University of Queensland
School of Information Technology and Electrical Engineering

FINAL YEAR THESIS

Rely/Guarantee Algebra: Model Verification

Julian Fell

supervised by
Ian Hayes and Larissa Meinicke

Julian Fell
101 The Promenade
Camp Hill QLD 4152

November 5, 2015

Prof Paul Strooper
Head of School
School of Information Technology and Electrical Engineering
The University of Queensland
St Lucia QLD 4072

Dear Professor Strooper,

In accordance with the requirement of the Degree of Bachelor of Engineering in the School of Information Technology and Electrical Engineering, I submit the following thesis entitled

Rely/Guarantee Algebra: Model Verification

The thesis was performed under the supervision of Ian Hayes and Larissa Meinicke. I declare that the work submitted in thesis is my own, except as acknowledged in the text and footnotes, and has not been previously submitted for a degree at The University of Queensland or any other institution.

Yours sincerely,

Julian Fell

Acknowledgements

Foremost, I would like to express my appreciation and gratitude to my supervisors Ian Hayes and Larissa Meinicke for their continued support and guidance throughout the entire project. Their patience and enthusiasm is matched only by their deep knowledge of the field.
Besides my supervisors, I would like to thank Kim Solin for being open to questions and generous with his time and expertise.

Last but not least, I would like to thank my family for their support and encouragement throughout my studies and life in general.

Abstract

Formal methods can be used as a basis for tools that assist reasoning about concurrent programs. In order to support reasoning about programs and their specifications, wide-spectrum languages integrate specification constructs with programming language primitives in a single language. This enables program commands to be manipulated, using semantic-preserving transforms from abstract specifications to concrete implementations, while maintaining a notion of correctness. Abstract algebras designed to model the behaviour of wide-spectrum languages allow concise proofs of general laws about the semantics of these transformations, while abstracting away the low level details of program implementations. One instance of these abstract algebras is the rely/guarantee algebra, which has been formalised in the generic proof assistant Isabelle/HOL.

We develop a formalisation of a semantic trace model for a wide-spectrum language in Isabelle/HOL to verify the validity of the rely/guarantee algebra. This is important at a mathematical level as it proves the consistency of the axiomatic definition of the algebra. In addition to this, the formalisation of a trace model for a wide-spectrum language allows reasoning about complex behaviours of realistic concurrent programs by providing a concrete basis for the language semantics.

Contents

List of Figures

1 Introduction

Concurrency in computer programs is a growing trend with most computer hardware, from smart phones to data centres, having multiple processors. As access to multiple-processor environments increases, programs need to take advantage of the performance improvements that are possible. The downside of this trend is that concurrent algorithms have an increased number of possible execution paths and are therefore necessarily more complex than single-threaded programs to write and maintain. These factors lead to increases in the number of incorrect and inefficient programs being developed. For these reasons, better methods of developing correct concurrent programs are essential.

One approach to creating better tools in computer science involves to deriving concrete implementations of programs from abstract specifications of how they should behave. This can be achieved through using a sequence of correctness-preserving transformations, a process generally referred to as program refinement.

In order to discover and apply these transformations, programs have to be represented in a way that allows efficiency of reasoning. Correctness preserving program transformations on program commands form a partial ordering called the refinement ordering ($\sqsubseteq$). Program commands and the operators which act on them, such as iteration and sequential composition, can be described by algebraic laws.

This idea has been well developed in other works, although most of the theory in this area focuses on sequential programs. Many papers [3, 26] have explored the idea of using algebras to prove theorems which merge and simplify sequential loops. A popular law for reasoning about the composition of loops is the *leapfrog* property of the iteration ($*$) and sequential composition ($;$) operators.

$$c \,;\, (d \,;\, c)^* = (c \,;\, d)^* \,;\, c$$

This rule amongst others describe the manipulations that can be safely made to expressions describing program specifications. Notably, the leapfrog rule can be used to show the equivalence of merged and split loops in the general case [3].

Expanding this idea to include parallelism allows derivation of concurrent algorithms in a similar fashion. As an example of this, commands composed sequentially ($;$) and in parallel ($\parallel$) satisfy interchange laws which preserve the correctness of their semantics.

$$(c_0 \,;\, d_0) \parallel (c_1 \,;\, d_1) \sqsubseteq (c_0 \parallel c_1) \,;\, (d_0 \parallel d_1)$$

An important concept in this area of computer science is the idea of axiomatic definitions. Abstract algebras are defined axiomatically, which means that a set of statements about the properties of the algebra are assumed to be true as a starting point. This set of axioms are then applied to prove all the theorems in the algebraic theory. The axiomatic approach to algebra results in elegant and simple proofs and forms the basis of many fields of modern mathematics. It is of course possible to start with a set of axioms that is inconsistent or not an accurate abstraction of the concrete subject matter (in this case program commands).

To ensure that the axiomatisation is consistent, relevant models can be shown to be instances of the algebra. The existence of models are an important part of demonstrating the validity of all algebraic theories, regardless of context. The relationship between the expressions in the algebra and code generated in a general parallel programming language can be demonstrated through the use of a relevant semantic model.

The rely/guarantee algebra is an abstract algebra that has been developed recently to improve reasoning about concurrent programs. Two novel operators are introduced in the rely/guarantee

algebra that model the affect of interference, which is necessary to reason compositionally about parallel composition. These operators are effectively ways to describe contracts between a command and its environment, which clearly cannot be directly translated into program constructs. Despite this, they are very useful for specifying the behaviour of a program component with respect to its environment.

A semantic description of a wide-spectrum specification language has been developed that provides a semantic trace model for the rely/guarantee algebra. This model represents program commands as the set of possible execution paths they can take.

1.1 Project Definition

The topic for this thesis is the construction and instantiation of a trace model [6] for the rely/guarantee algebra [12] in an automated theorem prover. The outcomes of this are twofold. The considered axioms of the rely/guarantee algebra are shown to be consistent, which adds to the validity of the theory as no other model has been verified to be an instantiation of this algebra. As the model is relevant to the context of concurrent programs, it provides a concrete basis for the semantics of the algebra.

In addition to this, all of the theorems proved for the algebra automatically apply to the trace model, which allows for complex reasoning about concurrent programs, including aborting, terminating, infinite and infeasible behaviour.

The generic theorem prover Isabelle/HOL is used to produce a mechanical instantiation of the rely/guarantee algebra formalisation in [10]. Mechanical proofs improve the confidence in the results derived and also potentially provide a basis for using the results for automated derivation of programs.

1.2 Thesis Outline

This thesis covers the formal representation of the trace model of the rely/guarantee algebra in Isabelle/HOL as well as a partial instantiation of the algebra. The project is broken down into three major sections that comprise the thesis contribution, which build on earlier work formalising the algebra itself [10] that is outside the scope of this thesis. These sections are preceded by two broad background sections that establish the context and theory to inform the details of the formalisation and instantiation.

1.2.1 Background

Chapter 2 introduces the key features and concepts in Isabelle/HOL, including some illuminating examples. This is intended to provide context for the discussion of Isabelle/HOL constructs throughout the thesis and to establish familiarity with the syntax for proofs and definitions.

The core concepts and motivations of formal methods in reasoning about programs are discussed in terms of program semantics and specifications in Chapter 3. A working definition of the rely/guarantee algebra is also built up to through a discussion of programming algebra, the refinement calculus and lattice theory.

1.2.2 Model Formalisation

The model is formalised in Isabelle/HOL and key properties about each definition are proved to enable easy reasoning for in the later proofs. The definition of the model is built up from a generic definition of program states to a complete representation of processes. A more detailed description of this process can be found in Chapter 4.

1.2.3 Operator Formalisation

The semantics of the operators are defined on the model, with consideration of the properties of each construct. The semantics of the non-deterministic choice, sequential composition and parallel composition operators are considered in detail. Due to time restrictions, the strict conjunction operator

is not considered in this thesis, while the iteration and rely quotient operators are defined in terms of other operators in the algebra so do not require explicit definitions in the model. A full description of these operators can be found in Chapter 5.

