified in terms of the standard functional program fib, which can be programmed in Isabelle, and the recursive function

$$\mathit{sum\text{-}efib}\ m\ 0 = 0,$$
$$\mathit{sum\text{-}efib}\ m\ (n+1) = \begin{cases} \mathit{sum\text{-}efib}\ m\ n & \text{for } \mathit{fib}\ n \text{ odd or } \mathit{fib}\ n > m, \\ \mathit{sum\text{-}efib}\ m\ n + \mathit{fib}\ n & \text{otherwise.} \end{cases}$$

The specification statement for our program is therefore, in Isabelle syntax,

$$[\![\mathit{True},\ \mathit{is\text{-}sum\text{-}efib}\ `\mathit{sum}\ m]\!].$$

We use the notation $[\![\_\_\ ,\ \_\_]\!]$ in Isabelle for the specification statement, because $[\_\_\ ,\ \_\_]$ is already reserved for functional lists.

[1] The algorithm is taken from http://toccata.lri.fr/gallery/euler002.en.html. Fibonacci numbers start as $1, 2, 3, 5, 8, ...$, which is perhaps nonstandard.

$[\![\ \mathit{True},\ \mathit{is\text{-}sum\text{-}efib}\ `\mathit{sum}\ m]\!]$

$\sqsubseteq$ (1)

$`x := 2;$

$[\![\ `x{=}\mathit{efib}\ 0 \wedge\ `x{=}\mathit{fib}\ 1,\ \mathit{is\text{-}sum\text{-}efib}\ `\mathit{sum}\ m]\!]$

by *morgan*

$\sqsubseteq$ (2)

$`x := 2;\ `y := 8;$

$[\![\ `x{=}\mathit{efib}\ 0\ \wedge\ `x{=}\mathit{fib}\ 1 \wedge\ `y{=}\mathit{efib}\ 1 \wedge\ `y{=}\mathit{fib}\ 4,\ \mathit{is\text{-}sum\text{-}efib}\ `\mathit{sum}\ m]\!]$

by *morgan* (*smt even-fib.simps*(*2*) *even-fib-correct*)

$\sqsubseteq$ (3)

$`x := 2;\ `y := 8;\ `\mathit{sum} := 0;$

$[\![\ `x{=}\mathit{efib}\ 0\ \wedge\ `x{=}\mathit{fib}\ 1 \wedge\ `y{=}\mathit{efib}\ 1 \wedge\ `y{=}\mathit{fib}\ 4 \wedge\ `\mathit{sum}{=}\mathit{sum\text{-}efib}\ m\ 1,$
$\mathit{is\text{-}sum\text{-}efib}\ `\mathit{sum}\ m]\!]$

by *morgan*

$\sqsubseteq$ (4)

$`x := 2;\ `y := 8;\ `\mathit{sum} := 0;\ `n := 0;\ `k := 1;$

$[\![\ `x{=}\mathit{efib}\ `n\ \wedge\ `x{=}\mathit{fib}\ `k \wedge\ `y{=}\mathit{efib}\ (`n{+}1) \wedge\ `y{=}\mathit{fib}\ (`k{+}3)$
$\wedge\ `n \geq 0 \wedge\ `k \geq 1 \wedge\ `\mathit{sum}{=}\mathit{sum\text{-}efib}\ m\ `k,\ \mathit{is\text{-}sum\text{-}efib}\ `\mathit{sum}\ m]\!]$

by *morgan*

$\sqsubseteq$ (5)

$`x := 2;\ `y := 8;\ `\mathit{sum} := 0;\ `n := 0;\ `k := 1;$

while $`x \leq m$ **do**

$[\![\ `x{=}\mathit{efib}\ `n\ \wedge\ `x{=}\mathit{fib}\ `k \wedge\ `y{=}\mathit{efib}\ (`n{+}1) \wedge\ `y{=}\mathit{fib}\ (`k{+}3)$
$\wedge\ `n \geq 0 \wedge\ `k \geq 1 \wedge\ `\mathit{sum}{=}\mathit{sum\text{-}efib}\ m\ `k \wedge\ `x \leq m,$
$`x{=}\mathit{efib}\ `n\ \wedge\ `x{=}\mathit{fib}\ `k \wedge\ `y{=}\mathit{efib}\ (`n{+}1) \wedge\ `y{=}\mathit{fib}\ (`k{+}3)$
$\wedge\ `n \geq 0 \wedge\ `k \geq 1 \wedge\ `\mathit{sum}{=}\mathit{sum\text{-}efib}\ m\ `k\ ]\!]$

od

by *morgan* (*smt is-sum-efib-def*)

$\sqsubseteq$ (6)

$`x := 2;\ `y := 8;\ `\mathit{sum} := 0;\ `n := 0;\ `k := 1;$

while $`x \leq m$ **do**

$[\![\ `x{=}\mathit{efib}\ `n\ \wedge\ `x{=}\mathit{fib}\ `k \wedge\ `y{=}\mathit{efib}\ (`n{+}1) \wedge\ `y{=}\mathit{fib}\ (`k{+}3)$
$\wedge\ `n \geq 0 \wedge\ `k \geq 1 \wedge\ `\mathit{sum}{=}\mathit{sum\text{-}efib}\ m\ `k \wedge\ `x \leq m,$
$`x{=}\mathit{efib}\ `n\ \wedge\ `x{=}\mathit{fib}\ (`k{+}3) \wedge\ `y{=}\mathit{efib}\ (`n{+}1)$
$\wedge\ `y{=}\mathit{fib}\ (6{+}`k) \wedge\ `n \geq 0 \wedge\ `\mathit{sum}{=}\mathit{sum\text{-}efib}\ m\ (`k{+}3)\ ]\!];$
$`k := `k{+}3$

od

by *morgan*

$\sqsubseteq$

$\ldots$

$$
\begin{array}{l}
\quad \ldots \\
\quad \sqsubseteq \\
\text{`}x := 2;\ \text{`}y := 8;\ \text{`}sum := 0;\ \text{`}n := 0;\ \text{`}k := 1; \\
\textbf{while } \text{`}x \leq m \textbf{ do} \\
\quad [\![\ \text{`}x{=}efib\ \text{`}n \ \wedge\ \text{`}x{=}fib\ \text{`}k \wedge \text{`}y{=}efib\ (\text{`}n{+}1) \wedge \text{`}y{=}fib\ (\text{`}k{+}3) \\
\qquad \wedge\ \text{`}n \geq 0 \wedge \text{`}k \geq 1 \wedge \text{`}sum{=}sum\text{-}efib\ m\ \text{`}k \wedge \text{`}x \leq m, \\
\qquad \text{`}y{=}efib\ (\text{`}n{+}1) \ \wedge\ \text{`}y{=}fib\ (\text{`}k{+}3) \wedge (4{*}\text{`}y{+}\text{`}x){=}efib\ (\text{`}n{+}2) \\
\qquad \wedge\ (4{*}\text{`}y{+}\text{`}x){=}fib\ (6{+}\text{`}k) \wedge (\text{`}sum{+}\text{`}x){=}sum\text{-}efib\ m\ (\text{`}k{+}3)\ ]\!]; \\
\quad \text{`}tmp := \text{`}x;\ \text{`}x := \text{`}y; \\
\quad \text{`}y := 4{*}\text{`}y + \text{`}tmp; \\
\quad \text{`}sum := \text{`}sum + \text{`}tmp; \\
\quad \text{`}n := \text{`}n{+}1;\ \text{`}k := \text{`}k{+}3 \\
\textbf{od} \\
\quad \textbf{by } morgan
\end{array}
\tag{7}
$$

Figure 5.1: Construction of the sum of even Fibonacci numbers program

In order to keep track of all even Fibonacci numbers up to m, we use the function

$$
\begin{aligned}
efib\ 0 &= 2, \\
efib\ 1 &= 8, \\
efib\ (n+2) &= 4 * efib\ (n+1) + efib\ n.
\end{aligned}
$$

We have verified by induction that all numbers computed by *efib* are even and that $efib\ n = fib\ (3n+1)$ holds for all $n \geq 0$. The following classical fact about Fibonacci numbers then implies that *efib* computes indeed all even terms

$$(fib\ n)\ mod\ 2 = 0 \Leftrightarrow n\ mod\ 3 = 1.$$

After this groundwork, which is an indispensable part of program construction and verification, we can start with the program construction itself, which it is shown in Figure 5.1. Since Fibonacci numbers are defined recursively from their two predecessors, we add the variables ‘x and ‘y to keep track of them. In (1) we initialise ‘x to 2—the first even Fibonacci number—applying the leading refinement law for assignments derived in Proposition 5.1.

Our *morgan* tactic automatically applies the assignment law. In (2) we then initialise ‘y to 8—the second even Fibonacci number. The *morgan* tactic now dictates the proof obligation $fib\ 4 = 8$, which is discharged by an integrated SMT solver. In (3) we initialise ‘*sum* to 0 and state that ‘$sum = sum\text{-}efib\ m\ 1$ by definition.

The main idea behind this program is to add the next even Fibonacci number to ‘*sum* as long as it is below m, while storing the previous numbers in ‘x and ‘y.

```
‘x := 2; ‘y := 8; ‘sum := 0;
while ‘x ≤ m do
  ‘tmp := ‘x; ‘x := ‘y;
  ‘y := 4∗‘y + ‘tmp;
  ‘sum := ‘sum + ‘tmp;
od
```

Figure 5.2: Sum of even Fibonacci numbers

In the actual state of development, we also want to keep track of the indices of these numbers in the *fib* and *efib* series. Hence in (4) we add the variables ‘n and ‘k. The facts proved about Fibonacci numbers imply that the numbers stored in ‘x and ‘y have distance 3 in the series of Fibonacci numbers. The precondition now stores the tentative loop invariant; so we can introduce the while-loop in (5). This requires that the precondition implies the postcondition, which follows from the definition of the predicate *is-sum-efib* by setting n to ‘k.

Deriving the body of the loop in (6) and (7) is quite straightforward; we just need to specify the variable updates. In (6), ‘k is updated; then, in (7), ‘*sum* is updated to ‘$sum + ‘x$, ‘x to ‘y, ‘y to the next even Fibonacci number, and so on. This can be achieved by applying the following or leading refinement law. This time we choose to apply the following law from Proposition 5.1, which forces a substitution in the postcondition. In (7) we also add a new variable ‘*tmp* to save the value of ‘x and proceed as before until all variables have been updated.

It now remains to eliminate the surviving specification statement. Refining it to **skip** requires that its precondition implies its postcondition. Accordingly, our *morgan* tactic generates the proof obligations

$$\begin{gathered} \mathit{fib}\ (k+6) = 4 * \mathit{fib}\ (k+3) + \mathit{fib}\ k, \\ \mathit{even}\ (\mathit{fib}\ k) \wedge \mathit{fib}\ k \leq m \Rightarrow \mathit{sum\text{-}efib}\ m\ (k+3) = \mathit{sum\text{-}efib}\ m\ k + \mathit{fib}\ k, \end{gathered}$$

which are discharged by automatic theorem proving, using induction on Fibonacci numbers. This finally gives us the program in Figure 5.2, which is partially correct by construction. For total correctness it remains to prove termination, for which Isabelle provides complementary support as well [107].

We conclude this development with three remarks. First, with good libraries for Fibonacci numbers in place, the algebra and particular Isabelle technology used for constructing this algorithm remain hidden behind an interface. Developers interact with Isabelle mainly by writing mathematical expressions and pseudocode in a specification language similar to Morgan's, and by calling the *refinement* tactic and Isabelle's theorem provers. Alternatively, they could invoke individual refinement rules. This is nicely supported by Isabelle's structured proof specification language Isar.

```
lemma ⊢ {⦇ True ⦈}
 'x := 2; 'y := 8; 'sum := 0; 'n := 0; 'k := 1;
 while 'x ≤ m
 inv
   ('k ≥ 1) ∧ ('x=efib 'n) ∧ ('x=fib 'k)
   ∧ ('y=efib ('n+1)) ∧ ('y=fib ('k+3))
   ∧ ('sum=sum-efib m 'k)
 do
   'tmp := 'x; 'x := 'y;
   'y := 4*'y + 'tmp;
   'sum := 'sum + 'tmp;
   'n := 'n+1; 'k := 'k+3
 od
{⦇ is-sum-efib 'sum m ⦈}
apply (hoare, auto)
apply (smt is-sum-efib-def)
apply (metis fib-6-n)
apply (metis efib-mod-2-eq-0 sum-efib-fib)
by (smt efib.simps(2) efib-correct)
```

Figure 5.3: Verification of the sum of even Fibonacci numbers program

Second, proof automation was very high. Most refinement steps were verified by *refinement* alone, the others by automated theorem proving. Isabelle thus supported a seamless refinement process at the level of textbook proofs.

Finally, it should be pointed out that we have used auxiliary variables such as $'n$ and $'k$ to prove correctness, which are not displayed in the final program, but have not been formally eliminated.

The Fibonacci algorithm can be verified with Hoare logic as well, as shown in Figure 5.3. Our *hoare* tactic generates the standard proof obligations, which can be inspected when executing our Isabelle theories, and *auto* discharges the trivial ones. The survivors are then proved by Isabelle's SMT solvers and external theorem provers, using the built-in theorem prover *metis* to verify external outputs. In this case, user interaction is restricted to calling tactics and theorem provers. Beyond that the verification is fully automatic.

5.4 Insertion Sort

Our next example stems from Morgan's book: the construction and verification of insertion sort. It shows that our tool can handle lists and nested loops.

We model an imperative list A by using Isabelle's functional lists, and therefore benefit from its excellent libraries developed for this data type. This includes the operation $A\ !\ i$ for retrieving the i-th element of A, the function *take* n A which extracts the first n elements of A, the function *list-update* A i e which updates the i-th value of A to e, and a *sorted* predicate. Using this, list assignments are defined merely as syntactic sugar:

$$`A\ !\ i := e \Leftrightarrow `A := \mathit{list\text{-}update}\ `A\ i\ e.$$

Our insertion sort algorithm takes a list A_0 of polymorphic data that can be linearly ordered. It returns a variable $`A$ which holds the sorted list; that is, the values in $A_0$ have been permuted so that $`A\ !\ i \leq `A\ !\ j$ whenever $i \leq j$. We write $`A \sim_\pi A_0$ if $`A$ stores a permutation of the values of A_0. We also require that A_0 has positive length. This suggests the specification statement

$$[\![\ |A_0| > 0 \wedge `A{=}A_0,\ \mathit{sorted}\ `A \wedge `A \sim_\pi A_0\ ]\!].$$

The idea behind insertion sort is well known and need not be repeated. To express that we successively sort larger prefixes, we introduce a variable $`i$ such that $1 \leq `i \leq |`A|$. For $`i = 1$, we have *sorted* (*take* $`i$ $`A$).

Our refinement steps are similar to Morgan's. We show only the most important ones in Figure 5.4. In (1) we initialise $`i$ to 1 and introduce a while-loop.

The resulting proof obligation is discharged by the *morgan* tactic. In the body of the loop we now wish to take the $`i$-th element of the list and insert it in a position $`j$ such that $`j \leq `i$ and *sorted* (*take* $(`i + 1)$ $`A$). To express this succinctly we define the predicate *sorted-but* A k, which states that A is sorted after removing its k-th element. We then rewrite the specification statement in (2). The *morgan* tactic generates four proof obligations which are discharged automatically.

Next we wish to set $`j$ to $`i$ and iteratively swap $`A\ !\ `j$ to $`A\ !\ (`j - 1)$ until $`A\ !\ (`j - 1) \leq `A\ !\ `j$ or $`j = 0$. This requires introducing a new while-loop in (3) which is justified by calling *morgan*.

Finally, in (4) we need to prove that the remaining specification statement is refined by swapping $`A\ !\ `j$ to $`A\ !\ (`j - 1)$. The *morgan* tactic generates six proof obligations. Discharging them automatically requires proving some general properties of sorted list and permutations absent in Isabelle's library, e.g., that swapping list elements yields a permutation. The construction of the insertion sort algorithm is then complete. The result is shown in Figure 5.5.

$[\![\ |A_0| > 0 \wedge\ `A{=}A_0,\ \mathit{sorted}\ `A \wedge\ `A \sim_\pi A_0]\!] \sqsubseteq$ (1)
$`i := 1;$
while $`i < |`A|$ **do**
  $[\![\ \mathit{sorted}\ (\mathit{take}\ `i\ `A) \wedge\ `i < |`A| \wedge\ `A \sim_\pi A_0,$
   $\mathit{sorted}\ (\mathit{take}\ (`i{+}1)\ `A) \wedge (`i{+}1) \leq |`A| \wedge\ `A \sim_\pi A_0]\!];$
  $`i := `i{+}1$
od
$\sqsubseteq$ (2)
$`i := 1;$
while $`i < |`A|$ **do**
  $[\![\ \mathit{sorted\text{-}but}\ (\mathit{take}\ (`i{+}1)\ `A)\ `i \wedge\ `i < |`A| \wedge\ `A \sim_\pi A_0,$
    $`j \leq `i \wedge \mathit{sorted\text{-}but}\ (\mathit{take}\ (`i{+}1)\ `A)\ `j \wedge (`j{\neq}`i \longrightarrow\ `A\ !\ `j \leq `A\ !\ (`j{+}1))$
    $\wedge\ (`i{+}1) \leq |`A|\ \ \wedge (`j{=}0 \vee\ `A\ !\ (`j{-}1) \leq `A\ !\ `j) \wedge\ `A \sim_\pi A_0]\!];$
  $`i := `i{+}1$
od
$\sqsubseteq$ (3)
$`i := 1;$
while $`i < |`A|$ **do**
  $`i := `j;$
  while $`j{\neq}0 \wedge\ `A\ !\ `j < `A\ !\ (`j{-}1)$ **do**
    $[\![\ `j \leq `i \wedge \mathit{sorted\text{-}but}\ (\mathit{take}\ (`i{+}1)\ `A)\ `j \wedge (`j{\neq}`i \longrightarrow\ `A\ !\ `j \leq `A\ !\ (`j{+}1))$
      $\wedge\ (`i{+}1) \leq |`A| \wedge\ `j{\neq}0 \wedge\ `A\ !\ `j < `A\ !\ (`j{-}1) \wedge\ `A \sim_\pi A_0,$
      $`j{-}1 \leq `i \wedge \mathit{sorted\text{-}but}\ (\mathit{take}\ (`i{+}1)\ `A)\ (`j{-}1) \wedge (`i{+}1) \leq |`A|$
      $\wedge\ (`j{-}1{\neq}`i \longrightarrow\ `A\ !\ (`j{-}1) \leq `A\ !\ `j) \wedge\ `j{\neq}0\ \ \wedge\ `A \sim_\pi A_0]\!];$
    $`j := `j{-}1$
  od;
  $`i := `i{+}1$
od
  ...
$[\![\ \ldots \mathit{previous\ specification\ statement} \ldots]\!]$
$\sqsubseteq$ (4)
$`k := `A\ !\ `j;$
$`A\ !\ `j := `A\ !\ (`j{-}1);$
$`A\ !\ (`j{-}1) := `k$

Figure 5.4: Construction of insertion sort algorithm (excerpts)

```
‘i := 1;
while ‘i < |‘A| do
  ‘i := ‘j;
  while ‘j≠0 ∧ ‘A ! ‘j < ‘A ! (‘j−1) do
    ‘k := ‘A ! ‘j;
    ‘A ! ‘j := ‘A ! (‘j−1);
    ‘A ! (‘j−1) := ‘k;
    ‘j := ‘j−1
  od;
  ‘i := ‘i+1
od
```

Figure 5.5: Insertion sort algorithm

Again, it is partially correct by construction; its termination can be proved by other means. Apart from adding some general-purpose lemmas about permutations and sorted lists to Isabelle's libraries, the development was fully automatic.

Decorating the algorithm with the pre and postcondition from the above specification statement, one can also verify this algorithm with Hoare logic. After calling the *hoare* and *auto* tactics we are left with seven proof obligations, the proof of which, mainly done with sledgehammer, is shown in Figure 5.6. Only the *unfold* step does not directly call a theorem prover. It unfolds two definitions and calls Isabelle's *auto* tool.

As already mentioned, our Isabelle developments use functional lists instead of imperative one, to which an imperative style syntactic sugar has been added. In an extension of our refinement tool it would be desirable to link these two data types formally by data refinement techniques. This is left as future work.

5.5 Conclusions

This chapter has derived a program construction tool in Isabelle/HOL, where adding one single axiom to Kleene algebra with tests yielded a basic Morgan-style refinement calculus. Two extended case studies showed the tool at work.

The examples suggest that algorithms of moderate complexity can be constructed step by step at the level of textbook granularity, with each refinement step being justified by the *morgan* tactic and having a high level of automation. Other examples confirm this impression and can be found in the online repository[1]. It should, however, be stressed that a certain amount of groundwork was needed to achieve this level of automation. A small library of generic facts about Fibonacci numbers and permutations were created in addition to those present in

[1] https://github.com/victorgomes/veritas

```
lemma ⊢ {| |A₀| > 0 ∧ `A=A₀ |}
 `i := 1;
 while `i < |`A| inv sorted (take `i `A) ∧ `A ∼π A₀ do
   `i := `j;
   while `j≠0 ∧ `A ! `j < `A ! (`j−1)
   inv
    (sorted_but (take (`i+1) `A) `j) ∧ (`i < length `A) ∧ (`j ≤ `i)
    ∧ (`j≠`i → `A ! `j ≤ `A ! (`j+1)) ∧ (`A ∼π A₀)
   do
     `k := `A ! `j;
     `A ! `j := `A ! (`j−1);
     `A ! (`j−1) := `k;
     `j := `j−1
   od;
   `i := `i+1
 od
{| sorted `A ∧ `A ∼π A₀ |}
apply (hoare, auto)
apply (metis One-nat-def take-sorted-butE-0)
apply (metis take-sorted-butE-n One-nat-def less-eq-Suc-le not-less-eq-eq)
apply (metis One-nat-def Suc-eq-plus1 le-less-linear less-Suc-eq-le take-sorted-butE)
apply (unfold sorted-equals-nth-mono sorted-but-def, auto)
apply (smt nth-list-update)
apply (metis (hide-lams, no-types) One-nat-def perm.trans perm-swap-array)
apply (smt nth-list-update)
by (smt perm.trans perm-swap-list)
```

Figure 5.6: Verification of insertion sort algorithm (proof steps)

Isabelle's libraries. This is a standard procedure and it is certainly typical when verifying larger programs.

It is also worth mentioning that the tools derived in this chapter and in the previous one are still at the level of prototypes. They are certainly stable enough for educational purposes, but not intended to compare with optimised industrial verification tools [49, 104]. There is certainly much scope for improvement (*cf.* §12). Nevertheless, they have clear advantages, such as the flexible algebraic layer, support to program construction and the fact to be correct by construction.

Chapter 6

Recursive Programs

This chapter extends the tool to support recursive programs. It uses a more expressive algebraic structure, called *quantales*, where recursivity is modelled by fixpoints. It shows that quantales subsume Kleene algebras and refinement Kleene algebras for program verification and refinement. This chapter is an extract of the work published on [11].

6.1 Verification of Recursive Programs

A limitation of modelling programs algebraically with KAT is that this formalism is not expressive enough for full recursion. It is, for instance, not possible to derive a Hoare rule for recursive programs in this setting. In this section, we present *quantales*, which subsume Kleene algebras, and which provide a setting in which classical fixpoint theory can be developed. This allows us to derive a Hoare-like inference rule for recursive programs. We restrict our attention to single recursive procedures, theoretically, mutual recursion could be solved as well with polyvariadic fixpoint combinators using an idea similar to the one presented here.

A *quantale* (or *standard Kleene algebra*) is a structure $(Q, \leq, \cdot, 1)$ such that $(Q, \leq)$ is a complete lattice, $(Q, \cdot, 1)$ is a monoid and the distributivity axioms

$$(\sum_{i \in I} x_i) \cdot y = \sum_{i \in I} x_i \cdot y,$$
$$x \cdot (\sum_{i \in I} y_i) = \sum_{i \in I} (x \cdot y_i)$$

hold, where $\sum X$ denotes the supremum of a set $X \subseteq Q$. Similarly, we write $\bigsqcap X$ for the infimum of X, and 0 for the least and $\top$ for the greatest element of the lattice. We write $\sqcap$ for the binary meet in a quantale. It is straightforward

to show that every quantale is a dioid with $x + y = \sum\{x, y\}$. The monotonicity laws

$$x \leq y \Rightarrow z \cdot x \leq z \cdot y,$$
$$x \leq y \Rightarrow x \cdot z \leq y \cdot z$$

follow from distributivity. The two annihilation laws $x \cdot 0 = 0 = 0 \cdot x$ follow from $\sum_{i \in \emptyset} x_i = \sum \emptyset = 0$. A quantale is *commutative* and *partial* if the underlying monoid is. It is *distributive* if

$$x \sqcap (\sum_{i \in I} y_i) = \sum_{i \in I} (x \sqcap y_i),$$
$$x + (\bigsqcap_{i \in I} y_i) = \bigsqcap_{i \in I} (x + y_i).$$

A *boolean quantale* is a complemented distributive quantale. The boolean quantale B of the booleans, where multiplication coincides with meet, is an important example.

Proposition 6.1. *Every quantale is a Kleene algebra.*

Proof. In quantales, the Kleene star can be defined explicitly as a sum of powers up to the first infinite ordinal:

$$x^* = \sum_{i \in \mathbb{N}} x^i,$$

where $x^0 = 1$ and $x^{i+1} = x \cdot x^i$. The proofs of the unfold axioms $1 + xx^* = x^*$ and $1 + x^*x = x^*$ depend on the left distributivity law for quantales, which, in the context of fixpoint theory, is a continuity law. The verification of the induction axioms $z + xy \leq y \Rightarrow x^*z \leq y$ and $z + yx \leq y \Rightarrow zx^* \leq y$ depends on continuity as well; they require

$$z(\sum_{i \in \mathbb{N}} x^i) = \sum_{i \in \mathbb{N}} (zx^i),$$
$$(\sum_{i \in \mathbb{N}} x^i)z = \sum_{i \in \mathbb{N}} (x^i z).$$

This explains the relationship between the induction axiom and continuity assumptions. □

More generally, least pre-fixpoints and fixpoints as well as greatest post-fixpoints and fixpoints of arbitrary isotone functions exist in quantales, and can be

obtained by iteration under appropriate (co)continuity assumptions. But before explaining this in more detail we set up the link between quantales and KAT.

One can further enrich quantales with a test dioid to form a *quantale with tests* as described in §3.2. It is desirable that, in the test algebra, the binary meet operation of the quantale coincides with multiplication. For that, it is sufficient to impose a closure condition for the meet operator on tests, so that $p \sqcap q$ is a test whenever p and q are.

class *quantales-tests* = *quantale* + *dioid-tests* +
  assumes *test-meet-closure: test p* $\Longrightarrow$ *test q* $\Longrightarrow$ *test* $(p \sqcap q)$

The proof that binary meet and multiplication coincide on tests is straightforward in Isabelle. We use antisymmetry of equality to create two subgoals that can be discharged simply.

lemma *test-meet: test p* $\Longrightarrow$ *test q* $\Longrightarrow$ $p \sqcap q = p \cdot q$
  apply (*auto simp: test-restrictl test-restrictr intro!: antisym*)
  by (*metis inf.boundedE test-meet-closure mult-inf-subdistr ...*)

Finally, Propositions 3.3 and 6.1 immediately yield the following corollaries.

Corollary 6.1. *Every quantale with test is a* KAT.

Corollary 6.2. *The inference rules of* PHL *are derivable in quantales with tests.*

As already mentioned, fixpoints beyond the Kleene star exist in quantales. A basic fixpoint calculus for quantales in Isabelle can then be implemented, extending a previous one for continuous lattices [12]. Two key ingredients are Knaster-Tarski's fixpoint theorem, according to which any isotone function $f : L \to L$ over a complete lattice L has a least fixpoint μf, and Kleene's fixpoint theorem, according to which each least fixpoint of a continous function over a complete lattice can be obtained by iteration to the first infinite ordinal: $\mu f = \sum_{i \in \mathbb{N}} f^i(0)$ for $f^0 = id$ and $f^{n+1} = f \circ f^n$.

theorem *knaster-tarski: mono f* $\Longrightarrow$ $\exists!x.$ *is-lfp x f*
  by (*metis knaster-tarski-lpp lfp-equality lpp-is-lfp*)

Using the Knaster-Tarski theorem, a Hoare rule for recursive programs can then be derived.

Lemma 6.1. *Let* $f : Q \to Q$ *be an isotone function on a test quantale* Q *and* $p, q \in t(Q)$. *Then*

$$(\forall x \in Q.\ \{p\}\ x\ \{q\} \Rightarrow \{p\}\ f\ x\ \{q\}) \Rightarrow \{p\}\ \mu f\ \{q\}. \qquad \text{(recursion rule)}$$

Proof. Let $M = \{x \in Q \mid x \leq \mu f \wedge \{p\}\ x\ \{q\}\}$. We claim that if $\{p\}\ x\ \{q\} \Rightarrow \{p\}\ f\ x\ \{q\}$ holds for all x, then $\sum M = \mu f$. Clearly $\sum M \leq \mu f$. Then by isotonicity and least fixpoint definition

$$f\ (\sum M) \leq f\ \mu f \leq \mu f.$$

But the Hoare triple $\{p\}\ \sum M\ \{q\}$ is valid and, by assumption, $\{p\}\ f\ (\sum M)\ \{q\}$ holds as well. Hence $f\ (\sum M) \in M$, which implies that $f\ (\sum M) \leq \sum M$. Thus, by the Knaster-Tarski theorem, $\mu f \leq \sum M$. Therefore $\mu f = \sum M$ and $\{p\}\ \mu f\ \{q\}$ follows from $\{p\}\ \sum M\ \{q\}$. □

The proof of this proposition in Isabelle is straightforward and automatic, which also shows its power when handling nontrivial proof tasks. Lemma 6.1 and Corollary 6.2 then imply the following fact.

Theorem 6.1. *The rules of* PHL *enriched with the recursion rule are derivable in quantales with tests.*

Finally, in order to build a tool based on quantales with tests, we need to prove soundness with respect to the relational model. In fact, this result subsumes all other soundness results so far.

Theorem 6.2. *Let A be a set. Then $(2^{A\times A}, \subseteq, \circ, Id_A)$ forms a quantale with tests.*

Nevertheless, PHL with the recursion rule is not (relatively) complete for recursive parameterless programs. Auxiliary variables, which are needed for verification tasks, may appear in assertions. As expected, they can be arbitrary and have no impact on the validity of a Hoare triple.

Let P and Q be functions from a tuple of auxiliary variables to state sets. One should be able to prove that $\{P\ u\}\ R\ \{Q\ u\}$ is equivalent to $\{P\ v\}\ R\ \{Q\ v\}$ when R does not depend on the auxiliary variables u and v. Clearly this is not possible with standard Hoare logic. The only rule that we could apply is the consequence rule, which would imply that $P\ u = P\ v$ and $Q\ u = Q\ v$.

To overcome this problem, Apt [5] has introduced three adaptation rules, but these are difficult to automate and rather unsuitable for interactive theorem provers. Nipkow [87] has therefore implemented a stronger version of the consequence rule in Isabelle which has been proposed by Kleymann [70]. We prove Kleymann's stronger version of the consequence rule in the relational model.

Proposition 6.2. *Let P, P', Q, Q' be maps from auxiliary variables to state sets and R a relation. If*

$$\forall \sigma\ \sigma'.\ (\forall u.\ \sigma \in P'\ u \Rightarrow \sigma' \in Q'\ u) \Rightarrow (\forall v.\ \sigma \in P\ v \Rightarrow \sigma' \in Q\ v),$$

then

$$\forall u.\ \{P'\ u\}\ R\ \{Q'\ u\} \Rightarrow \forall v.\ \{P\ v\}\ R\ \{Q\ v\}.$$

The importance of this rule when verifying recursive programs is shown by an example. Here and henceforth we write **letrec** f **in** S **end** as a syntactic sugar for $\mu(\lambda f.\ S)$.

Factorial Example. A program that calculates the factorial of a number can be written as follows.

```
letrec Fac in
  if `x = 0 then
    `y := 1
  else
    `x := `x - 1;
    Fac;
    `x := `x + 1;
    `y := `y·`x
  fi
end
```

We would like to prove that $\{u = \text{`}x \wedge \text{`}x \geq 0\}$ *Fac* $\{u = \text{`}x \wedge \text{`}y = \text{`}x!\}$ holds. When applying the tactic *hoare* expanded with the recursion rule, the verification condition

$$\{u = \text{`}x \wedge \text{`}x \geq 0\}\ \mathit{Fac}\ \{u = \text{`}x \wedge \text{`}y = \text{`}x!\} \Rightarrow \{u = \text{`}x + 1 \wedge \text{`}x \geq 0\}\ \mathit{Fac}\ \{u = \text{`}x + 1 \wedge \text{`}y = \text{`}x!\}$$

is generated, but it is unprovable by standard Hoare logic. One should be able to instantiate u in the assumption; after all, u is arbitrary and does not occur in *Fac*. By applying the usual consequence rule, the assertion $u = \text{`}x + 1 \wedge \text{`}x \geq 0$ for instance would need to imply that $u = \text{`}x \wedge \text{`}x \geq 0$, which is obviously not the case. Yet with Kleymann's rule, Isabelle can prove it completely automatically with *force* or *auto*. □

6.2 Binary Search

As a larger example of an algorithmic verification in the presence of recursive procedures we prove the correctness of binary search. This example is drawn from the book [5]. The idea behind binary search is commonly known. It is a dichotomic divide and conquer search algorithm, which searches an element in

a sorted array by halving the number of items to check within each call. The algorithm is as follows.

```
letrec BinSearch in
  'mid := ('first + 'last)/2;
  if 'first ≠ 'last then
    if A ('mid) < val then
      'first := 'mid +1;
      BinSearch
    else if A ('mid) > val then
      'last := 'mid;
      BinSearch
    fi
  fi
end
```

The sorted array A and the element to search val is given as input to the program. The variables $`first$ and $`last$ correspond respectively the first and the last element of the array. The variable $`mid$ is its output; it corresponds to the position of the element val in the array A whenever such an element exists.

As explained in §4.4, we model arrays as functions from integers to a polymorphic linear ordered type. Define an integer interval to be the set

$$[\![i : j]\!] = \{k \mid i \leq k \leq j\},$$

and the predicate *sorted* as

$$sorted\ A\ i\ j = \forall m\ n.\ i \leq m \leq n \leq j \Rightarrow A(m) \leq A(n).$$

By using the defined set and predicate together with auxiliary variables f and l, we can express the precondition of the binary search algorithm as

$$f = `first \wedge l = `last \wedge `first \leq `last \wedge sorted\ A\ `first\ `last,$$

and its postcondition as

$$f \leq `mid \wedge `mid \leq l \wedge (A\ (`mid) = val \Leftrightarrow (\exists x \in [\![f : l]\!].\ A(x) = val)).$$

Figure 6.1 shows the complete annotated algorithm. Note that for each *BinSearch*, we have annotated the call with pre and postconditions. We also have annotations before some if statements to guide the verification condition generator. Applying the *hoare* tactic generates 10 proof obligations. Eight of them are discharged by calling Isabelle's tactical *force* or *simp* together with the

$$
\begin{array}{l}
\textbf{lemma}\ \vdash \{\!|\ f = \text{`}first \wedge l = \text{`}last \wedge \text{`}first \leq \text{`}last \wedge sorted\ A\ \text{`}first\ \text{`}last\ |\!\} \\
\ \textbf{letrec}\ BinSearch\ \textbf{in} \\
\quad \text{`}mid := (\text{`}first + \text{`}last)/2; \\
\quad \{\!|\ f = \text{`}first \wedge l = \text{`}last \wedge \text{`}first \leq \text{`}last \wedge sorted\ A\ \text{`}first\ \text{`}last \wedge \text{`}mid = (\text{`}first + \text{`}last)/2\ |\!\} \\
\quad \textbf{if}\ \text{`}first \neq \text{`}last\ \textbf{then} \\
\qquad \textbf{if}\ A\ (\text{`}mid) < val\ \textbf{then} \\
\qquad\quad \text{`}first := \text{`}mid + 1; \\
\qquad\quad \{\!|\ (f{+}l)/2 + 1 = \text{`}first \wedge l = \text{`}last \wedge \text{`}first \leq \text{`}last \wedge sorted\ A\ \text{`}first\ \text{`}last \\
\qquad\qquad \wedge\ sorted\ A\ f\ l \wedge A((f{+}l)/2) < val\ |\!\} \\
\qquad\quad BinSearch \\
\qquad\quad \{\!|\ (f{+}l)/2 + 1 \leq \text{`}mid \wedge \text{`}mid \leq l \wedge (A\ (\text{`}mid) = val \Leftrightarrow (\exists x \in [\![(f{+}l)/2 + 1 : l]\!].\ A(x) = val)) \\
\qquad\qquad \wedge\ sorted\ A\ f\ l \wedge A((f{+}l)/2) < val\ |\!\} \\
\qquad \textbf{else} \\
\qquad\quad \{\!|\ f = \text{`}first \wedge l = \text{`}last \wedge \text{`}first \leq \text{`}last \wedge sorted\ A\ \text{`}first\ \text{`}last \wedge \text{`}mid = (\text{`}first + \text{`}last)/2 \\
\qquad\qquad \wedge\ \text{`}first \neq \text{`}last \wedge A((f{+}l)/2) < val\ |\!\} \\
\qquad\quad \textbf{if}\ A\ (\text{`}mid) > val\ \textbf{then} \\
\qquad\qquad \text{`}last := \text{`}mid; \\
\qquad\qquad \{\!|\ f = \text{`}first \wedge (f{+}l)/2 = \text{`}last \wedge \text{`}first \leq \text{`}last \wedge sorted\ A\ \text{`}first\ \text{`}last \\
\qquad\qquad\quad \wedge\ sorted\ A\ f\ l \wedge val < A((f{+}l)/2)\ |\!\} \\
\qquad\qquad BinSearch \\
\qquad\qquad \{\!|\ f \leq \text{`}mid \wedge \text{`}mid \leq (f{+}l)/2 \wedge (A\ (\text{`}mid) = val \Leftrightarrow (\exists x \in [\![f : (f{+}l)/2\ ]\!].\ A(x) = val)) \\
\qquad\qquad\quad \wedge\ sorted\ A\ f\ l \wedge val < A((f{+}l)/2)\ |\!\} \\
\qquad\quad \textbf{fi} \\
\quad \textbf{fi} \\
\ \textbf{end} \\
\{\!|\ f \leq \text{`}mid \wedge \text{`}mid \leq l \wedge (A\ (\text{`}mid) = val \Leftrightarrow (\exists\ x \in [\![f : l]\!].\ A\ !\ x = val))\ |\!\}
\end{array}
$$

Figure 6.1: Verification of binary search (annotated algorithm)

definition of the predicate *sorted*. The remaining two conditions, derived with the consequence rule of Hoare logic, are less automatic. To deal with them, following [5], we have derived a condition that limits the search to a smaller array section,

$$(sorted\ A\ f\ l \wedge f \leq m < l \wedge A(m) < val) \Rightarrow \\ (\exists x \in [\![m+1 : l]\!].\ A(x) = val \Leftrightarrow \exists x \in [\![f : l]\!].\ A(x) = val).$$

Using this new lemma suffices to finish the verification of binary search.

Although termination remains to be proven, as usual, this example shows that our approach is robust enough to verify correctness of program with parameterless recursion. In addition, the items used in the array are polymorphic to any linearly ordered type.

6.3 Quantales and Program Transformations

In this final section, we present two examples for the applicability of quantales in program transformation and construction.

The first one is a classical program transformation example used in compilers. In the context of quantales with tests, one can relate tail recursion to a while loop, which is useful for compiler optimisation. The Isabelle proof is fully automatic.

lemma while b **do** x **od**; $y =$ **letrec** F **in** (**if** b **then** x; F **else** y **fi**) **end**
by (*force simp: cwhile-def cond-def lfp-lowerbound lfp-greatest antisym*)

The second example uses Galois connections. It shows that, in the context of quantales, specification statements (§5.1) can be defined as upper adjoints of a Galois connection. A *Galois connection* between two posets $(A, \leq_A)$ and $(B, \leq_B)$ is a pair of functions $f : A \to B$ and $g : B \to A$ such that

$$f\ x \leq_B y \ \Leftrightarrow\ x \leq_A g\ y.$$

The function f is called the *lower adjoint* and g the *upper adjoint* of the Galois connection. Identifying functions as adjoints of Galois connections is often desirable, as they are known to satisfy many useful properties. For a detailed overview of properties of Galois connections, see [1] for a survey.

It is a general property of Galois connections that any given function f on a complete lattice, which is continuous and hence satisfies $f\ (\sum_{i\in I} x_i) = \sum_{i\in I} f\ x_i$, has an upper adjoint g defined by

$$g\ y = \sum \{x \mid f\ x \leq y\}.$$

Clearly, for arbitrary tests p and q in a quantale with tests, the function $\lambda x.\ px\overline{q}$ distributes over suprema. Hence in test quantales, the specification statement can be defined explicitly as

$$[p, q] = \sum\{x \mid px\overline{q} \leq 0\},$$

where the Galois connection

$$\{p\}\ x\ \{q\} \Leftrightarrow px\overline{q} = 0 \Leftrightarrow x \leq [p, q].$$

holds by definition. Perhaps a better way to see this Galois connection is via the use of modal operators. In the relational model, the statement $px\overline{q} = 0$ expresses that the image of the set p under the relation x is contained in the set q. This can be rewritten as $\langle x|p \leq q$, using the backward diamond $\lambda x, y.\ \langle x|y$, as an image operator (*cf.* Chapter 7). The above Galois connection can then be written in more standard form as

$$(\lambda y.\ \langle x|y)\ p \leq q \Leftrightarrow \langle x|p \leq q \Leftrightarrow x \leq [p, q] \Leftrightarrow x \leq (\lambda y.[p, y])\ q,$$

which makes the adjoints clearly visible. In Isabelle the fact that our specification operator is an upper adjoint lets us automatically instantiate many useful theorems about Galois connections, and use them with the specification statement.

The explicit definition of the specification statement in the quantale context and the fact that every test quantale is a KAT implies the following fact.

Corollary 6.3. *Every quantale with tests forms a* rKAT*.*

By duality, in the context of quantales, a refinement law for recursive procedures can be derived as well.

Proposition 6.3. *Let $f : Q \to Q$ be an isotone function on a quantale with tests Q and $p, q \in t(Q)$. Then*

$$(\forall x \in Q.\ [p, q] \sqsubseteq x \Rightarrow [p, q] \sqsubseteq f\ x) \Rightarrow [p, q] \sqsubseteq \mu f$$

Finally Corollary 6.3 and Proposition 6.3 yield a refinement tool that supports recursive procedures.

Theorem 6.3. *Morgan's refinement calculus with a recursion law is derivable in quantales with tests.*

The applicability of this tool for the refinement and transformation of imperative programs remains to be explored. The derivation of suitable transformation and refinement rules is an interesting direction for future research.

6.4 Conclusions

This chapter has further generalised the correctness tools developed in the previous chapters. It documents the implementation of a tool based on quantales with tests, which is more expressive that the previous ones and is able to handle recursive procedures. It also demonstrated that quantales are important structures for program transformation and expressive enough to define a specification statement operator.

Chapter 7

Predicate Transformers Semantics

The *weakest precondition calculus* serves as an alternative way for reasoning about programs. This chapter shows how to add this feature to our verification tool framework. Since variants of KAT are not expressive enough to derive the weakest precondition of an arbitrary element [110], Kleene Algebras with Domain (KAD) [37] are presented, implemented in Isabelle/HOL and then linked to the relational model.

7.1 Alternative Semantics

Predicate transformers form an alternative denotational semantics for imperative programs by assigning to each command a corresponding total function between two predicates. They are inspired by the *weakest preconditions* (or the *strongest postconditions*) calculus, where the problem of verifying a Hoare triple is reduced to verifying a predicate implication, or isomorphically, a subset relation.

Let 2^A be the set of predicates over an arbitrary set A by powerset lifting, then the structure $(2^A, \subseteq)$ form a complete boolean algebra. Moreover, the set of predicate transformers $F : 2^B \to 2^A$ is denoted by $(2^A)^{2^B}$, where the following operators and constants are lifted pointwise

$$\begin{aligned} 0\ q &= \bot, \\ \top\ q &= \top, \\ (F \sqcap G)\ q &= F\ q \cap G\ q, \\ (F + G)\ q &= F\ q \cup G\ q. \end{aligned}$$

The boolean complement of a predicate in 2^A has no counterpart in the algebra of predicate transformers. We write p, q and r for predicates and F and G for

predicate transformers. Notation of the operators is overloaded and let implicit by the context. The following result is an immediate consequence of the lifting definition.

Proposition 7.1 ([19])**.** *Predicate transformers in* $(2^A)^{2^B}$ *form complete distributive lattices.*

In addition, predicate transformers in $(2^A)^{2^A}$ form a monoid under function composition with the identity function as the unit. Such predicate transformers form a distributive *near-quantale*, which is a quantale such that the left distributivity law,

$$x \cdot (\sum_{i \in I} y_i) = \sum_{i \in I} (x \cdot y_i),$$

need not hold. The monotone predicate transformers in $(2^A)^{2^A}$, which satisfy $p \leq q \Rightarrow F\ p \leq F\ q$, form a distributive *pre-quantale*, which is a near-quantale in which the left monotonicity law, $x \leq y \Rightarrow z \cdot x \leq z \cdot y$, holds. This yields the following result.

Theorem 7.1. *Let* A *be a set and* 2^A *its powerset lifting. The monotone predicate transformers over* 2^A *form a distributive pre-quantale.*

The proof consists of showing that the predicate transformers over 2^A form a near quantale and checking that the monotone predicate transformers form a subalgebra of this near-quantale. In fact, the unit predicate transformer has to be monotone—which is the case—and composition have to preserve monotonicity.

Proposition 7.2. *The healthiness condition imposed by Dikstra on predicate transformers,* cf. §*2.3, hold in a pre-quantale.*

Similarly, as in Kleene algebra with test, the quantale of predicate transformers supports a shallow algebraic embedding of a simple while language. We distinguish, however, two semantics: the *backward* predicate transformer and its opposite, the *forward* predicate transformer.

7.2 Backward Predicate Transformers

The backward predicate transformer semantics correspond closely to the weakest precondition calculus. For a given predicate q and program F, the result of the function $F\ q$ is its weakest precondition. That is, the smallest condition on the initial state ensuring that execution of the program F terminates in a final state satisfying q.

A predicate is lifted to backward predicate transformers by

$$[b] = \lambda q.\ \bar{b} + q,$$

where $\bar{p}$ denotes the boolean complement of p. The notation $[\_]$ is the same one used by Back and von Wright [19] for their *assumption* operator and it is inspired by the modal box operator in dynamic logic [59] and modal Kleene algebra/KAD. For a given postcondition q, the predicate $[b]\ q$ indicates that if we assume b then q must hold, hence logically this is an implication. Moreover, let F be a predicate transformer, if the predicate b is true on the initial states, then $[b] \cdot F = F$; otherwise $[b] \cdot F = \top$.

The semantics of a while language is then given by

$$\begin{aligned}
\textbf{abort} &= 0,\\
\textbf{skip} &= 1,\\
F; G &= F \cdot G,\\
\textbf{if } b \textbf{ then } F \textbf{ else } G \textbf{ fi} &= [b] \cdot F \sqcap [\bar{b}] \cdot G,\\
\textbf{while } b \textbf{ do } F \textbf{ od} &= ([b] \cdot F)^* \cdot [\bar{b}].
\end{aligned}$$

The meet $\sqcap$ operation is known as demonic choice [19]. We write F^* for the iteration of the predicate transformer F, which is the greatest fixpoint of $\lambda\alpha.\ 1 \sqcap \alpha \cdot F$. The quantale setting guarantees its existence by Knaster-Tarski theorem. An explicit definition can also be given by iteration to the first infinite ordinal

$$F^* = \bigsqcap_{i \in \mathbb{N}} F^i$$

where $F^0 = 1$ and $F^{n+1} = F \cdot F^n$, as usual.

Hoare logic can easily be derived for monotone backward predicate transformers, where validity of a Hoare triple is defined as

$$\{p\}\ F\ \{q\} \Leftrightarrow p \leq F\ q.$$

Note the backward reasoning strategy when building a valid deduction on Hoare logic. From a postcondition q, we calculate its weakest precondition $F\ q$ and verify a subset relation.

7.3 Forward Predicate Transformers

Similarly, the forward predicate transformer semantics correspond to the strongest postcondition calculus. The result of a program F applied to states satisfying a

precondition p is the strongest postcondition $F\ q$, that is, the largest condition held on final states, after the execution of F.

A predicate is lifted to forward predicate transformer by

$$\langle p \rangle = \lambda q.\ p \sqcap q.$$

This is called the *assertion* operator by [19]. Our notation differs from the literature and it is used to contrast with the box notation. This will become clear in the remaining sections.

The semantics of a while language is the same as before, where $\sqcap$ is substituted by $+$, the angelic choice; $[\_]$ by $\langle \_ \rangle$; and the order of composition is swapped, yielding a somehow similar algebraic semantics of the one in the context of KAT.

$$\begin{aligned} \mathbf{abort} &= 0, \\ \mathbf{skip} &= 1, \\ F; G &= G \cdot F, \\ \mathbf{if}\ b\ \mathbf{then}\ F\ \mathbf{else}\ G\ \mathbf{fi} &= \langle b \rangle \cdot F + \langle \bar{b} \rangle \cdot G, \\ \mathbf{while}\ b\ \mathbf{do}\ F\ \mathbf{od} &= (\langle b \rangle \cdot F)^* \cdot \langle \bar{b} \rangle. \end{aligned}$$

Here, iteration F^* is modelled using the least fixpoint of $\lambda\alpha.\ 1 + \alpha \cdot F$. Note that, since the order in forward predicate transformers is the inverse dual of the one for backward predicate transformers, iteration is defined by a least fixpoint instead of a greatest fixpoint.

Hoare logic is again derivable for monotone forward predicate transformers with the Hoare triple defined as

$$\{p\}\ F\ \{q\} \Leftrightarrow F\ p \leq q.$$

Similarly, this correspond to a forward reasoning; from a precondition p, we derive the strongest postcondition $F\ p$.

7.4 Relating Semantics

This section shows a well known relationship between relational semantics and forward/backward predicate transformers [19].

A *state transformer* $f_R : A \to 2^B$ is often associated with a relation $R \subseteq A \times B$ from set A to set B by defining

$$f_R\ a = \{b \mid (a, b) \in R\}.$$

It can be lifted to a forward predicate transformer, a function $\langle R \rangle : 2^A \to 2^B$ defined for all $X \subseteq A$ by

$$\langle R \rangle X = \bigcup_{a \in X} f_R\ a = \bigcup_{a \in X} \{b \mid (a, b) \in R\}.$$

Notation is overloaded to the lifting of predicate into predicate transformers. In fact, let P be a subidentity relation, then

$$\langle P\rangle X = \bigcup_{a\in X}\{b \mid (a,b)\in P\} = \bigcup_{a\in X}\lceil P\rceil = \lceil P\rceil \cap X,$$

where $\lceil P\rceil$ denotes the set isomorphic to the subidentity P. Similarly, relations are lifted to backward predicate transformers $[R] : 2^B \to 2^A$ by defining

$$[R]\ Y = \{x \mid f_R\ x \subseteq Y\} = \{x \mid \forall y \in B.\ (x,y)\in R \Rightarrow y \in Y\}$$

for all $Y \subseteq B$. The modal box and diamond notation is justified by the correspondence between diamond operators and Hoare triples as well as box operators and weakest precondition operators in the context of modal semirings and modal Kleene algebras [37]. In fact we obtain the Galois connection

$$\langle R\rangle X \subseteq Y \Leftrightarrow X \subseteq [R]Y$$

from the above definitions. For a subidentity P, the adjunction becomes the classical Galois connection of a boolean algebra

$$\lceil P\rceil \cap X \subseteq Y \Leftrightarrow X \subseteq \lceil \bar{P}\rceil \cup Y.$$

Proposition 7.3. *The following properties hold for all relations R and S.*

$$\begin{gathered}
\langle 0\rangle = [0] = \lambda x.\ \emptyset\\
\langle 1\rangle = [1] = \lambda x.\ x\\
\langle R\cup S\rangle = \langle R\rangle + \langle S\rangle,\\
[R\cup S] = [R]\sqcap[S],\\
\langle R\circ S\rangle = \langle S\rangle\cdot\langle R\rangle = \lambda x.\ \langle S\rangle\ (\langle R\rangle\ x),\\
[R\circ S] = [R]\cdot[S] = \lambda x.\ [R]\ ([S]\ x).
\end{gathered}$$

Proposition 7.3 justifies the alternative semantics given for a simple while language using forward or backward predicate transformers. Additionally, their Hoare triple are equivalent.

Theorem 7.2. *The Hoare triples in all semantics considered (relational, forward and backward predicate transformers) are equivalent. That is, let P and Q be arbitrary sets, and let R be a relation, then,*

$$\lfloor P\rfloor \circ R \circ \lfloor \bar{Q}\rfloor = \emptyset \Leftrightarrow \lfloor P\rfloor \circ R \subseteq R\circ\lfloor Q\rfloor \Leftrightarrow \langle R\rangle P \subseteq Q \Leftrightarrow P \subseteq [R]Q.$$

7.5 Modal Kleene Algebras

KAT is not expressive enough to model algebraically this intricate relationship between relations and backward/forward predicate transformers. In this section, Kleene algebras are expanded to modal Kleene algebras, which are algebraic relatives of propositional dynamic logic [59]. The idea is to integrate programs into a predicate language, allowing actions to be modal operators [50]. Modal Kleene algebras [37] can be axiomatised in various ways, three of which are introduced briefly in this section. The general approach is to embed, like in KAT, a suitable boolean algebra B of tests or assertions into a Kleene algebra K modelling the actions of a program.

Modal diamond operators are axiomatised in dynamic algebras as mappings $\langle \_ \rangle \_ : K \to B \to B$ and modal box operators as $[\_]\_ : K \to B \to B$. The former model the strongest postcondition operator, while the latter the weakest (liberal) precondition one. Validity of Hoare triple is then defined algebraically within a dynamic algebra as

$$\{p\}\ x\ \{q\} \Leftrightarrow \langle x \rangle p \leq q \Leftrightarrow p \leq [x]q.$$

The notation of the modal operators are overloaded with its corresponding semantics in the relational Kleene algebra, which are the forward and backward lifting operators from relations to predicate transformers.

Single sorted axiomatisations

Axiomatisations of modal boxes and diamonds over a Kleene algebra can be based on that of a domain or antidomain operation [37, 38]. A *domain semiring* is a semiring S expanded by a *domain operation* $d : S \to S$ that satisfies

$$\begin{aligned} x + d(x) \cdot x &= d(x) \cdot x, \\ d(x \cdot y) &= d(x \cdot d(y)), \\ d(x) + 1 &= 1, \\ d(0) &= 0, \\ d(x + y) &= d(x) + d(y). \end{aligned}$$

In fact, every domain semiring is additively idempotent and therefore a dioid. It can be checked that the domain operation is a retraction, that is, $d^2 = d$. Hence the set $d(S)$ of domain elements of S consists precisely of the fixpoints of d. This is useful for typing domain elements and checking their closure conditions.

Proposition 7.4. *The domain elements $d(S)$ of a domain semiring S form a bounded distributive lattice in which $\cdot$ is meet, 0 is the least element and 1 is the greatest element.*

The domain algebra $d(S)$ does in fact contain the greatest boolean subalgebra of S between 0 and 1, but does not need to be a boolean algebra itself. Due to this structure, domain elements can serve as tests or assertions. Henceforth we write p, q, r for them.

Domain semirings are too weak to express the negation of a test. Boolean domain algebras can be obtained by axiomatising an antidomain operator, which yields the boolean complement of a domain element.

An *antidomain semiring* is a semiring S expanded by an *antidomain operation* $a : S \to S$ that satisfies

$$\begin{aligned} a(x) \cdot x &= 0, \\ a(x \cdot y) + a(x \cdot d(y)) &= a(x \cdot d(y)), \\ d(x) + a(x) &= 1, \end{aligned}$$

where we have written $d = a^2$. This is justified because every antidomain semiring is a domain semiring with respect to this definition.

Antidomain elements are domain elements since $d \circ a = a$. Moreover $d(x) \cdot a(x) = 0$ for all $x \in S$, hence a implements boolean complementation on $d(S)$. We henceforth also write $\overline{p}$ instead of $a(p)$.

Theorem 7.3 ([38])**.** *The domain algebra of an antidomain semiring forms the maximal boolean subalgebra of the semiring of subidentites.*

Proposition 7.5. *Let A be a set. The structure $(2^{A \times A}, \cup, \circ, a, \emptyset, Id_A)$ forms an antidomain semiring with $a(R) = \{(a, a) \mid \forall b \in A.\ (a, b) \notin R\}$.*

We call this structure the *full relational antidomain semiring* over A. It follows that every subalgebra is a again an antidomain semiring—a *relational antidomain semiring*. Moreover, every relational antidomain semiring is also a relational domain semiring with $d(R) = \{(a, a) \mid \forall b \in A.\ (a, b) \in R\}$.

It follows from the definitions that $\langle R \rangle P = d(R \circ \lfloor P \rfloor)$ in a relational domain semiring. More abstractly we can define, for any antidomain semiring,

$$\begin{aligned} \langle x \rangle y &= d(x \cdot y), \\ [x]^o y &= a(\langle x \rangle a(y)) = a(x \cdot a(y)), \end{aligned}$$

where $[x]^o$ is called the dual of the modal box operator and it is defined by Morgan duality. These are sometimes called forward diamond and box operators and are written as $|x\rangle$ and $|x]$. Here, we stick with the classical notation of dynamic algebras.

Extending domain semirings with a Kleene star * is straightforward and yields a Kleene algebra with domain (KAD) [37].

Modal algebras

The dual S^o of a semiring S is a semiring where multiplication is applied backward, that is, if $(S, \odot)$ is the dual of $(S, \cdot)$, then $x \odot y = y \cdot x$. The duals of a domain and an antidomain semiring define the operators d^o and a^o respectively. They are also called range and antirange, since in the relational model the following holds

$$d^o(R) = \{(b, b) \mid \exists a \in A.\ (a, b) \in R\}.$$

Proposition 7.6. *The structure* $(2^{A \times A}, \cup, \bullet, ar, \emptyset, Id_A)$ *forms a dual antidomain semiring with* $R \bullet S = S \circ R$ *and* $ar(R) = \{(b, b) \mid \forall a \in A.\ (a, b) \notin R\}$.

Similarly, for any dual antidomain semiring, we define the modal operators

$$\begin{aligned} \langle x \rangle^o y &= d^o(y \cdot x), \\ [x]y &= a^o(\langle x \rangle a^o(y)) = a^o(a^o(y) \cdot x). \end{aligned}$$

These are also called backward diamond and box operators, and are written as $\langle x|$ and $[x|$.

In the relational model, the domain of the range of a relation R is the range of R. Similarly, the range of its domain is equal to its domain. This result entails the definition of a *modal semiring*, which is formed by an antidomain semiring and its dual, such that the following holds:

$$\begin{aligned} d(d^o(x)) &= d^o(x), \\ d^o(d(x)) &= d(x). \end{aligned}$$

Proposition 7.7 ([37])**.** *In a modal semiring, boxes and diamonds are upper and lower adjoints in a Galois connection. Additionally, they form conjugated pairs with their duals.*

$$\begin{aligned} \langle x \rangle p \leq q &\Leftrightarrow p \leq [x]q \\ \langle x \rangle^o p \leq q &\Leftrightarrow p \leq [x]^o q \\ p \cdot \langle x \rangle q = 0 &\Leftrightarrow q \cdot \langle x \rangle^o p = 0 \\ p \cdot [x]q = 0 &\Leftrightarrow q \cdot [x]^o p = 0 \end{aligned}$$

Finally, a *modal Kleene algebra* (mKA) is a modal semiring enriched with a Kleene star.

Two sorted axiomatisations

Alternatively, domain semirings can be axiomatised in a two-sorted setting [37]. In that case, a *test dioid* is a structure $(S, B, +, \cdot, ^-, 0, 1\}$ such that S is a dioid

and $B \subseteq S$ a boolean algebra with join $+$, meet $\cdot$, complementation $^-$, least element 0 and greatest element 1. A domain operation $d : S \to B$ can now be axiomatised, for all $x, y \in S$ and $p \in B$, by

$$\begin{aligned} x &\leq d(x) \cdot x, \\ d(p \cdot x) &\leq p, \\ d(x \cdot d(y)) &\leq d(x \cdot y). \end{aligned}$$

Similar reasoning can be used to axiomatise antidomains [37]. It is easy to see that $(S, d(S), +, \cdot, a, 0, 1)$ is a test dioid in that sense for every antidomain semiring S. The extension of this setting with duals and modal variants is straightforward.

Domain modules

Finally, a *semiring module* [47, 78] $(S, L, :)$ consists of a dioid $(S, +, \cdot, 0, 1)$, a bounded join-semillatice L with least upper bound operation $+$ and least element 0 and the scalar product $:$ of type $S \to L \to L$ such that, for all $x, y \in S$ and $p, q \in L$,

$$\begin{aligned} (x + y) : p &= x : p + y : p, \\ x : (p + q) &= x : p + y : p, \\ (x \cdot y) : p &= x : y : p, \\ 1 : p &= p, \\ x : 0 &= 0. \end{aligned}$$

Every domain semiring S defines a semiring module $(S, d(S), \lambda x, p.\ d(x \cdot p))$ and every antidomain semiring S forms a semiring module in which $L = d(S)$ forms a boolean algebra. Conversely, one can define $d = \lambda x.\ x : 1$ for each semiring module in which L has a greatest element 1. However it turns out that domain elements need not be closed with respect to multiplication under that definition.

In every semiring module $(S, L, :)$, diamond operators $\langle \_ \rangle \_ : S \times L \to L$ can be defined as $\langle x \rangle p = x : p$. Moreover, when $L = B$ is a boolean algebra, box operators $[\_]\_ : S \times B \to B$ can be defined by De Morgan duality as $[x]p = \overline{\langle x \rangle \overline{p}}$, as usual. Once again, an extension for modal module variants is straightforward.

7.6 Domain Quantale

It is desirable to be able to define directly the domain operation in a quantale; yielding a Kleene algebra with domain. Naïvely, domain could be defined as $d(x) = 1 \sqcap x \cdot \top$, which holds in the relational model. Nevertheless, in a quantale,

we can easily find a counterexample for the equation $d(x) \cdot x = x$. Consider for instance a three-elements quantale 0, t and $1 = \top$, where $t \cdot t = 0$ and $0 < t < 1$. Then

$$d(t) \cdot t = (1 \sqcap t \cdot \top) \cdot t = (\top \sqcap t \cdot 1) \cdot t = (\top \sqcap t) \cdot t = t \cdot t = 0 \neq t.$$

To overcome this issue, we consider quantales where the following distributivity law holds

$$(z \sqcap x \cdot \top) \cdot y = z \cdot y \sqcap x \cdot \top,$$

for all $x, y, z \in Q$. This is called the *right distributivity law for vectors*, because in relation algebra an element x where $x = x \cdot \top$ is usually called a *vector*. The proof of $d(x) \cdot x = x$ becomes trivial, since $x \leq x \cdot \top$, then

$$d(x) \cdot x = (1 \sqcap x \cdot \top) \cdot x = 1 \cdot x \sqcap x \cdot \top = x.$$

Similarly, we have a left distributivity law for vectors.

Lemma 7.1. *Every quantale satisfying the right distributivity law is a domain semiring where $d(x) = 1 \sqcap x \cdot \top$; moreover it is a Kleene algebra with domain. If it satisfies the left distributivity law, then it is a dual domain semiring with $d^o(x) = 1 \sqcap \top \cdot x$.*

A quantale is called a *boolean quantale* if its underlying lattice is a boolean algebra, that is, it is distributive and complemented. As previously, we would like to define an antidomain operation $a(x) = 1 \sqcap \overline{d(x)} = 1 \sqcap \overline{x \cdot \top}$ directly in a boolean quantale. However it is straightforward to find a counterexample for $a(x) \cdot x = 0$. Nevertheless, when considering a quantale where the right distributivity law for vectors holds, it is sufficient to prove that $\overline{x \cdot \top} \cdot \top = \overline{x \cdot \top}$, since $x \leq x \cdot \top$, then

$$a(x) \cdot x = (1 \sqcap \overline{x \cdot \top}) \cdot x = (1 \sqcap \overline{x \cdot \top} \cdot \top) \cdot x = 1 \cdot x \sqcap \overline{x \cdot \top} \cdot \top = x \sqcap \overline{x \cdot \top} = 0.$$

We use the conjugation of multiplication to prove the remaining claim. It is well known that a quantale is residuated [68]. A residuated boolean algebra induces the conjugates

$$x \cdot y \sqcap z = 0 \Leftrightarrow y \sqcap (x \rhd z) = 0 \Leftrightarrow x \sqcap (z \lhd y) = 0$$

for all $x, y, z \in Q$. In fact, one can give an equational definition of conjugates in a boolean quantale [55]

$$x \rhd y = \bigsqcap \{z.\ x \cdot \overline{z} \sqcap y = 0\},$$
$$x \lhd y = \bigsqcap \{z.\ x \sqcap \overline{z} \cdot y = 0\}.$$

Lemma 7.2. *Let Q be a quantale where the right distributivity law for vectors is satisfied, then the following holds for all $x, y \in Q$.*

$$x \cdot \top \sqcap y = 0 \Leftrightarrow x \sqcap y \cdot \top = 0,$$
$$x \lhd \top = x \cdot \top,$$
$$\overline{y \cdot x} \lhd x \sqcap y = 0.$$

Proof. Suppose $x \cdot \top \sqcap y = 0$. Then by the right distributivity law for vectors

$$x \cdot \top \sqcap y \cdot \top = (x \cdot \top \sqcap y) \cdot \top = 0 \cdot \top = 0,$$

and since $x \leq x \cdot \top$, we get by isotonicity of meet

$$x \sqcap y \cdot \top \leq x \cdot \top \sqcap y \cdot \top = 0.$$

Dually for the converse. Now by the equational definition of conjugation, $x \lhd \top$ is the smallest element z such that $x \sqcap \overline{z} \cdot \top = 0$. Clearly, by the previous fact, $x \sqcap \overline{x \cdot \top} \cdot \top = 0$, since $x \cdot \top \sqcap \overline{x \cdot \top} = 0$. Moreover $x \cdot \top$ is the smallest of such elements. Suppose by contradiction that exists z where $z < x \cdot \top$. Then $\overline{x \cdot \top} < \overline{z}$ and $0 = x \sqcap \overline{x \cdot \top} \cdot \top < x \sqcap z \cdot \top = 0$. Hence $x \lhd \top = x \cdot \top$. The last equation is easily derived by using the complementation and conjugation, $\overline{y \cdot x} \sqcap y \cdot x = 0 \Leftrightarrow \overline{y \cdot x} \lhd x \sqcap y = 0$. □

It is then straightforward to prove the desired equation $\overline{x \cdot \top} \cdot \top = \overline{x \cdot \top}$ by using Lemma 7.2 and that $\overline{x \cdot \top} \leq \overline{x \cdot \top} \cdot \top$, that is,

$$\overline{x \cdot \top} \cdot \top = \overline{(x \cdot \top) \cdot \top} \cdot \top = \overline{(x \cdot \top) \cdot \top} \lhd \top \leq \overline{x \cdot \top}.$$

Lemma 7.3. *Every boolean quantale satisfying the right distributivity law for vectors is an antidomain semiring where $a(x) = 1 \sqcap \overline{x \cdot \top}$. If it satisfies the left distributivity law, then it is a dual antidomain semiring with $a^o(x) = 1 \sqcap \overline{\top \cdot x}$. If it satisfies both laws, then its reduct forms a modal semiring.*

A *domain quantale* is any boolean quantale satisfying both the right and the left distributivity law for vectors.

Corollary 7.1. *Every domain quantale is a modal Kleene algebra.*

Jònsson and Tsinakis [68] have considered a residuated boolean algebra B where the following equality holds

$$x \rhd y = (x \rhd 1) \cdot y$$

for all $x, y \in B$. They have proved, and we have mechanically verified in Isabelle/HOL [55], that the resulting structure is isomorphic to a relation algebra.