1.2.4 Algebra Instantiation

The axioms of non-deterministic choice and sequential composition in the algebra are proved to be satisfied by the model. The instantiation of the parallel composition operator is not complete, so this operator is not discussed in this section. This section of the project imports the Isabelle/HOL definition of the rely/guarantee algebra to complete the instantiation command. Figure 1.1 illustrates the relationship between the algebra and semantic model through instantiation and inheritance constructs in the Isabelle/HOL theory files.

A more detailed description of this process can be found in Chapter 6.

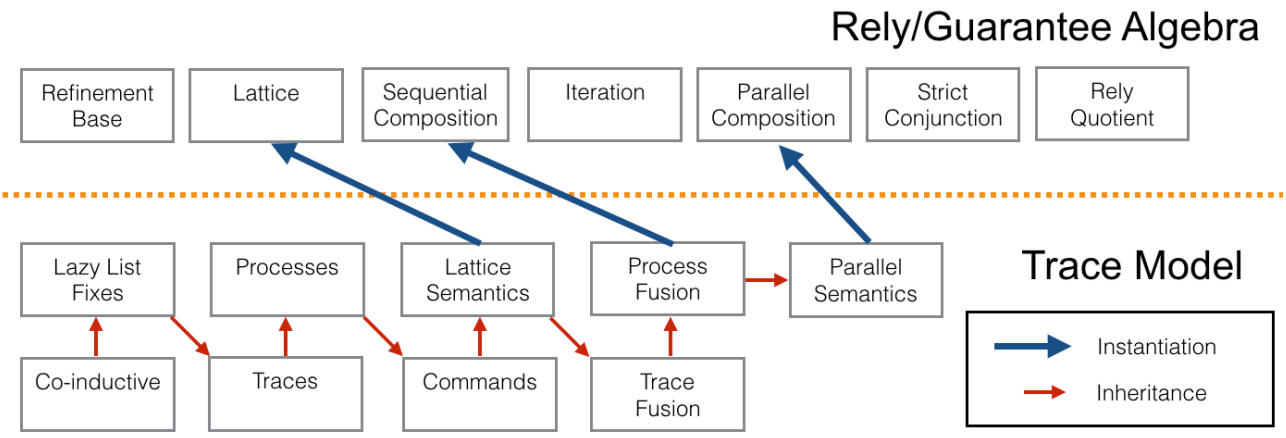

Figure 1.1: Isabelle/HOL theory file structure

2 Isabelle/HOL

Isabelle is a generic proof assistant, based on a logical framework, which incorporates an integrated development environment (IDE) and a number of automated theorem provers to produce mechanical proof documents [23]. Isabelle/HOL, the specialisation of Isabelle for higher order logic, is becoming increasingly common in the field of theoretical computer science for verifying theories and proofs.

2.1 Types and Functions

Isabelle/HOL uses a type system which consists of exclusively total functions, so there is no concept of undefinedness. For this reason, partial functions are only supported in that a domain predicate can characterise the values which a function is defined over. This is achieved through partial-simplification properties which are guarded by the domain predicates [21]. As in the following example, partial functions are declared with the *domintros* option that instructs Isabelle to generate the domain predicates.

```
function (domintros) findzero  :: "(nat ⇒ nat) ⇒ nat ⇒ nat" where
  "findzero f n = (if f n = 0 then n else findzero f (Suc n))"
  apply (pat_completeness)
     by (auto)
```

As all functions require a termination proof to justify that they are defined through well-founded recursion, a partial function requires a suitable domain predicate to prove that it terminates over its domain. In the example of *findzero*, it is intuitively clear that this function only terminates if there is a zero which is greater or equal to n. The automatically generated *findzero.domintros* domain predicate

```
lemma findzero_domintros: "(0 < f n ⟹ findzero_dom (f, Suc n)) ⟹ findzero_dom
(f, n)"
```

can be used to prove this property of this function [21].

```
lemma findzero_termination:
  assumes zero: "x >= n"
  assumes [simp]: "f x = 0"
  shows "findzero_dom (f, n)"
  using zero
  apply (induct rule:inc_induct)
     by (auto intro: findzero.domintros)
```

Subsets of already existing types can be defined through a *typedef*. This form of declaration requires a proof that it is not empty, and is accompanied by implicitly defined morphisms which define the injection from type to set (Rep_T), and the inverse (Abs_T). Basic predicates concerning the properties of the newly defined set are also instantiated [27].

```
typedef type = "{n::nat. n ≤ 3}"
  by blast

lemma "(Rep_type a) ≤ 3"
  using Rep_type by force
```

2.2 Proof Methods

Isabelle/HOL supports numerous formal proof methods, each of which are suitable in different circumstances. The traditional mathematical proof methods of induction, co-induction, anti-symmetry and contradiction all function as expected, allowing most conventional proofs to be reproduced in Isabelle.

Applying these proof methods can be difficult, so Isabelle has a number of tools to assist in goal simplification and interactive proof construction. Isabelle has an inbuilt simplification engine as well as external provers which automatically simplify proof goals to suit the main theorem provers. Trivial proofs can be completed using these tools exclusively, although their limitations become apparent when proofs become more complex.

Sledgehammer is a subsystem of Isabelle that automatically locates relevant lemmas and invokes external automated theorem provers [24] to construct proofs. This simplifies and speeds up the proof process for both straightforward and complex proofs.

All proof methods can be employed using apply-style proofs, which are concise but inexpressive. A trivial example of an apply-style proof applying induction to a lemma demonstrates the syntax of this style of proof in Isabelle.

```
lemma rev_rev [simp]: "rev (rev xs) = xs"
  apply (induct xs)
     by (auto)
```

The lemma is declared and given a name on the first line to set up the proof goals. After this, a series of proof methods are applied, in this case induction is employed and both resulting subgoals are proved by the *auto* simplifier.

Structured proofs can also be written using the inbuilt Isar proof language. Isar is used in Isabelle to enable the production of human-readable mechanical proofs which are explicit enough to be maintainable and easy to understand. Each proof step can be proved by any of the automated theorem provers compatible with Isabelle/HOL using a combination of the internal higher order logic (HOL), locale assumptions and any previously proved lemmas.

```
lemma subst: "y = x ⟹ B x ⟹ B y"
proof -
  assume a: "B x"
  assume b: "y = x"
  then show "B y" by (metis a b)
qed
```

The declaration of a lemma is the same as for apply-style proofs, but the proof itself is more explicit. Assumptions and intermediate steps are entered and proved individually. The *show* keyword specifies the final step in the proof and must match up with the original lemma declaration [27].

2.3 Locales

A key feature of Isabelle/HOL is the intuitive structuring of formal theories, combining assumptions, definitions and proofs in a self-contained theory. Within an Isabelle/HOL theory, assumptions can be organised in locales at an abstract level along with the accompanying theorems. These locales can also be built up by combining other locales along with additional assumptions in an inheritance hierarchy [27].

The following example defining a partial order demonstrates the syntax for defining a locale.

```
locale partial_order =
  fixes le :: "'a ⇒ 'a ⇒ bool" (infixl "≪" 50)
  assumes refl [intro, simp]: "x ≪ x"
  and anti_sym [intro]:       "⟦ x ≪ y; y ≪ x ⟧ ⟹ x = y"
  and trans [trans]:          "⟦ x ≪ y; y ≪ z ⟧ ⟹ x ≪ z"
```

2.4 Instantiation

Once a locale has been defined, it can be instantiated with the *interpretation* command in Isabelle/HOL. This command requires proof that the chosen operators and set of mathematical objects satisfy the axioms of the locale [27]. The proof goals created by this command correspond directly with the axioms of the selected locale.

As a locale is an abstract grouping of assumptions and theorems, the *interpretation* command allows great flexibility in reasoning jointly about algebras and models with shared properties [4]. This allows theories to be reused for varying instances without having to prove all the properties that have already been completed for the abstractly defined algebra. The inheritance structure of the locales enables this instantiation to be completed in stages, allowing modular structuring of the interpretation process.