Since the distributivity laws for vectors hold in a relation algebra, one could consequently use the quantale expansion of this structure to derive a modal semiring. However, the question whether the reduct of a domain quantale forms a relation algebra remains open.

Finally, we prove soundness of the last algebraic structure in our desired model for programs, the relational model.

Theorem 7.4. *Let A be a set. The structure $(2^{A\times A}, \subseteq, \circ, Id_A)$ forms a boolean quantale satisfying the distributivity laws for vectors.*

7.7 Implementation in Isabelle/HOL

This section adds support to our correctness tools for reasoning about while program using the modal box operator. We use Gomes *et al.* formalisation of KAD in Isabelle/HOL [54] and, once again, we prove as an example the correctness of Euclid's greatest common divisor algorithm.

First, soundness with respect to the relational model is proved as usual. The structure is called *full relational* KAD.

interpretation *rel-kad:*
kad (*op* $\cup$) (*op* O) *Id* $\{\}$ (*op* $\subseteq$) (*op* $\subset$) *rtrancl* ($\lambda R.\ \{(a,a) \mid a.\ \forall b.\ (a,b) \in R\}$)
⟨ *proof* ⟩

Second, since the same relational model of Chapter 4 is used, it is straightforward to derive the weakest predicate transformer for the graph $\langle f \rangle$ of a state transformer $f : \Sigma \to \Sigma$. Note that the notation for the graph of a function and the modal diamond operator is overloaded and it can be distinguished by typing.

Lemma 7.4. *Let $f : \Sigma \to \Sigma$, the following holds in relational* KAD.

$$P \subseteq \{\sigma \mid f(\sigma) \in Q\} \Rightarrow P \subseteq [\langle f \rangle]Q.$$

The assignment axiom is then a corollary of Lemma 7.4.

$$P \subseteq Q[e/`x] \Rightarrow P \subseteq [`x := e]\ Q.$$

Third, all the weakest predicate transformer rules are grouped in Isabelle/HOL by a named theorem list *wlp-rules*, with the except of the consequence rule, which is basically a monotonicity law for predicate transformers. The reason for this exception is that the list would not be safe to be applied as an introduction rule and it would generate schematic variables.

named-theorems *wlp-rules*

```
record state =
    x :: nat
    y :: nat
    z :: nat

lemma euclids:
  (‘x = xo ∧ ‘y = yo) ⊆
    [while ‘y ≠ 0
    inv gcd ‘x ‘y = gcd xo yo
    do
      ‘z := ‘y;
      ‘y := ‘x mod ‘y;
      ‘x := ‘z
    od]
    (‘x = gcd xo yo)
apply (rule wlp-rules)+
by auto (metis gcd-red-nat)
```

Figure 7.1: Euclid's algorithm by weakest liberal precondition

Finally, we are able to prove correctness of our toy example, presented in Figure 7.1. The subset relation notation is used to explicitly show the weakest precondition semantics, however, the program and its specification could have been expressed by a Hoare triple. When calling (*rule wlp-rules*), Isabelle tries to apply one of the facts as an introduction rule in the named theorem list *wlp-rules*. The + symbol indicates that Isabelle can safely apply the rules until all of them fails. Since we have one rule per program construct, it is clear that, after the application of the rules, the proof state contains all the verification conditions generated by the predicate transformer semantics. The conditions are of the form $P \subseteq Q$, where P and Q are predicates. They are essentially the same conditions generated in §4.3 by Hoare logic and can be dispatched in the same way.

Note that we have not created a tactic to generate these proof obligations, since all the rules are applied in the same fashion, simply grouping the facts is sufficient. However, if more program constructs need to be supported, such as procedure calls and recursion, one might need to create its own tactic, since these rules might not be safe or need to be applied in a specific manner. For these extensions, domain quantales can be used to derive a recursion rule, following the same recipe presented in Chapter 6.

Although the predicate transformer semantics is more expressive that Hoare logic [110], they entail the same verification conditions. In §4.2, *hoare* tactic

chooses the largest possible set of states whenever it encounters a schematic variable, essentially yielding a weakest precondition reasoning. For the purposes of a simple imperative while language, they are therefore equivalent and we do not pursue this method any longer. If one is interested in program transformation and refinement however, the predicate transformers semantics can be more useful [19, 118]. Additionally, for while programs with pointers, reasoning with predicate transformer yields a simpler approach (*cf.* Chapter 10).

7.8 Conclusions

This chapter presented alternative ways of defining the semantics of imperative while programs. We have described (three) axiomatisations of modal variants of Kleene algebras, which are more expressive and are able to characterise the weakest precondition (or the strongest postcondition) reasoning by modal operators. Additionally, these algebraic structures can be extended to incorporate quantales; we have derived conditions that they must satisfy to define domain and antidomain operations explicitly. Finally, the components are linked to our tool framework by a soundness proof with respect to the relational model and an example shows the approach at work.

Although for verification condition generation, the alternative semantics does not bring a clear benefit, the chapter demonstrates the robustness and flexibility of the algebraic approach to correctness tools presented in this thesis.

Chapter 8

Nondeterministic Programs

This chapter introduces the guarded command language proposed by Dijkstra for nondeterministic programs. We present an algebraic semantics for this language in the context of quantales with test and derive propositional Hoare logic equationally. A soundness proof links the language with the relational semantics and examples show the extended tool in practice. Finally, we present a way of verifying concurrent programs using nondeterminism and we prove the correctness in Isabelle of well-known concurrent algorithms.

8.1 Guarded Command Language (GCL)

The guarded command language (GCL) was proposed by Dijkstra [40] for its simplicity and symmetry when reasoning about nondeterministic programs. A GCL program C is a string of symbols generated by the following BNF grammar:

$$GC ::= [b \to C] \mid [GC_1, GC_2]$$
$$C ::= \textbf{abort} \mid \textbf{skip} \mid u := t \mid C_1; C_2 \mid \textbf{if } GC \textbf{ fi} \mid \textbf{do } GC \textbf{ od}$$

The commands **abort** and **skip**, sequential composition ; and assignment $u := t$ are clear and denote the same as in the while language. A guarded command $[b \to C]$ is the most basic construct in GCL, where a command C is guarded by a proposition, or test, b. Informally, if b is true in the initial state, then C is executed. Otherwise, the command fails or aborts. Guarded commands can be a single pair of test b and a command C or be listed by commas "," as in $[b_1 \to GC_1,\ b_2 \to C_2]$. A guarded command is not however a full statement of the language and it needs to be enclosed by a selection operator (or *alternative construct*) **if** _ **fi** or a repetition operator (or *repetitive construct*) **do** _ **od**. The former nondeterministically selects any command to be executed from a list of guarded commands where the guard is evaluated to true. If none of the guards are

true, the statement aborts. The latter executes the guarded commands repeatedly until none of the guards in the list are true.

One of the main advantages of GCL according to Dijkstra is the presentation of an algorithm in a symmetric fashion, enhancing its clarity. Consider, for example, Euclid's greatest common divisor algorithm; using the repetition operator, it can be written as

$$\mathbf{do}\ [x > y \to x := x - y,\ x < y \to y := y - x]\ \mathbf{od},$$

which is more readable and symmetric than the one previously written in a while language (*cf.* Chapter 4).

An algebraic semantics of GCL without assignment statement can be easily given in the context of quantales with tests. The semantics of a guarded command $[b \to x]$ is simply $b \cdot x$, where b is a test and x is an arbitrary element of the quantale. We write $[b_i \to x_i]_{i\leq n}$ for a list of guarded commands.

$$\begin{aligned}
\mathbf{abort} &= 0,\\
\mathbf{skip} &= 1,\\
x; y &= x \cdot y,\\
\mathbf{if}\ [b_i \to x_i]_{i\leq n}\ \mathbf{fi} &= \sum_{i\leq n}(b_i \cdot x_i),\\
\mathbf{do}\ [b_i \to x_i]_{i\leq n}\ \mathbf{od} &= (\sum_{i\leq n}(b_i \cdot x_i))^* \cdot \prod_{i\leq n} \bar{b_i}.
\end{aligned}$$

The semantics given here are as usual for partial correctness only. A total correctness semantics in this case would not only change the repetitive construct, but also the alternative one. It needs to ensure that $\sum_{i\leq n} b_i$ holds in the initial state of $\mathbf{if}\ [b_i \to x_i]_{i\leq n}\ \mathbf{fi}$, otherwise it would fail and do not terminate.

GCL is clearly more expressive than a while program; the next proposition shows that the conventional conditional and loop statements in a while language can be modelled in GCL.

Proposition 8.1. *Let b be a test, and x and y be arbitrary elements in a quantale with tests, then the following correspondence identities hold.*

$$\begin{aligned}
\mathbf{if}\ b\ \mathbf{then}\ x\ \mathbf{else}\ y\ \mathbf{fi} &= \mathbf{if}\ [b \to x,\ \bar{b} \to y]\ \mathbf{fi},\\
\mathbf{while}\ b\ \mathbf{do}\ x\ \mathbf{od} &= \mathbf{do}\ [b \to x]\ \mathbf{od}.
\end{aligned}$$

Therefore, GCL can be seen as an extension of the while language supporting nondeterminism.

Dijkstra has initialled proposed GCL for a predicate transformer semantics. In a quantale with domain, one can apply the backward box operator to derive

a similar semantics. His weakest precondition semantics for GCL varies a little from the one presented here, since he considers termination.

$$\begin{aligned}
[\mathbf{abort}]\ q &= 0,\\
[\mathbf{skip}]\ q &= q,\\
[x; y]\ q &= [x]\ [y]\ q,\\
[\mathbf{if}\ [b_i \to x_i]_{i\leq n}\ \mathbf{fi}]\ q &= \prod_{i\leq n}(\bar{b_i} + [x_i]\ q,)\\
[\mathbf{do}\ [b_i \to x_i]_{i\leq n}\ \mathbf{od}]\ q &= (\prod_{i\leq n}(\bar{b_i} + [x_i](\sum_{i\leq n} b_i + q)))^*.
\end{aligned}$$

8.2 Propositional Hoare Logic for GCL

Section 3.4 presents propositional Hoare logic for a while language based on Kleene algebra with tests. Since GCL is an extension of the while language, to derive a version of PHL for GCL, it is enough to derive Hoare rules for the alternative and repetitive constructs. As in KAT, this can easily be done by equational reasoning in the context of quantale with tests. Note that the validity of a Hoare triple remains the same.

Lemma 8.1. *Let p and q be tests. Moreover, for all i in I, let b_i be tests and x_i be arbitrary elements in a quantale with tests, then the following Hoare rule for the alternative construct can be derived.*

$$\forall i \leq n.\ \{p \cdot b_i\}\ x_i\ \{q\} \Rightarrow \{p\}\ \mathbf{if}\ [b_i \to x_i]_{i\leq n}\ \mathbf{fi}\ \{q\}.$$

Proof. Unfolding the definition of a Hoare triple and applying distributivity of arbitrary suprema, we have

$$\begin{aligned}
p \cdot \sum_{i\leq n}(b_i \cdot x_i) &= \sum_{i\leq n}(p \cdot b_i \cdot x_i)\\
&\leq \sum_{i\leq n}(b_i \cdot x_i \cdot q)\\
&= (\sum_{i\leq n}(b_i \cdot x_i)) \cdot q.
\end{aligned}$$

From the first line to the second, we apply monotonicity of suprema and the assumption $\{p \cdot b_i\}\ x_i\ \{q\}$, then $p \cdot b_i \cdot x_i \leq b_i \cdot x_i \cdot q$. □

Lemma 8.1 shows the classic Hoare rule for the alternative construct [5]. Nevertheless, we propose two others that are equivalent to the classical one, but they

are more suitable for machine automation and for program construction. They inductively reduce the size of the list of guarded commands inside the alternative construct.

$$\{p \cdot b\}\ x\ \{q\} \Rightarrow \{p\}\ \mathbf{if}\ [b \to x]\ \mathbf{fi}\ \{q\},$$
$$\{p \cdot b_n\}\ x_n\ \{q\} \wedge \{p\}\ \mathbf{if}\ [b_i \to x_i]_{i\leq n-1}\ \mathbf{fi}\ \{q\} \Rightarrow \{p\}\ \mathbf{if}\ [b_i \to x_i]_{i\leq n}\ \mathbf{fi}\ \{q\}.$$

There is one rule for the base step and another one for the induction step. Although all the 3 rules were proved in Isabelle/HOL, only the last two are added to the *hoare* tactic for verification condition generation.

Lemma 8.2. *Similarly, let p and q be tests, and for all i in I let b_i be tests and x_i be arbitrary elements in a quantale with tests, then the following Hoare rule for the repetitive construct can be derived.*

$$\forall i \leq n.\ \{p \cdot b_i\}\ x_i\ \{p\} \Rightarrow \{p\}\ \mathbf{do}\ [b_i \to x_i]_{i\leq n}\ \mathbf{od}\ \{p \cdot \prod_{i\leq n} \bar{b_i}\}.$$

Proof. Following similar reasoning from Lemma 8.1, we have

$$p \cdot \sum_{i\leq n}(b_i \cdot x_i) \leq (\sum_{i\leq n}(b_i \cdot x_i)) \cdot p,$$

applying simulation of *, yields

$$p \cdot (\sum_{i\leq n}(b_i \cdot x_i))^* \leq (\sum_{i\leq n}(b_i \cdot x_i))^* \cdot p.$$

By applying monotonicity of $\cdot$, we get

$$p \cdot (\sum_{i\leq n}(b_i \cdot x_i))^* \cdot \prod_{i\leq n} \bar{b_i} \leq (\sum_{i\leq n}(b_i \cdot x_i))^* \cdot p \cdot \prod_{i\leq n} \bar{b_i},$$

which completes the proof. □

Once more, we propose an equivalent rule more suitable for machine automation, reducing the problem to an alternative construct.

$$\{p \cdot \sum_{i\leq n} b_i\}\ \mathbf{if}\ [b_i \to x_i]_{i\leq n}\ \mathbf{fi}\ \{p\} \Rightarrow \{p\}\ \mathbf{do}\ [b_i \to x_i]_{i\leq n}\ \mathbf{od}\ \{p \cdot \prod_{i\leq n} \bar{b_i}\}.$$

Similarly, it is hard to find a suitable invariant for the repetitive construct. In general, it is in fact an undecidable problem. During the verification of a nondeterministic algorithm, the repetitive construct needs to be correctly annotated. We write $\mathbf{inv}\ r\ \mathbf{do}\ [b_i \to x_i]_{i\leq n}\ \mathbf{od}$ for a construct annotated with invariant r. In order to automatically generate proof obligations, the following secondary Hoare rule is added to the tactic *hoare*.

Lemma 8.3. *Let p, q and r be tests, and, for all $i \leq n$, let b_i be tests and x_i be arbitrary elements in a quantale with tests. If $p \leq r$ and $r \cdot \bigsqcap_{i \leq n} b_i \leq q$, then*

$$\{r \cdot \sum_{i \leq n} b_i\}\ \mathbf{if}\ [b_i \to x_i]_{i \leq n}\ \mathbf{fi}\ \{r\} \Rightarrow \{p\}\ \mathbf{inv}\ r\ \mathbf{do}\ [b_i \to x_i]_{i \leq n}\ \mathbf{od}\ \{q\}.$$

These results yield the following theorem.

Theorem 8.1. PHL *for* GCL *is derivable in a quantale with tests.*

Similarly, we can derive refinement laws for GCL, since the specification operator is available in the context of quantales. The derivation is straightforward.

$$[p, q] \sqsubseteq \mathbf{if}\ [b_i \to [b_i \cdot p, q]]_{i \leq n}\ \mathbf{fi},$$
$$[p, p \cdot \bigsqcap_{i \leq n} \overline{b}_i] \sqsubseteq \mathbf{do}\ [b_i \to [b_i \cdot p, p]]_{i \leq n}\ \mathbf{od}.$$

8.3 GCL Examples

The whole extension to support GCL has been done so far in the algebraic level. This means that any computational model, that forms a quantale with tests, supports reasoning in GCL and its propositional Hoare logic.

In order to finish the extension of our tool to GCL in Isabelle/HOL, it sufficient to link the algebra with its model. §4.2 shows that relations form quantale with tests, therefore the derivation of the Hoare rules are sound in the relational model. This yields the following result.

Theorem 8.2. *The inference rules of Hoare logic for* GCL *hold in the full relational quantale with tests.*

Finally, since the tool already has a concrete relational model with assignments, we are able to prove the correctness of programs using GCL. Two toy examples are shown in Figures 8.1 and 8.2. Both examples are straightforward. Note that in this version of Euclid's algorithm, both variables 'x and 'y are the same and store the greatest common divisor of their initial values. The verification of both examples are completely automatic and it was done by calling the tactic *hoare*, which generates all the necessary proof obligations. These were discharged by *auto* or Sledgehammer. In Euclid's example, the tactic *hoare* generates 4 conditions: the loop invariant initialisation, one condition per each guarded command and the establishment of the postcondition. Similarly, the other example generates 2 conditions, one per guarded command.

```
record state =
  x :: nat
  y :: nat
  z :: nat

lemma ⊢ {| 'x = xo ∧ 'y = yo |}
    inv gcd xo yo = gcd 'x 'y
    do [
      'x > 'y → 'x := 'x − 'y,
      'x < 'y → 'y := 'y − 'x
    ] od
  {| 'x = gcd xo yo ∧ 'x = 'y |}
  apply hoare
  using gcd-diff1-nat gcd-nat.commute by auto
```

Figure 8.1: Verification of Euclid's greatest common divisor algorithm in GCL

```
lemma ⊢ {| True |}
    if [
      'x ≥ 'y → 'z := 'x,
      'y ≥ 'x → 'z := 'y
    ] fi
  {| 'z = max 'x 'y |}
  by hoare auto
```

Figure 8.2: Verification of maximum algorithm in GCL

8.4 Parallelism by Nondeterminism

The tool implemented so far only supports sequential nondeterministic programs. In this section, we present a well-known transformation of parallel programs in nondeterministic ones, adding support to a parallel version of while programs [5]. For the remaining of this chapter, an *action* will be any arbitrary element of a quantale with tests (which represents a sequential **while**/GCL program), and a (parallel) *program* will be a list of actions with an associated *program counter*. The semantics of a program will be done by a nondeterministic *scheduler*.

A program counter p is a list of tests p_i and actions $\widehat{p_i}$ satisfying the following

properties for all elements of the lists:

$$\begin{gathered}\widehat{p_i} \cdot p_i = p_i, \\ p_i \cdot \widehat{p_i} = \widehat{p_i}, \\ p_i \cdot p_j = 0 \text{ for all } j \neq i, \\ \sum_{i \leq n} p_i = 1.\end{gathered}$$

$\widehat{p_i}$ is the action that turns the test p_i true, every p_i in the list is pairwise disjoint, and one of the p_i must be true. The actions of a program should not interfere with a program counter, and vice-versa. That is, an action in a program x should never stablish p_i and $x_m \cdot \widehat{p_i} \cdot x_n = x_m \cdot x_n$ for all m, n and i. Semantically, the actions $\widehat{p_i}$ represents an assignment of a program variable pc to i that is not in any action of x.

A scheduler is a part of an operating system that decides which process should run at a certain point. The repetitive construct of GCL can be thought of a very simple and naïve scheduler. That is, let $x_0, x_1, ..., x_n$ be programs and $b_0, b_1, ..., b_n$ conditions that they need respectively to satisfy in order to be executed, in other words their guards, then

$$(b_0 \cdot x_0 + b_1 \cdot x_1 + .. + b_n \cdot x_n)^* \cdot \overline{(b_0 + b_1 + ... + b_n)}$$

models the semantics of a nondeterministic scheduler. The scheduler only finishes when all the guards are false. Note that there is no guarantee of fairness and it could potentially abort (or in total correctness, run forever).

We use the following BNF grammar to generate parallel programs:

$$\langle a \rangle \mid x;;y \mid x \parallel y \mid \textbf{if } b \textbf{ then } x \textbf{ else } y \textbf{ fi} \mid \textbf{while } b \textbf{ do } x \textbf{ od} \mid \textbf{await } b \textbf{ then } a \textbf{ end}.$$

The program $\langle a \rangle$ is an atomic statement which corresponds to the action a. $x;;y$ denotes sequential composition and $x \parallel y$ parallel composition. There are two models of parallel composition in the literature: the interleaving model and the true concurrency model [79]. Here, we are interested in the former, where the execution of the parallel composition of two programs x and y is the interleaving of all actions within x and y. The conditional and loop construct are clear. The last statement, or the synchronisation construct, was proposed by Owicki and Gries [94]. Informally, whenever b is true, the construct **await** b **then** a **end** executes the action a. Otherwise, the program gets blocked until another program running in parallel turns b to true. It is used for synchronisation purposes. We also introduce the following abbreviation:

$$\textbf{wait } b = \textbf{await } b \textbf{ then skip end}.$$

The semantics of these programs can be expressed by a nondeterministic scheduler. For instance, let a sequential program x be a list of programs $x_0, ..., x_n$, and let p be the program counter associated to x, then the semantics of x can be given by

$$[\![x]\!] = \widehat{p}_0 \cdot (T^p_{0,\#x}\ (x))^* \cdot p_{n+1},$$

where $T^p_{i,j}$ is a function that transforms the program into a suitable list of guarded command actions, updating the program counter in the end of each action execution. The program counter starts as i and finish as j. We denote the number of actions in a program x by $\#x$. If x were a sequence of atomic actions, then $T^p_{0,\#x}(x) = \sum_{i \leq n} p_i \cdot x_i \cdot \widehat{p}_{i+1}$. The scheduler would start by initialising the program counter to 0. Next it would choose nondeterministically one of the actions where p_i is true, and then update p after the execution of x_i. This is done until p_{n+1}, in which case the scheduler does not have any more option and would finish its execution. Note that in this case, if all x_i were deterministic actions, x would also be deterministic. Also note that, due to the nature of program counters, the operation is indeed a nondeterministic scheduler, or a repetitive construct of GCL, since $\overline{\sum_{i \leq n} p_i} = \overline{1 - p_{n+1}} = p_{n+1}$.

Similarly, let q ($q \neq p$) be the program counter associated to another sequential program y which is a list of actions $y_0, y_1, ..., y_m$, then the parallel composition of x and y is

$$[\![x \parallel y]\!] = \widehat{p}_0 \cdot \widehat{q}_0 \cdot (T^p_{0,\#x}(x) + T^q_{0,\#y}(y))^* \cdot p_{n+1} \cdot q_{m+1}.$$

This parallel operator can easily be extended to any arbitrary number of programs.

The transformation function $T^p_{i,j}$ is defined by induction on the structure of the program:

$$\begin{aligned}
&T^p_{i,j}(\langle a \rangle) && = p_i \cdot a \cdot \widehat{p}_j \\
&T^p_{i,j}(x;;y) && = T^p_{i,i+\#x+1}(x) + T^p_{i+\#x+1,j}(y) \\
&T^p_{i,j}(\textbf{if } b \textbf{ then } x \textbf{ else } y \textbf{ fi}) && = p_i \cdot b \cdot \widehat{p}_{i+1} + T^p_{i+1,j}(x) \\
& && + p_i \cdot \overline{b} \cdot \widehat{p}_{i+\#x+1} + T^p_{i+\#x+1,j}(y) \\
&T^p_{i,j}(\textbf{while } b \textbf{ do } x \textbf{ od}) && = p_i \cdot b \cdot \widehat{p}_{i+1} + p_i \cdot \overline{b} \cdot \widehat{p}_{i+\#x+1} + T^p_{i+1,i}(x) \\
&T^p_{i,j}(\textbf{await } b \textbf{ then } a \textbf{ end}) && = p_i \cdot b \cdot a \cdot \widehat{p}_j
\end{aligned}$$

Clearly, whenever the program is an atomic action, the transformation tries to execute a if p_i is true and then update the program counter. For the sequential composition case, it generates the guarded commands for x starting from i and finishing in the end of x, that is, $i + \#x + 1$. Similarly, for y, it will start in the end of x and finish in j. The conditional construct follows a similar reasoning, it

will execute x if b is true from $i+1$ or execute y if b is false from $i+\#x+1$. The while loop construct will jump to the end of the command if b is false, otherwise it will set the program counter p to $i+1$, execute x and loop back to i. Note that $T^p_{i+1,i}(x)$ will force the last action in x to set the program counter p to i. Finally, the await statement works simply as a conditional atomic region, that is, it will execute the atomic action a only if b is true. Note that, since T is not defined for $\|$, nested application of parallel composition is not allowed.

8.5 Isabelle Implementation

In order to simplify the Isabelle implementation, the programs need to be annotated with labels, *cf.* Figure 8.3. The statement S when labelled by k becomes:

$$k =: S.$$

This allows more expressivity when writing invariants for a program. One can write an assertion Q_a about the execution of the program that holds immediately before k by simply stating $pc = k \Rightarrow Q_a$. Likewise, an assertion Q_b that always hold after k could be expressed as $pc \geq k+1 \Rightarrow Q_b$. Although, we do not implement this here, temporal logic [79] could have been used instead, hiding the program counters.

Since a scheduler is simply a repetitive construct, an invariant should be added to the parallel construct in order to generate the necessary proof obligations by the tactic *hoare*. We add the following construct to our parallel language:

$$\textbf{inv}\ b.\ x \parallel y,$$

where x and y are labelled programs. Any other invariants of loops inside a parallel component can be annotated directly here by the program counters.

Finally, programs counters are modelled by variables *'pcX* and *'pcY*.

record *pc-state* =
  pcX :: *nat*
  pcY :: *nat*

The tests in a program counter becomes simply *'pcX* = *i*, and an update action becomes an assignment *'pcX := j*. Any record state used needs to extend the basic record *pc-state*. Additionally, in order to comply with the last requirement for a program counter, the invariants *'pcX* $\leq \#x+1$ and *'pcY* $\leq \#y+1$ are automatically added to the parallel construct.

Finally, we implement a tactic *transform* that takes our labelled and parallel code and transforms into a GCL program by the method discussed in the previous

section. The implementation is straightforward and done in the Eisbach language. For more information, please refer to the Isabelle files[1].

8.6 Examples

Figures 8.3 and 8.4 show the tool in work. They are classic parallel algorithms extracted from Feijen and van Gasteren's monograph [48].

The first example (Figure 8.3) is a handshake problem, called the initialization protocol, where an initial state needs to be established before any of the component could start executing its own code. Each component would run an initialisation code and then enter in a synchronisation phase, and it would only continue to run the rest after the second component has finished its initialisation phase. Note that the postcondition does not establish the desired property, it only states that both components finish their initialisation phases. The property needs to be expressed as a global invariant; therefore we add the following specification as an invariant of the parallel operator:

$$\begin{aligned} &`pcX \leq 1 \longrightarrow `pcY \leq 2, \\ &`pcY \leq 1 \longrightarrow `pcX \leq 2. \end{aligned}$$

It states that while the first component has not yet finished its initialisation phase ($`pcX \leq 1$), which is marked as comment in the Isabelle code, the second component needs to be in its waiting statement or still running its own initialisation code ($`pcY \leq 2$); similarly for the second component.

Additionally, one needs to prove that the components do not *deadlock*, that is, that they do not wait forever. This is verified by the following invariant

$$`pcX = 2 \wedge `pcY = 2 \longrightarrow `x \vee `y,$$

which simply states that if both components are in their waiting statement, then $`x$ or $`y$ must be true. Hence, at least one of the components will be able to proceed.

The other invariants can be seen as program annotations for each of the commands and directly follows the ones from Feijen and van Gasteren. These are not quite straightforward, and the reader is invited to check their comments [48]. Note that a ghost variable $`c$ was introduced to break the symmetry between both programs. Although $`c$ is not necessary for the final version of the program and is not use in the synchronisation, since the first component does not read the value of $`c$, it is needed to write a correct invariant for the problem.

[1] https://github.com/victorgomes/veritas

If correctly annotated, the tactic *transform* re-write the program to an equivalent one in GCL, *hoare* generates 10 verification conditions (initialisation of the invariant, its maintenance–one condition for each of the 8 commands–and establishment of the postcondition) and *force*+ dispatches all the proof obligations.

As an enhancement to this correctness tool, one could formally eliminate ghost variables (such as $`c$ in this example) and allow command annotation. For instance, one could annotate the statement 0 of the first component with $`x \longrightarrow \neg`c$, and the tool would automatically introduce the invariant $`pcX = 0 \longrightarrow `x \longrightarrow \neg`c$. This enhancement would be extremely useful if the tool also could support temporal logic [79]. However, we leave this for future work.

The second example (Figure 8.4) is the solution to the mutual inclusion problem proposed by the Dutch computer science community. In this problem, two reactive components loop forever; they are synchronised in each loop and each of them run a critical section. Their solution is inspired by the CSP (Communicating Sequential Process) [62] primitive constructs, in which a component "send" a message to the other component, granting permission to enter in its critical section. The symmetry of the problem is broken by allowing the variable $`y$ to be bound between $`x$ and $`x + 1$. These two examples have similar properties and, in fact, form a family of concurrent problems [48].

Every time that the first component enters in its critical section, $`c$ is true and whenever the second component is in its critical section, then $`c$ is false. Additionally, when a component enter in its critical section, $`x$ and $`y$ are equal, that is, the components enter in their section alternately. The variable $`x$ is always equal or below $`y$, since $`x$ is only incremented whenever the second component increments first $`y$ and set $`c$ to true. Moreover $`y$ is at most $`x + 1$. These constraints are added as global invariants. Once again, the tactic *transform* rewrites the program to an equivalent one in GCL, *hoare* generates verification conditions and *force*+ dispatches all the proof obligations.

8.7 Conclusions

This chapter extended our framework with a guarded command language (GCL), supporting nondeterminism. In the context of quantales with tests, we have presented an algebraic semantics for the language and derived a version of Hoare logic. Additionally, from this GCL extension, we derived a simple correctness tool for two concurrent processes based on an interleaving model. Examples drawn from Feijen and van Gasteren's monograph [48] has shown the tool at work. Numerous improvements in the tool can be drawn for these examples, such as command annotation, temporal logic support and ghost variable elimination. A major drawback of the tool, however, is its lack of *fairness*. Since parallelism

```
record init-state = pc-state +
  x :: bool
  y :: bool
  c :: bool

lemma ⊢ {| ¬ 'c |}
    inv (
      ('pcX = 0 ⟶ 'x ⟶ ¬ 'c) ∧
      ('pcX = 1 ⟶ ('x ⟶ ¬ 'c) ∧ ('y ⟶ 'c)) ∧
      ('pcX = 2 ⟶ 'y ⟶ 'c) ∧
      ('pcX = 3 ⟶ 'c) ∧
      ('pcX = 4 ⟶ 'x ∧ 'c) ∧

      ('pcY ≤ 1 ⟶ ¬ 'c) ∧
      ('pcY = 1 ⟶ 'x ⟶ 'pcX > 1) ∧
      ('pcY ≥ 2 ⟶ 'c) ∧
      ('pcY = 4 ⟶ 'y ∧ 'c) ∧

      ('pcX = 2 ∧ 'pcY = 2 ⟶ 'x ∨ 'y) ∧
      ('pcX ≤ 1 ⟶ 'pcY ≤ 2) ∧
      ('pcY ≤ 1 ⟶ 'pcX ≤ 2)
    )
     (* Initialisation Phase 1 *)
    0 =: 'y := False ;;
    1 =: 'x := True ;;
    2 =: wait 'y ;;
    3 =: 'x := True
     (* Process 1 *)
      ||
     (* Initialisation Phase 2 *)
    0 =: 'x := False ;;
    1 =: 'y := True; 'c := True ;;
    2 =: wait 'x ;;
    3 =: 'y := True; 'c := True
     (* Process 2 *)
  {| 'x ∧ 'y ∧ 'c |}
  apply transform
  apply hoare
  by force+
```

Figure 8.3: The initialisation protocol example

```
record mutual-incl = pc-state +
  x :: int
  y :: int
  c :: bool

lemma ⊢ {| 'x = 0 ∧ 'y = 0 ∧ 'c |}
    inv (
      ('x ≤ 'y) ∧ ('y ≤ 'x + 1) ∧

      ('pcX ≤ 1 ⟶  'c ∧ ('x = 'y)) ∧
      ('pcX = 2 ⟶ ¬'c ∨ ('x + 1 = 'y)) ∧
      ('pcX = 3 ⟶ 'c ∧ ('x + 1 = 'y)) ∧

      ('pcY ≤ 1 ⟶ 'c ∨ ('x = 'y)) ∧
      ('pcY = 2 ⟶ ¬'c ∧ ('x = 'y)) ∧
      ('pcY = 3 ⟶ ¬'c ∧ ('x + 1 = 'y))
    )
    0 =: while True do
           (* Critical Section 1 *)
          1 =: 'c := False ;;
          2 =: wait 'c ;;
          3 =: 'x := 'x + 1
        od
      ||
    0 =: while True do
          1 =: wait (¬ 'c) ;;
           (* Critical Section 2 *)
          2 =: 'y := 'y + 1 ;;
          3 =: 'c := True
        od
  {| True |}
  apply transform
  apply hoare
  by force+
```

Figure 8.4: Mutual inclusion problem example

is model as nondeterministic choice, one component could potentially be chosen forever and *starve* the second one. Nevertheless, the aim of the chapter was to demonstrate how to rapidly derive an useful, albeit simple, tool for concurrent programs from algebraic principles.

Part II

Verification of *while* programs with Pointers

Chapter 9

Separation Logic

The remaining chapters of this thesis are dedicated to add support to shared mutable data structures and local reasoning for our tool framework by using separation logic. The approach to separation logic used in this thesis is constructive, where separating conjunction is defined by convolution. This chapter focus on the algebraic structure formed by the heap, the assertion language and the program semantics. Here, the backward predicate transformers semantics is used, which is a much simpler approach for programs with pointers, where assertions are only over *proper* states, excluding hence the *fault* state. The inference rules of separation logic and the refinement laws for this extension are easily derived in a pre-quantale setting. Finally, we show a correspondence between our model and a variant of the RAM model [93], where the fault state is explicit. A discussion about dynamic separation algebras finishes the chapter.