The partial ordering defined above can be instantiated for the *less than* operator over integers.

```
interpretation int: partial_order "op ≤ :: [int, int] ⇒ bool"
  apply (unfold_locales)
  apply (metis order_refl)
  apply (metis eq_iff)
     by (metis order_trans)
```

2.5 Co-Induction

Iteration and looping constructs in programming language semantics are usually described in terms of least or greatest fixed-points. Least fixed-points restrict the iteration to being finite, while greatest fixed points include the possibility of infinite iteration. As the rely/guarantee algebra considers both forms of iteration, the denotational model for traces requires infinite data structures. A co-inductive approach allows elegant definitions of these data structures, as co-induction supports building theoretically infinite types [20]. The two forms of definitions for lists are declared similarly, with the only difference being normal lists are defined inductively and lazy lists are defined co-inductively.

```
codatatype (lset: 'a) llist =
    lnull: LNil
  | LCons (lhd: 'a) (ltl: "'a llist")    (infixr "##" 65)

datatype 'a list =
    Nil    ("[]")
  | Cons 'a  "'a list"                   (infixr "#" 65)
```

The Isabelle Archive of Formal Proofs contains a library written by Andreas Lochbihler, which contains definitions of co-inductive lists and sets [22]. Proofs of many useful and well-known properties of these datatypes are also included.

3 Program Semantics

The semantics of a programming language, in contrast with the syntax, are concerned with the mathematical meaning of programs. Semantic descriptions of programming languages have been dominated by two approaches. The original and most popular approach is that of operational semantics, which defines programs in terms of transitions between states of an interpreter [18]. In contrast, a denotational approach is concerned with definitions of programs as a direct mappings to their meanings. This takes the form of the construction of mathematical objects that are compositionally combined to define the semantics of programs [25]. A denotational approach is taken in this thesis, as it aligns with the goals of the project.

3.1 Program Specifications

An abstract program specification describes the desired behaviour of a program without defining how that behaviour is to be realised. In contrast, a concrete implementation describes the behaviour of a program in terms of an acceptable program implementation. The challenge is to prove that a concrete implementation is correct in relation to its abstract specification. C.A.R Hoare developed a notation for describing the specifications of sequential programs in terms of pre- and post-conditions. A program, c, and the pre-conditions, P, and post-conditions, Q, on the program state are expressed as a *partial correctness specification* or Hoare Triple [14]. The Hoare triple,

$$\{P\}c\{Q\}$$

specifies that a program, c, when executed from a state satisfying a pre-condition, P, will terminate in a state satisfying a post-condition, Q. As a concrete example, if we have a specification of

$$\{x > 0\}c\{x > 2\}$$

then an implementation of c that meets that specification is:

$$\mathrm{x} := \mathrm{x} + 2$$

In this example, the implementation trivially meets the specification. As the specification becomes more complex, it quickly becomes non-trivial to prove that a program implements the specification correctly.

3.2 Rely and Guarantee Conditions

While programs executing in isolation can be effectively modelled using just pre- and post-conditions, when concurrency is considered, a new construction is required to express interference between parallel programs. Rely and guarantee conditions are an extension of Hoare triples to quintuples to accommodate specifying interactions with the environment. These conditions constrain the changes that a process and its environment can make to the global state [16]. The Hoare quintuple,

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

specifies that a program, c, when executed from a state satisfying P, and an environment satisfying R, will terminate in a state satisfying Q, with each atomic step satisfying G in relation to its environment.

A rely condition, R, is a condition that a command assumes is met by the environment between the before and after states of every zero or more atomic steps. Every atomic step of the command must satisfy the guarantee condition G [13]. The relations specified as rely and guarantee conditions are required to be reflexive to allow stuttering steps.

3.3 Wide-Spectrum Languages

A wide-spectrum language combines a set of specification constructs with the commands of a programming language into a single language. This allows Hoare's partial correctness specifications, including rely and guarantee conditions, to coexist with program commands [6]. By supporting expressions simultaneously including both high-level specification constructs and low-level programming language primitives, wide-spectrum languages are useful for developing programs based on correctness-preserving transformations.

3.4 Program Refinement

Specifications are only useful when programs can be proved to implement them. The commands in a wide-spectrum language can be related by a partial-ordering on their semantics, called the refinement ordering ($\sqsubseteq$), based on whether an command can implement or be *refined* by another command. Therefore a program implementation (consisting of only the subset of the wide-spectrum language that is directly implementable) can be shown to be a *refinement* of a specification.

The non-deterministic choice operator ($\sqcap$) is a specification operator that models the situation where two commands, c and d, are possible then a non-deterministic or "free" choice can be made between executing c and d. This choice operator is defined in terms of refinement.

$$c \sqsubseteq d \iff c \sqcap d = c$$

This relationship dictates that if a command, c, refines a command, d, the choice between the two commands is equivalent to just c. Intuitively, this means that for an expression to refine a specification it must not allow any behaviour that is not allowed by that specification.

3.5 Lattice Theory

Mathematically, the refinement ordering forms a lattice on the language of program commands with non-deterministic choice and its dual ($\sqcup$) taking the place of the infimum and supremum operators [26]. Two specific primitive commands with special meanings, **magic** ($\top$) and **abort** ($\bot$), form the greatest and least elements on the lattice. In the model considered in this thesis, the lattice of program commands is a distributive lattice, meaning compositional combination of the commands satisfies certain distributive properties.

By the Knaster-Tarski Fixpoint Theorem, any order-preserving map, F, on a complete lattice has both least and greatest fixed points [7]. As the lattice of program commands is a complete lattice, it has both least and greatest fixed points. In addition to their utility in defining iteration of program commands, fixed points provide powerful properties for reasoning about iteration through their induction and fusion rules.

3.6 Abstract Algebra

Abstract algebra is the study of mathematical structures through considering their properties abstractly rather than studying concrete instances of mathematical structures. This allows properties proved for the abstract algebra to apply to any mathematical object that can be shown to be an

instance of the algebra. They are defined axiomatically, meaning that a set of axioms of a carrier set and operators are taken to be true as a starting point for the theory [9]. The carrier set is required to be closed under all operators in the algebra in order to keep the theory self-contained.

A common example of an axiomatic definition is the Peano Axioms for natural numbers, in which the natural numbers are uniquely defined using only the number zero and a successor function. There are five axioms that describe the properties of zero and the successor function on the carrier set N [11].

$$\exists 0 \quad : \quad 0 \in N \tag{3.1}$$
$$\forall x \in N \quad : \quad \exists S(x) \in N \tag{3.2}$$
$$\forall x \in N \quad : \quad S(x) \neq 0 \tag{3.3}$$
$$\forall x, y \in N \quad : \quad S(x) = S(y) \Longrightarrow x = y \tag{3.4}$$
$$\forall A \subseteq N \quad : \quad (0 \in A \wedge (x \in A \Longrightarrow S(x) \in A)) \Longrightarrow A = N \tag{3.5}$$

3.7 Algebraic Models

As abstract algebras are defined by a set of axioms, the consistency and correctness of these axioms need to be verified. Models establish a concrete instance that can be verified to satisfy all of the axioms of an abstract algebra.

To continue the example of the Peano Axioms for natural numbers, there exists a model for this axiomatisation based on set theory. The empty set is defined as zero and the successor function is defined as

$$S(x) = x \cup \{x\}$$

Each of the axioms (3.1 - 3.5) and closure under the successor operator can be shown to be true using this definition and basic set theory [11]. Instantiating the elements of the algebra and proving the axioms hold proves the consistency of the axiomatisation.

3.8 Programming Algebra

Programming algebras, such as the rely/guarantee algebra, are abstract algebras with models that describe the semantics of commands in a wide-spectrum language [5]. The abstract properties of compositional operators are taken as axioms to develop theories about the combination and transformation of computer programs. Semantic models of programming languages can then be used to verify the consistency of the axiomatisations.

3.9 Kleene and Omega Algebras

Kleene Algebras (KA) are commonly known from their applications in the automata for regular expressions [19]. They can also be interpreted over program commands (rather than an alphabet), with the operators taking on slightly different meanings in each interpretation. The union operator in regular expressions corresponds to non-deterministic choice, while concatenation corresponds to sequential composition of program commands. The Kleene Star represents finite iteration in both contexts. The **abort** and **magic** represent the set of all finite strings and the empty set, respectively.