9.1 Design Approach

Over the last decade, separation logic has been receiving considerable attention. Inspired by Burstall's work on mutable data structures [29], Reynolds [100] derived a programming logic similar to Hoare's where the part of a system or resource affected by an action can be isolated and verification can occur locally. He introduced, as it would be latter called, the separating conjunction operator and the frame rule. A quick overview of separation logic was given in §2.4. Its main application lies in the verification of programs with pointers [101, 93], but it has been used in concurrency verification as well [90, 115, 26].

Currently, separation logic is supported by a large number of tools, some of which were discussed in §1.1. The correctness tools developed in the second part of this thesis add to this tool chain and present yet another implementation within Isabelle/HOL. Nevertheless, the approach is considerably different and has obvi-

ous advantages. As before, it focuses on a clear separation between control flow and data level by an algebraic semantic layer. This is achieved by developing a novel algebraic approach to separation logic which combines the abstractness and elegance of O'Hearn and Pym's categorical logic of bunched implications [92] in a way suitable for simple and efficient implementation within an interactive theorem assistant environment. The approach is entirely constructive and based on *power series* [44], which have several applications in computer science, including formal language and automata theory [24, 45].

For the purpose of this thesis, a *power series* is a function $f : M \to Q$ from a partial monoid M into a quantale Q. Addition is defined pointwise and multiplication as *convolution*

$$(f \otimes g)\ x = \sum_{x=y \circ z} f\ y \odot g\ z,$$

where $\cdot$ acts on M, $\odot$ on Q and $\otimes$ on Q^M. The function space Q^M of power series itself forms then a quantale [44]. Furthermore, if M is commutative (a *resource monoid* [30]) and Q is the quantale of booleans B (with $\odot$ as meet), the power series can be seen as characteristic functions of assertions or predicates over M. Separating conjunction then arises as a special case of convolution, and the function space B^M forms the assertion quantale of separation logic.

Inspired by previous approach [120], power series are lifted again yielding an algebraic semantics for predicate transformers over assertion quantales. Moreover, we characterise the subspace of monotone predicate transformers which forms a so-called pre-quantale and where the inferences rules of propositional Hoare logic for partial correctness can be derived equationally. These can be used for verification condition generation at the algebraic level. The frame rule of separation logic is then derived in the subalgebra of local monotone predicate transformers, thus subsuming propositional separation logic. Furthermore, Morgan's specification statement [83] can be defined in this subalgebra as usual, yielding tools for program construction. Finally, the refinement variant of the frame rule is easily obtained in this setting. Predicate transformer semantics, instead of the more common state transformer one [30], fits well into the power series approach and simplifies the development.

At the data-domain level, states σ are instantiated as concrete store-heap pairs (s, h) enriched with a fault element $\bot$. In this layer, assignments and mutation rules of separation logic are derived as well as their refinement laws counterparts. As usual, Isabelle's concrete data domain models are linked formally with the abstract algebras by soundness proofs, where algebraic facts can be picked up automatically for reasoning with the concrete model.

The approach thus provides a very simple and elegant formalisation of separation logic, which yields a modular tool for program construction and verification

with a good degree of automation. The entire technical development has been formalised in Isabelle; all proofs have been formally verified.

9.2 Partial Algebras

This section presents an algebraic structure that underlies our approach to separation logic. It abstracts the reasoning about shared resources by a partial operator, which adds them together when separated in some sense.

A *partial semigroup* $(S, D, \odot)$ is defined as a set S and a relation $D \subseteq S \times S$ with a composition $\odot : D \to S$ that satisfies the usual associativity law in the sense that if either side is defined then so is the other side and both are equal [23]. The element $x \odot y$ is said to be defined for $x, y \in S$ if and only if $(x, y) \in D$.

An obvious extension is a *partial monoid* $(M, D, \odot, 1)$, where 1 is the unit and satisfies

$$\begin{aligned} x \odot 1 &= x, \qquad (x, 1) \in D, \\ 1 \odot x &= x, \qquad (1, x) \in D, \end{aligned}$$

for all $x \in S$. A partial monoid M is said to be *commutative* if $x \odot y = y \odot x$ for all x and y such that $(x, y) \in D \Leftrightarrow (y, x) \in D$. Henceforth $\odot$ is used for a general partial operation and $\oplus$ for the commutative version. Commutative partial monoids are also known as *resource monoids* or *separation algebra*. Moreover, the *cancellative* property, that is,

$$x \oplus y = x \oplus z \Rightarrow y = z$$

for all $x, y, z \in M$, is usually said to hold in a separation algebra.

Several examples of partial algebras can be drawn from the literature [44].

1. **Ordered Pairs.** Let A be a set, then $(A \times A, \odot)$ forms a partial semigroup, where for $a, b, c, d \in A$,

   $$(a, b) \odot (c, d) = (a, d),$$

   and the operation is defined whenever $b = c$.

2. **Traces.** Let Σ be a finite set of state symbols and T a finite set of transition symbols. A *trace* is a finite word over $(\Sigma \cup T)^*$ in which state and transition alternate. For all $p_1, p_2, q_1, q_2 \in \Sigma$ and $\alpha_1, \alpha_2 \in T$, the *fusion product*

   $$p_1 \alpha_1 q_1 \odot p_2 \alpha_2 q_2 = p_1 \alpha_1 \alpha_2 q_2$$

   is defined whenever $q_1 = p_2$ and forms a commutative partial monoid with the empty trace as unit.

3. **Partial Functions.** The structure $([A \rightharpoonup B], \uplus, e)$ forms a commutative partial monoid, where $[A \rightharpoonup B]$ is the set of partial functions from A to B, $f \uplus g$ is their union whenever f and g have disjoint domains of definition and e is the empty partial function. A special case of this example, which will be used in the remaining of this thesis, is the *heaplet model*, where the heap is modelled as partial function between natural numbers.

Other examples include *permission algebras* [26], variable as resources [27], multisets and others [41, 90].

We have implemented partial algebras in Isabelle/HOL, including partial quantales, in which its monoidal operator is partial. We have also proved soundness of the algebras with respect to their standard models.

9.3 Power Series and Convolution

This section presents the remaining algebraic structures that underlie our approach to separation logic. Further details on power series and lifting constructions can be found in [44].

A *power series* is a function $f : M \to Q$, from a partial monoid M into a quantale Q. For $f, g : M \to Q$ and a family of functions $f_i : M \to Q$, $i \in I$, we define

$$(f \cdot g)\ x = \sum_{x = y \odot z} f\ y \cdot g\ z,$$
$$(\sum_{i \in I} f_i)\ x = \sum_{i \in I} f_i\ x,$$

where $y, z \in M$. The composition $f \cdot g$ is called *convolution*; the multiplication symbol is often overloaded to be used on Q and the function space Q^M. The idea behind convolution is simple: element x is split into y and z, the functions f and g are applied in parallel to y and z to calculate the values $f\ y$ and $g\ z$, and their results are composed to form a value for the summation with respect to all possible splits of x. Additionally, $(f + g)\ x = f\ x + g\ x$ arises as a special case of the supremum. Finally, we define the power series $0 : M \to Q$ and $1 : M \to Q$ as

$$0 = \lambda x.\ 0,$$
$$1 = \lambda x.\ \begin{cases} 1, & \text{if } x = 1, \\ 0, & \text{otherwise.} \end{cases}$$

The quantale structure lifts from Q to the function space Q^M of power series.

Theorem 9.1 ([44]). *Let M be a partial monoid. If Q is a (boolean) quantale, then so is $(Q^M, \leq, \cdot, 1)$. If M and Q are commutative, then so is Q^M.*

On the logic of bunched implications, developed by O'Hearn and Pym [92], this quantale is known as boolean BI algebra, or simply BBI.

The power series approach generalises from one to n dimensions [44]. For separation logic, the two-dimensional case with power series $f : S \times M \to Q$ from set S and partial commutative monoid M into the commutative quantale Q is needed. Now

$$(f * g)\ (x, y) = \sum_{y = y_1 \oplus y_2} f\ (x, y_1) * g\ (x, y_2),$$
$$(\sum_{i \in I} f_i)\ (x, y) = \sum_{i \in I} f_i\ (x, y).$$

The convolution $f * g$ acts solely on the second coordinate. We write $*$ whenever the operation is commutative. Finally, we define two-dimensional units as

$$0 = \lambda x\ y.\ 0,$$
$$1 = \lambda x\ y.\ \begin{cases} 1, & \text{if } y = 1, \\ 0, & \text{otherwise.} \end{cases}$$

Theorem 9.2 ([44]). *Let S be a set and M a partial (commutative) monoid. If Q is a (commutative boolean) quantale, then so is $Q^{S \times M}$.*

A power series $f_C : A \to B$ can be isomorphically associated to a set $C \subseteq A$ by defining its characteristic function $f_C(a) = \top \Leftrightarrow a \in A$, where B is the booleans $\{\bot, \top\}$. Moreover, B also forms a degenerated quantale, where composition $\cdot$ is meet $\sqcap$. Example of convolution are ubiquitous in mathematics.

1. **Formal Languages.** Language product is an instance of convolution

$$(f_X * f_Y)(w) = \sum_{w = w_1 w_2} f_X(w_1) \sqcap f_Y(w_2),$$

   where the monoid is word concatenation.

2. **Binary Relations.** Relational composition can be defined by convolution

$$(f_R \cdot f_S)(a, b) = \sum_c f_R(a, c) \sqcap f_S(c, b).$$

   Let $R \cdot S = \{(a, b) \mid \exists c.\ (a, c) \in R \wedge (c, b) \in S\}$, it is easy to check that $f_{R \cdot S} = f_R \cdot f_S$.

3. **Matrices.** Matrix multiplication is also given by convolution

$$(A \cdot B)(i,j) = \sum_{k=1}^{m} A(i,k) \cdot B(k,j),$$

where A is a $n \times m$ matrix and B is an $m \times p$ matrix.

We have implemented partial monoids and quantales by using Isabelle's type class and locale infrastructure, building on existing libraries for monoids, quantales and complete lattices. The implementation of power series uses Isabelle's well developed libraries for functions. This makes proofs in this setting simple and highly automatic.

9.4 Assertion Quantale

In language theory, power series have been introduced for modelling formal languages. Here, M is the free monoid X^* and Q can be taken as a semiring $(Q, +, \cdot, 0, 1)$, because there are only finitely many ways of splitting words into prefix/suffix pairs in convolutions. Infinite suprema in the definition of convolution are therefore not needed. In the particular case of the boolean semiring B, where composition $\cdot$ is meet $\sqcap$, power series $f : X^* \to B$ are interpreted as characteristic functions (i.e., predicates) that indicate whether a word is in a set. In this case, sets are languages, and hence, convolution specialises to

$$(f \cdot g)\ x = \sum_{x=yz} f\ x \sqcap g\ y,$$

identifying predicates with their extensions to the language product

$$p \cdot q = \{yz \mid y \in p \wedge z \in q\}.$$

More generally, we consider power series $S \to B$ from a partial monoid S into the boolean quantale B and set up a connection with separation logic. There, one is interested in modelling assertions or predicates over the memory heap. The heap can be represented abstractly by a resource monoid [30], which is a partial commutative monoid. By analogy to the language case, an assertion p of separation logic is a boolean-valued function from a resource monoid M, hence a power series $p : M \to B$. Thus Theorem 9.1 applies.

Corollary 9.1. *The assertions B^M over resource monoid M form a commutative boolean quantale with convolution as separating conjunction.*

The logical structure of the assertion quantale B^M is as follows. The predicate 0 is a contradiction whereas 1 (unit of B^M) holds when its extension contains the empty resource 1 (unit of M). The operations $\sum$ and $\prod$ correspond to existential and universal quantification; their finite cases yield conjunctions and disjunctions. The order $\leq$ is implication. Convolution becomes

$$(p * q)\ x = \sum_{x=y*z} p\ y \sqcap q\ z,$$

and it gives a simple algebraic account of separating conjunction. By $x = y * z$, resource x is separated into resources y and z. By $p\ y \sqcap q\ z$, the value of predicate p on y is conjoined with that of q on z. Finally, the supremum is true if one of the conjunctions holds for some splitting of x.

As for languages, one can again identify predicates with their extensions. Then

$$p * q = \{y * z \in M \mid y \in p \wedge z \in q\},$$

and separating conjunction becomes a language product over resources (cf. [63]). The analogy to language theory is even more striking when considering the paradigmatic kind of resource: multisets or Parikh vectors over a finite set X. These form the free commutative monoids over X.

Applications of separation logic, however, require program states which are store-heap pairs. Now Theorem 9.2 applies.

Corollary 9.2. *The assertions $B^{S\times M}$ over set S (the store) and resource monoid M form a commutative boolean quantale with convolution as separating conjunction. For all $p, q : S \times M \to B$, $s \in S$ and $h \in M$,*

$$(p * q)\ (s, h) = \sum_{h=h_1*h_2} p\ (s, h_1) \sqcap\ q\ (s, h_2).$$

Written in language-product style, therefore,

$$p * q = \{(s, h_1 * h_2) \in S \times M \mid (s, h_1) \in p \wedge (s, h_2) \in q\}.$$

The definition of convolution and the associated lifting is obviously flexible enough to encompass situations where pairs are extended to tuples or where the store as well as the heap are split by convolution. Isabelle also supports uncurried representations of such tuples and translations between them. The constructive functional approach of power series is very convenient for the functional programming style of Isabelle.

One can think of the power series approach to separation logic as a simpler account of the category-theoretical approach in O'Hearn and Pym's logic of bunched

implication [92] in which convolution generalises to coends and the quantale lifting is embodied by Day's construction [36]. For the design of verification tools and our implementation in Isabelle, the simplicity of the power series approach is certainly an advantage.

Quantales carry a rich algebraic structure. Their distributivity laws give rise to continuity or co-continuity properties. Therefore, many functions constructed from the quantale operations have adjoints as well as fixpoints, which can be iterated to the first limit ordinal. This is well known in denotational semantics and important for our approach to program verification. In particular, separating conjunction $*$ distributes over arbitrary suprema in B^M and $B^{S\times M}$ and therefore has an upper adjoint: the *magic wand* operation $\mathrel{-\!\!*}$, which is widely used in separation logic.

$$p * q \leq r \Leftrightarrow p \leq q \mathrel{-\!\!*} r$$

In the quantale setting, the adjunction gives us theorems for the magic wand for free. This and other residuals that arise on the assertion quantale of separation logic have been studied, for instance, in [35]. Lemma 9.1 shows some properties of magic wand (or spacial implication).

Lemma 9.1. *The following hold in a commutative quantale.*

a. $p \leq q \Rightarrow r \mathrel{-\!\!*} p \leq r \mathrel{-\!\!*} q$

b. $p \leq q \Rightarrow q \mathrel{-\!\!*} r \leq p \mathrel{-\!\!*} r$

c. $s \leq q \wedge p \leq q \mathrel{-\!\!*} r \Rightarrow p * s \leq r$

d. $q * (q \mathrel{-\!\!*} p) \leq p$

e. $q \leq p \mathrel{-\!\!*} (p * q)$

f. $(r \mathrel{-\!\!*} q) * (q \mathrel{-\!\!*} p) \leq r \mathrel{-\!\!*} p$

g. $p \mathrel{-\!\!*} \top = \top$

h. $\top \mathrel{-\!\!*} p \leq p$

i. $0 \mathrel{-\!\!*} p = \top$

j. $(p + q) \mathrel{-\!\!*} r \leq (p \mathrel{-\!\!*} r) + (q \mathrel{-\!\!*} r)$

k. $p \mathrel{-\!\!*} (q \sqcap r) \leq (p \mathrel{-\!\!*} q) \sqcap (p \mathrel{-\!\!*} r)$

l. $(p \mathrel{-\!\!*} q) + (p \mathrel{-\!\!*} r) \leq p \mathrel{-\!\!*} (q + r)$

m. $p \mathrel{-\!\!*} (q \sqcap r) \leq (p \mathrel{-\!\!*} q) \sqcap (p \mathrel{-\!\!*} r)$

Local Assertions. An assertion p is *local* if and only if, for all stores s and heaps h and h', whenever $dom(h) \subseteq dom(h')$ and $(s,h) \in p$, then $(s,h') \in p$. Local assertions can be easily characterised algebraically by

$$p * \top \leq p.$$

Additionally, it can be proved that $\top$ is unit of $*$ for local assertions, that is, the equalities $p * \top = p$ and $p = \top \mathrel{-\!\!*} p$ hold for an local assertion p.

Lemma 9.2. *Let r be any assertion, then the assertions $\top$, 0, $r * \top$ and $\top \mathrel{-\!\!*} r$ are local.*

Proof. $\top$ and 0 are clearly local, $(r * \top) * \top = r * (\top * \top) = r * \top$, and finally, $(\top \mathrel{-\!\!*} r) * \top = (\top \mathrel{-\!\!*} \top) * (\top \mathrel{-\!\!*} r) = \top \mathrel{-\!\!*} r$. □

The first primitive version of separation logic was developed by Reynolds [100] as a logic for reasoning about mutable data structures in which all predicates are written as local assertions. The assertions were later called *intuitionistic*, because his development was constructive. However, the term local is used in this thesis, since when lifted to predicate transformers, these assertions satisfy the *locality property* [30] (*cf.* §9.5).

Lemma 9.3. *Local assertions are closed under meet $\sqcap$, join $+$, separating conjunction $*$ and magic wand $\mathrel{-\!\!*}$. Moreover, if p_i is local for all $i \in I$, then so are $\sum_{i\in I} p_i$ and $\prod_{i\in I} p_i$. Therefore, the subspace of local assertions forms a quantale.*

Local assertion are not closed under negation, but it is possible to define an intuitionistic negation on assertions [100], which is always local, by

$$\bar{p}^i = \top \mathrel{-\!\!*} \bar{p}.$$

Lemma 9.4. *Let p be a local assertion, then the following hold.*

a. $p * q \leq p$,

b. $(p \sqcap 1) * \top \leq p$,

c. $p * q \leq p \sqcap (q * \top)$,

d. $p * q \leq p \sqcap q$, *when q is also local,*

e. $(p \sqcap q) * r \leq p \sqcap (q * r)$.

All the facts in this section were formally proved in Isabelle and are available in our theory files about commutative quantales.

9.5 Predicate Transformer Quantales

Our algebraic approach to separation logic is based on predicate transformers (cf. [19]). This is in contrast to most previous state-transformer-based approaches and implementations [30, 69, 112], with [120, 63] being exceptions. First of all, predicate transformers are more amenable to algebraic reasoning [19]—simply because their source and target types are both at powerset level. Second, the approach is coherent and easily implementable within our framework. Predicate transformers can be seen once more as power series and instances of Theorem 9.1 describe their algebras.

Predicate transformers in $(2^A)^{2^B}$ form complete distributive lattices [19]. In the power series setting, this follows from Theorem 9.1 in two steps, ignoring the monoidal structure. Since B forms a complete distributive lattice, so do $2^B \cong B^B$ and $2^A \cong B^A$ in the first step, and so does $(2^A)^{2^B}$ in the second one.

In addition, as seen in §7.1, predicate transformers in $(2^A)^{2^A}$ form a distributive near-quantale and if monotone, a distributive pre-quantale [19]. In these cases, the monoidal parts of the lifting are not obtained with the power series technique. The monoidal operation on predicate transformers is function composition and not convolution. Of course it is associative and the identity function is a unit of composition.

Adapting these results to separation logic requires the consideration of assertion quantales B^M or $B^{S\times M}$ with store S and resource monoid M instead of the powerset algebra over a set A. Instead of lifting these quantales, we only lift their boolean algebra reduct, disregarding separating conjuction by not lifting it to a convolution on predicate transformers, which is not needed for separation logic. The quantale structure of predicate transformers is again obtained by considering function composition as the monoidal operation. This yields the following result.

Theorem 9.3. *Let S be a set, M a resource monoid and $B^{S\times M}$ an assertion quantale. The monotone predicate transformers over $B^{S\times M}$ form a distributive pre-quantale.*

Monotone predicate transformers can be used to derive the standard inference rules of Hoare logic as verification conditions (*cf.* §9.6) and the usual rules of Morgan's refinement calculus (*cf.* §9.7). Derivation of the frame rule of separation logic, however, requires a smaller class of predicate transformers.

A backward predicate transformer F is said to be local with respect to a predicate r if for all q

$$(F\ q) * r \leq F\ (q * r).$$

This definition differs slightly from the classical definition of locality [93], where r is also quantified. This difference will become clear in §9.8. If r is arbitrary, we simply say that F is local.

The final theorem in this section establishes the local monotone predicate transformers as a suitable algebraic framework for separation logic.

Theorem 9.4. *Let S be a set and M a resource monoid. The local monotone predicate transformers w.r.t an arbitrary assertion r over the assertion quantale $B^{S\times M}$ form a distributive pre-quantale.*

The fact that the local monotone predicate transformers form a complete lattice has been observed previously [120]. Once again it must be checked that the zero predicate transformer is local—which is the case—and that the quantale operations preserve locality and monotonicity.

We have implemented the whole approach in Isabelle; all theorems have been formally verified, mainly using Theorem 9.1 for the lifting to predicate transformers. Apart from the local case, ours is not the first Isabelle formalisation of predicate transformers; it is based on previous work by Preoteasa [98].

9.6 Verification Conditions

The pre-quantale of local monotone predicate transformers supports the derivation of verification conditions by equational reasoning. A standard set of such conditions are the inference rules of Hoare logic.

The quantale setting supports a shallow algebraic embedding of a simple while language with the standard intermediate language for the verification of while-programs (*cf.* §7.2). In order to derive the frame rule of separation logic, this language needs to be closed with respect to locality, which forces a test to be a local predicate.

Lemma 9.5. *Let b be a local predicate, then $[b]$ is a local predicate transformer. Moreover, if F and G are local w.r.t. a predicate r, then* **if** b **then** F **else** G **fi** *and* **while** b **do** F **od** *are local w.r.t the predicate r.*

We provide the usual assertions notation for programs via Hoare triple syntax:

$$\{p\}\ F\ \{q\} \Leftrightarrow p \leq F\ q.$$

Proposition 9.1. *Let $p, q, r, p', q' \in B^{S\times M}$ be predicates and b a local predicate. Let F, G, H be monotone predicate transformers over $B^{S\times M}$, with H being local with respect to r. Then the rules of propositional Hoare logic (no assignment rule) and the frame rule of separation logic are derivable.*

$$\{p\}\ H\ \{q\} \Rightarrow \{p * r\}\ H\ \{q * r\}.$$

Proof. We derive the frame rule as an example. Suppose $p \leq H\ q$. Then, by isotonicity of $*$ and locality, $p * r \leq (H\ q) * r \leq H(q * r)$. □

9.7 Refinement Laws

To demonstrate the power of the predicate transformer approach to separation logic we now outline its applicability to local reasoning in program construction and transformation. We show that the standard laws of Morgan's refinement calculus [83] plus an additional framing law for resources can be derived and programmed in Isabelle with little effort. It only requires defining one single additional concept—Morgan's specification statement—which is definable in every predicate transformer quantale (*cf.* §9.4).

Formally, for predicates $p, q \in B^{S\times M}$, we define the *specification statement* as

$$[p, q] = \bigsqcap\{F \mid p \leq F\ q\}.$$

It models the most general predicate transformer or program that links postcondition q with precondition p. It is easy to see that

$$\{p\}\ F\ \{q\} \Leftrightarrow [p, q] \leq F,$$

which entails the characteristic properties

$$\{p\}\ [p, q]\ \{q\}, \qquad \{p\}\ F\ \{q\} \Rightarrow [p, q] \leq F$$

of the specification statement: program $[p, q]$ relates precondition p with postcondition q whenever it terminates; and it is the largest program with that property. It is easy to check that specification statements over the pre-quantale of local monotone predicate transformers are themselves local and monotone.

Like Hoare logic, Morgan's basic refinement calculus provides one refinement law per program construct. Once more we ignore assignments at this stage. We also switch to standard refinement notation with refinement order being $\sqsubseteq$.

Proposition 9.2. *For $p, q, r, p', q' \in B^{S\times M}$, b a local predicate, and predicate transformer F the following refinement laws are derivable in the algebra of local monotone predicate transformers with respect to r.*

$$[p * r, q * r] \sqsubseteq [p, q].$$

Proof. Using the frame rule, we derive the framing law as an example:

$$\{p\}\ [p, q]\ \{q\} \Rightarrow \{p * r\}\ [p, q]\ \{q * r\} \Leftrightarrow [p * r, q * r] \sqsubseteq [p, q].$$

The first step uses the frame rule from Proposition 9.1, the second one the Galois connection for the specification statement. The proofs of the other refinement laws are equally simple, using the corresponding Hoare rules in their proofs. They are fully automatic in Isabelle. A refinement law for recursive programs can be derived as well. □

The entire theory hierarchy discussed so far, from partial monoids to predicate transformer quantales, has been formalised in modular fashion as algebraic components in Isabelle/HOL, much of which was highly automatic and required only a moderate effort. It benefits, to a large extent, from Isabelle's integrated first-order theorem proving, SMT-solving and counterexample generation technology. These tools are highly optimised for equational reasoning, interacting efficiently with the algebraic layer.

9.8 Relational Fault Model

In this section, we show an equivalence between the backward predicate transformer model constructed to a relational model where the *fault* state is explicitly defined. That validates our model and shows its simplicity.

The classical relational model used for simple **while** programs is clearly not expressive enough to model memory *fault*, by for instance dereferencing a dangling pointer. Hence, different models were proposed and proved sound for separation logic [93, 101]. We implement a variant of the RAM model proposed by [93, 30], where a state σ is a pair of (s, h) of store and heap enriched by a bottom $\perp$ (fault) element. Elements of the store s range over an arbitrary set S, and elements of the heap h over a partial commutative monoid H. A command (or program) is then a relation

$$R \subseteq ((S \times H) \cup \{\perp\}) \times ((S \times H) \cup \{\perp\}).$$

We say that a program R starting for a state $\sigma = (s, h)$ delivers a final state $\sigma' = (s', h')$ whenever $(\sigma, \sigma') \in R$. If $(\sigma, \perp) \in R$, then a memory fault has occurred. In fact, we are interested in a subclass of these relations where the *safe* and *frame* properties with respect to a set of proper states A hold [120]. The former says that if a command executes safely (no fault occurs) in a state with a heap h, then it will execute safely in a state where the heap h is enlarged by a heap h' of A. Formally, for all s, h and h', if $(s, h') \in A$ then

$$((s, h), \perp) \notin R \Rightarrow ((s, h \oplus h'), \perp) \notin R.$$

The latter says that if a command R can execute safely in a small heap h_0, then the execution on a larger heap can be tracked back to the small one, isolating the part of the system affected by the action, that is, for all s, s', h_0, h, dh and h', if $((s_0, h_0), \perp) \notin R$, $h = h_0 \oplus dh$, $(s, dh) \in A$ and $((s, h), (s', h')) \in R$, then

$$\exists h'_0\ dh'.\ h' = h'_0 \oplus dh' \wedge (s', dh') \in A \wedge ((s, h_0), (s', h'_0)) \in R.$$

Note that these properties are only concerned about the heap and leave the store unchanged. The definition slightly differs from the literature, since they are

defined with respect to a predicate (or a set of proper state) A, if we set A to $\top$, these definitions are equivalent. In [30], these properties are presented for a partial monoidal state. Here, we needed to present the store separately, since the state does not form a partial monoid. One could think that the states form a partial semigroup where $(s, h) \oplus (s', h') = (s, h \oplus h')$ whenever $s = s'$, but this would oblige that s and s' become the same in the frame property, which does not make sense. An alternative is to treat the store as a partial resource too, however this would entail a different assignment rule for Hoare logic [27].

A test and a proper assertion in this model is a set of states $P \subseteq S \times H$. Similarly, tests can be lifted to subidentities $\lfloor P \rfloor$. Note that lifted sets are always safe and do not contain any $\bot$ element.

Lemma 9.6. *Let P be a local assertion, then $\lfloor P \rfloor$ satisfies the safe and frame properties with respect to an arbitrary assertion Q.*

In addition, the operations union $\cup$, intersection $\cap$ and reflexive-transitive closure * preserve safety and frame properties. This yields the following soundness result.

Theorem 9.5. *Relations on the relational fault model forms a Kleene algebra with tests.*

Therefore, the validity of a Hoare triple in this model can be written as

$$\{P\}\ R\ \{Q\} \Leftrightarrow \lfloor P \rfloor \circ R \subseteq R \circ \lfloor Q \rfloor .$$

In order to prove the equivalence of models, we prove that the validity of Hoare triples yields the same result and that safe and frame properties are equivalent to locality.

Following [30], we lift relations R to state transformers

$$\langle R \rangle : (S \times H) \to (2^{S \times H} \cup \{\mathbf{fail}\})$$

by

$$\langle R \rangle\ \sigma = \begin{cases} \{\sigma' \mid (\sigma, \sigma') \in R\} & \text{if } (\sigma, \bot) \notin R, \\ \mathbf{fail} & \text{otherwise.} \end{cases}$$

Notice that the range of $\langle R \rangle\ \sigma$ is an assertion enriched with a **fail** element. Order-theoretically, **fail** is a Scott's top [58] value, that is, faulting is mapped to this inconsistent top element. Separating conjunction can be further extended $*^\top$ to support this new element by $p *^\top \mathbf{fail} = \mathbf{fail} *^\top p = \mathbf{fail}$.

It is straightforward to encode the validity of a Hoare triple for state transformers. Let P and Q be proper assertions and f a state transformer, then

$$\{P\}\ f\ \{Q\} \Leftrightarrow \forall \sigma \in P.\ f\ \sigma \subseteq^\top Q.$$

We write $\subseteq^\top$ to indicate that the order is enriched with a top element **fail**. If a fault occurs for a state σ, then $f\ \sigma = \textbf{fail}$, which is greater than Q.

Lemma 9.7. *Validity of Hoare triple are equivalent in the relational semantics and in its state transformer counterpart.*

$$\lfloor P \rfloor \circ R \subseteq R \circ \lfloor Q \rfloor \Leftrightarrow \forall \sigma \in P.\ \langle R \rangle\ \sigma \subseteq^\top Q$$

We can also define locality with respect to a proper assertion A for this setting. f is a local state transformer if for all s, h and h' such that $(s, h') \in A$ then

$$f\ (s, h \oplus h') \subseteq^\top f\ (s, h) *^\top A.$$

Again, this condition differs slightly from the literature [120].

Lemma 9.8. *Let R be a relation between states, then $\langle R \rangle$ is local w.r.t. a proper assertion A iff the safety and frame properties hold for R w.r.t. A.*

Since locality is a much simpler condition than safety and frame, this explains why the state transformer approach is useful and widely used [30]. However, a **fail** element still exists and to implement this in Isabelle/HOL, one would need to wrap their states around with an *option* datatype. All operations would need to be unwrapped to be used, this would reduce automation and increase the complexity of the implementation. That is also the reason why, differently from the literature, we explicitly write $\subseteq^\top$ and $*^\top$ to indicate that these operations are wrapped around with this top element.

Finally, state transformers can be lifted to backward predicate transformers by

$$[f]\ Q = \{\sigma \mid f\ \sigma \subseteq^\top Q\}.$$

Lemma 9.9. *Validity of Hoare triple of a state transformer f is equivalent to the one for the predicate transformer $[f]$.*

$$\forall \sigma \in P.\ f\ \sigma \subseteq^\top Q \Leftrightarrow P \subseteq [f]\ Q.$$

Lemma 9.10. *f is a local state transformer w.r.t A iff $[f]$ is local predicate transformer w.r.t. A.*

The lemmas in this section entails the following result.

Theorem 9.6. *The relational fault model and the local monotone predicate transformer semantics are equivalent up to the validity of Hoare triples.*

Of all these 3 equivalent models, the backward predicate transformer is the simplest one. Although it does not contain explicitly a fault element, its backward reasoning excludes all possible dangling pointers and faulty states.

9.9 Modal Separation Algebras

Since we follow a predicate transformer approach for separation logic, it seems natural that one can define dynamic algebras over assertion quantales. This section adapts modal algebras (*cf.* §7.5) for separation logic.

Domain Separation Algebras

We use domain semirings to define a predicate transformer algebra over the assertion quantale of separation logic. As with domain semirings, the idea is to use a domain operation d over a semiring S to induce an assertion algebra $d(S)$ that contains separating conjunction. In the domain semiring case, the meet of the boolean algebra was induced by semiring composition, complementation was obtained by antidomain. Now, however, there is no semiring operation that could give us separating conjunction.

We therefore enrich an antidomain semiring S with an operation $* : S \times S \to S$ and a constant e to form a *domain separation semiring*, where the following axioms hold

$$\begin{aligned} d(d(x) * d(y)) &= d(x) * d(y), \\ d(x * (y * z)) &= d((x * y) * z), \\ d(x * (y + z)) &= d(x * y) + d(x * z), \\ d(x * y) &= d(y * x), \\ d(x * e) &= d(x), \\ d(x \cdot (y * z)) &\leq d(x \cdot y) * d(z). \end{aligned}$$

The last axiom is the *locality* axiom for domains, it yields the locality property of separation logic for predicate transformers. Hoare *et al.* [63] uses a similar axiom in concurrent Kleene algebra inspired by category theory ideas, which they call it *small exchange law*. This locality property can be expressed using the diamond modal operator as

$$\langle x \rangle (p * q) \leq \langle x \rangle p * q.$$

Lemma 9.11. *Let S be a domain separation semiring and $d(S)$ the set of elements x in S such that $d(x) = x$, then the structure $(d(S), +, *, 0, e)$ form a commutative dioid. Moreover, $(d(S), +, \cdot, *, a, 0, 1, e)$ is a boolean lattice-ordered commutative monoid, that is, it has a boolean algebra and a monoid reduct.*

Note that, since all the axioms are wrapped by the domain operator d, the separating conjunction $*$ only needs to be defined for domain elements in the model. For instance, let $**$ be this operation in the relation model, then $R{**}S$ is undefined for arbitrary relations R and S, whereas for subidentities P and Q,

$P{*}{*}Q$ needs to be defined. However, one cannot use the definition of separating conjunction proposed in the previous chapter, since the subidentities P and Q might contain the *fault* state.

Therefore, we split the subidentity space into two, P is called a *fail* assertion if $\bot \in P$, otherwise it is a *proper* assertion. The top element $\top$ of the set of proper assertions is isomorphic to the predicate **true**, whereas the top element of all assertions is the element **fail** introduced in the previous section.