A Kleene Algebra is an idempotent semi-ring, over the lattice of program commands, that is extended with a finite iteration operator ($*$).

$$(C, \sqcap, \ ;\ , *, \mathbf{abort}, \mathbf{magic})$$

The non-deterministic choice operator is associative, commutative, idempotent and has an identity element (**magic**). These properties follow intuitively from the concept of a choice operator. More interestingly, the sequential composition operator (**;**) models program commands being executed sequentially. This is the most basic form of composition in imperative programming languages and is also used in the definition of the iteration. As per the definition of a semi-ring, this operator is associative and has an identity and a left annihilator. Notably, it is not commutative, as switching the order of two program commands will not return the same result in general.

The Kleene Star operator is an unary operator that models finite iteration of program commands and is defined through the greatest fixed point operator (ν). An Omega Algebra is similar to a Kleene Algebra with the addition of a strong iteration operator, which is defined similarly to the Kleene star in terms of the least fixed point operator (μ).

3.10 Concurrent Kleene Algebra

Kleene and Omega Algebras are of limited used for modelling concurrent programs as all of the operators deal with programs composed sequentially. A parallel composition operator is introduced to this setting in [15], to consider concurrency in the Concurrent Kleene Algebra (CKA). The sequential composition and parallel composition operators are related by an inequational exchange law.

$$(c_0 \mathbin{;} d_0) \parallel (c_1 \mathbin{;} d_1) \sqsubseteq (c_0 \parallel c_1) \mathbin{;} (d_0 \parallel d_1)$$

The concurrent Kleene algebra handles interference conditions in a separate algebra, with explicit *rely* and *guar* clauses.

3.11 Specifications in the Refinement Ordering

A feature of wide-spectrum languages is the ability to model specifications as program commands. Algebras on the refinement ordering achieve this through a special assumption ($\{c\}$) and specification ($[c]$) syntax. Therefore, Hoare logic [14] pre- and post-conditions can be expressed in a wide-spectrum language on the refinement ordering as:

$$\{P\}c\{Q\} \Longrightarrow \{P\} \mathbin{;} [Q] \sqsubseteq c$$

Less formally, if the command, c, establishes a post-condition, Q, from states satisfying a pre-condition, P, then c refines the specification of $\{P\}$ sequentially composed by $[Q]$.

3.12 Interference in the Refinement Ordering

Interference between parallel processes can be modelled in the refinement ordering using the strict conjunction and rely quotient operators as a generalisation of C.B. Jones rely/guarantee relations [17] to processes. This provides a more expressive approach to defining rely and guarantee conditions that enables describing more complex requirements than binary relations on states.

Strict conjunction ($\Cap$) is a specification construct that acts as a weak conjunction operator on program commands. A strict conjunction of two commands behaves as the conjunction of both commands until one of them aborts, in which case the process aborts. This enables the behaviour of a program to be constrained by an arbitrary process, allowing guarantee conditions to be specified as part of the algebra. The axioms relating to strict conjunction can be found in Figure 3.2.

The rely quotient operator ($/\!\!/$) constrains the behaviour of arbitrary environment processes, which allows rely conditions to be generalised to processes [12]. It is defined as the non-deterministic choice over all commands d satisfying the defining property of the rely quotient ($c \sqsubseteq d \parallel i$), where i is an arbitrary process representing the rely specification:

$$c /\!\!/ i = \bigsqcap \{d \mid (c \sqsubseteq d \parallel i)\}$$

Hoare quintuples can be expressed in the refinement ordering using the rely quotient and strict conjunction operators.

3.13 Rely/Guarantee Algebra

The algebraic underpinnings of the rely/guarantee algebra [12] are an Omega Algebra on the refinement ordering, extended by the three concurrency focused operators: parallel composition ($\|$), strict conjunction ($\Cap$) and rely quotient ($/\!\!/$).

3.13.1 Lattice and Fixed Point Axioms

The lattice and fixed point axioms are standard properties for any complete lattice structure with monotonic functions.

Lattice

$$
\begin{aligned}
c_0 \sqcap (c_1 \sqcap c_2) &= (c_0 \sqcap c_1) \sqcap c_2 \\
c_0 \sqcap c_1 &= c_1 \sqcap c_0 \\
c \sqcap c &= c \\
c \sqcap \mathbf{magic} &= c \\
c_0 \sqcup (c_1 \sqcup c_2) &= (c_0 \sqcup c_1) \sqcup c_2 \\
c_0 \sqcup c_1 &= c_1 \sqcup c_0 \\
c \sqcup c &= c \\
c \sqcup \mathbf{abort} &= c \\
c_0 \sqcap (c_0 \sqcup c_1) &= c_0 \\
c_0 \sqcup (c_0 \sqcap c_1) &= c_0
\end{aligned}
$$

Distributive lattice

$$c_0 \sqcup (c_1 \sqcap c_2) = (c_0 \sqcup c_1) \sqcap (c_0 \sqcup c_2)$$

Complete lattice

$$
\begin{aligned}
c \in C &\implies \bigsqcap C \sqsubseteq c \\
(\forall c \in C \bullet d \sqsubseteq c) &\implies d \sqsubseteq \bigsqcap C \\
c \in C &\implies c \sqsubseteq \bigsqcup C \\
(\forall c \in C \bullet c \sqsubseteq d) &\implies \bigsqcup C \sqsubseteq d
\end{aligned}
$$

Fixed point axioms

$$
\begin{aligned}
\mu f &= f(\mu f) & \qquad \nu f &= f(\nu f) \\
f(x) \sqsubseteq x &\implies \mu f \sqsubseteq x & \qquad x \sqsubseteq f(x) &\implies x \sqsubseteq \nu f
\end{aligned}
$$

Figure 3.1: Axioms for lattices and fixed points [12]

The non-deterministic choice operator ($\sqcap$) is a specification construct that models the possibility of a program following different paths based on chance. Along with its dual ($\sqcup$), non-deterministic choice satisfies the distributive and continuity properties of the infimum and supremum of a distributive lattice of program commands. The fixed point axioms are basic properties of any complete lattice, and consist of an unfolding and induction lemma for each of the greatest (μ) and least (ν) fixed point operators. These axioms, along with the fixed-point fusion lemmas that follow from them, are sufficient to prove the fundamental properties of iteration ($*$ and ω). A further discussion of the lattice theory can be found in Section 3.5.

3.13.2 Compositional Operator Axioms

Each of the compositional operators on program commands have basic properties that define their behaviour. In addition to these, there are weak interchange laws that allow refinement of specifications toward implementable expressions.

The notation $\{c \in C \bullet f\}$ stands for the set of values of the expression f for c an element of C. Let C be any set of commands and D any non-empty set of commands.

Sequential

$$
\begin{aligned}
c_0 \,;\, (c_1 \,;\, c_2) &= (c_0 \,;\, c_1) \,;\, c_2 \\
c \,;\, \mathbf{nil} &= c \\
\mathbf{nil} \,;\, c &= c \\
c \,;\, (d_0 \sqcap d_1) &= (c \,;\, d_0) \sqcap (c \,;\, d_1) \\
(\bigsqcap C) \,;\, d &= \bigsqcap \{c \in C \bullet c \,;\, d\} \\
(\bigsqcup C) \,;\, d &= \bigsqcup \{c \in C \bullet c \,;\, d\}
\end{aligned}
$$

Parallel

$$
\begin{aligned}
c_0 \parallel (c_1 \parallel c_2) &= (c_0 \parallel c_1) \parallel c_2 \\
c_0 \parallel c_1 &= c_1 \parallel c_0 \\
c \parallel \mathbf{skip} &= c \\
\mathbf{skip} \,;\, \mathbf{skip} &= \mathbf{skip} \\
(\bigsqcap C) \parallel d &= \bigsqcap \{c \in C \bullet c \parallel d\}
\end{aligned}
$$