The separating conjunction $P{*}{*}Q$ is then extended such that whenever $\bot$ is in P or Q, then it is also in $P{*}{*}Q$. Formally,

$$P{*}{*}Q = \{(s, h_1 * h_2) \mid (s, h_1) \in P \wedge (s, h_2) \in Q\} \cup \{\bot \mid \bot \in P \cup Q\}.$$

Theorem 9.7. *Let S be a set, H be a partial commutative monoid and Σ be the state space $(S \times H) \cup \{\bot\}$. The structure $(L, \cup, \circ, {*}{*}, \mathsf{ad}, \emptyset, Id_R, [\mathbf{emp}])$ forms a domain separation semiring, where*

$$L = \{R \in \Sigma \times \Sigma \mid R \textit{ satisfies the safety and frame property}\}$$

and

$$\mathsf{ad}(R) = \{(\sigma, \sigma) \mid \forall \sigma' \in \Sigma.\ (\sigma, \sigma') \in R\}.$$

Proof. The structure clearly forms a domain semiring. We prove by case analysis for each of the axioms of a domain separation semiring. We prove that if x, y and z does not contain the $\bot$ element, then the axiom hold. This is easily done in Isabelle/HOL. Otherwise, we need to prove that if the bottom element $\bot$ is contained in the left hand side, then it is also contained in the right hand side; and vice-versa. For the locality axiom, just one direction is needed. □

Extending a separation semiring by a Kleene star operation ${}^* : S \to S$, which satisfies the unfold and induction axioms, yields a *separation Kleene algebra with domain*. All the classical inference rules of propositional separation logic can be derived in a separation Kleene algebra, that is, all except the rules concerning the mutation of store and heap.

Corollary 9.3. *The structure $(L, \cup, \circ, {*}{*}, \mathsf{ad}, {}^*, \emptyset, Id_R)$ forms a separation Kleene algebra, where * corresponds to the reflexive-transitive closure relation.*

Another possible extension is a *demonic refinement algebra with domain*, which introduces a infinite iteration operator ${}^\omega$ and a possible finite operator ${}^\infty$. Those are related by the equation $x^\infty = x^* + x^\omega \cdot 0$. In this algebra, the right annihilation law $x \cdot 0 = 0$ does not hold. It is useful for a total correctness setting, where termination is considered.

Following the same approach of §7.5, a *modal separation semiring* can be defined, yielding a modal box operator by De Morgan duality. It is then straightforward to derive the locality counterpart for boxes.

Lemma 9.12. *Let S be a modal separation semiring, then for all $x \in S$ and $p, q \in d(S)$, we have*

$$[x]p * q \leq [x](p * q).$$

Proof. By reflexivity, we have $[x]p \leq [x]p$. Hence,

$$\begin{aligned}[x]p \leq [x]p \Leftrightarrow & \langle x\rangle[x]p \leq p \\ \Leftrightarrow & \langle x\rangle[x]p * q \leq p * q \\ \Rightarrow & \langle x\rangle([x]p * q) \leq p * q \\ \Leftrightarrow & [x]p * q \leq [x](p * q).\end{aligned}$$

The proof uses the Galois connection between the modal operators, monotonicity of $*$ and locality for the diamond operation. □

Finally, we can similarly define a *modal separation Kleene algebra*.

Separation Quantale

A domain separation semiring is expressive enough to derive forward and backward modal operators and propositional separation logic. However it cannot express infinitary sum operation, which can be useful when writing predicates or invariants for proving correctness of a program. Here, we extend domain quantales to a separation quantale, where one can define, explicitly from the algebra, the domain elements. Nonetheless it has the drawback of not being first order, and consequently more difficult to automate.

A *bi-quantale* Q is a structure formed by a quantale $(Q, \leq, \cdot, 1)$ and a commutative quantale $(Q, \leq, *, e)$. It is a domain bi-quantale if the distributivity laws for vectors hold. A *separation quantale* is a domain bi-quantale where the locality axiom

$$d(x \cdot (y * z)) \leq d(x \cdot y) * d(z)$$

hold for all $x, y, z \in Q$. The locality axiom is also known as the small interchange law in concurrent Kleene algebra setting. The non-commutative quantale multiplication models the sequential operation of programs and the commutative operation models separation conjunction.

Lemma 9.13. *Every separation quantale is a modal separation Kleene algebra.*

The definition of a separation quantale is justified by the relational model of programs.

Theorem 9.8. *Let S be a set, H be a partial commutative monoid, Σ be the state space $(S \times H) \cup \{\perp\}$ and L the set of relation satisfying the safety and frame property, then the structure $(L, \subseteq, \circ, **, \emptyset, Id_R, [\mathbf{emp}])$ forms a separation quantale.*

Implementing these modal algebras in Isabelle/HOL is fairly simple. However if one wants to use the facts derived in the algebra directly in the model, many typing conversions are needed and automation is compromised. For that reason, we stick with the backward predicate transformer as our main model and pre-quantale as the underlying algebraic structure.

9.10 Conclusions

This chapter has introduced a novel algebraic approach to separation logic. More than just an implementation of it, it is a complete conceptual reconstruction. The formalisation in Isabelle is small, automatic and modular as possible, the approach provides a minimalist setting from which useful program transformation can be executed and powerful verification conditions and refinement laws can be derived. It was strongly inspired by abstract separation logic [30] and the logic of bunched implications [92], but a different combination of simplicity and mathematical abstraction were achieved. In contrast to the latter, power series were used instead of higher categories, which yield a simpler and generic lifting constructions throughout the approach. In particular, the notion of separating conjunction as convolution over resource algebras appears a more convincing and natural definition.

The chapter has also presented two different semantics for **while** programs with pointers and proved that they were equivalent up to the validity of Hoare triples. The backward predicate transformer is however much simpler to reason with and automate in Isabelle/HOL. The inferences of propositional separation logic and its refinement laws counterpart were derived in this model. The chapter has also extended dynamic algebras to accommodate separating conjunction. The facts derived in these algebras however are not in the correctness tool implemented, since many type conversion are needed and automation is reduced.

Chapter 10

Separation Logic Assertions

This chapter instantiates separation logic assertions with the heaplet model and proposes tactics in Isabelle/HOL to reason about these assertions.

10.1 Assertions over Heaplets

In the previous chapter, we have constructed assertions for separation logic by using convolution, which are independent of any model. In this section, we present the *singleton heap* predicate, $e \mapsto e'$, which is built as a relation of expressions and is used to make statements about the contents of heap cells. The assertion is presented in the heaplet model, since it is difficult to define (or to axiomatise) in a generic fashion. An incomplete axiom schema was given by Reynolds [101]. We write **emp** for the quantale unit in this model.

The predicate $e \mapsto e'$ asserts that the heap contains a single cell at address e, which has the value e', *i.e.*,

$$e \mapsto e' = \{(s,h) \mid dom(h) = \{e(s)\} \wedge h(e(s)) = e'(s)\}.$$

Note that e and e' are expressions evaluated only over the store. We use the abbreviation $e \mapsto -$ to assert that the value in the address e is arbitrary.

A *points-to* assertion, $e \hookrightarrow e'$, can also be defined using the singleton heap predicate. It asserts that the address e points to the value e' in the heap.

$$e \hookrightarrow e' = (e \mapsto e') * \top.$$

As another more complex abbreviation, we define the *doublet heap* predicate, and its points-to counterpart.

$$\begin{aligned} e \mapsto e_1, e_2 &= (e \mapsto e_1) * (e+1 \mapsto e_2) \\ e \hookrightarrow e_1, e_2 &= (e \hookrightarrow e_1) * (e+1 \hookrightarrow e_2) \\ &= (e \mapsto e_1) * (e+1 \mapsto e_2) * \top \end{aligned}$$

The extension to varying length heap predicates is straightforward.

Lemma 10.1. *Points-to $\hookrightarrow$ predicates are local assertions.*

Pure Assertions. An assertion is said to be *pure* if, for any store, it is independent of the heap. In an pure assertion, separating conjunction is degenerated to simple conjunctions, and the magic wand becomes implication. Dang [35] gives an algebraic characterization of pure assertions on a quantale. An element p is pure if and only if for all q and r, the following holds

$$p * \top \leq p,$$
$$p \sqcap (q * r) \leq (p \sqcap q) * (p \sqcap r).$$

That is, p is local and its meet distributes over separating conjunction.

Lemma 10.2. *Let p_1 and p_2 be pure assertions, then the following hold.*

a. $p_1 \sqcap p_2 = p_1 * p_2$,

b. $(p_1 \sqcap q) * r \leq p_1 \sqcap (q * r)$,

c. $p_1 \mathrel{-\!\!*} q \leq p_1 \to q$,

d. $p_1 \mathrel{-\!\!*} p_2 = p_1 \to p_2$,

where $p \to q$ is defined as $\bar{p} + q$.

Lemma 10.3. *Pure assertions do not contain* **emp**, $\mapsto$, *or* $\hookrightarrow$.

Precise Assertions [30]. An assertion p is *precise* if and only if, for all s and h, there exists at most one h' such that $dom(h') \subseteq dom(h)$ and $(s, h') \in p$. Intuitively, it says that for each state with heap h, a precise assertion unambiguously specifies the part of the heap h that is concerned. The property can be characterised algebraically, an assertion p is precise if it distributes over meets of arbitrary assertions.

$$p * (q \sqcap r) = (p * q) \sqcap (p * r)$$

Lemma 10.4. *Let p and q be precise assertions, r be an arbitrary assertion and b a pure assertion. Then $p \sqcap r$, $p * q$ and $(b \sqcap p) + (\bar{b} \sqcap q)$ are precise.*

Precise assertions were used by Bornat [25] to express spatial separation in traditional Hoare logic and by O'Hearn *et al.* [93] to prove completeness of their framework.

Exact Assertions. In his proof of Schorr-Wait graph marking algorithm by so-called *local reasoning in BI pointer logic*, Yang [120] identifies a class of *strictly exact* assertions, where P is strictly exact if and only if, for all s, h and h',

$$(s,h) \in P \wedge (s,h') \in P \Rightarrow h = h'.$$

When p is strictly exact, the following hold

$$(P * \top) \sqcap Q \leq P * (P \mathrel{-\!\!*} Q).$$

Reynolds [101] further defines a broader class, the domain-exact assertions. An assertion P is *domain-exact* if and only if, for all s, h and h',

$$(s,h) \in P \wedge (s,h') \in P \Rightarrow dom(h) = dom(h').$$

Clearly, a strictly exact assertion is domain exact. When P is domain exact, meet of arbitrary assertions distributes over P.

$$P * (Q \sqcap R) = (P * Q) \sqcap (P * R)$$
$$P * \bigsqcap_{i \in I} Q_i = \bigsqcap_{i \in I} (P * Q_i)$$

Lemma 10.5. *Precise predicates are (domain and strictly) exact assertions. Moreover, the singleton heap $e \mapsto e'$ is precise.*

Many assertions require an empty heap, that is, they are of the form $p \sqcap \mathbf{emp}$, where P is pure. Following [31], we define the operator

$$\langle P \rangle = \{(s,h).\ s \in P\} \sqcap \mathbf{emp},$$

which lifts a store predicate to a separation assertion in which the heap is empty.

Lemma 10.6. *Let P be a pure predicate, then $\langle P \rangle$ is precise and*

$$\langle P \rangle \sqcap Q = \langle P \rangle * Q,$$

for any assertion Q.

10.2 Tactics for Separation Logic

This section extends Isabelle tactics *simp*, *safe* and *auto*, which are excellent for classical predicate logic, to be able to reason about assertions of separation logic. These tactics were mainly built upon [31], which creates practical tactics for Coq in order to verify C programs, and upon [3], which specifies generic tactics for LCF-style proof assistants.

A *spatial fragment* P' of an assertion R is syntactically defined as a part of R which does not have any conjunction and disjunction construct. For instance, $(e \mapsto e') * \top$ and **emp** are spatial fragments of $(e \mapsto e') * \top \sqcap \mathbf{emp} \sqcap (P \sqcup Q)$. A spatial fragment P is *normalised* if it has the form of $\exists x_1, ..., x_2.\ P'$, where P' satisfies:

1. it does not contain any existential quantification;
2. all the separation conjunctions are associated to the right;
3. if $P' \neq \mathbf{emp}$, then it does not contain **emp**;
4. it has at most one $\top$ on the right side;
5. it has at most one $\langle P'' \rangle$ on the left side.

The first, and most basic tactic developed, is called *sep-simp*, which is responsible for normalising separation fragments and simplifying assertions by unfolding definitions, such as doublet heap and points-to predicates, and by removing redundant $\top$ and **emp**. New predicate definitions can be added to the simplifier by writing [*sep-simp*] after a lemma in Isabelle. For instance, let i, j, and k be natural values, then applying *sep-simp* to the predicate

$$(\exists x.\ (x \mapsto i) * \top * \langle x = j \rangle * \mathbf{emp}) * (\exists x.\ (x \hookrightarrow j) * \top) * \mathbf{emp} * (k \mapsto i, j) * \langle i = j \rangle$$

yields the predicate

$$\exists x\ y.\ \langle i = j \sqcap x = j \rangle * (x \mapsto i * (y \mapsto j * (k \mapsto i * ((k + 1 \mapsto j) * \top)))).$$

We have explicitly written the predicate with parenthesis to show that all the separation conjunctions are associated to the right.

The goal of the tactics *sep-safe* and *sep-auto* is to prove $P \leq Q$, where p and q are conjunctions of spatial fragments. The tactic *sep-safe* only uses safe introduction and elimination rules, whereas *sep-auto* also tries rules that might put the predicate in an unprovable state. The tactic begins by applying *sep-simp* to normalise P and Q, then it transforms the goal into

$$(s, h) \in P \Longrightarrow (s, h) \in Q.$$

```
method sep-simp =
  (subst sep-simp | subst (asm) sep-simp | subst spred | subst (asm) spred)+

method sep-safe-split =
    ((rule Set.IntI)+)?;
      (
        ((erule Set.IntE)+)?,
        ((erule pred-exE)+)?,
        ((erule spE)+)?,
        sep-same?,
        (sep-simp, sep-safe-split)?,
        (assumption | rule HOL.refl)?
      )

method sep-split =
    sep-safe-split;
    ((rule pred-exI)+)?,
    ((rule spI)+)?;
    simp?;
    (
      (sep-same | sep-simp, sep-split)?,
      (assumption | rule HOL.refl)?
    )

method sep-safe =
    sep-simp?,
    (rule subrelI)?,
    sep-safe-split?;
    (((erule sep-safe-elim)+, force); sep-safe-split?)?;
    sep-same?;
    (assumption | rule HOL.refl)?

method sep-auto =
    sep-simp?,
    (rule subrelI)?,
    sep-split?;
    ((erule sep-elim | rule sep-intro)+; simp?, sep-split?)?;
    sep-same?;
    (assumption | rule HOL.refl)?;
    auto?
```

Figure 10.1: Tactics for separation logic

Next, the tactic tries to separate the spatial fragments by eliminating conjunctions according to

lemma *infI*: $(s,\ h) \in P \Longrightarrow (s,\ h) \in Q \Longrightarrow (s,\ h) \in P \sqcap Q$

lemma *infE*: $(s,\ h) \in P \sqcap Q \Longrightarrow ((s,\ h) \in P \Longrightarrow (s,\ h) \in Q \Longrightarrow G) \Longrightarrow G$,

which yields subgoals of the form

$$(s,h) \in P_1 \Longrightarrow ... \Longrightarrow (s,h) \in P_n \Longrightarrow (s,h) \in Q_i.$$

Every P_i is a normalised spacial fragment, therefore if it contains any existential quantification, we can apply the elimination rule

lemma *pred-exE*: $(s,\ h) \in (\exists x.\ P\ x) \Longrightarrow (\bigwedge x.\ (s,\ h) \in P\ x \Longrightarrow Q) \Longrightarrow Q$,

which yields

$$(s,h) \in P'_1 \Longrightarrow ... \Longrightarrow (s,h) \in P'_n \Longrightarrow (s,h) \in Q_i,$$

where P'_i does not contain any $\exists$. The assertions P'_i might also contain a prefix $\langle P'' \rangle$, in which case, it can be eliminated by

lemma *spE*: $(s,\ h) \in <P> * Q \Longrightarrow (s \in P \Longrightarrow (s,\ h) \in Q \Longrightarrow R) \Longrightarrow R$.

Note that P'' depends only on the store s, this is one of the main reasons why we split the subset relation $P \leq Q$ into a set membership implication. The second reason is heap reduction. The heap is reduced by monotonicity of separating conjunction, that is, if one of the premises has the same predicate as the conclusion, we apply the elimination rule

lemma: $(s,\ h) \in P * Q \Longrightarrow (\bigwedge h'.\ (s,\ h') \in Q \Longrightarrow (s,\ h') \in R) \Longrightarrow (s,\ h) \in P * R$,

where it is easy to show that domain of h' is smaller than the one of h. The rule only eliminates if the same parts are prefixes of both premise and conclusion. Since $*$ is associative and commutative, we can also reduce the heap regarding where the same parts are located in the premise and in the conclusion, by applying variants of the previous elimination rule.

At this point, *sep-safe* will call Isabelle's *safe* tactic, which will try to solve all the subgoals generated so far and will present the remaining simplified subgoals to the user.

The tactic *sep-auto* goes beyond this point by applying the introduction rule of existential quantifier

schematic_lemma: $(s,\ h) \in P\ ?x \Longrightarrow (s,\ h) \in (\exists\ x.\ P\ x)$.

The variable $?x$ is an schematic variable [89] and it can be substituted with any term in scope. This type of variable makes the tactic extremely powerful, but if it matches with an undesirable term according to some rule, it can lead to an unprovable state. If the conclusion contains a prefix $\langle - \rangle$, the tactic will apply the introduction rule

lemma *spI*: $s \in P \Longrightarrow (s,\ h) \in Q \Longrightarrow (s,\ h) \in <P> * Q$.

Next, heap reduction is applied again, this time, it tries to match terms on the premises with schematic variables on the conclusion. Finally, it tries to apply elimination and introduction rules added by the user when annotating lemmas with [*sep-elim*] and [*sep-intro*]. Isabelle's *auto* is then called and the user is presented with any remaining subgoals. Figure 10.1 shows the tactics written in Eisbach.

10.3 Conclusions

The chapter has presented a set of practical tactics implemented in Isabelle/HOL to decide (or simplify) separation assertion implications. The assertions use the standard model of separation logic, the so-called heaplet model. They were motivated by Appel's note [3] and built upon Cao *et al.* [31]. The tactics were implemented in Eisbach and are based in the introduction/elimination reasoning inspired by other tactics within Isabelle/HOL. They are therefore easily extensible by lemma annotation, which is a common practice inside the Isabelle proving environment.

Chapter 11

Programs with Pointers

This chapter finishes the integration of separation logic to the verification framework developed in this thesis, by instantiating a concrete store-heap model to the backward predicate transformer semantics developed in the previous chapter. Several examples show our tool at work.

11.1 Program State Integration

This section describes the integration of the data domain layer into our Isabelle tools for program construction and verification. It uses an important Isabelle feature, namely that the mathematical structures formalised in Isabelle are all polymorphic. We can therefore instantiate the abstract algebras for the control flow with various concrete models by soundness proofs, that is, quantales with predicate transformers, predicate transformers with binary relations and functions which update program states. In particular, abstract resource monoids are linked with various concrete models for resources, including store-heap pairs.

Our data domain integration can build on excellent Isabelle libraries and decades-long experience in reasoning with functions and relations, all sorts of data structures and data types. In particular, for program construction and verification with separation logic, Isabelle already provides support for reasoning with pointers and the heap [119]. This is predominantly based on set theory.

Program states in separation logic are store-heap pairs (s, h) enriched with a fault element $\bot$. We call a state *proper* if it is different from $\bot$. Program stores are implemented in Isabelle as records of program variables, each of which has a *retrieve* and an *update* function. On the one hand, this approach is polymorphic and supports variables of any Isabelle type. For instance, Isabelle's built-in list data type and list libraries can be used to reason about list-based programs. On the other hand, Isabelle records are static, which makes it difficult to accom-

modate dynamic features such as variable scoping, as considered in the framing laws of Morgan's refinement calculus. Heaps, as previously mentioned, have been modelled in Isabelle as partial functions on natural numbers [119]; they therefore have type **nat** $\rightarrow$ **nat option**. Since stores are defined and instantiated later, state type is defined in Isabelle as follows.

type_synonym $'a$ *state* = $('a \times (\textbf{nat} \rightarrow \textbf{nat option}))$ **option**

We implement assignments first as functions from proper states to states (a deterministic state transformer),

$$(`x := e) = \lambda(s, h).\ \mathit{Some}\ (x\_\mathit{update}\ s\ (e\ s), h),$$

where $`x$ is a program variable, *x_update* the update function for $`x$, (s, h) a state and e an evaluated expression of the same type as $`x$. Note that assignments do not change or use the heap, therefore they cannot fault. It is a safe command. Classical separation logic offers a *lookup* command, which assigns a value pointed by a heap cell to a program variable.

$$(`x := @e) = \lambda(s, h).\ \begin{cases} \mathit{Some}\ (x\_\mathit{update}\ s\ (h\ (e\ s)), h) & \text{if } e\ s \in \mathit{dom}\ h, \\ \mathit{None} & \text{otherwise.} \end{cases}$$

If the heap does not contain the address e in that state, then the program is trying to reference a null pointer and therefore it faults.

Next, we implement the typical commands for heap manipulation of separation logic. For heap allocation, we use Hilbert's ε operator, where $\varepsilon\ x.\ P\ x$ denotes some x such that $P\ x$ provided it exists. Heap allocation is then programmed as

$$\begin{aligned}(`x := \textbf{cons}\ e) = \lambda(s, h).\ &\textbf{let}\ n = \varepsilon\ m.\ m \notin \mathit{dom}\ h \\ &\textbf{in}\ \mathit{Some}\ (x\_\mathit{update}\ s\ n, h[n \mapsto (e\ s)]),\end{aligned}$$

where $\mathit{dom}\ h$ is the domain of the heap h, expression e is of natural number type, and $h[n \mapsto e]$ maps n to e and is the same as h for all other parameters, namely, $h[n \mapsto e] = \lambda m.\ \textbf{if}\ m = n\ \textbf{then}\ e\ \textbf{else}\ h(m)$. In a similar fashion, we implement deallocation and mutation as

$$(\textbf{dispose}\ e) = \lambda(s, h).\ \begin{cases} \mathit{Some}\ (s, h[(e\ s) := \mathit{None}]) & \text{if } e\ s \in \mathit{dom}\ h, \\ \mathit{None} & \text{otherwise.} \end{cases}$$

$$(@e := e') = \lambda(s, h).\ \begin{cases} \mathit{Some}\ (s, h[(e\ s) \mapsto (e'\ s)]) & \text{if } e\ s \in \mathit{dom}\ h, \\ \mathit{None} & \text{otherwise.} \end{cases}$$

where the expressions e and e' evaluate to natural numbers and $h[e := \mathit{None}]$ removes e from the domain of h.

Following the same lines of §7.4, we lift these atomic commands to predicate transformers by

$$\langle f \rangle : (S \times H) \to (S \times H) \cup \{\bot\}$$
$$\langle f \rangle = \{(s,h) \mid f\ (s,h)\ \mathbf{in}\ Q\},$$

where $x\ \mathbf{in}\ A$ is defined to be equal to $x \in A$ if $x \neq \bot$, and false otherwise. As previously, we generally do not write the lifting brackets explicitly, identifying program pseudocode with predicate transformers to simplify verification notation. It then remains to show that these atomic commands are local and monotonic.

Lemma 11.1. *The 5 atomic commands for separation logic: assignment, lookup, heap allocation, heap deallocation and heap mutation are local and monotone.*

In fact, heap allocation is only local if $\mathit{dom}\ h$ is a finite set, since one needs to be able to always pick a heap location that is not in the domain of h.

With this infrastructure in place we can prove Hoare's assignment rule and Reynolds' local axioms (or O'Hearn's small axioms) for lookup, allocation, deallocation and mutation of separation logic [101] in the concrete heap model. We write $q[e/\text{`}x]$ for the substitution of variable $\text{`}x$ by expression e in q.

Proposition 11.1. *The following inference rules of separation logic are derivable in the store-heap model:*

$$\{q[e/\text{`}x]\}\ \text{`}x := e\ \{q\},$$
$$\{(e \mapsto n) * \langle \text{`}x = m\rangle\}\ \text{`}x := @e\ \{(e[m/\text{`}x] \mapsto n) * \langle \text{`}x = n\rangle\},$$
$$\{\mathbf{emp}\}\ \text{`}x := \mathbf{cons}\ e\ \{\text{`}x \mapsto e\},$$
$$\{e \mapsto -\}\ \mathbf{dispose}\ e\ \{\mathbf{emp}\},$$
$$\{e \mapsto -\}\ @e := e'\ \{e \mapsto e'\}.$$

Variants of these rules can easily be derived. For example, global rules, obtained by a simple application of the frame rule, emphasise that anything in the heap different from the mutated location is left unchanged:

$$\{q[e/\text{`}x] * r\}\ \text{`}x := e\ \{q * r\},$$
$$\{(e \mapsto n) * \langle \text{`}x = m\rangle * r\}\ \text{`}x := @e\ \{(e[m/\text{`}x] \mapsto n) * \langle \text{`}x = n\rangle * r\},$$
$$\{r\}\ \text{`}x := \mathbf{cons}\ e\ \ \{(\text{`}x \mapsto e) * r\},$$
$$\{(e \mapsto -) * r\}\ \ \mathbf{dispose}\ e\ \{r\},$$
$$\{(e \mapsto -) * r\}\ @e := e'\ \{(e \mapsto e') * r\}.$$

Applying the weakening rule and the identity $p * (p \mathrel{-\!\!*} q) \leq q$ yields global rules for backward reasoning, which is more suitable for automation:

$$\{\exists n.\ (e \mapsto n) * ((e \mapsto n) \mathrel{-\!\!*} q[n/\text{`}x]\}\ \text{`}x := @e\ \{q\},$$
$$\{(e \mapsto -) * ((e \mapsto e') \mathrel{-\!\!*} q)\}\ \ @e := e'\ \ \{q\}.$$

An easier global backward rule for lookup can further be derived by applying the identity $(e \hookrightarrow e') \sqcap p \leq (e \mapsto e') * ((e \mapsto e') \mathrel{-\!\!*} p)$.

$$\{\exists n.\ (e \hookrightarrow n) \sqcap q[n/\text{`}x]\}\ \text{`}x := @e\ \{q\}$$

The resulting set of control-flow and data-domain inference rules for separation logic allows us to program the Isabelle proof tactic *hoare* (*cf.* §4.2), which generates verification conditions automatically and eliminates the entire control structure when the invariants of while loops are annotated.

One can also use the assignment rules to derive their refinement counterparts:

$$p \leq q[e/\text{`}x] \Rightarrow [p, q] \sqsubseteq (\text{`}x := e),$$
$$q' \leq q[e/\text{`}x] \Rightarrow [p, q] \sqsubseteq [p, q'];\ (\text{`}x := e),$$
$$p' \leq p[e/\text{`}x] \Rightarrow [p, q] \sqsubseteq (\text{`}x := e);\ [p', q].$$

The second and third laws are called the *following* and *leading* refinement law for assignments [83]. They are useful for program construction. We have derived analogous laws for lookup, heap allocation, deallocation and mutation.

$$[\exists n.\ (e \mapsto n) * p[n/\text{`}x], (e \mapsto \text{`}x) * p] \sqsubseteq \text{`}x := @e$$
$$[p, (\text{`}x \mapsto e) * p] \sqsubseteq \text{`}x := \textbf{cons}\ e$$
$$[(e \mapsto -) * p, p] \sqsubseteq \textbf{dispose}\ e$$
$$[(e \mapsto -) * p, (e \mapsto e') * p] \sqsubseteq @e := e'$$

We have also programmed the tactic *morgan*, which automatically tries to apply all the rules of this refinement calculus in construction steps of pointed programs.

11.2 Linked Lists

To show our approach at work, we present three examples, among them the obligatory correctness proof of the classical in-situ linked-list reversal algorithm. The post-hoc verification of this algorithm in Isabelle has been considered before [119]. However, we follow Reynolds [101], who gave an informal annotated proof, and reconstruct his proof step-by-step in refinement style. As usual for verification with interactive theorem provers, functional specifications are related

to imperative data structures. The former are defined recursively in functional programming style and are hence amenable to proof by induction. Such detailed functional specifications of data structures used in separation logic are usually not amenable to pure first-order reasoning and are therefore beyond the scope of first-order tools such as SMT solvers.

Following Reynolds, we define a predicate *list_seg xs i j* that indicates whether a heap, starting from position i, contains a linked list segment ending in a position j, represented by a functional list xs.

$$\begin{aligned} &\mathit{list\_seg}\ [\,]\ i\ j &&= (i = j) \wedge \mathbf{emp},\\ &\mathit{list\_seg}\ (x\#xs)\ i\ j &&= \exists k.\ (i \mapsto x, k) * \mathit{list\_seg}\ xs\ k\ j. \end{aligned}$$

We use the doublet heap predicate $(i \mapsto x, k)$ to indicate that i points to the cells containing the value x and the address k of the next list block.

A linked list is then a segment ending into a null pointer. Formally,

$$\mathit{llist}\ xs\ i = (\mathit{list\_seg}\ xs\ i\ 0) \sqcap \mathbf{null},$$

where **null** simply states that the value zero is used as a null pointer, that is, the value 0 is not in the heap domain, $\mathbf{null} = \{(s, h) \mid 0 \notin \mathit{dom}\ h\}$. Note that this value is arbitrary.

We enrich the *sep-simp* tactic with the following simplification rules:

$$\begin{aligned} \mathit{llist}\ [\,]\ i &= \langle i = 0\rangle\\ \mathit{llist}\ [\,]\ 0 &= \mathbf{emp}\\ \mathit{llist}\ (x\#xs)\ i &= \langle i \neq 0\rangle * (\exists k.\ (i \mapsto x, k) * \mathit{llist}\ xs\ k). \end{aligned}$$

Finally, we add the following elimination rules to *sep-auto*:

$$(s, h) \in \mathit{llist}\ x\ i * R \Longrightarrow i\ s = 0 \Longrightarrow (x = [\,] \Longrightarrow (s, h) \in R \Longrightarrow Q) \Longrightarrow Q$$
$$(s, h) \in \mathit{llist}\ x\ i * R \Longrightarrow i\ s \neq 0 \Longrightarrow ((s, h) \in \mathit{llist}\ (\mathit{hd}\ x)\#(\mathit{tl}\ x)\ i * R \Longrightarrow Q) \Longrightarrow Q$$

The first rule says that in a state (s, h) where $i\ s = 0$, the predicate *llist x i* simplifies to **emp** and can be eliminated. Otherwise, x can be split into head and tail in order to be further simplified.

11.3 Examples

Our first example, though somewhat artificial, explicitly demonstrates the use of the frame rule. The following Isabelle code fragment shows the Hoare triple used for verification.

$$\{\ x \mapsto \text{-},\ j\ *\ \mathit{llist}\ \mathit{as}\ j\ \}\ @x := a\ \{\ x \mapsto a,\ j\ *\ \mathit{llist}\ \mathit{as}\ j\ \}$$

Calling the *hoare* tactic for verification condition generation was sufficient for proving the correctness of this simple example automatically. Internally, the frame and the mutation rule have been applied.

In the next example, we show how to deallocate the first cell of a list segment. Figure 11.1 shows the algorithm written in Isabelle. It uses lookup, the most complex command in our language, and heap deallocation. First, the address of the second cell of the list ($`i + 1$) is saved into the variable $`k$. Second, the heap cells with address $`i$ and ($`i+1$) are deallocated. Finally, $`i$ gets the address stored in $`k$. Calling the *hoare* tactic, the following condition is generated:

$$\mathit{list\_seg}\ (x\ \#\ xs)\ i\ j \leq \exists x.\ (i \mapsto -, x) * \mathit{list\_seg}\ xs\ x\ j.$$

When adding the definition of *list_seg* to *sep-simp*, the condition can be automatically proved by calling *sep-auto*.

lemma *llist-seg-dealloc*: $\vdash$ {‖ *list_seg* $(x\ \#\ xs)$ $`i$ $`j$ ‖}
  $`k := @(`i + 1)$;
  dispose $`i$;
  dispose $(`i + 1)$;
  $`i := `k$
  {‖ *list_seg* xs $`i$ $`j$ ‖}
by *hoare sep-auto+*

Figure 11.1: Deallocation of the first element in a list segment

To deallocate the whole list segment, one can loop until the variable $`i$ is equal to $`j$, or equal to 0 if the segment is a full linked list. In Figure 11.2, we show the complete verification of an algorithm for linked list deallocation in a post-hoc fashion. It has been annotated in the standard way with a precondition, a postcondition and a loop invariant. The latter simply states that there exists a linked list xs starting from the pointer $`i$ while $`i$ is not equal to 0. Calling the *hoare* tactic generates three verification conditions for the data domain, which are easily discharged by *sep-auto*.

11.4 Constructing a Linked List Reversal Algorithm.

In this section, we reconstruct Reynolds' classical proof relative to the standard recursive function *rev* for functional list reversal. The initial specification statement is

$$[\mathit{llist}\ xs_0\ `i,\ \mathit{llist}\ (\mathit{rev}\ xs_0)\ `j],$$

```
lemma llist-dealloc: ⊢ {| llist xs 'i |}
  while 'i ≠ 0
  inv {| ∃xs. list xs 'i |}
  do
    'j := 'i;
    'i := @('i + 1);
    dispose 'j;
    dispose ('j + 1)
  od
  {| emp |}
by hoare sep-auto+
```

Figure 11.2: Linked list deallocation algorithm

where xs_0 is the input list and $`i$ and $`j$ are pointers to the head of the list on the heap.

The main idea behind Reynolds' proof is to split the heap into two lists, initially xs_0 and an empty list, and then iteratively swing the pointer of the first element of the first list to the second list. The full proof is shown in Figure 11.3; we now explain its details.