Strict Conjunction

$$
\begin{aligned}
c_0 \Cap (c_1 \Cap c_2) &= (c_0 \Cap c_1) \Cap c_2 \\
c_0 \Cap c_1 &= c_1 \Cap c_0 \\
c \Cap c &= c \\
c \Cap \mathbf{chaos} &= c \\
\mathbf{chaos} &\sqsubseteq \mathbf{skip} \\
\mathbf{chaos} \parallel \mathbf{chaos} &= \mathbf{chaos} \\
D \neq \emptyset \Longrightarrow c \Cap (\bigsqcap D) &= \bigsqcap \{d \in D \bullet c \Cap d\} \\
c \Cap (\bigsqcup D) &= \bigsqcup \{d \in D \bullet c \Cap d\}
\end{aligned}
$$

Weak interchange axioms

$$
\begin{aligned}
(c_0 \parallel c_1) \Cap (d_0 \parallel d_1) &\sqsubseteq (c_0 \Cap d_0) \parallel (c_1 \Cap d_1) \\
(c_0 \,;\, c_1) \Cap (d_0 \,;\, d_1) &\sqsubseteq (c_0 \Cap d_0) \,;\, (c_1 \Cap d_1)
\end{aligned}
$$

Figure 3.2: Axioms for core language of commands [12]

3.14 Semantic Models

The relationships between Kleene and Omega Algebras and the semantics of wide-spectrum languages have been explored in other works with some discussion of the parallel composition operator. These relationships are expressed through models of program semantics being shown to be instances of relevant programming algebras. The Concurrent Kleene Algebra is applied to a model based on a dependence relation between program commands [15], while the Kleene Algebra has a number of program-based models [3, 26].

A similar trace model to the one considered in this thesis has been proved to be an instance of a Kleene Algebra using the Isabelle/HOL theorem prover [2]. This trace model developed for the Kleene Algebra does not define the possibilities of terminating, aborting and infinite traces and only considers the sequential composition of traces because the concurrency focused operators are not in the algebra. Despite the weaker assumptions and lack of detail, the approach taken in this work is similar in structure to this thesis.

3.15 Semantic Trace Model

The semantic trace model for a wide-spectrum language considered in this thesis [6] provides a semantics for both specification constructs and standard programming language primitives, with each operator in the rely/guarantee algebra defined in the model. The concept of program refinement is central to the model, and is defined as a partial ordering over program commands.

4 Model Formalisation

The semantic trace model [6] represents programs as the set of possible execution paths that a program command can take. These sets can then be combined compositionally to model choice, sequential composition, parallel composition, iteration and the interference specification operators.

Execution paths are defined in the model using a compact representation, based on Aczel traces [1, 8]. Traces are defined as an initial state followed by a sequence of state transitions which specify the result of each atomic operation as a new state. For example, a trace that starts in state σ, makes two program steps then terminates, ending at state σ'', can be expressed as

$$(\sigma, [\text{pgm } \sigma',\ \text{pgm } \sigma'',\ \text{stop}])$$

Steps taken by the environment and the program are explicitly distinguished within the traces because this is important to model parallel composition (see Section 5.4). Terminating behaviour is a separate type of step that must be synchronised with the environment to be feasible. This is separate from aborting behaviour, which is caused by failure and results in undefined behaviour after reaching an undefined program state. A trace is considered infeasible when it is finite and does not terminate or abort, as this not possible in practice. Terminating, aborting and infeasible traces are encompassed by the model in order to match the semantics of real-world concurrent programs and specifications.

4.1 Formalisation Overview

The trace model described in [6] is well-suited to formal representation in Isabelle/HOL, but some key changes are needed to reduce the complexity of the instantiation. The model is constructed from generic program states and potentially infinite lists to represent realistic processes. It is important to prove key properties of each definition as this enables the later proofs to leverage the properties of each datatype and function without the complexity of the full definition.

4.2 Co-inductive Lists

The *llist* (lazy list) data structure in the co-inductive library by Andreas Lochbihler [22] is used to model potentially infinite sequences. This is to allow reasoning about programs such as servers which run indefinitely, and therefore do not terminate.

4.2.1 Key Functions

Most of the relevant functions acting on the *llist* datatype are included with the library, including *lprefix*, *lappend* (@@), and *llast*. One omission is the *lbutlast* function that returns all elements in an *llist* except the last one, which is used extensively in the model. This is defined co-recursively as *llist* is a co-inductive datatype.

```
primcorec lbutlast :: "'a llist ⇒ 'a llist" where
"lbutlast xs = (if lfinite xs then
                    (case xs of [] ⇒ [] |
                          x ## xs' ⇒ (if xs' = [] then [] else (x ## (lbutlast xs'))))
                else xs)"
```

4.2.2 Key Properties

It is useful to have a small library of properties about the *lbutlast* function before attempting to reason about the properties of the definitions built using the function. The properties with the most utility in later proofs are the two variations on *lbutlast_lappend*, which distribute *lbutlast* across the *lappend* operator,

```
lemma lbutlast_lappend: "ys = [] ⟹ lbutlast (xs @@ ys) = lbutlast xs"

lemma lbutlast_lappend2: "ys ≠ [] ⟹ lbutlast (xs @@ ys) = xs @@ lbutlast ys"
```

and *lbutlast_snoc*, which demonstrates cancelling an *lappend* operation.

```
lemma lbutlast_snoc [simp]: "lbutlast (xs @@ [x]) = xs"
  apply (coinduction arbitrary: xs)
  apply (auto)
  apply (metis LNil_eq_lappend_iff llist.distinct(1))
  apply (smt2 lappend_inf lbutlast.code lbutlast.simps(4) lfinite_ltl)
  apply (smt2 lappend.disc(2) lappend_ltl lhd_lappend llist.case_eq_if)
  apply (smt2 lappend.disc(2) lappend_ltl llist.case_eq_if)
     by (metis (mono_tags, lifting) lappend_inf lbutlast.code lbutlast.simps(4) lfinite_ltl)
```

Each of these proofs require co-induction as the *llist* datatype is defined co-inductively. The proof goals generated by applying co-induction all follow from the automatically generated properties of *lbutlast* and other theorems proved in the co-inductive library [22].

Finite *llists* also require an induction law for building up sequences in reverse. This is important for reasoning about the final state of a finite sequence of program steps.

```
lemma lfinite_induct_rev [consumes 1, case_names LNil Lsnoc]:
  assumes lfinite: "lfinite xs"
  and LNil: "⋀xs. lnull xs ⟹ P xs"
  and Lsnoc: "⋀xs x. ⟦ lfinite xs; P xs ⟧ ⟹ P (xs @@ [x])"
  shows "P xs"
```

4.3 Program States

The Isabelle/HOL formalisation of states is kept as general as possible to avoid restricting the theory to a specific memory model. Aborting states are used to represent undefined states after which the behaviour of the program is entirely undefined. The parameter *'a* denotes a generic type variable, which is an extension of higher-order logic in Isabelle to allow polymorphism and type-inference [27].

```
datatype ('a) Σa =
    state 'a
|   abortState
```

datatype declarations in Isabelle/HOL define new types based on generic types, which is appropriate for this situation as it places no restriction on the form of the actual program states being modelled.

4.4 Program Steps

Program steps are defined as a transition between two states, and are partitioned into three categories: program steps, environment steps and termination steps.

```
datatype ('σ) step =
    pgm 'σ
|   env 'σ
|   stop
```

4.5 Tseqs and Pre_Tseqs

Sequences of program steps are central to representing programs as the set of their possible traces. These sequences have a number of properties that must be specified in order to correctly model real-world programs.

Sequences of program steps must conform to healthiness conditions so they are restricted to a subset of the possible *llists* of program steps. This is defined in Isabelle/HOL using the *typedef* declaration style (see Section 2.1). Only the last program step of a sequence can be a termination or aborting step, as after this point there can be no defined execution path. This constraint is expressed through a Pre_Tseq, a sequence containing no aborting or terminating steps, being all but the last element of any legitimate Tseq.

```
typedef ('a) Pre_Tseq = "{(t::(('a) Σa) step llist). (∀ s ∈ (lset t). s ∉ {stop,
(pgm abortState), (env abortState)})}"

typedef ('a) Tseq = "{(t::(('a) Σa) step llist). (∃ s. (lbutlast t) = (Rep_Pre_Tseq
s))}"
```

The automatically generated *Rep_Pre_Tseq* mutator, that converts the abstract type into a concrete representation (in this case an *llist*), is used in the definition of a Tseq. The non-empty set obligation of these declarations (see Section 2.1) are trivial to prove as the empty *llist* is in both sets.