In (1), we strengthen the precondition, splitting the heap into two lists xs and ys, and inserting a variable $`j$ initially assigned to 0. The equation

$$(rev\ xs_0) = (rev\ xs)\ @\ ys$$

then holds for these lists, where @ denotes the append operation on linked lists. Justifying this step in Isabelle requires calling the *morgan* tactic from §9.7, which applies the leading law for assignment. This obliges us to prove that the lists xs and ys *de facto* exist, which is discharged automatically by calling *sep-auto*. In fact, all the 8 proof steps in our construction are essentially automatic: they only require calling *morgan* followed by *sep-safe* or *sep-auto* tactics.

The new precondition generated then becomes the loop invariant of the algorithm. It allows us to refine our specification statement to a while loop in step (2), where we iterate $`i$ until it becomes 0. Calling the *morgan* tactic applies the while law for refinement. From step (3) to (8), we refine the body of the while loop and do not display the outer part of the program.

Because now $`i \neq 0$, the list $xs$ has at least an element $a$. We can thus expand the definition of *llist* in step (3). Next, we assign the value pointed to by $`i + 1$ to $`k$—our first list now starts at $`k$ and $`i$ points to $[a, `k]$.

Step (5) performs a mutation on the heap, changing the cell $`i + 1$ to $`j$; consequently $`i$ now points to $[a, `j]$. Because $*$ is commutative, we can strengthen the precondition accordingly. We now work backwards, folding the definition of

$$[\![\ \mathit{llist}\ \mathit{xs}\ {}^{\backprime}i_0,\ \mathit{llist}\ (\mathit{rev}\ \mathit{xs}_0)\ {}^{\backprime}j\ ]\!]$$
$$\sqsubseteq \tag{1}$$
$${}^{\backprime}j := 0;$$
$$[\![\ \exists \mathit{xs}\ \mathit{ys}.\ \mathit{llist}\ \mathit{xs}\ {}^{\backprime}i * \mathit{llist}\ \mathit{ys}\ {}^{\backprime}j * \langle \mathit{rev}\ \mathit{xs}_0 = (\mathit{rev}\ \mathit{xs}) \mathbin{@} \mathit{ys}\rangle,\ \mathit{llist}\ (\mathit{rev}\ \mathit{xs}_0)\ {}^{\backprime}j\ ]\!]$$
$$\sqsubseteq \tag{2}$$
$${}^{\backprime}j := 0;$$
$$\textbf{while}\ {}^{\backprime}i \neq 0\ \textbf{do}$$
$$[\![\ \exists \mathit{xs}\ \mathit{ys}.\ \mathit{llist}\ \mathit{xs}\ {}^{\backprime}i * \mathit{llist}\ \mathit{ys}\ {}^{\backprime}j * \langle \mathit{rev}\ \mathit{xs}_0 = (\mathit{rev}\ \mathit{xs}) \mathbin{@} \mathit{ys}\rangle * \langle {}^{\backprime}i \neq 0\rangle,$$
$$\exists \mathit{xs}\ \mathit{ys}.\ \mathit{llist}\ \mathit{xs}\ {}^{\backprime}i * \mathit{llist}\ \mathit{ys}\ {}^{\backprime}j * \langle \mathit{rev}\ \mathit{xs}_0) = (\mathit{rev}\ \mathit{xs}) \mathbin{@} \mathit{ys}\rangle\ ]\!]$$
$$\textbf{od}$$

$$[\![\ \exists \mathit{xs}\ \mathit{ys}.\ \mathit{llist}\ \mathit{xs}\ {}^{\backprime}i * \mathit{llist}\ \mathit{ys}\ {}^{\backprime}j * \langle \mathit{rev}\ \mathit{xs}_0 = (\mathit{rev}\ \mathit{xs}) \mathbin{@} \mathit{ys}\rangle * \langle {}^{\backprime}i \neq 0\rangle,$$
$$\exists \mathit{xs}\ \mathit{ys}.\ \mathit{llist}\ \mathit{xs}\ {}^{\backprime}i * \mathit{llist}\ \mathit{ys}\ {}^{\backprime}j * \langle \mathit{rev}\ \mathit{xs}_0) = (\mathit{rev}\ \mathit{xs}) \mathbin{@} \mathit{ys}\rangle\ ]\!]$$
$$\sqsubseteq \tag{3}$$
$$[\![\ \exists \mathit{xs}\ \mathit{ys}\ a\ k.\ ({}^{\backprime}i \mapsto a,\ k) * \mathit{llist}\ \mathit{xs}\ k * \mathit{llist}\ \mathit{ys}\ {}^{\backprime}j * \langle \mathit{rev}\ \mathit{xs}_0 = (\mathit{rev}\ a{\#}\mathit{xs}) \mathbin{@} \mathit{ys} \sqcap {}^{\backprime}i \neq 0\rangle,$$
$$\exists \mathit{xs}\ \mathit{ys}.\ \mathit{llist}\ \mathit{xs}\ {}^{\backprime}i * \mathit{llist}\ \mathit{ys}\ {}^{\backprime}j * \langle \mathit{rev}\ \mathit{xs}_0) = (\mathit{rev}\ \mathit{xs}) \mathbin{@} \mathit{ys}\rangle\ ]\!]$$
$$\sqsubseteq \tag{4}$$
$${}^{\backprime}k := @({}^{\backprime}i + 1);$$
$$[\![\ \exists \mathit{xs}\ \mathit{ys}\ a.\ ({}^{\backprime}i \mapsto a,\ {}^{\backprime}k) * \mathit{llist}\ \mathit{xs}\ {}^{\backprime}k * \mathit{llist}\ \mathit{ys}\ {}^{\backprime}j * \langle \mathit{rev}\ \mathit{xs}_0 = (\mathit{rev}\ a{\#}\mathit{xs}) \mathbin{@} \mathit{ys} \sqcap {}^{\backprime}i \neq 0\rangle,$$
$$\exists \mathit{xs}\ \mathit{ys}.\ \mathit{llist}\ \mathit{xs}\ {}^{\backprime}i * \mathit{llist}\ \mathit{ys}\ {}^{\backprime}j * \langle \mathit{rev}\ \mathit{xs}_0) = (\mathit{rev}\ \mathit{xs}) \mathbin{@} \mathit{ys}\rangle\ ]\!]$$
$$\sqsubseteq \tag{5}$$
$${}^{\backprime}k := @({}^{\backprime}i + 1);$$
$$@({}^{\backprime}i + 1) := {}^{\backprime}j;$$
$$[\![\ \exists \mathit{xs}\ \mathit{ys}\ a.\ ({}^{\backprime}i \mapsto a,\ {}^{\backprime}j) * \mathit{llist}\ \mathit{xs}\ {}^{\backprime}k * \mathit{llist}\ \mathit{ys}\ {}^{\backprime}j * \langle \mathit{rev}\ \mathit{xs}_0 = (\mathit{rev}\ a{\#}\mathit{xs}) \mathbin{@} \mathit{ys} \sqcap {}^{\backprime}i \neq 0\rangle,$$
$$\exists \mathit{xs}\ \mathit{ys}.\ \mathit{llist}\ \mathit{xs}\ {}^{\backprime}i * \mathit{llist}\ \mathit{ys}\ {}^{\backprime}j * \langle \mathit{rev}\ \mathit{xs}_0) = (\mathit{rev}\ \mathit{xs}) \mathbin{@} \mathit{ys}\rangle\ ]\!]$$
$$\sqsubseteq \tag{6}$$
$${}^{\backprime}k := @({}^{\backprime}i + 1);$$
$$@({}^{\backprime}i + 1) := {}^{\backprime}j;$$
$$[\![\ \exists \mathit{xs}\ \mathit{ys}.\ \mathit{llist}\ \mathit{xs}\ {}^{\backprime}k * \mathit{llist}\ \mathit{ys}\ {}^{\backprime}i * \langle \mathit{rev}\ \mathit{xs}_0 = (\mathit{rev}\ \mathit{xs}) \mathbin{@} \mathit{ys} \sqcap {}^{\backprime}i \neq 0\rangle,$$
$$\exists \mathit{xs}\ \mathit{ys}.\ \mathit{llist}\ \mathit{xs}\ {}^{\backprime}i * \mathit{llist}\ \mathit{ys}\ {}^{\backprime}j * \langle \mathit{rev}\ \mathit{xs}_0) = (\mathit{rev}\ \mathit{xs}) \mathbin{@} \mathit{ys}\rangle\ ]\!]$$
$$\sqsubseteq \tag{7}$$
$${}^{\backprime}k := @({}^{\backprime}i + 1);$$
$$@({}^{\backprime}i + 1) := {}^{\backprime}j;$$
$${}^{\backprime}j := {}^{\backprime}i;$$
$$[\![\ \exists \mathit{xs}\ \mathit{ys}.\ \mathit{llist}\ \mathit{xs}\ {}^{\backprime}k * \mathit{llist}\ \mathit{ys}\ {}^{\backprime}j * \langle \mathit{rev}\ \mathit{xs}_0 = (\mathit{rev}\ \mathit{xs}) \mathbin{@} \mathit{ys} \sqcap {}^{\backprime}j \neq 0\rangle,$$
$$\exists \mathit{xs}\ \mathit{ys}.\ \mathit{llist}\ \mathit{xs}\ {}^{\backprime}i * \mathit{llist}\ \mathit{ys}\ {}^{\backprime}j * \langle \mathit{rev}\ \mathit{xs}_0) = (\mathit{rev}\ \mathit{xs}) \mathbin{@} \mathit{ys}\rangle\ ]\!]$$
$$\sqsubseteq \tag{8}$$
$${}^{\backprime}k := @({}^{\backprime}i + 1);$$
$$@({}^{\backprime}i + 1) := {}^{\backprime}j;$$
$${}^{\backprime}j := {}^{\backprime}i;$$
$${}^{\backprime}i := {}^{\backprime}k$$

Figure 11.3: *In situ* linked list reversal algorithm by refinement. The first block shows the refinement up to the introduction of the while-loop. The second block shows the refinement of the body of the loop.

$$
\begin{array}{l}
[\![\ \mathit{llist}\ \mathit{xs}_0\ \ `i\ ,\ \mathit{llist}\ \ (\mathit{rev}\ \mathit{xs}_0)\ \ `j\ ]\!] \\
\quad \sqsubseteq \\
`j := 0; \\
\textbf{while}\ `i \neq 0\ \textbf{do} \\
\quad `k := @(`i + 1); \\
\quad @(`i + 1) := `j; \\
\quad `j := `i; \\
\quad `i := `k \\
\textbf{od}
\end{array}
$$

Figure 11.4: *In situ* linked list reversal algorithm

llist in and removing the existential of a in step (6). Last, to establish the invariant, we only need to swap the pointers $`j$ to $`i$ and $`i$ to $`k$ as in steps (7) and (8). The resulting algorithm is highlighted in Figure 11.3 and shown in Figure 11.4.

In addition, we have performed a post-hoc verification of the list reversal algorithm using two different approaches. The first, previously taken by Weber [119], uses Reynolds' list predicate, as we have used it in the above refinement proof. The verification can be done automatically by *hoare* and *sep-auto*. The second follows Nipkow in using separating conjunction in the pre- and postcondition, but not in the definition of the list predicate. Since our approach is modular with respect to the underlying data model, it was straightforward to replay Nipkow's proof in our setting.

11.5 Conclusions

This chapter integrated a concrete store-heap model to the backward predicate semantics. Inference rules and refinement laws of the five atomic statements of separation logic were proven sound with respect to this model. Finally, we have applied the tool to several examples.

In summary, the approach supports the program construction and verification of pointer-based programs with separation logic, but larger case studies are desirable to assess the performance of the tool. Apart from soundness proofs and the certification of the tool itself, the algebraic approach has predominantly been used in the derivation and implementation of tactics for program construction and verification. In the end, tactics for the data level, such as *sep-auto* and *sep-safe*, dramatically increased tool automation.

Chapter 12

Conclusion

The tool prototypes presented in this thesis has so far allowed us to verify several program examples with a relatively high degree of automation. Although these examples were simple, they have interesting and varied properties, such as recursivity, nondeterminism and pointers. So far the tools have been shown to be useful for educational and research purposes; Prof. Georg Struth has used them in his lectures for *Software and Hardware Verification* at the University of Sheffield. However, extensions and optimisations beyond the mere proof of concept are desirable, including the development of more sophisticated proof tactics, and the integration of tools and techniques for automatic data-level reasoning in Sledgehammer style. There are yet many verification issues that this thesis has not addressed. These include:

- termination – proof that a program will always terminate;
- abrupt termination commands, such as *break*, *return* and *continue* in C;
- exceptions;
- mutually recursive procedures;
- local name for a local variable – our tools support local variables, but these need to have their names globally defined;
- variable aliases;
- variable framing refinement laws;
- pointers to procedures; and
- concurrency – although we have a tool able to verify simple concurrent algorithm by an interleaving model, it does not consider liveness, fairness, starvation and weak memory models.

12.1 Summary

This thesis has covered three main contributions.

Formalisation of algebraic structures. Various algebraic structures for reasoning about imperative programs were formalised in Isabelle/HOL and made available for the theorem proving community through the Archive of Formal Proofs [9, 55, 54].

Principled approach for correctness tools. A coherent approach based on algebraic principles to the design and development of program verification and refinement tools, which are correct by construction, was implemented within the Isabelle/HOL theorem proving environment. It aimed at a clean separation of concerns between the control flow and the data domain level of programs and focussed on developing a lightweight algebraic layer from which verification conditions or transformation and refinement laws can be developed by simple equational reasoning. Variants were also proposed, each with a different underlying algebraic structure.

Correctness tool for separation logic. The novel algebraic approach for separation logic, proposed by Dongol, Hayes and Struth [44] and developed in this thesis, yields a clean design for verification and construction tools for programs with pointers. These design choices allow us to use power series, quantales and generic lifting constructions throughout the approach, which leads indeed to a very small and highly automated Isabelle implementation. A particular feature of this approach is the view on separating conjunction as a notion of convolution over resources. The tool has a high level of automation with powerful tactics for the control and data levels.

12.2 Future Work

Concurrent Separation Logic The approach to separation logic in this thesis suggests a new lifting construct, in which a new operation $\|$ can be defined as

$$(F \| G)\ Q = \sum_{Q \leq Q_1 * Q_2} F\ Q_1 * G\ Q_2,$$

where F and G are backwards predicate transformers and Q is a predicate. In fact, Hoare *et al.* [63] uses a model where this equation holds. However, more investigation is needed to certify that this model is indeed good for concurrent

programs. Perhaps one can find a bisimulation between this model and trace semantics.

Algebraic Structures. At least two questions remain for future work. First, the question of whether or not a domain quantale forms a relational algebra is still open. Second and more importantly, separation quantales define an operation $*$ over the whole carrier set; it would be interesting to know if a model exists where $*$ is a parallel operator for generic elements and a separating conjunction for subidentities. More research is needed nonetheless to investigate this intrinsic relationship between both operations.

Correctness Tools. The development of fully-fledged correctness tools instead of prototypes with a better interface and other integrated techniques is desirable. Other opportunities for future work lie in the integration of categorical approaches to data type constructions [53, 84], and the consolidation with Preoteasa's approach to predicate transformers in Isabelle [98].

Concurrency. The algebraic approach for correctness tools presented in this thesis has been implemented for a rely-guarantee algebra in a concurrent setting [7, 6]. However, it fails to address many interesting issues of concurrency. For instance, the area of weak memory models is being researched intensely and new verification logics are being created. These promising logics, such as GPS [114], have never had an algebraic treatment. In addition, many open questions still remain in the area, such as the development of inference rules for fences, definition of atomicity and refinement under weak consistent models.

Appendix A

Algebraic Structures

This appendix collects the main algebraic axiom systems used in the thesis (cf. Figure A.1).

Definition A.1. *A* join semilattice *is a poset $(L, \leq)$ such that every pair of elements $x, y \in L$ has a least upper bound $x + y$ in L. A join semilattice L is* complete *if every set $X \subseteq L$ has a least upper bound $\sum X$ in L. Alternatively, a join semilattice is a structure $(L, +)$ which satisfies, for all $x, y, z \in L$, the axioms*

$$(x + y) + z = x + (y + z), \tag{A.1}$$
$$x + y = y + x, \tag{A.2}$$
$$x + x = x. \tag{A.3}$$

Definition A.2. *A* lattice *is a poset $(L, \leq)$ such that every pair of elements $x, y \in L$ has a least upper bound $x + y$ and a greatest lower bound $x \sqcap y$ in L. A lattice is* complete *if every set X has a least upper bound $\sum X$ and a greatest lower bound $\prod X$ in L.*

Definition A.3. *A* semiring *is a structure $(S, +, \cdot, 0, 1)$ which satisfies for all $x, y, z \in S$, the axioms*

$$(x + y) + z = x + (y + z), \tag{A.4}$$
$$0 + x = x + 0 = x, \tag{A.5}$$
$$x + y = y + x, \tag{A.6}$$
$$(x \cdot y) \cdot z = x \cdot (y \cdot z), \tag{A.7}$$
$$1 \cdot x = x \cdot 1 = x, \tag{A.8}$$
$$x \cdot (y + z) = x \cdot y + y \cdot z, \tag{A.9}$$
$$0 \cdot x = x \cdot 0 = 0. \tag{A.10}$$

Definition A.4. *A* dioid *is a semiring S which is additively idempotent, that is, for all $x \in S$,*

$$x + x = x. \tag{A.11}$$

Definition A.5. *A* test dioid *is a structure $(S, B, +, \cdot, ^-, 0, 1)$ such that $(S, +, \cdot, 0, 1)$ is a dioid, $B \subseteq S$ and $(B, +, \cdot, ^-, 0, 1)$ is a boolean algebra.*

Definition A.6. *A* Kleene algebra (KA) *is a structure $(K, +, \cdot, ^*, 0, 1)$ such that $(K, +, \cdot, 0, 1)$ is a dioid and the star satisfies the axioms*

$$1 + x^* \cdot x \leq x^*, \tag{A.12}$$

$$1 + x \cdot x^* \leq x^*, \tag{A.13}$$

$$z + y \cdot x \leq y \Rightarrow z \cdot x^* \leq y, \tag{A.14}$$

$$z + x \cdot y \leq y \Rightarrow x^* \cdot z \leq y. \tag{A.15}$$

Definition A.7. Kleene algebras with tests (KAT) *are Kleene algebras which are also test dioids.*

Definition A.8. *A* refinement Kleene algebra with tests (rKAT) *is a* KAT *expanded by the operation $[\_\_, \_\_] : B \times B \to K$ which satisfies*

$$px \leq xq \Leftrightarrow x \leq [p, q]. \tag{A.16}$$

Definition A.9. *A* quantale *is an algebraic structure $(Q, \leq, \cdot, 1)$ such that $(Q, \leq)$ is a complete lattice, $(Q, \cdot, 1)$ a monoid and the following distributivity laws hold*

$$(\sum_{i \in I} x_i) \cdot y = \sum_{i \in I} x_i \cdot y, \tag{A.17}$$

$$x \cdot (\sum_{i \in I} y_i) = \sum_{i \in I} (x \cdot y_i). \tag{A.18}$$

Definition A.10. *A* quantale with tests *is a quantale that is also a test dioid.*

Definition A.11. *A* domain semiring *is a semiring S expanded by a domain operation $d : S \to S$ that satisfies*

$$x + d(x) \cdot x = d(x) \cdot x, \tag{A.19}$$

$$d(x \cdot y) = d(x \cdot d(y)), \tag{A.20}$$

$$d(x) + 1 = 1, \tag{A.21}$$

$$d(0) = 0, \tag{A.22}$$

$$d(x + y) = d(x) + d(y). \tag{A.23}$$

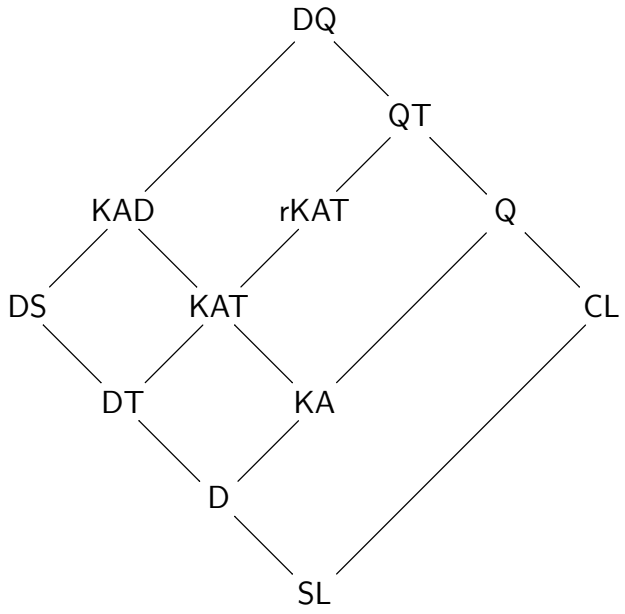

Figure A.1: Class inclusions for join semilattices (SL), dioids (D), dioids with tests (DT), domain semiring (DS), Kleene algebras (KA), Kleene algebras with tests (KAT), complete lattices (CL), refinement Kleene algebra with tests (rKAT), Kleene algebra with domain (KAD), quantales (Q), quantales with tests (QT) and domain quantales (DQ)

Definition A.12. *An* antidomain semiring *is a semiring S expanded by an antidomain operation $a : S \to S$ that satisfies*

$$a(x) \cdot x = 0, \tag{A.24}$$
$$a(x \cdot y) + a(x \cdot d(y)) = a(x \cdot d(y)), \tag{A.25}$$
$$d(x) + a(x) = 1, \tag{A.26}$$

where $d(x) = a(a(x))$.

Definition A.13. Kleene algebras with domain (KAD) *are Kleene algebra which are also domain semirings.*

Definition A.14. *A* domain quantale *is a quantale where the left and right distributivity laws for vectors hold,* i.e.*,*

$$y \cdot (z \sqcap \top \cdot x) = y \cdot z \sqcap \top \cdot x, \tag{A.27}$$
$$(z \sqcap x \cdot \top) \cdot y = z \cdot y \sqcap x \cdot \top. \tag{A.28}$$

Appendix B

Introduction to Isabelle/HOL

This appendix is an introduction to Isabelle/HOL, covering the essentials needed to understand this thesis. For further information on Isabelle/HOL we refer to its excellent standard documentation [89].

Isabelle is an interactive proof assistant with embedded first-order automatic theorem provers, SMT-solvers and counterexample generators, apart from provers, solvers and simplifiers for higher-order logic. As an LCF-style framework it is based on a small logical core to guarantee correctness. All algebras and models implemented are consistent with respect to this core and all theorems proved are correct relative to it. In particular, all proof outputs produced by external theorem provers must be internally reconstructed in order to be accepted. Isabelle has been used to formalise a wide range of mathematical theories and applied in numerous computing applications, including program correctness and verification. Isabelle/HOL, in particular, is based on a typed higher-order logic which supports reasoning with sets, polymorphic data types, inductive definitions and recursive functions.

B.1 Proving in Isabelle

When proving a proposition in Isabelle, the statement is preceded by the keywords **theorem** or **lemma**. There are no differences internally between these keywords, but it depends how the user perceives the importance of the given proposition. Isabelle enters then in a *proof mode*, where the output window shows the *goals* or the *proof obligations*. The proof proceeds by applying *tactics* or *methods* to these. The goals are then reduced to simpler *subgoals* until they become trivial. A tactic can be invoked by the command **apply**. When no goals remains to be proved, the user might exit the proof mode by the keyword **done**. The command **by** is used as an abbreviation for applying a tactic and exiting the proof mode.

lemma *star-cosim:* $z{\cdot}x \leq y{\cdot}z \longrightarrow z{\cdot}x^\star \leq y^\star{\cdot}z$
proof
  assume $z{\cdot}x \leq y{\cdot}z$
  hence $y^\star{\cdot}z{\cdot}x \leq y^\star{\cdot}y{\cdot}z$
    by *(metis mult-isol mult-assoc)*
  also have $... \leq y^\star{\cdot}z$
    by *(metis mult-isor star-1r)*
  finally have $z + y^\star{\cdot}z{\cdot}x \leq y{\cdot}z$
    by *(metis add-lub-var mult-1-left mult-isor star-ref)*
  thus *?thesis*
    by *(metis star-inductr)*
qed

Figure B.1: Example of proof in Isar, *co*-simulation law for Kleene star.

A fully proved lemma can be stored under a given name and henceforth be used by the tactics.

Tactics and tacticals. Tactics are usually based on natural deduction and *resolution*. Simplifiers, such as the tactic *simp*, are based on *rewritting* by substitution of *equals for equals*. Powerful tactics, also known as *tacticals*, combine multiple tactics and are able to automatically prove complicated statements. A extremely useful tactic is *auto*, which combines higher-order resolution and rewritting. However, *auto* might reach an unprovable subgoal, the tactical *safe* instead applies only safe rules. Additionally, *force* works similar than *auto*, but fails if it does not fully prove the subgoal.

Sledgehammer. Isabelle contains a range of built-in tactics, provers and simplifiers in addition to *auto*. In particular, its internal Sledgehammer tool is able to invoke external automated theorem provers and SMT solvers and reconstruct their output internally in order to guarantee trustworthiness. The tool can be invoked by the command **sledgehammer** in the proof mode, or by clicking in the sledgehammer button in the Isabelle IDE. The output window then displays a list of tactics or provers with the necessary arguments able to prove the required subgoal. The user selects a desired tactic, this line is copied to the Isabelle file and the command **sledgehammer** is erased.

Isar. Isabelle also offers different modes of interactive reasoning, notably the proof scripting language *Isar* which supports human-readable proofs. The example in Figure B.1 proves a *co*-simulation law for the Kleene star in the context

of Kleene algebras. The proof is split into simple human-readable steps, which are proved automatically by Sledgehammer and internally verified by the prover metis. The lemma is named *star-cosim* and can be used in future proofs. *metis* is the name of Isabelle's internal theorem prover which reconstructs proofs given by the external provers called by Sledgehammer. Lemmas such as *mult-isol* or *star-ref*, which are used by metis, are not provided by the user, but selected by the Sledgehammer tool according to syntactic criteria and by using machine learning.

B.2 Isabelle Axiomatic Type Class

Algebraic hierarchies like the one presented in Chapter 3 are typically formalised within Isabelle's axiomatic type class infrastructure. As an example, the following type class formalises dioids using the existing class for semirings and expanding it with the idempotency axiom for addition.

class *dioid* = *semiring* +
  assumes *add-idem*: $x + x = x$

The operation of addition used by Isabelle is polymorphic; it has type $\alpha \Rightarrow \alpha \Rightarrow \alpha$. The type class mechanism supports theory expansion and the formalisation of subclass relationships. Theorems proved for reducts or superclasses thus become automatically available in expansions or subclasses. The fact that dioids form a join-semilattice can, for instance, be captured by the following statement.

subclass *(***in** *dioid) join-semilattice*
  by *unfold-locales (auto simp add: add.commute add.left-commute)*

The first line of the statement indicates the proof obligation, which is to prove that the join semilattices form a subclass of dioids. When calling the tactic *unfold-locales*, Isabelle generates all the subgoals necessary to prove this claim. Because a join-semilattice is a set equipped with an associative, commutative and idempotent join operation, these are as follows.

1. $\forall\ x\ y\ z.\ (x + y) + z = x + (y + z)$
2. $\forall\ x\ y.\ x + y = y + x$
3. $\forall\ x.\ x + x = x$

These subgoals are automatically discharged by calling Isabelle's internal tactical *auto* with lemmas named *add.commute* and *add.left-commute* as parameters. These lemmas have been proved in the context of semirings and are available in the type class of *dioid* due its definition by expansion.

An important Isabelle feature is that the mathematical structures formalised are all polymorphic—their elements can have various types. Within this infrastructure, abstract algebras can be linked formally with their models by instantiation or interpretation statements. This, for instance, allows us to verify that relations form a dioid.

interpretation *rel-dioid: dioid (op ∪) (op O)*
by *auto*

B.3 Isabelle Main Library

This thesis extensively uses Isabelle standard library, where types and notations are similar to ML.

Function Type: $\alpha \Rightarrow \beta$**.** Function type, or arrow type, is a basic type in HOL. This can be construct by lambda abstraction, let x be of type α and y of type β, then $\lambda x.\ y$ is of type $\alpha \Rightarrow \beta$. Isabelle offers a composition function $\circ :: (\alpha \Rightarrow \beta) \Rightarrow (\beta \Rightarrow \gamma) \Rightarrow \alpha \Rightarrow \gamma$ and an identity function $id :: \alpha \Rightarrow \alpha$. Function types associate to the right, *i.e.*, $\alpha \Rightarrow \beta \Rightarrow \gamma \equiv \alpha \Rightarrow (\beta \Rightarrow \gamma)$. The notation $f(x := y)$ updates the function f, such that $f\ x = y$.

Product Type: $\alpha \times \beta$**.** Isabelle offers projection functions $fst :: \alpha \times \beta \Rightarrow \alpha$ and $snd :: \alpha \times \beta \Rightarrow \beta$. The construction $Pair :: \alpha \Rightarrow \beta \Rightarrow \alpha \times \beta$ creates a pair (x, y). Additionally, tuple patterns such as (x, y, z) can also be used. Tuples are pairs associated to the right, *i.e.*, $(x, y, z) \equiv (x, (y, z))$.

Sum Type: $\alpha + \beta$**.** The disjoint sum of two types can be manipulated by its projections and constructors. It offers the projections $projl :: \alpha + \beta \Rightarrow \alpha$ and $projr :: \alpha + \beta \Rightarrow \beta$, and the constructors $Inl :: \alpha \Rightarrow \alpha + \beta$ and $Inr :: \beta \Rightarrow \alpha + \beta$.

Option Type: α *option***.** The datatype α *option* = *None* | *Some* α is useful to add an extra element to a particular type, usually a bottom $\bot$ or a top $\top$ element. The function $the :: \alpha\ option \Rightarrow \alpha$ is such that $the\ (Some\ x) = x$. Option types are used in this thesis to model partial functions.

List: α *list***.** Isabelle offers classic methods to reason about functional list. The empty list is [ ] or *Nil*. The constructor *cons* is denotated by # and the operation append by @. The i-th element of a list xs is written $xs\ !\ i$. Isabelle has functions to retrieve the head of a list $hd :: \alpha\ list \Rightarrow \alpha$, its tail $tail :: \alpha\ list \Rightarrow \alpha\ list$, its

last element *last* :: α *list* $\Rightarrow$ α and its length *length* :: α *list* $\Rightarrow$ *nat*. The notation $xs[i := y]$ updates the i-th element of the list xs by y. For more advanced functions, please see Isabelle's manual [89] or its source code.

Sets: α *set*. The notation $\{x.\ P\ x\}$ denotes set comprehension, that is, the set of all elements satisfying the predicate P. Isabelle also offers all the classic set operations such union $A \cup B$, intersection $A \cap B$ and complementation $-A$. The syntax $\{\}$ denotes the empty set and *UNIV* is the universe for a given type. We write $x \in A$ and $A \subseteq B$ for the membership and subset relations. Additionally, the image of a set P under a function f is denoted by $f`A$.

Relations: α *rel*. Relations are defined in Isabelle/HOL as a set of pairs. All the operations for sets are then available for relations. In addition, relation composition is $R\ O\ S = \{(x, z).\ \exists y.\ (x, y) \in R \wedge (y, z) \in S\}$, and the identity relation is $Id = \{(x, x).\ x \in UNIV\}$.

B.4 Eisbach

Isabelle 2015 offers the new proof method language Eisbach [80]. A tactic is created by the keyword **method**, which might include lists of lemmas as arguments declared after **uses**. The syntax used to define a method is essentially the same one when proving a lemma in Isabelle. The operations are similar to the ones for regular expressions, where '|' is choice, ',' is sequential operation, and '+' and '?' are quantifiers indicating *one or more* and *zero or one* respectively. Eisbach has more powerful features not used in this thesis, for a full overview please refer to its manual [80].

Appendix C

Source Code

The implementation of the tools discussed in this thesis is based on the formalisation of algebraic structures available in the Archive of Formal Proofs:

- Kleene algebras [13],
- Kleene algebras with tests and demonic refinement algebras [9],
- Residuated lattices [55],
- Kleene algebras with domain [54].

In addition, the full Isabelle implementation of the correctness tools and the formalisation of refinement Kleene algebras, quantales and partial algebras, together with all the program construction and verification examples presented in this thesis, can be found online:

https://github.com/victorgomes/veritas.

Previous versions of the tools, presented in [8, 10, 11, 43], are also available:

- http://afp.sourceforge.net/entries/KAT_and_DRA.shtml,
- http://www.dcs.shef.ac.uk/~victor/refinement,
- http://www.dcs.shef.ac.uk/~victor/verification,
- https://github.com/victorgomes/sep-logic.

All the facts in this thesis were formally verified, except the ones in §9.9 and the examples of partial algebras and convolution drawn from the literature in §9.2 and §9.3.

The remaining sections of this appendix present the most important parts of the implementation. The formalisation of DRA, KAD and quantales are not presented, since they follow a similar reasoning to the one for Kleene algebras. Similarly, the implementation of the correctness tools for GCL and separation logic is not shown here.