4.5.1 Key Properties

The most important properties for Tseqs and Pre_Tseqs concern the concatenation of, and transformation between the two types.

```
lemma pre_tseq_lappend: "∃ t'. (Rep_Pre_Tseq a) @@ (Rep_Pre_Tseq b) = Rep_Pre_Tseq
t'"

lemma tseq_lappend: "(∃ t'. (lbutlast (Rep_Tseq a)) @@ (Rep_Tseq b) = Rep_Tseq t')"

lemma pre_tseq_to_tseq: "(∃ t'. (Rep_Pre_Tseq t) @@ [a] = (Rep_Tseq t'))"

lemma tseq_to_pre_tseq: "(∃ t'. (lbutlast (Rep_Tseq t)) = (Rep_Pre_Tseq t'))"
```

The proofs for these properties are trivial, but establishing these rules simplify later proofs significantly as sequential composition involves concatenation of Tseqs and Pre_Tseqs in the trace fusion operation.

4.6 Traces

A trace is simple to define at this point, as all the required building blocks are already in place. A trace consists of a starting state and a sequence of program steps. The *type_synonym* declaration creates a alias for a combination of other types.

```
type_synonym ('a) Tr = "'a × ('a) Tseq"
```

4.7 Startstate and Laststate

The initial and final states of a trace are important for reasoning about compositional operators, as the corresponding states must match up for the resulting trace to be feasible. The initial state is trivial to define.

```
definition startstate :: "('a) Tr ⇒ 'a" where
  "startstate t ≡ fst t"
```

The final state is only defined for finite sequences, so *laststate* is a partial function. The definition is split up into two stages to simplify the proofs for key properties about this function. *laststate_seq* is defined over all sequences of steps so the well-formedness of traces is not considered at this point. The *domintros* flag is included in the definition to indicate a partial-function and generate a domain intro rule for each case of the function definition.

```
function (domintros) laststate_seq :: "('a × ('a Σa step llist)) ⇒ ('a Σa)" where

  "laststate_seq (σ, []) = state σ"
| "laststate_seq (σ, xs @@ [x]) = (if (lfinite (xs)) then (case (x) of
                                      pgm σ' ⇒ σ'|
                                      env σ' ⇒ σ'|
                                      stop   ⇒ (laststate_seq (σ, xs))
                                  ) else undefined)"
```

The first case is trivial, as if there are no program steps, the final state and initial state are the same. The second case is more complicated as there are a number of possible situations to consider. Recursion is needed for terminating traces, as the final state is the state before the *stop* step, while infinite traces return an undefined result. Finally, finite traces ending with a regular program or environment step return a straightforward result.

The actual definition of *laststate* follows easily from the definition of *laststate_seq*.

```
definition laststate :: "('a) Tr ⇒ ('a Σa)" where
  "laststate t = laststate_seq (startstate t, (steps t))"
```

4.7.1 Key Properties

Many other proofs require that *laststate_seq* terminates over an appropriate domain. As the recursive call is for terminating traces, finite traces ending in *stop* can be shown to be in the domain of the function. With this shown to hold for all terminating traces, partial simplification properties can be used as the domain introduction predicate is satisfied.

```
lemma laststate_dom: "⟦lfinite xs ; llast xs = stop⟧ ⟹ laststate_seq_dom (σ, xs)"
  apply (induct rule: lfinite_induct_rev)
  apply (metis lappend.disc(2) laststate_seq.domintros(1) llist.collapse(1) llist.disc(2))
     by (metis (erased, lifting) lappend_lbutlast_last_id laststate_seq.domintros(1)
laststate_seq.domintros(2) lbutlast_snoc llast_lappend_LCons llast_singleton)
```

The proof is completed by applying the reverse induction of finite *llists* discussed earlier, which breaks the proof down into two subgoals. Both the base case and inductive step are then straightforward to prove with previous properties of *llists* and the properties created by the definition of *laststate_seq*. Of particular interest, both subgoals require the *laststate_seq.domintros* rules as these govern the domain of the function definition.

4.8 Processes

Processes are sets of traces that satisfy healthiness requirements. These requirements need to be constructed before the set of well-formed processes can be defined. As these definitions deal with sets of traces, the morphisms between abstract and base-type representations of Tseqs and Pre_Tseqs are required. Essentially, *Rep_Tseq* and *Rep_Pre_Tseq* convert an abstract sequence into a list of steps that will always satisfy the properties of that sequence type. *Abs_Tseq* and *Abs_Pre_Tseq* are the reverse operation, converting a list of steps into an abstract sequence, and require proof that the list satisfies the properties of that sequence type.

4.8.1 Prefix Closure

A key element in this definition is the prefix closure, which adds in all the possible traces that share the initial state and have a sequence of steps that is a prefix of each existing trace in the set.

```
definition pc :: "('a Tr) set ⇒ ('a Tr) set" where
  "pc s ≡ {(σ, t) | σ t. (∃ t'. ( (σ, t') ∈ s
                        ∧ (lprefix (Rep_Tseq t) (Rep_Tseq t'))))
                        ∨ (Rep_Tseq t) = []}"
```

The *lprefix* function used in this definition is a non-strict prefix, which allows the definition in the original model [6] to be simplified. All the traces in the original set are preserved by the non-strict prefix function so it is redundant to explicitly include them in the definition.

4.8.2 Abort Closure

The other healthiness property for processes is abort closure. After a program aborts, the behaviour of the program is undefined from that point on. Therefore, each trace in the set ending in abort has all possible execution paths from this point are added in by the abort closure.

```
definition ac  :: "('a Tr) set ⇒ ('a Tr) set" where
   "ac s ≡ s ∪ { (σ, t) | σ t. (∃ t'. (lprefix (Rep_Pre_Tseq t') (Rep_Tseq t))
                                    ∧ lfinite (Rep_Pre_Tseq t')
              ∧ ((σ, Abs_Tseq ((Rep_Pre_Tseq t') @@ [pgm abortState])) ∈ s))}"
```

4.8.3 Process Definition

A process is then defined as the subset of sets of traces which are both prefix and abort closed using a *typedef* declaration.

```
typedef ('a) Pr = "{(s::'a Tr set). (s = pc s) ∧ (s = ac s)}"
```

4.8.4 Key Properties

The prefix closure and abort closure functions distribute over union and intersection.

```
lemma pc_union_eq: "pc (s ∪ t) = pc s ∪ pc t"

lemma ac_union_eq: "ac (s ∪ t) = ac s ∪ ac t"
```

The order that they are applied has no impact, and applying either closure twice is equivalent to applying it once.

```
lemma ac_pc_order: "ac (pc s) = pc (ac s)"

lemma ac_double: "ac (ac s) = ac s"

lemma pc_double: "pc (pc s) = pc s"
```

4.9 Primitive Commands

It is useful to define primitive commands which can be built into more interesting programs. The obvious primitives are single program, environment and terminating steps lifted to the command level. These can be used to describe any program with a single step satisfying a relation, r, between program states.

definition *π :: "('a, 'a) relation ⇒ 'a Pr"*
where *"π (r) ≡ Abs_Pr (pc {(σ, Abs_Tseq [pgm (state σ'), stop]) | σ σ'. (σ, σ') ∈ r})"*

definition *ε :: "('a, 'a) relation ⇒ 'a Pr"*
where *"ε (r) ≡ Abs_Pr (pc {(σ, Abs_Tseq [env (state σ'), stop]) | σ σ'. (σ, σ') ∈ r})"*

definition *τ :: "'a predicate ⇒ 'a Pr"*
where *"τ (p) ≡ Abs_Pr (pc {(σ, Abs_Tseq [stop]) | σ. σ ∈ p})"*

The top and bottom of the lattice of program commands are defined specifically as primitive commands. The greatest element, **magic**, is defined as the set of all possible starting states with empty sequences.

definition *magic :: "'a Pr"*
where *"magic ≡ τ({})"*

Similarly, the least element, **abort**, is defined as the set of all possible traces.

definition *abort :: "'a Pr"*
where *"abort ≡ Abs_Pr {(s::'a Tr) . True}"*

Finally, **nil** is defined as the set of all possible starting states with either an empty sequence or an immediately terminating step.

definition *nill :: "'a Pr"*
where *"nill ≡ τ({σ . True})"*

4.10 Termination

The concept of terminating traces is important in the model, especially for sequentially composing traces. A trace is classed as terminating when it is finite and ends in a *stop* step.

definition *terminates :: "'a Tr ⇒ bool"* **where**
"terminates t ≡ (lfinite (steps t)) ∧ llast (steps t) = stop ∧ (steps t ≠ []) "

As processes are represented as sets of traces, it is useful to split sets of traces into the terminating and non-terminating subsets. These sets are straightforward to define.

definition *𝒯 :: "'a Tr set ⇒ 'a Tr set"* **where**
"𝒯 A = {t. t ∈ A ∧ terminates t}"

definition *n𝒯 :: "'a Tr set ⇒ 'a Tr set"* **where**
"n𝒯 A = {t. t ∈ A ∧ ¬(terminates t)}"

4.10.1 Key Properties

The rule to split any set of traces into these two subsets follows directly from their definitions.

lemma *Tr_set_term_split: "s = (𝒯 s) ∪ (n𝒯 s)"*

5 Operator Formalisation

Each of the operators in the rely/guarantee algebra is defined on program commands in the concrete model. These definitions are designed to accurately represent the semantics of combining real-world programs, while exhibiting the properties of the corresponding operators in the algebra. As the model represents programs as a set of possible execution paths, the paths of both original programs are combined to reflect the composition of every possible path in the result.

The primitive commands in the model are combined using compositional operators. Some of the commands are specification constructs and can not be directly implemented in practice. These specification operators improve the expressiveness of the language and allow reasoning about the interactions between a program and its environment.

5.1 Refinement

Refinement in the model is defined using trace inclusion. A program, C, is refined by another, D, if its set of possible traces are a superset of the other programs set of traces. Refinement forms a partial ordering over the lattice of program commands and requires no extra properties to be proved, as all lemmas relating to subset inclusion are directly applicable.

definition *refinement :: "'a Pr ⇒ 'a Pr ⇒ bool" (***infix** *"⊑" 50)* **where**
"refinement c d ≡ (Rep_Pr c ⊇ Rep_Pr d)"

5.2 Lattice

The non-deterministic choice operator is used to model conditional statements and other non-deterministic control structures. In algebraic theory, it becomes the infimum operator in the lattice formed over program commands. As the program can take any of the possible paths in the composed commands, the respective sets of possible traces are combined using the union operator. The binary case is very simple.

definition *nd_choice :: "'a Pr ⇒ 'a Pr ⇒ 'a Pr" (***infix** *"⊓⊓" 65)* **where**
"nd_choice c_1 c_2 *≡ Abs_Pr ((Rep_Pr* c_1*) ∪ (Rep_Pr* c_2*))"*

There is also a generalised operator, which acts on a set of program commands. In the original model, the prefix closure is applied to the result as this causes the empty set to return **magic**, rather than the empty set which is not a valid process. This introduced considerable complexity to reasoning about the operator, so the alternative of taking the union of the result and **magic** is used instead. This is an equivalent definition as **magic** is the identity of non-deterministic choice.

definition *nd_choice_gen :: "'a Pr set ⇒ 'a Pr" ("⨅⨅_" [900] 901)* **where**
"nd_choice_gen C ≡ Abs_Pr (⋃ { (Rep_Pr c) | c. c ∈ C} ∪ (Rep_Pr magic))"

The duals of these two operators, the angelic choice operators, also exist in the algebra. Naturally, this leads to similar definitions using intersection rather than union to achieve a supremum operator on the lattice of program commands. The angelic choice operators are strictly specification operators, with no direct relation to any programming language constructs.

```
definition a_choice :: "'a Pr ⇒ 'a Pr ⇒ 'a Pr" (infix "⊔⊔" 70) where
  "a_choice c₁ c₂ ≡ Abs_Pr ((Rep_Pr c₁) ∩ (Rep_Pr c₂))"

definition a_choice_gen :: "'a Pr set ⇒ 'a Pr" ("⨆⨆_" [900] 901) where
  "a_choice_gen C ≡ Abs_Pr (⋂ { (Rep_Pr c) | c. c ∈ C} ∪ (Rep_Pr magic))"
```

5.2.1 Key Properties

The most important properties for the lattice operators deal with the language of commands being closed under each operator. The binary case is quite simple, requiring only the distributive properties of prefix and abort closure for the proof.

```
lemma Pr_union: "Rep_Pr (Abs_Pr (Rep_Pr c ∪ Rep_Pr d)) = Rep_Pr c ∪ Rep_Pr d"
  by (metis (mono_tags, lifting) ac_union_eq pc_union_eq Abs_Pr_inverse Rep_Pr mem_Collect_eq)
```

The general case needs to be split into two cases so that the empty set can be dealt with separately. Each of the subgoals are straightforward to prove.

```
lemma Inf_Pr: "∃ D. Rep_Pr D = ⋃ {Rep_Pr c |c. c ∈ C} ∪ (Rep_Pr magic)"
  apply (cases "C = {}")
  apply (smt2 Collect_empty_eq Union_empty emptyE sup_bot_left)
     by (metis (no_types) Union_closed magic_unit)
```

The dual operators have identical lemmas and proofs for the corresponding properties.

5.3 Sequential Composition

Programs executed sequentially are combined compositionally using the concept of trace fusion. A terminating trace has all but the termination step concatenated with a second trace to form a trace consisting of both sequences of steps. This operation is undefined in cases where the second trace does not start in the same state that the first ends at, as this would result in inconsistent traces.

Trace fusion builds on the definitions of *laststate*, *startstate* and *terminates*.

```
definition Tr_fusion :: "'a Tr ⇒ 'a Tr ⇒ 'a Tr" where
  "Tr_fusion t g ≡ (if ¬(terminates t) then t else
                    (if (laststate t = state (startstate g))
                    then (startstate t, Abs_Tseq ((lbutlast (steps t)) @@ (steps g)))
                    else undefined))"
```

Trace fusion is then lifted up to the set level, with *Tr_set_fusion* being the set of all terminating traces in the first process fused with all the traces in the second. Non-terminating traces in the first set are retained in the fusion.

```
definition Tr_set_fusion :: "'a Tr set ⇒ 'a Tr set ⇒ 'a Tr set" (infix "⌢" 90) where
    "Tr_set_fusion f g ≡ { Tr_fusion a b | a b . (a ∈ f) ∧ (b ∈  g)
                        ∧ ((terminates a) ⟶ (state (startstate b) = laststate a))}"
```

Finally, the abort closure is applied to the result in order to ensure the sequential composition of two processes is also a process.

```
definition Pr_fusion :: "'a Pr ⇒ 'a Pr ⇒ 'a Pr" (infix ";;" 90) where
  "Pr_fusion f g ≡ Abs_Pr (ac ((Rep_Pr f) ⌢ (Rep_Pr g)))"
```

5.3.1 Key Properties

The fusion of sets of traces distributes across deterministic choice, which follows directly from the definition.

lemma *Tr_set_fusion_dist_r: "(f ∪ g) ⌢ h = (f ⌢ h) ∪ (g ⌢ h)"*

lemma *Tr_set_fusion_dist_l: "h ⌢ (f ∪ g) = (h ⌢ f) ∪ (h ⌢ g)"*

Properties of the abort and prefix closure distributing over process fusion are key for proving that the algebra is closed for commands under sequential composition.

lemma *Tr_set_fusion_to_Pr: "∃ C. ac ((Rep_Pr f) ⌢ (Rep_Pr g)) = Rep_Pr C"*

5.4 Parallel Composition

Composing commands in parallel requires a synchronisation between the steps of two processes. This is achieved by synchronising a program step in one trace with an environment step transition to the same state in the other. The special case of terminating steps requires that a *stop* step can only synchronise with another *stop* step.