C.1 Kleene Algebras

Diods, or commutative semiring, are implemented from Isabelle's semiring class. A *near-dioid* is a dioid without left distributivity, whereas a *pre-dioid* is a near-dioid with sub-distributivity. This hierarchy of near and pre algebras is preserved for more complex algebraic structures.

class *near-dioid* = *ab-near-semiring* + *plus-ord* +
  assumes *add-idem'* [*simp*]: $x + x = x$

class *pre-dioid* = *near-dioid* +
  assumes *subdistl*: $z \cdot x \leq z \cdot (x + y)$

class *dioid* = *near-dioid* + *semiring*

class *near-dioid-one* = *near-dioid* + *ab-near-semiring-one*

class *pre-dioid-one* = *pre-dioid* + *near-dioid-one*

class *dioid-one* = *dioid* + *near-dioid-one*

class *dioid-one-zerol* = *dioid-one* + *ab-near-semiring-one-zerol*

class *dioid-one-zero* = *dioid-one-zerol* + *ab-near-semiring-one-zero*

The left and the right star induction axiom is considered independently. In addition, the right unfold axiom can be derived in a left Kleene algebra.

class *left-near-kleene-algebra* = *near-dioid-one* + *star-op* +
  assumes *star-unfoldl*: $1 + x \cdot x^\star \leq x^\star$
  and *star-inductl*: $z + x \cdot y \leq y \longrightarrow x^\star \cdot z \leq y$

class *left-pre-kleene-algebra* = *left-near-kleene-algebra* + *pre-dioid-one*
begin

lemma *star-unfoldr*: $1 + x^\star \cdot x \leq x^\star$

by (*metis add-lub star-1r star-ref*)

end

class *left-kleene-algebra* = *left-pre-kleene-algebra* + *dioid-one*

class *left-kleene-algebra-zerol* = *left-kleene-algebra* + *dioid-one-zerol*

class *left-kleene-algebra-zero* = *left-kleene-algebra-zerol* + *dioid-one-zero*

class *kleene-algebra* = *left-kleene-algebra-zero* +
assumes *star-inductr*: $z + y \cdot x \leq y \longrightarrow z \cdot x^\star \leq y$

Before defining Kleene algebra with tests, a dioid with tests is formalised.

class *near-dioid-tests-zerol* = *ab-near-semiring-one-zerol* + *plus-ord* +
fixes *comp-op* :: $'a \Rightarrow 'a$ (*n*- [*90*] *91*)
assumes *test-one*: $n\ n\ 1 = 1$
and *test-mult*: $n\ n\ (n\ n\ x \cdot n\ n\ y) = n\ n\ y \cdot n\ n\ x$
and *test-mult-comp*: $n\ x \cdot n\ n\ x = 0$
and *test-de-morgan*: $n\ x + n\ y = n\ (n\ n\ x \cdot n\ n\ y)$

class *pre-dioid-test-zerol* = *near-dioid-test-zerol-dist* + *pre-dioid*

class *dioid-tests-zerol* = *dioid-one-zerol* + *pre-dioid-test-zerol*

class *dioid-tests* = *dioid-tests-zerol* + *dioid-one-zero*

class *kat* = *kleene-algebra* + *dioid-tests*
context *kat*
begin

We encode validity of Hoare triples and derive the inference rules of propositional Hoare logic, that is, Hoare logic without the assignement rule, in Kleene algebra with tests.

definition *hoare-triple* :: $'a \Rightarrow 'a \Rightarrow 'a \Rightarrow$ *bool* ({|-|}-{|-|}) **where**
$\{|p|\}\ x\ \{|q|\} \equiv p{\cdot}x = p{\cdot}x{\cdot}q \wedge test\ p \wedge test\ q$

lemma *hoare-triple-def-var*: $\{|p|\}\ x\ \{|q|\} = (p{\cdot}x \leq x{\cdot}q \wedge test\ p \wedge test\ q)$
by (*metis hoare-triple-def kat-eq1 kat-eq2*)

lemma *skip-rule*: $test\ p \Longrightarrow \{|p|\}1\{|p|\}$
by (*simp add*: *hoare-triple-def*)

lemma *sequence-rule*: $\{\!|p|\!\}x\{\!|q'|\!\} \Longrightarrow \{\!|q'|\!\}x'\{\!|q|\!\} \Longrightarrow \{\!|p|\!\}x{\cdot}x'\{\!|q|\!\}$
by (*simp add*: *hoare-triple-def*, *metis mult.assoc*)

lemma *conditional-rule*: $[\![\{\!|p{\cdot}b|\!\}x\{\!|q|\!\}; \{\!|p{\cdot}!b|\!\}x'\{\!|q|\!\}; \mathit{test}\ p; \mathit{test}\ b]\!] \Longrightarrow \{\!|p|\!\}b{\cdot}x + !b{\cdot}x'\{\!|q|\!\}$
by (*simp add*: *hoare-triple-def*, *metis mult.assoc distrib-left distrib-right*)

lemma *consequence-rule*: $[\![\mathit{test}\ p;\ p \leq p';\ \{\!|p'|\!\}x\{\!|q'|\!\};\ q' \leq q;\ \mathit{test}\ q]\!] \Longrightarrow \{\!|p|\!\}x\{\!|q|\!\}$
by (*unfold hoare-triple-def*, *metis* (*full-types*) *mult.assoc test-leq-mult-def-var*)

lemma *while-rule-var*: $\{\!|p|\!\}x\{\!|p|\!\} \Longrightarrow \{\!|p|\!\}x^\star\{\!|p|\!\}$
by (*metis hoare-triple-def-var star-sim2*)

lemma *while-rule*: $[\![\mathit{test}\ q;\ \{\!|p{\cdot}q|\!\}x\{\!|p|\!\}]\!] \Longrightarrow \{\!|p|\!\}(q{\cdot}x)^\star{\cdot}!q\{\!|p{\cdot}!q|\!\}$
proof (*unfold hoare-triple-def-var*, *auto*)
assume *assms*: *test p test q* $p{\cdot}q{\cdot}x \leq x{\cdot}p$
hence $q{\cdot}p{\cdot}q{\cdot}x \leq q{\cdot}x{\cdot}p$
by (*metis mult.assoc mult-isol*)
thus $p{\cdot}((q{\cdot}x)^\star{\cdot}!q) \leq (q{\cdot}x)^\star{\cdot}!q{\cdot}(p{\cdot}!q)$
by (*metis assms*(*1*,*2*) *test-mult-comm-var star-sim2 mult-isor kat-eq3 mult.assoc test2*)
qed (*metis test-comp-closed-var test-mult-closed*)

definition (**in** *kat*) *while-inv* :: $'a \Rightarrow 'b \Rightarrow 'a \Rightarrow 'a$ (*while - inv - do - [64,64,64] 63*)
where
$\mathit{while}\ b\ \mathit{inv}\ i\ \mathit{do}\ p = (b{\cdot}p)^\star \cdot !b$

lemma *hoare-while-inv*:
assumes *tb*: *test b* **and** *tp*: *test p* **and** *tq*: *test q*
and *pi*: $p \leq i$ **and** *iq*: $i \cdot !b \leq q$
and *inv-loop*: $\{\!|i \cdot b|\!\}\ c\ \{\!|i|\!\}$
shows $\{\!|p|\!\}\ \mathit{while}\ b\ \mathit{inv}\ i\ \mathit{do}\ c\ \{\!|q|\!\}$
by (*metis assms while-inv-def while-rule consequence-rule*)

end

C.2 Kozen's Loop Transformation Theorem

Kozen's transformation theorem for while loops [72] is proved in a weak setting that unifies previous proofs in Kleene algebra with tests, demonic refinement algebras and a variant of probabilistic Kleene algebra.

context *pre-conway*
begin

abbreviation *preservation* :: $'a \Rightarrow 'a \Rightarrow bool$ (**infix** *preserves 60*) **where**
$x \text{ preserves } p \equiv test\ p \wedge p{\cdot}x{\cdot}p = p{\cdot}x \wedge\ !p{\cdot}x{\cdot}!p = !p{\cdot}x$

lemma *preserves-test-closed*: $[\![test\ p;\ x \text{ preserves } q]\!] \Longrightarrow p{\cdot}x \text{ preserves } q$
apply (*auto*, *metis mult.assoc test-mult-comm-var*)
by (*metis mult.assoc test-comp-closed-var test-mult-comm-var*)

lemma *conditional-helper1*:
assumes *test r1*
x1 preserves q y1 preserves q
x2 preserves q y2 preserves q
shows $p{\cdot}q{\cdot}x1{\cdot}(q{\cdot}r1{\cdot}y1 + !q{\cdot}r2{\cdot}y2)^\dagger{\cdot}(q{\cdot}!r1 + !q{\cdot}!r2) = p{\cdot}q{\cdot}x1{\cdot}(r1{\cdot}y1)^\dagger{\cdot}!r1$
proof −
let $?B = q{\cdot}!r1 + !q{\cdot}!r2$
have *pres*: $q{\cdot}(r1{\cdot}y1) = q \cdot (r1{\cdot}y1) \cdot q$
by (*metis assms preserves-test-closed*)
hence $q{\cdot}(q{\cdot}r1{\cdot}y1 + !q{\cdot}r2{\cdot}y2)^\dagger = (q{\cdot}r1{\cdot}y1)^\dagger{\cdot}q$
by (*metis assms*(2−) *test-preserve1 dagger-slide mult.assoc*)
hence $p{\cdot}q{\cdot}x1{\cdot}(q{\cdot}r1{\cdot}y1 + !q{\cdot}r2{\cdot}y2)^\dagger{\cdot}?B = p{\cdot}q{\cdot}x1{\cdot}(q{\cdot}r1{\cdot}y1)^\dagger{\cdot}q{\cdot}?B$
by (*metis assms*(2) *mult.assoc*)
also have $\ldots = p{\cdot}q{\cdot}x1{\cdot}(q{\cdot}r1{\cdot}y1)^\dagger{\cdot}q{\cdot}!r1$
by (*metis assms*(5) *mult.assoc weak-distrib-left-var test-comp-mult annil add-zeror test-mult-idem-var*)
also have $\ldots = p{\cdot}q{\cdot}x1{\cdot}(r1{\cdot}y1)^\dagger{\cdot}!r1$
by (*metis pres assms*(2) *mult.assoc test-preserve*)
finally show *?thesis* **.**
qed

lemma *conditional-helper2*:
assumes *test r2*
x1 preserves q y1 preserves q
x2 preserves q y2 preserves q
shows $p{\cdot}!q{\cdot}x2{\cdot}(q{\cdot}r1{\cdot}y1 + !q{\cdot}r2{\cdot}y2)^\dagger{\cdot}(q{\cdot}!r1 + !q{\cdot}!r2) = p{\cdot}!q{\cdot}x2{\cdot}(r2{\cdot}y2)^\dagger{\cdot}!r2$
proof −
let $?B = q{\cdot}!r1 + !q{\cdot}!r2$
have *pres*: $!q{\cdot}(r2{\cdot}y2) = !q \cdot (r2{\cdot}y2) \cdot !q$
by (*metis assms preserves-test-closed*)
hence $!q{\cdot}(q{\cdot}r1{\cdot}y1 + !q{\cdot}r2{\cdot}y2)^\dagger = (!q{\cdot}r2{\cdot}y2)^\dagger{\cdot}!q$
by (*metis assms*(2−) *test-preserve1*[*of* $!q\ r2{\cdot}y2\ r1{\cdot}y1$] *add.commute mult.assoc test-comp-closed-var test-double-comp-var*)

hence $p{\cdot}!q{\cdot}x2{\cdot}(q{\cdot}r1{\cdot}y1 + !q{\cdot}r2{\cdot}y2)^\dagger{\cdot}?B = p{\cdot}!q{\cdot}x2{\cdot}(!q{\cdot}r2{\cdot}y2)^\dagger{\cdot}!q{\cdot}?B$
by (*metis assms*(4) *mult.assoc*)
also have ... = $p{\cdot}!q{\cdot}x2{\cdot}(!q{\cdot}r2{\cdot}y2)^\dagger{\cdot}!q{\cdot}!r2$
by (*metis assms*(5) *mult.assoc test-comp-closed-var weak-distrib-left-var test-comp-mult2 test-mult-idem-var add-zerol annil*)
also have ... = $p{\cdot}!q{\cdot}x2{\cdot}(r2{\cdot}y2)^\dagger{\cdot}!r2$
by (*metis assms*(4) *mult.assoc pres test-comp-closed-var test-preserve*)
finally show *?thesis* **.**
qed

lemma *cond-distr*:
assumes *test p test q test r*
shows $(p{\cdot}q + !p{\cdot}r){\cdot}(p{\cdot}x + !p{\cdot}y) = p{\cdot}q{\cdot}x + !p{\cdot}r{\cdot}y$
proof −
have $(p{\cdot}q + !p{\cdot}r){\cdot}(p{\cdot}x + !p{\cdot}y) = p{\cdot}q{\cdot}p{\cdot}x + p{\cdot}q{\cdot}!p{\cdot}y + !p{\cdot}r{\cdot}p{\cdot}x + !p{\cdot}r{\cdot}!p{\cdot}y$
by (*metis assms distrib-right′ mult.assoc weak-distrib-left-var add.assoc test-comp-closed-var*)
thus *?thesis*
by (*metis assms mult.assoc test2 test3 test4 annil add-zeror test-comp-closed-var*)
qed

theorem *conditional*:
assumes *test p test r1 test r2*
x1 preserves q y1 preserves q
x2 preserves q y2 preserves q
shows $(p{\cdot}q + !p{\cdot}!q){\cdot}(p{\cdot}x1{\cdot}(r1{\cdot}y1)^\dagger{\cdot}!r1 + !p{\cdot}x2{\cdot}(r2{\cdot}y2)^\dagger{\cdot}!r2) =$
$(p{\cdot}q + !p{\cdot}!q){\cdot}(q{\cdot}x1 + !q{\cdot}x2){\cdot}((q{\cdot}r1 + !q{\cdot}r2){\cdot}(q{\cdot}y1 + !q{\cdot}y2))^\dagger{\cdot}!(q{\cdot}r1 + !q{\cdot}r2)$
proof −
have $p{\cdot}q{\cdot}(x1{\cdot}(r1{\cdot}y1)^\dagger{\cdot}!r1) = p{\cdot}q{\cdot}x1{\cdot}(q{\cdot}r1{\cdot}y1 + !q{\cdot}r2{\cdot}y2)^\dagger{\cdot}(q{\cdot}!r1 + !q{\cdot}!r2)$ **and** $!p{\cdot}!q{\cdot}(x2{\cdot}(r2{\cdot}y2)^\dagger{\cdot}!r2) = !p{\cdot}!q{\cdot}x2{\cdot}(q{\cdot}r1{\cdot}y1 + !q{\cdot}r2{\cdot}y2)^\dagger{\cdot}(q{\cdot}!r1 + !q{\cdot}!r2)$
apply (*metis assms*(2,4−) *conditional-helper1*[*of r1 q x1 y1 x2 y2 p*] *mult.assoc*)
by (*metis assms*(3−) *conditional-helper2*[*of r2 q x1 y1 x2 y2 !p*] *mult.assoc*)
moreover have $(p{\cdot}q + !p{\cdot}!q){\cdot}(p{\cdot}x1{\cdot}(r1{\cdot}y1)^\dagger{\cdot}!r1 + !p{\cdot}x2{\cdot}(r2{\cdot}y2)^\dagger{\cdot}!r2) = p{\cdot}q{\cdot}(x1{\cdot}(r1{\cdot}y1)^\dagger{\cdot}!r1) + !p{\cdot}!q{\cdot}(x2{\cdot}(r2{\cdot}y2)^\dagger{\cdot}!r2)$
by (*metis assms*(1,4−) *cond-distr mult.assoc test-def*)
moreover have ... = $(p{\cdot}q{\cdot}x1 + !p{\cdot}!q{\cdot}x2){\cdot}(q{\cdot}r1{\cdot}y1 + !q{\cdot}r2{\cdot}y2)^\dagger{\cdot}(q{\cdot}!r1 + !q{\cdot}!r2)$
by (*metis calculation*(1) *calculation*(2) *distrib-right′*)
moreover have ... = $(q{\cdot}p{\cdot}x1 + !q{\cdot}!p{\cdot}x2){\cdot}(q{\cdot}r1{\cdot}y1 + !q{\cdot}r2{\cdot}y2)^\dagger{\cdot}(q{\cdot}!r1 + !q{\cdot}!r2)$
by (*metis assms*(1) *assms*(5) *test-comp-closed-var test-mult-comm-var*)
moreover have ... = $(q{\cdot}p + !q{\cdot}!p){\cdot}(q{\cdot}x1 + !q{\cdot}x2){\cdot}((q{\cdot}r1 + !q{\cdot}r2){\cdot}(q{\cdot}y1 + !q{\cdot}y2))^\dagger{\cdot}!(q{\cdot}r1 + !q{\cdot}r2)$
by (*metis assms*(1−3,5) *cond-distr de-morgan-var2 test-comp-closed-var*)
ultimately show *?thesis*
by (*metis assms*(1,5) *test-comp-closed-var test-mult-comm-var*)

qed

theorem *nested-loops*:
assumes *test p test q*
shows $(p{\cdot}x{\cdot}(q{\cdot}y)^\dagger{\cdot}!q)^\dagger{\cdot}!p = p{\cdot}x{\cdot}((p + q){\cdot}(q{\cdot}y + !q{\cdot}x))^\dagger{\cdot}!(p + q) + !p$
proof −
have $p{\cdot}x{\cdot}((p + q){\cdot}(q{\cdot}y + !q{\cdot}x))^\dagger{\cdot}!(p + q) + !p = p{\cdot}x{\cdot}(q{\cdot}y)^\dagger{\cdot}(!q{\cdot}p{\cdot}x{\cdot}(q{\cdot}y)^\dagger)^\dagger{\cdot}!p{\cdot}!q + !p$
by (*metis assms test-distrib mult.assoc de-morgan2 dagger-denest2*)
thus *?thesis*
by (*metis assms mult.assoc test-comp-closed-var test-mult-comm-var add.commute dagger-slide dagger-unfoldl-distr*)
qed

lemma *postcomputation*:
assumes *y preserves p*
shows $(p{\cdot}x)^\dagger{\cdot}!p{\cdot}y = !p{\cdot}y + p{\cdot}(p{\cdot}x{\cdot}(!p{\cdot}y + p))^\dagger{\cdot}!p$
proof −
have $p{\cdot}(p{\cdot}x{\cdot}(!p{\cdot}y + p))^\dagger{\cdot}!p = p{\cdot}(1 + p{\cdot}x{\cdot}((!p{\cdot}y + p){\cdot}p{\cdot}x)^\dagger{\cdot}(!p{\cdot}y + p)){\cdot}!p$
by (*metis dagger-prod-unfold mult.assoc*)
also have $\ldots = (p + p{\cdot}p{\cdot}x{\cdot}((!p{\cdot}y + p){\cdot}p{\cdot}x)^\dagger{\cdot}(!p{\cdot}y + p)){\cdot}!p$
by (*metis assms mult.assoc weak-distrib-left-var distrib-right' mult-1-left*)
also have $\ldots = p{\cdot}!p + p{\cdot}x{\cdot}(!p{\cdot}y{\cdot}p{\cdot}x + p{\cdot}p{\cdot}x)^\dagger{\cdot}(!p{\cdot}y + p){\cdot}!p$
by (*metis assms mult.assoc distrib-right' test-mult-idem-var*)
also have $\ldots = p{\cdot}!p + p{\cdot}x{\cdot}(!p{\cdot}y{\cdot}p{\cdot}x + p{\cdot}p{\cdot}x)^\dagger{\cdot}(!p{\cdot}y{\cdot}!p + p{\cdot}!p)$
by (*metis distrib-right' mult.assoc*)
also have $\ldots = p{\cdot}x{\cdot}(!p{\cdot}y{\cdot}!p{\cdot}p{\cdot}x + p{\cdot}x)^\dagger{\cdot}(!p{\cdot}y)$
by (*metis assms test-comp-mult test-mult-idem-var add-zerol add-zeror*)
also have $\ldots = p{\cdot}x{\cdot}(!p{\cdot}y{\cdot}0 + p{\cdot}x)^\dagger{\cdot}!p{\cdot}y$
by (*metis assms mult.assoc test-double-comp-var test-mult-comp annil*)
moreover have $\ldots = p \cdot x \cdot (p \cdot x)^\dagger{\cdot}(!p \cdot y \cdot 0 \cdot (p \cdot x)^\dagger)^\dagger{\cdot}!p \cdot y$
by (*metis mult.assoc add.commute dagger-denest2*)
moreover have $\ldots = p \cdot x \cdot (p \cdot x)^\dagger{\cdot}!p \cdot y \cdot (0{\cdot}!p \cdot y)^\dagger$
by (*metis annil dagger-slide mult.assoc*)
ultimately have $p{\cdot}(p{\cdot}x{\cdot}(!p{\cdot}y + p))^\dagger{\cdot}!p = p \cdot x \cdot (p \cdot x)^\dagger{\cdot}!p \cdot y$
by (*metis zero-dagger annil mult-1-right*)
thus $(p{\cdot}x)^\dagger{\cdot}!p{\cdot}y = !p{\cdot}y + p{\cdot}(p{\cdot}x{\cdot}(!p{\cdot}y + p))^\dagger{\cdot}!p$
by (*metis dagger-unfoldl-distr mult.assoc*)
qed

lemma *composition-helper*:
assumes *test g test h* $g{\cdot}y = y{\cdot}g$
shows $g{\cdot}(h{\cdot}y)^\dagger{\cdot}!h{\cdot}g = g{\cdot}(h{\cdot}y)^\dagger{\cdot}!h$

apply (*subgoal-tac* $g{\cdot}(h{\cdot}y)^\dagger{\cdot}!h \leq (h{\cdot}y)^\dagger{\cdot}!h{\cdot}g$)
apply (*metis assms*(*1*) *test-eq3 mult.assoc*)
by (*metis assms mult.assoc test-mult-comm-var order-refl dagger-simr mult-isor test-comp-closed-var*)

theorem *composition*:
assumes *test g test h* $g{\cdot}y = y{\cdot}g$ $!g{\cdot}y = y{\cdot}!g$
shows $(g{\cdot}x)^\dagger{\cdot}!g{\cdot}(h{\cdot}y)^\dagger{\cdot}!h = !g{\cdot}(h{\cdot}y)^\dagger{\cdot}!h + g{\cdot}(g{\cdot}x{\cdot}(!g{\cdot}(h{\cdot}y)^\dagger{\cdot}!h + g))^\dagger{\cdot}!g$
apply (*subgoal-tac* $(h{\cdot}y)^\dagger{\cdot}!h$ *preserves g*)
by (*metis postcomputation mult.assoc*, *metis assms composition-helper test-comp-closed-var mult.assoc*)

end

Kleene algebras with tests form pre-Conway algebras, therefore the transformation theorem is valid for KAT as well.

sublocale *kat* $\subseteq$ *pre-conway star*
apply (*default*, *simp-all only*: *star-prod-unfold star-sim2*)
by (*metis star-denest-var star-slide*)

Demonic refinement algebras form pre-Conway algebras, therefore the transformation theorem is valid for DRA as well.

sublocale *dra-tests* $\subseteq$ *pre-conway strong-iteration*
apply (*default*, *metis iteration-denest iteration-slide*)
by (*metis iteration-prod-unfold*, *metis iteration-sim*)

C.3 Refinement Kleene Algebra

rKAT extends KAT with a specification statement and an axiom. The propositional version of Morgan's refinement calculus is derived in this setting.

class *rkat* = *kat* +
fixes *spec* :: $'a \Rightarrow 'a \Rightarrow 'a$
assumes *spec-def*: *test p* $\Longrightarrow$ *test q* $\Longrightarrow$ $p{\cdot}x \leq x{\cdot}q \longleftrightarrow x \leq$ *spec p q*
begin

definition *ref-order* :: $'a \Rightarrow 'a \Rightarrow$ *bool* (**infix** $\sqsubseteq$ *50*) **where**
$x \sqsubseteq y \equiv y \leq x$

lemma *ref-order-refl* [*simp*]: *spec p q* $\sqsubseteq$ *spec p q*
by (*auto simp*: *ref-order-def*)

lemma *spec-char* [*simp*]: *test p* $\Longrightarrow$ *test q* $\Longrightarrow$ $p{\cdot}($*spec p q*$) \leq ($*spec p q*$){\cdot}q$
by (*simp add*: *spec-def*)

lemma *specI* [*intro*]: *test p* $\Longrightarrow$ *test q* $\Longrightarrow$ $p \cdot x \leq x \cdot q \Longrightarrow$ (*spec p q*) $\sqsubseteq x$
by (*simp add*: *spec-def ref-order-def*)

lemma *spec-mult-isol* [*intro*]: $x \sqsubseteq y \Longrightarrow z \cdot x \sqsubseteq z \cdot y$
by (*simp add*: *mult-isol ref-order-def*)

lemma *spec-mult-isor* [*intro*]: $x \sqsubseteq y \Longrightarrow x \cdot z \sqsubseteq y \cdot z$
by (*simp add*: *mult-isor ref-order-def*)

lemma *star-iso* [*intro*]: $x \sqsubseteq y \Longrightarrow x^\star \sqsubseteq y^\star$
by (*simp add*: *star-iso ref-order-def*)

lemma *ref-order-trans* [*trans*]: $x \sqsubseteq y \Longrightarrow y \sqsubseteq z \Longrightarrow x \sqsubseteq z$
by (*simp add*: *ref-order-def*)

lemma *strengthen-post* [*intro*]: *test p* $\Longrightarrow$ *test q* $\Longrightarrow$ *test* $q' \Longrightarrow q' \leq q \Longrightarrow$ *spec p q* $\sqsubseteq$ *spec p* q'
by (*meson specI local.mult-isol local.order-trans spec-char*)

lemma *weaken-pre* [*intro*]: *test p* $\Longrightarrow$ *test q* $\Longrightarrow$ *test* $p' \Longrightarrow p \leq p' \Longrightarrow$ *spec p q* $\sqsubseteq$ *spec* p' *q*
by (*meson specI local.mult-isor local.order-trans spec-char*)

lemma *weaken-and-strengthen* [*intro*]: *test p* $\Longrightarrow$ *test q* $\Longrightarrow$ *test* $p' \Longrightarrow$ *test* $q' \Longrightarrow p \leq p' \Longrightarrow q' \leq q \Longrightarrow$ *spec p q* $\sqsubseteq$ *spec* p' q'
by (*metis ref-order-trans strengthen-post weaken-pre*)

lemma *skip-ref* [*simp*]: *spec 1 1* $\sqsubseteq$ *1*
using *local.test-one-var* **by** *blast*

lemma *abort-ref* [*simp*]: *spec 0 1* $\sqsubseteq x$
using *local.test-one-var local.test-zero-var* **by** *auto*

lemma *magic-ref* [*simp*]: $x \sqsubseteq$ *spec 1 0*
by (*metis local.annir local.mult-1-left local.zero-least local.zero-unique ref-order-def spec-char test-one-var test-zero-var*)

lemma *magic-eq-zero* [*simp*]: *spec 1 0* = *0*
using *local.zero-unique magic-ref ref-order-def* **by** *blast*

lemma *skip-ref-var1*: *test p* $\Longrightarrow$ *spec p 1* $\sqsubseteq$ *1*
by (*auto simp*: *test-one-var local.test-ub-var*)

lemma *skip-ref-var2* [*intro*!]: *test p* $\Longrightarrow$ *test q* $\Longrightarrow$ $p \leq q \Longrightarrow$ *spec p q* $\sqsubseteq$ *1*
by *auto*

lemma *sequence-ref* [*intro*]: *test p* $\Longrightarrow$ *test q* $\Longrightarrow$ *test r* $\Longrightarrow$ *spec p q* $\sqsubseteq$ (*spec p r*) $\cdot$ (*spec r q*)
proof −
assume *hyp*: *test p test q test r*
hence $p \cdot$ (*spec p r*) $\cdot$ (*spec r q*) $\leq$ (*spec p r*) $\cdot r \cdot$ (*spec r q*)
by (*simp add*: *local.mult-isor*)
also have ... $\leq$ (*spec p r*) $\cdot$ (*spec r q*) $\cdot q$
by (*simp add*: *hyp local.mult-isol mult-assoc*)
finally show *?thesis*
by (*auto simp*: *hyp mult-assoc*)
qed

lemma *choice-ref* [*intro*]: *test p* $\Longrightarrow$ *test q* $\Longrightarrow$ *spec p q* $\sqsubseteq x \Longrightarrow$ *spec p q* $\sqsubseteq y \Longrightarrow$ *spec p q* $\sqsubseteq x + y$
by (*simp add*: *local.add-lub ref-order-def*)

lemma *test-expl-ref*: *test b* $\Longrightarrow$ *test p* $\Longrightarrow$ *test q* $\Longrightarrow$ *spec p q* $\sqsubseteq$ $b \cdot$ (*spec* ($p \cdot b$) *q*)
by (*metis local.test-eq3 local.test-mult-closed mult-assoc specI spec-char*)
oops

lemma *conditional-ref*: *test b* $\Longrightarrow$ *test p* $\Longrightarrow$ *test q* $\Longrightarrow$ *spec p q* $\sqsubseteq$ $b \cdot$ (*spec* ($p \cdot b$) *q*) + $!b \cdot$ (*spec* ($p \cdot !b$) *q*)
by (*auto intro*: *test-expl-ref test-comp-closed*)

end

C.4 Correctness Tools

A single while language is defined by a shallow embedding in Isabelle. The implementation is done in Isabelle/ML.

no-notation *comp-op* (*n-* [*90*] *91*)
and *comp-op* (!- [*101*] *100*)
and *test-operator* (*t-* [*100*] *101*)
and *floor* (⌊-⌋)
and *ceiling* (⌈-⌉)
and *Set.image* (**infixr** ‘ *90*)

syntax

-quote :: $'a \Rightarrow ('s \Rightarrow 'a)$ ((≪-≫) [0] 1000)
-antiquote :: $('s \Rightarrow 'a) \Rightarrow 's$ (‘- [1000] 1000)
-assert :: $('s \Rightarrow bool) \Rightarrow 's\ set$

-subst :: $'s\ set \Rightarrow 'v \Rightarrow idt \Rightarrow 's\ set$ (-[-'/‘-] [1000] 999)
-assign :: $idt \Rightarrow 'v \Rightarrow 's\ rel$ ((‘- :=/ -) [0, 65] 62)
-asgn-array :: $idt \Rightarrow nat \Rightarrow 'v \Rightarrow 's\ rel$ ((‘- (-) :=/ -) [0, 0, 65] 62)

-cond :: $'s\ set \Rightarrow 's\ rel \Rightarrow 's\ rel \Rightarrow 's\ rel$ ((0if - then// -//else// -//fi) [0,0,0] 62)
-cond-skip :: $'s\ set \Rightarrow 's\ rel \Rightarrow 's\ rel$ ((0if -//then -//fi) [0,0] 62)
-while :: $'s\ set \Rightarrow 's\ rel \Rightarrow 's\ rel$ ((0while -//do -//od) [0, 0] 62)
-awhile :: $'s\ set \Rightarrow 's\ set \Rightarrow 's\ rel \Rightarrow 's\ rel$ ((0while -//inv -//do// -//od) [0, 0, 0] 62)

-for :: $idt \Rightarrow 'v \Rightarrow idt \Rightarrow 's\ rel \Rightarrow 's\ rel$ ((0for ‘- := - to ‘- do// -//od) [0, 65, 50, 0] 62)
-afor :: $idt \Rightarrow 'v \Rightarrow 'v \Rightarrow 's\ set \Rightarrow 's\ rel \Rightarrow 's\ rel$ ((0for ‘- := - to -//inv - //do -//od) [0, 0] 62)

-apre :: $'s\ set \Rightarrow 's\ rel \Rightarrow 's\ rel$ ({| - |}// - [0, 62] 62)
-apre-aux :: $'v \Rightarrow ('v \Rightarrow 's\ set) \Rightarrow 's\ rel \Rightarrow 's\ rel$ ({| - . - |} - [0, 0, 62] 62)

-aprog-aux :: $'v \Rightarrow 's\ set \Rightarrow 's\ rel \Rightarrow 's\ set \Rightarrow bool$ ({| - . - |}// -// {| - |} [0, 0, 62, 0] 62)

-proc :: $'s\ rel \Rightarrow 's\ rel$ (begin// -//end)
-fun :: $'s\ rel \Rightarrow 'a \Rightarrow ('s\ rel \times 'a)$ (begin// -// return ‘-//end)
-local :: $idt \Rightarrow 'b \Rightarrow 'a\ rel \Rightarrow 'a\ rel$ ((0local ‘- := - in// -//end) [0, 65, 55] 62)
-call :: $idt \Rightarrow ('s\ rel \times 'a) \Rightarrow 's\ rel$ (‘- := call (0-) [0, 65] 62)

-rec :: $'s\ rel \Rightarrow 's\ rel \Rightarrow 's\ rel$ ((0rec - in// -//end) [0, 55] 62)

-ht :: $'s\ set \Rightarrow 's\ rel \Rightarrow 's\ set \Rightarrow bool$ ($\vdash$ {| - |}// -// {| - |} [0, 55, 0] 50)
-ht-aux :: $'v \Rightarrow 's\ set \Rightarrow 's\ rel \Rightarrow 's\ set \Rightarrow bool$ ($\vdash$ {| - . - |}// -// {| - |} [0, 55, 0] 50)

-Spec :: $bool \Rightarrow bool \Rightarrow 'a$ ([|-,-|] [10, 10] 100)

syntax *-ListUpdate :: idt list ⇒ nat ⇒ 'b ⇒ 'a rel (‘- ! - := - [70, 70, 70] 62)*

ML ⟪

```
fun antiquote-tr name =
  let
    fun tr i ((t as Const (c, -)) $ u) =
          if c = name then tr i u $ Bound i
          else tr i t $ tr i u
      | tr i (t $ u) = tr i t $ tr i u
      | tr i (Abs (x, T, t)) = Abs (x, T, tr (i + 1) t)
      | tr - a = a;
  in tr 0 end;

fun quote-tr [t] = Syntax-Trans.quote-tr @{syntax-const -antiquote} t
    | quote-tr ts = raise TERM (quote-tr, ts)
```

⟫

parse-translation ⟪

```
[(@{syntax-const -quote}, K quote-tr)]
```

⟫

translations

```
p [t/‘u]              == -update-name u (λ-. t) ∈ p
‘u := t               == CONST assign (-update-name u) (-quote t)
‘a(i) := t            => CONST assign (-update-name a) ≪(CONST fun-upd (‘a) i t)≫

-assert b             => CONST Collect (≪b≫)

if b then x else y fi  => CONST cond (-assert b) x y
if b then x fi         == if b then x else skip fi
while b do x od         == CONST cwhile (-assert b) x
while b inv i do x od   == CONST awhile (-assert i) (-assert b) x

 for ‘i := n to ‘m do x od=> CONST cfor (CONST Pair (-update-name i) i) ≪n≫ (CONST Pair (-update-name m) m) x

{| p |} x             == CONST apre (-assert p) x
{| u . p |} x          => CONST apre-aux (%u. -assert p) x
```

```
  {| u . p |} x {| q |}        => CONST aprog-aux (%u. -assert p) x (%u. -assert q)

  begin x end                 => x
  begin x return 'z end   => CONST fun-block x (CONST Pair (-update-name z) z)
  local 'u := t in x end  => CONST loc-block (CONST Pair (-update-name u) u)
«t» x
  'z := call R                => CONST fun-call (-update-name z) R

  rec f in x end              => CONST lfp (%f. x)

  ⊢ {| p |} x {| q |}          => CONST ht (-assert p) x (-assert q)
  ⊢ {| u . p |} x {| q |}     => CONST All (%u. CONST ht (-assert p) x (-assert q))

  [[p, q]]                    == CONST Spec (CONST Collect «p») (CONST Collect «q»)

  'A ! k := a => 'A := CONST list-update 'A k a

syntax ( output)
  -assert    :: 's ⇒ 's set                          ([-] [0] 1000)
  -seq       :: 's rel ⇒ 's rel ⇒ 'a rel              (-;// - [59, 59] 60 )
  -ht        :: 's set ⇒ 's rel ⇒ 's set ⇒ bool       (⊢ {| - |}// -// {| - |} [0, 55, 0] 50)

ML ⟪
  fun quote-tr' f (t :: ts) =
        Term.list-comb (f $ Syntax-Trans.quote-tr' @{syntax-const -antiquote} t, ts)
    | quote-tr' - - = raise Match;

  val assert-tr' = quote-tr' (Syntax.const @{syntax-const -assert});

  fun subst-tr' (p :: x :: ts) = (quote-tr' (Syntax.const @{syntax-const -subst} $ p) ts)
$ Syntax-Trans.update-name-tr' x
    | subst-tr' - = raise Match;

  fun assign-tr' (x :: ts) =  quote-tr' (Syntax.const @{syntax-const -assign} $
Syntax-Trans.update-name-tr' x) ts
    | assign-tr' - = raise Match;

  fun local-tr' [(Const - $ - $ x), t, y] = (quote-tr' (Syntax.const @{syntax-const -local}
$ x) [t]) $ y
    | local-tr' - = raise Match;

  fun call-tr' [z, f] = Syntax.const @{syntax-const -call} $ Syntax-Trans.update-name-tr'
z $ f
```

```
  | call-tr' - = raise Match;

fun fun-tr' [x, (Const - $ - $ z)] = Syntax.const @{syntax-const -fun} $ x $ z
  | fun-tr' - = raise Match;

fun for-tr' [(Const - $ - $ i), n, (Const - $ - $ m), x] = (quote-tr' (Syntax.const
@{syntax-const -for} $ i) [n]) $ m $ x
  | for-tr' - = raise Match;

fun print-tr' name [x, y, z] = Syntax.const name $ x $ y $ z
  | print-tr' name [x, y] = Syntax.const name $ x $ y
  | print-tr' name [x] = Syntax.const name $ x
  | print-tr' - - = raise Match;

⟫
```

Hoare's assignment axiom can easily proved for this language.

```
lemma hl-graph [hl-rules]: P ⊆ {s. f s ∈ Q} ⟹ ht P (graph f) Q
  by (auto simp: ht-def graph-def seq-def test-def)

lemma hl-assign [hl-rules]: P ⊆ subst Q z-upd t ⟹ ht P (assign z-upd t) Q
  by (auto simp: assign-def subst-def intro!: hl-graph)
```

A tactic *hoare*, which automatically generates verification conditions, is implemented in Eisbach.

```
named-theorems hl-rules

method hoare-init uses simp =
  ((subst simp | subst fst-conv | subst snd-conv)+)?

method hoare-step uses simp hl =
   (hoare-init simp: simp, (assumption | rule subset-refl | rule mono-rules | rule hl
hl-rules | rule allI | rule ballI))

method hoare-ind uses simp hl =
  (hoare-step simp: simp hl: hl; (hoare-ind simp: simp hl: hl)?)+

method hoare uses simp hl =
  (hoare-init simp: simp; (hoare-ind simp: simp hl: hl)?)
```

method *hl-aux* **uses** *rule* =
(*rule allI*, *rule rule*; ((*erule-tac x=u* **in** *allE*)+, *assumption*))

Finally, some examples of correctness proofs are presented.

record *state* =
x :: *nat*
y :: *nat*
z :: *nat*

lemma *swap*:
⊢ {| ‘*x* = *xo* ∧ ‘*y* = *yo* |}
‘*z* := ‘*x* ;
‘*x* := ‘*y*;
‘*y* := ‘*z*
{| ‘*x* = *yo* ∧ ‘*y* = *xo*|}
by *hoare auto*

lemma ⊢ {| ‘*x* = *n* |}
local ‘*x* := ‘*x* + *1 in*
‘*x* := *2*;
‘*y* := ‘*x* + *1*
end
{| ‘*x* = *n* ∧ ‘*y* = *3* |}
by *hoare-split auto*

lemma ⊢ {| ‘*x* = *u* |} *local* ‘*x* := *t in R end* {| ‘*x* = *u* |}
by *hoare auto*

definition *MAX xo yo* ≡ *begin*
local ‘*x* := ‘*xo in*
‘*y* := ‘*yo*;
if ‘*x* ≥ ‘*y then*
‘*y* := ‘*x*
fi
end
return ‘*y*
end

lemma ⊢ {| *True* |} *proc* (*MAX* ≪*xo*≫ ≪*yo*≫) {| ‘*y* ≥ *xo* ∧ ‘*y* ≥ *yo* |}
by (*hoare simp*: *MAX-def*) *auto*

lemma ⊢ {| ‘*x* = *xo* |} ‘*z* := *call* (*MAX* ≪*xo*≫ ≪*yo*≫) {| ‘*x* = *xo* |}

apply (*hoare-step simp*: *MAX-def* , *simp*)
by *hoare auto*

lemma $\vdash \{\!|\ `y = yo\ |\!\}\ `z := call\ (MAX\ \ll xo\gg\ \ll yo\gg)\ \{\!|\ `y = yo\ |\!\}$
by (*hoare simp*: *MAX-def*) *auto*

lemma $\vdash \{\!|\ `y = yo\ |\!\}\ `z := call\ (MAX\ \ll xo\gg\ \ll yo\gg)\ \{\!|\ `y = yo \wedge\ `z \geq xo \wedge\ `z \geq yo\ |\!\}$
by (*hoare-split simp*: *MAX-def*) *auto*

lemma *swap-annotated*:
$\vdash \{\!|`x = xo \wedge\ `y = yo\ |\!\}$
$`z := `x;$
$\{\!|\ `x = xo \wedge\ `y = yo \wedge\ `z = xo\ |\!\}$
$`x := `y;$
$\{\!|\ `x = yo \wedge\ `y = yo \wedge\ `z = xo\ |\!\}$
$`y := `z$
$\{\!|`x = yo \wedge\ `y = xo|\!\}$
by (*hoare hl*: *hl-apre-classic*) *auto*

record *sum-state* =
s :: *nat*
i :: *nat*

lemma *array-sum*: $\vdash \{\!|\ True\ |\!\}$
$`i := 0;$
$`s := 0;$
while $`i < N$
inv $`s = array\text{-}sum\ a\ 1\ (`i) \wedge\ `i \leq N$
do
$`i := `i + 1;$
$`s := `s + a<`i>$
od
$\{\!|\ `s = array\text{-}sum\ a\ 1\ N\ |\!\}$
by *hoare auto*

hide-const *s i*

record *power-state* =
b:: *nat*
i :: *nat*
n :: *nat*

lemma *power*:
⊢ {| *True* |}
'i := *0*;
'b := *1*;
while *'i* < *'n*
inv *'b* = *a* ^ *'i* ∧ *'i* ≤ *'n*
do
'b := *'b* ∗ *a*;
'i := *'i* + *1*
od
{| *'b* = *a* ^ *'n* |}
by *hoare auto*

lemma *power'*: ⊢ {| *True* |}
'b := *1*;
for *'i* := *0* *to* *'n* *do*
'b := *'b* ∗ *a*
od
{| *'b* = *a* ^ *'n* |}
by *hoare auto*

hide-const *i b n*

record *ls-state* =
i :: *nat*
j :: *nat*
n :: *nat*

lemma *linear-search*:
⊢ {| *'n* > *0* |}
'i := *1*;
while *'i* < *'n*
inv ((∀ *k*. *1* ≤ *k* ∧ *k* < *'i* ⟶ *a*(*k*) ≠ *m*) ∨ (*a*(*'j*) = *m*)) ∧ (*'i* ≤ *'n*)
do
if *a*(*'i*) = *m* *then*
'j := *'i*
fi;
'i := *'i* + *1*
od
{| (∀ *k*. *1* ≤ *k* ∧ *k* < *'n* ⟶ *a*(*k*) ≠ *m*) ∨ (*a*(*'j*) = *m*) |}
apply (*hoare*, *auto*)
using *less-SucE* **by** *blast*

lemma *linear-search'*: $\vdash$ {| $`n > 0$ |}
  for $`i := 1$ *to* $`n$ *do*
    if $a(`i) = m$ *then*
      $`j := `i$
    fi
  od
 {| $(\forall k.\ 1 \leq k \wedge k < `n \longrightarrow a(k) \neq m) \vee (a(`j) = m)$ |}
 apply (*hoare*, *auto*)
 using *less-SucE* **by** *blast*

lemma *linear-search''*:
 $\vdash$ {| $`n > 0$ |}
  $`i := 1$;
  $`j := 0$;
  while $`i < `n$
  inv (*if* $\forall k.\ 1 \leq k \wedge k < `i \longrightarrow a(k) \neq m$ *then* $`j = 0$ *else* $(a(`j) = m)$) $\wedge$ ($`i \leq `n$)
  do
    if $a(`i) = m$ *then*
      $`j := `i$
    fi;
    $`i := `i + 1$
  od
 {| *if* $(\forall k.\ 1 \leq k \wedge k < `n \longrightarrow a(k) \neq m)$ *then* $`j = 0$ *else* $(a(`j) = m)$ |}
 apply *hoare*
 apply *auto*
 using *le-less-linear* **apply** *blast*
 using *le-less-linear* **apply** *blast*
 using *less-SucE* **by** *blast*

hide-const *i j n*
primrec *fact* :: *nat* $\Rightarrow$ *nat* **where**
 fact 0 = *1*
| *fact* (*Suc n*) = (*Suc n*) $*$ *fact n*

lemma *fact*: $\vdash$ {| *True* |}
 $`x := 0$;
 $`y := 1$;
 while $`x \neq xo$
 inv $`y = fact\ `x$
 do
  $`x := `x + 1$;

```
    'y := 'y · 'x
  od
  {| 'y = fact xo |}
  by hoare auto

lemma fact-rec: ∀ xo. ⊢ {| xo = 'x |}
  rec Fact in
    if 'x = 0 then
      'y := 1
    else
      'x := 'x − 1;
      {|xo. xo = 'x + 1|}
      Fact;
      'x := 'x + 1;
      'y := 'y · 'x
    fi
  end
  {| xo = 'x ∧ 'y = fact 'x |}
  by hoare auto

lemma euclids:
  ⊢ {|'x = xo ∧ 'y = yo|}
    while 'y ≠ 0
    inv gcd 'x 'y = gcd xo yo
    do
      'z := 'y;
      'y := 'x mod 'y;
      'x := 'z
    od
    {|gcd xo yo = 'x|}
  by (hoare, auto) (metis gcd-red-nat)

record div-state = state +
  q :: nat
  r :: nat

lemma div:
  ⊢ {| 'x ≥ 0 |}
    'q :=  0; 'r := 'x;
    while 'y ≤ 'r
    inv 'x = 'q * 'y + 'r ∧ 'r ≥ 0
    do
      'q := 'q + 1;
```

```
    ‘r := ‘r − ‘y
  od
 {‘x = ‘q ∗ ‘y + ‘r ∧ ‘r ≥ 0 ∧ ‘r < ‘y}
 by hoare auto

hide-const x y z q r

lemma extend-euclid-invariant:
 assumes (a′ :: int)·m + b′·n = c a·m + b·n = d c = q·d + r
 shows (a′ − q·a)·m + (b′− q·b)·n = r
 using assms int-distrib(2)
 by (auto simp: int-distrib(3))

record extended-euclid-state =
 a :: int
 b :: int
 a′:: int
 b′:: int
 c :: int
 d :: int
 r :: int
 q :: int
 t :: int

lemma extended-euclid: ⊢ {True}
    ‘b := 1;
    ‘a′:= 1;
    ‘b′ := 0;
    ‘a := 0;
    ‘c := m;
    ‘d := n;
    ‘q := ‘c div ‘d;
    ‘r := ‘c mod ‘d;
    while ‘r ≠ 0
    inv
      ‘a′·m + ‘b′·n = ‘c ∧ ‘a·m + ‘b·n = ‘d ∧ ‘c = ‘q·‘d + ‘r
    do
      ‘c := ‘d;
      ‘d := ‘r;
      ‘t := ‘a′;
      ‘a′ := ‘a;
      ‘a := ‘t − ‘q·‘a;
      ‘t := ‘b′;
```

```
    'b' := 'b;
    'b := 't - 'q·'b;
    'q := 'c div 'd;
    'r := 'c mod 'd
  od
{| 'a·m + 'b·n = 'd |}
by hoare (auto simp: extend-euclid-invariant)

hide-const a b a' b' c d r q t
```

References

[1] Aarts, C. Galois connections presented calculationally. *Afstudeer verslag (Graduating Dissertation), Department of Computing Science, Eindhoven University of Technology*, 1992.

[2] Ahrendt, W., Beckert, B., Bruns, D., Bubel, R., Gladisch, C., Grebing, S., Hähnle, R., Hentschel, M., Herda, M., Klebanov, V., Mostowski, W., Scheben, C., Schmitt, P. H., and Ulbrich, M. The KeY platform for verification and analysis of java programs. In Giannakopoulou, D. and Kroening, D., editors, *VSTTE 2014*, volume 8471 of *LNCS*, pages 55–71. Springer, 2014.

[3] Appel, A. W. Tactics for separation logic. *INRIA Rocquencourt and Princeton University, Early Draft*, 2006.

[4] Apt, K. R. Ten years of Hoare's logic: A survey - part 1. *ACM Transactions on Programming Languages and Systems*, 3(4):431–483, 1981.

[5] Apt, K. R., de Boer, F. S., and Olderog, E. *Verification of Sequential and Concurrent Programs*. Texts in Computer Science. Springer, 2009.

[6] Armstrong, A. *Formal Analysis of Concurrent Programs* (to appear). PhD thesis, University of Sheffield, 2015.

[7] Armstrong, A., Gomes, V. B. F., and Struth, G. Algebraic principles for rely-guarantee style concurrency verification tools. In Jones, C. B., Pihlajasaari, P., and Sun, J., editors, *FM 2014*, volume 8442 of *LNCS*, pages 78–93. Springer, 2014.

[8] Armstrong, A., Gomes, V. B. F., and Struth, G. Algebras for program correctness in Isabelle/HOL. In Höfner, P., Jipsen, P., Kahl, W., and Müller, M. E., editors, *RAMiCS 2014*, volume 8428 of *LNCS*, pages 49–64. Springer, 2014.

[9] Armstrong, A., Gomes, V. B. F., and Struth, G. Kleene algebra with tests and demonic refinement algebras. *Archive of Formal Proofs*, 2014.

[10] Armstrong, A., Gomes, V. B. F., and Struth, G. Lightweight program construction and verification tools in Isabelle/HOL. In Giannakopoulou, D. and Salaün, G., editors, *SEFM 2014*, volume 8702 of *LNCS*, pages 5–19. Springer, 2014.

[11] Armstrong, A., Gomes, V. B. F., and Struth, G. Building program construction and verification tools from algebraic principles. *Formal Aspects of Computing*, pages 1–29, 2015.

[12] Armstrong, A. and Struth, G. Automated reasoning in higher-order regular algebra. In Kahl, W. and Griffin, T. G., editors, *RAMiCS 2012*, volume 7560 of *LNCS*, pages 66–81. Springer, 2012.

[13] Armstrong, A., Struth, G., and Weber, T. Kleene algebra. *Archive of Formal Proofs*, 2013.

[14] Armstrong, A., Struth, G., and Weber, T. Program analysis and verification based on Kleene algebra in Isabelle/HOL. In *ITP 2013*, pages 197–212, 2013.

[15] Armstrong, A., Struth, G., and Weber, T. Programming and automating mathematics in the Tarski-Kleene hierarchy. *Journal of Logical and Algebraic Methods in Programming*, 83(2):87–102, 2014.

[16] Back, R.-J. *On the correctness of refinement steps in program development.* PhD thesis, Department of Computer Science, University of Helsinki, 1978.

[17] Back, R. A method for refining atomicity in parallel algorithms. In Odijk, E., Rem, M., and Syre, J., editors, *PARLE '89*, volume 366 of *LNCS*, pages 199–216. Springer, 1989.

[18] Back, R. and Kurki-Suonio, R. Distributed cooperation with action systems. *ACM TOPLAS*, 10(4):513–554, 1988.

[19] Back, R. and von Wright, J. *Refinement Calculus - A Systematic Introduction.* Graduate Texts in Computer Science. Springer, 1998.

[20] Balser, M., Reif, W., Schellhorn, G., Stenzel, K., and Thums, A. Formal system development with KIV. In Maibaum, T. S. E., editor, *FASE 2000*, volume 1783 of *LNCS*, pages 363–366. Springer, 2000.

[21] Barendregt, H. and Barendsen, E. Autarkic computations in formal proofs. *Journal of Automated Reasoning*, 28(3):321–336, 2002.

[22] Berdine, J., Calcagno, C., and O'Hearn, P. W. Verification condition generation and variable conditions in smallfoot. *CoRR*, abs/1204.4804, 2012.

[23] Bergelson, V., Blass, A., and Hindman, N. Partition theorems for spaces of variable words. *Proceedings of the London Mathematical Society*, 68(3):449–476, 1994.

[24] Berstel, J. and Reutenauer, C. *Les séries rationnelles et leurs langages*. Masson, 1984.

[25] Bornat, R. Proving pointer programs in hoare logic. In Backhouse, R. C. and Oliveira, J. N., editors, *MPC 2000*, volume 1837 of *LNCS*, pages 102–126. Springer, 2000.

[26] Bornat, R., Calcagno, C., O'Hearn, P. W., and Parkinson, M. J. Permission accounting in separation logic. In Palsberg, J. and Abadi, M., editors, *POPL 2005*, pages 259–270. ACM, 2005.

[27] Bornat, R., Calcagno, C., and Yang, H. Variables as resource in separation logic. *Electronic Notes in Theoretical Computer Science*, 155:247–276, 2006.

[28] Bulwahn, L., Krauss, A., Haftmann, F., Erkök, L., and Matthews, J. Imperative functional programming with Isabelle/HOL. In Mohamed, O. A., Muñoz, C. A., and Tahar, S., editors, *TPHOLs 2008*, volume 5170 of *LNCS*, pages 134–149. Springer, 2008.

[29] Burstall, R. M. Some techniques for proving correctness of programs which alter data structures. *Machine intelligence*, 7(23–50):3, 1972.

[30] Calcagno, C., O'Hearn, P. W., and Yang, H. Local action and abstract separation logic. In *LICS 2007*, pages 366–378. IEEE Computer Society, 2007.

[31] Cao, J., Fu, M., and Feng, X. Practical tactics for verifying C programs in Coq. In Leroy, X. and Tiu, A., editors, *CPP 2015*, pages 97–108. ACM, 2015.

[32] Chlipala, A., Malecha, J. G., Morrisett, G., Shinnar, A., and Wisnesky, R. Effective interactive proofs for higher-order imperative programs. In Hutton, G. and Tolmach, A. P., editors, *ICFP 2009*, pages 79–90. ACM, 2009.

[33] Cohen, E. Separation and reduction. In Backhouse, R. C. and Oliveira, J. N., editors, *MPC 2000*, volume 1837 of *LNCS*, pages 45–59. Springer, 2000.

[34] Conway, J. H. *Regular Algebra and Finite Machines*. Chapman and Hall, 1971.

[35] Dang, H., Höfner, P., and Möller, B. Algebraic separation logic. *The Journal of Logic and Algebraic Programming*, 80(6):221–247, 2011.

[36] Day, B. On closed categories of functors. In *Reports of the Midwest Category Seminar IV*, pages 1–38. Springer, 1970.

[37] Desharnais, J., Möller, B., and Struth, G. Kleene algebra with domain. *ACM Transactions on Computational Logic*, 7(4):798–833, 2006.

[38] Desharnais, J. and Struth, G. Internal axioms for domain semirings. *Science of Computer Programming*, 76(3):181–203, 2011.

[39] Dijkstra, E. W. A constructive approach to the problem of program correctness. *BIT Numerical Mathematics*, 8(3):174–186, 1968.

[40] Dijkstra, E. W. *A Discipline of Programming*. Prentice-Hall, 1976.

[41] Dinsdale-Young, T., Birkedal, L., Gardner, P., Parkinson, M. J., and Yang, H. Views: compositional reasoning for concurrent programs. In Giacobazzi, R. and Cousot, R., editors, *POPL 2013*, pages 287–300. ACM, 2013.

[42] Distefano, D. and Parkinson, M. J. jstar: towards practical verification for java. In Harris, G. E., editor, *OOPSLA 2008*, pages 213–226. ACM, 2008.

[43] Dongol, B., Gomes, V. B. F., and Struth, G. A program construction and verification tool for separation logic. In Hinze, R. and Voigtländer, J., editors, *MPC 2015*, volume 9129 of *LNCS*, pages 137–158. Springer, 2015.

[44] Dongol, B., Hayes, I. J., and Struth, G. Convolution, separation and concurrency. *ACM Transactions on Computational Logic* (in press), 2015.

[45] Droste, M., Kuich, W., and Vogler, H. *Handbook of weighted automata*. Springer, 2009.

[46] Dudka, K., Müller, P., Peringer, P., and Vojnar, T. Predator: A tool for verification of low-level list manipulation - (competition contribution). In Piterman, N. and Smolka, S. A., editors, *TACAS 2013*, volume 7795 of *LNCS*, pages 627–629. Springer, 2013.

[47] Ehm, T., Möller, B., and Struth, G. Kleene modules. In Berghammer, R., Möller, B., and Struth, G., editors, *RelMiCS 2003*, volume 3051 of *LNCS*, pages 112–124. Springer, 2003.

[48] Feijen, W. H. J. and van Gasteren, A. J. M. *On a Method of Multiprogramming*. Monographs in Computer Science. Springer, 1999.

[49] Filliâtre, J. and Paskevich, A. Why3 - where programs meet provers. In Felleisen, M. and Gardner, P., editors, *ESOP 2013*, volume 7792 of *LNCS*, pages 125–128. Springer, 2013.

[50] Fischer, M. J. and Ladner, R. E. Propositional dynamic logic of regular programs. *Journal of Computer and System Sciences*, 18(2):194–211, 1979.

[51] Floyd, R. W. Assigning meanings to programs. In *Proceeding of Symposia in Applied Mathematics*, volume 19, pages 19–32, 1967.

[52] Foster, S. and Struth, G. On the fine-structure of regular algebra. *Journal of Automating Reasoning*, 54(2):165–197, 2015.

[53] Gardiner, P. H. B., Martin, C. E., and de Moor, O. An algebraic construction of predicate transformers. *Science of Computer Programming*, 22(1-2):21–44, 1994.

[54] Gomes, V. B. F., Guttman, W., Höfner, P., Struth, G., and Weber, T. Kleene algebra with domain. *Archive of Formal Proofs* (to appear), 2015.

[55] Gomes, V. B. F. and Struth, G. Residuated lattices. *Archive of Formal Proofs*, 2015.

[56] Gordon, M. J. C. *The denotational description of programming languages - an introduction*. Springer, 1979.

[57] Gordon, M. J. C. Mechanizing programming logic in higher order logic. In Birtwistle, G. and Subrahmanyam, P. A., editors, *Current Trends in Hardware Verification and Automated Theorem Proving*, pages 387–439. Springer, 1989.

[58] Gunter, C. A. and Scott, D. S. Semantic domains. In *Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics (B)*, pages 633–674. The MIT Press, 1990.

[59] Harel, D., Kozen, D., and Tiuryn, J. *Dynamic logic*. MIT press, 2000.

[60] Hennessy, M. and Plotkin, G. D. Full abstraction for a simple parallel programming language. In Becvár, J., editor, *Mathematical Foundations of Computer Science 1979*, volume 74 of *LNCS*, pages 108–120. Springer, 1979.

[61] Hoare, C. A. R. An axiomatic basis for computer programming. *Communication of ACM*, 12(10):576–580, 1969.

[62] Hoare, C. A. R. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[63] Hoare, C. A. R., Hussain, A., Möller, B., O'Hearn, P. W., Petersen, R. L., and Struth, G. On locality and the exchange law for concurrent processes. In Katoen, J. and König, B., editors, *CONCUR 2011*, volume 6901 of *LNCS*, pages 250–264. Springer, 2011.

[64] Höfner, P. and Struth, G. Automated reasoning in Kleene algebra. In Pfenning, F., editor, *CADE 2007*, volume 4603 of *LNCS*, pages 279–294. Springer, 2007.

[65] Höfner, P., Struth, G., and Sutcliffe, G. Automated verification of refinement laws. *Annals of Mathematics and Artificial Intelligence*, 55(1-2):35–62, 2009.

[66] Ishtiaq, S. S. and O'Hearn, P. W. BI as an assertion language for mutable data structures. In Hankin, C. and Schmidt, D., editors, *POPL 2001*, pages 14–26. ACM, 2001.

[67] Jacobs, B., Smans, J., and Piessens, F. A quick tour of the VeriFast program verifier. In Ueda, K., editor, *APLAS 2010*, volume 6461 of *LNCS*, pages 304–311. Springer, 2010.

[68] Jónsson, B. and Tsinakis, C. Relation algebras as residuated boolean algebras. *Algebra Universalis*, 30(4):469–478, 1993.

[69] Klein, G., Kolanski, R., and Boyton, A. Mechanised separation algebra. In Beringer, L. and Felty, A. P., editors, *ITP 2012*, volume 7406 of *LNCS*, pages 332–337. Springer, 2012.

[70] Kleymann, T. Hoare logic and auxiliary variables. *Formal Aspects of Computing*, 11(5):541–566, 1999.

[71] Kozen, D. *Automata and computability*. Undergraduate texts in computer science. Springer, 1997.

[72] Kozen, D. Kleene algebra with tests. *ACM TOPLAS*, 19(3):427–443, 1997.

[73] Kozen, D. On Hoare logic and Kleene algebra with tests. *ACM Transactions on Computational Logic*, 1(1):60–76, 2000.

[74] Kozen, D. Kleene algebras with tests and the static analysis of programs. Technical Report TR2003-1915, Cornell University, 2003.

[75] Kozen, D. and Patron, M. Certification of compiler optimizations using Kleene algebra with tests. In Lloyd, J. W. and al, editors, *CL 2000*, volume 1861 of *LNCS*, pages 568–582. Springer, 2000.

[76] Kozen, D. and Smith, F. Kleene algebra with tests: Completeness and decidability. In van Dalen, D. and Bezem, M., editors, *CSL '96*, volume 1258 of *LNCS*, pages 244–259. Springer, 1996.

[77] Krauss, A. and Nipkow, T. Proof pearl: Regular expression equivalence and relation algebra. *Journal of Automating Reasoning*, 49(1):95–106, 2012.

[78] Leiß, H. Kleene modules and linear languages. *The Journal of Logic and Algebraic Programming*, 66(2):185–194, 2006.

[79] Manna, Z. and Pnueli, A. *The temporal logic of reactive and concurrent systems: Specification*. Springer, 1992.

[80] Matichuk, D., Wenzel, M., and Murray, T. *The Eisbach user manual*. Isabelle Community, 2015.

[81] The Coq development team. *The Coq proof assistant reference manual*. LogiCal Project, 2015. Version 8.4.pl6.

[82] Morgan, C. The specification statement. *ACM Transactions on Programming Languages and Systems*, 10(3):403–419, 1988.

[83] Morgan, C. C. *Programming from specifications, 2nd Edition*. Prentice Hall International series in computer science. Prentice Hall, 1994.

[84] Naumann, D. A. A categorical model for higher order imperative programming. *Mathematical Structures in Computer Science*, 8(4):351–399, 1998.

[85] Nieto, L. P. The rely-guarantee method in isabelle/hol. In Degano, P., editor, *ESOP 2003*, volume 2618 of *LNCS*, pages 348–362. Springer, 2003.

[86] Nipkow, T. Winskel is (almost) right: Towards a mechanized semantics. *Formal Aspects of Computing*, 10(2):171–186, 1998.

[87] Nipkow, T. Hoare logics for recursive procedures and unbounded nondeterminism. In Bradfield, J. C., editor, *CSL 2002*, volume 2471 of *LNCS*, pages 103–119. Springer, 2002.

[88] Nipkow, T. and Klein, G. *Concrete Semantics - With Isabelle/HOL*. Springer, 2014.

[89] Nipkow, T., Paulson, L. C., and Wenzel, M. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[90] O'Hearn, P. W. Separation logic and concurrent resource management. In Morrisett, G. and Sagiv, M., editors, *ISMM 2007*, page 1. ACM, 2007.

[91] O'Hearn, P. W. A primer on separation logic (and automatic program verification and analysis). In Nipkow, T., Grumberg, O., and Hauptmann, B., editors, *Software Safety and Security - Tools for Analysis and Verification*, volume 33 of *NATO Science for Peace and Security Series - D: Information and Communication Security*, pages 286–318. IOS Press, 2012.

[92] O'Hearn, P. W. and Pym, D. J. The logic of bunched implications. *Bulletin of Symbolic Logic*, 5(2):215–244, 1999.

[93] O'Hearn, P. W., Reynolds, J. C., and Yang, H. Local reasoning about programs that alter data structures. In Fribourg, L., editor, *CSL 2001*, volume 2142 of *LNCS*, pages 1–19. Springer, 2001.

[94] Owicki, S. S. and Gries, D. An axiomatic proof technique for parallel programs I. *Acta Informatica*, 6:319–340, 1976.

[95] Owre, S. and Shankar, N. A brief overview of PVS. In Mohamed, O. A., Muñoz, C. A., and Tahar, S., editors, *TPHOLs 2008*, volume 5170 of *LNCS*, pages 22–27. Springer, 2008.

[96] Plotkin, G. D. A structural approach to operational semantics. *The Journal of Logic and Algebraic Programming*, 60-61:17–139, 2004.

[97] Pous, D. Kleene algebra with tests and coq tools for while programs. In Blazy, S., Paulin-Mohring, C., and Pichardie, D., editors, *ITP 2013*, volume 7998 of *LNCS*, pages 180–196. Springer, 2013.

[98] Preoteasa, V. Algebra of monotonic boolean transformers. *Archive of Formal Proofs*, 2011, 2011.

[99] Rabehaja, T. M. and Sanders, J. W. Refinement algebra with explicit probabilism. In Chin, W. and Qin, S., editors, *TASE 2009*, pages 63–70. IEEE Computer Society, 2009.

[100] Reynolds, J. C. Intuitionistic reasoning about shared mutable data structure. *Millennial perspectives in computer science*, 2(1):303–321, 2000.

[101] Reynolds, J. C. Separation logic: A logic for shared mutable data structures. In *LICS 2002*, pages 55–74. IEEE Computer Society, 2002.

[102] Schirmer, N. *Verification of sequential imperative programs in Isabelle/HOL*. PhD thesis, Technical University Munich, 2006.

[103] Schirmer, N. and Wenzel, M. State spaces - the locale way. *Electronic Notes in Theoretical Computer Science*, 254:161–179, 2009.

[104] Smans, J., Jacobs, B., and Piessens, F. VeriFast for Java: A tutorial. In Clarke, D., Noble, J., and Wrigstad, T., editors, *Aliasing in Object-Oriented Programming. Types, Analysis and Verification*, volume 7850 of *LNCS*, pages 407–442. Springer, 2013.

[105] Solin, K. Normal forms in total correctness for while programs and action systems. *Journal of Logic and Algebraic Programming*, 80(6):362–375, 2011.

[106] Stephan, W., Langenstein, B., Nonnengart, A., and Rock, G. Verification support environment. In Hutter, D. and Stephan, W., editors, *Mechanizing Mathematical Reasoning, Essays in Honor of Jörg H. Siekmann on the Occasion of His 60th Birthday*, volume 2605 of *LNCS*, pages 476–493. Springer, 2005.

[107] Sternagel, C. and Thiemann, R. Certification of nontermination proofs. In Beringer, L. and Felty, A. P., editors, *ITP 2012*, volume 7406 of *LNCS*, pages 266–282. Springer, 2012.

[108] Stoy, J. E. *Denotational semantics: the Scott-Strachey approach to programming language theory*. MIT press, 1977.

[109] Stoy, J. E. Foundations of denotational semantics. In Bjørner, D., editor, *Abstract Software Specifications*, volume 86 of *LNCS*, pages 43–99. Springer, 1979.

[110] Struth, G. On the expressive power of Kleene algebra with domain. *Information Processing Letters*, 2015.

[111] Tuch, H. *Formal Memory Models for Verifying C Systems Code*. PhD thesis, The University of New South Wales, 2008.

[112] Tuerk, T. A separation logic framework for HOL. Technical Report UCAM-CL-TR-799, Computer Laboratory, University of Cambridge, 2011.

[113] Turing, A. Checking a large routine. In *The early British computer conferences*, pages 70–72. MIT Press, 1989.

[114] Turon, A., Vafeiadis, V., and Dreyer, D. GPS: navigating weak memory with ghosts, protocols, and separation. In Black, A. P. and Millstein, T. D., editors, *OOPSLA 2014*, pages 691–707. ACM, 2014.

[115] Vafeiadis, V. Modular fine-grained concurrency verification. Technical Report UCAM-CL-TR-726, Computer Laboratory, University of Cambridge, 2008.

[116] von Neumann, J. and Goldstine, H. Planning and coding of problems for an electronic computing instrument. *Institute for Advanced Study, Princeton, New Jersey*, 1948.

[117] von Wright, J. From Kleene algebra to refinement algebra. In Boiten, E. A. and Möller, B., editors, *MPC 2002*, volume 2386 of *LNCS*, pages 233–262. Springer, 2002.

[118] von Wright, J. Towards a refinement algebra. *Science of Computer Programming*, 51(1-2):23–45, 2004.

[119] Weber, T. Towards mechanized program verification with separation logic. In Marcinkowski, J. and Tarlecki, A., editors, *CSL 2004*, volume 3210 of *LNCS*, pages 250–264. Springer, 2004.

[120] Yang, H. and O'Hearn, P. W. A semantic basis for local reasoning. In Nielsen, M. and Engberg, U., editors, *FOSSACS 2002*, volume 2303 of *LNCS*, pages 402–416. Springer, 2002.