This definition is broken up into four parts to reduce the complexity of individual proofs. First, the cases for matching individual steps are considered. This definition effectively matches steps f and g to return the result of h.

```
definition match :: "'a Σa step ⇒ 'a Σa step ⇒ 'a Σa step ⇒ bool" where
    "match f g h ≡ ∃ σ. (f = pgm σ ∧ g = env σ ∧ h = pgm σ) ∨
                        (f = env σ ∧ g = pgm σ ∧ h = pgm σ) ∨
                        (f = env σ ∧ g = env σ ∧ h = env σ) ∨
                        (f = stop ∧ g = stop ∧ h = stop)"
```

Next, the definition of match is applied recursively to match two sequences together. For sequences to match, they must be of the same length. The pattern matching capability of Isabelle/HOL is leveraged in this definition to simplify reasoning about the recursive case to only consider equally long sequences. While the definition does not look optimal, what is lost in conciseness here is made up for in the remaining proofs.

```
function (sequential, domintros) match_seq  :: "'a Σa step llist ⇒ 'a Σa step llist
⇒ 'a Σa step llist ⇒ bool" where
    "match_seq ([]) (g##gs) (h##hs) = False"
|   "match_seq (f##fs) ([]) (h##hs) = False"
|   "match_seq ([]) ([]) (h##hs) = False"
|   "match_seq (f##fs) (g##gs) ([]) = False"
|   "match_seq (f##fs) ([]) ([]) = False"
|   "match_seq ([]) (g##gs) ([]) = False"
|   "match_seq ([]) ([]) ([]) = True"
|   "match_seq (f##fs) (g##gs) (h##hs) = (if (match f g h) then (match_seq fs gs hs)
else False)"
```

The parallel composition of two sets of traces are then defined by lifting the definition of *match_seq* to the set level, without consideration of whether the result constitutes a process.

```
definition Parallel_Comp_set :: "'a Tr set ⇒ 'a Tr set ⇒ 'a Tr set" (infix "||" 90)
where
    "Parallel_Comp_set f g ≡ ({(σ, t) | σ t. (∃ t' t''. (σ, t') ∈ f
          ∧ (σ, t'') ∈ g ∧ (match_seq (Rep_Tseq t') (Rep_Tseq t'') (Rep_Tseq t)))})"
```

Finally, the abort closure is applied to the result of the previous definition to ensure that parallel composition between two processes always results in a process.

```
definition Parallel_Comp :: "'a Pr ⇒ 'a Pr ⇒ 'a Pr" (infix "|||" 90) where
    "Parallel_Comp f g ≡ Abs_Pr (ac ((Rep_Pr f) || (Rep_Pr g)))"
```

6 Algebra Instantiation

Each axiom of an algebra needs to be proved to hold for the model to show it is an instantiation of that algebra. In Isabelle/HOL, these proof obligations can be separated into subsections called locales. The different operators provide logical groupings for the axioms. The full list of axioms for the rely/guarantee algebra can be found in Figures 3.1 and 3.2.

For clarity, the *lemma* keyword is used for non-axiom properties and *theorem* is used for axioms. It is important to note that there is no functional difference between the two commands in Isabelle/HOL.

The *interpretation* command (discussed in Section 2.4) completes the instantiation process using the theorems corresponding to the axioms for the current locale.

6.1 Refinement

The three basic properties that define a partial ordering follow directly from the definition of refinement as subset inclusion (a partial ordering itself).

theorem *refine_refl [iff]:* "$c \sqsubseteq c$"

theorem *refine_trans [trans]:* "$[\![c_0 \sqsubseteq c_1;\ c_1 \sqsubseteq c_2]\!] \Longrightarrow c_0 \sqsubseteq c_2$"

theorem *refine_antisym [intro]:* "$[\![c_0 \sqsubseteq c_1;\ c_1 \sqsubseteq c_0]\!] \Longrightarrow c_0 = c_1$"

6.2 Lattice

The non-deterministic choice operator and its dual satisfy the axioms for the infimum and supremum operators in the rely/guarantee algebra. The basic definition of the infimum operator on a lattice follows from the definition of non-deterministic choice as the union of two sets.

theorem *refine_inf_def:* "$(c \sqsubseteq d) \longleftrightarrow (c \sqcap\sqcap d = c)$"

The proofs for the lattice operators all follow a similar style of Isar proof so a single representative proof is presented here.

The infimum lower bound axiom effectively states that if every command in C is refined by a command d, the generalised non-deterministic choice of C is also refined by the command d.

theorem *inf_lb:* "$(\forall c \in C.\ d \sqcap\sqcap c = d) \Longrightarrow d \sqcap\sqcap (\bigsqcap\bigsqcap C) = d$"
proof *cases*

The Isar proof is declared using the *cases* proof structure. This allows the case when C is empty to be handled separately to when it is not.

The non-empty case is the more involved proof, which begins by expanding the starting assumption to use subset inclusion rather than the equivalent infimum definition. This results in the statement labelled *refines*, which is used in the final step of the proof. Next the definitions of the binary and generalised non-deterministic choice are expanded out and the resulting expression is simplified. Finally, the *refines* expression is used to absorb **magic** and the generalised non-deterministic choice of C into the command d.

```
  assume not_empty: "C ≠ {}"

  assume "(∀c ∈ C. d ⊓⊓ c = d)"
  then have "(∀c ∈ C. (Rep_Pr c) ⊆ (Rep_Pr d))"
              by (metis Lattice_Sem.refine_inf_def refinement_def)
  then have refines: "(⋃ {(Rep_Pr c) | c. c ∈ C} ∪ (Rep_Pr magic)) ⊆ (Rep_Pr d)"
              by (smt magic_bot sup.boundedI Sup_least mem_Collect_eq)

  then have "d ⊓⊓ (⨅⨅ C) = Abs_Pr ((Rep_Pr d) ∪ (Rep_Pr (Abs_Pr
                           (⋃ { (Rep_Pr c) | c. c ∈ C} ∪ (Rep_Pr magic)))))"
              by (metis nd_choice_def nd_choice_gen_def)
  then have "d ⊓⊓ (⨅⨅ C) = Abs_Pr ((Rep_Pr d) ∪
                           (⋃ { (Rep_Pr c) | c. c ∈ C} ∪ (Rep_Pr magic)))"
              by (metis Inf_Pr Pr_Union)
  thus ?thesis by (metis (erased, lifting) refines Rep_Pr_inverse Un_absorb2)
next
```

The empty set case is fairly straightforward as the generalised non-deterministic choice of C simplifies to **magic**, which is then absorbed into d as in the first case.

```
  assume empty: "¬ (C ≠ {})"
  then have "d ⊓⊓ (⨅⨅ C) = d ⊓⊓ Abs_Pr
                           (⋃ { (Rep_Pr c) | c. c ∈ C} ∪ (Rep_Pr magic))"
              by (metis nd_choice_gen_def)
  then have "d ⊓⊓ (⨅⨅ C) = d ⊓⊓ magic"
              by (smt2 Lattice_Sem.inf_idemp Sup_empty empty empty_Collect_eq equals0D
inf_sup_aci(5) magic_unit nd_choice_def nd_choice_gen_def sup_bot.right_neutral)
  thus ?thesis by (metis inf_top)
qed
```

Finally, the proved theorem is used in the interpretation command, along with the rest of the axioms, to complete the instantiation for the *Dist_Lattice* locale of the algebra.

```
interpretation Lattice_Sem : Dist_Lattice "op ⊑" magic abort "op ⊓⊓" nd_choice_gen
"op ⊔⊔" a_choice_gen
  apply (unfold_locales)
  apply (auto simp add: refine_inf_def)
  apply (auto simp add: inf_assoc)
  apply (auto simp add: inf_commu)
  apply (auto simp add: inf_idemp)
  apply (auto simp add: inf_top)
  apply (auto simp add: inf_glb)
  apply (auto simp add: inf_lb)
  apply (auto simp add: sup_assoc)
  apply (auto simp add: sup_commu)
  apply (auto simp add: sup_idemp)
  apply (auto simp add: sup_bot)
  apply (auto simp add: absorption1)
  apply (auto simp add: absorption2)
  apply (auto simp add: sup_ub)
  apply (auto simp add: sup_lub)
  by (auto simp add: sup_dist)
```

6.3 Sequential Composition

Most proofs for sequential composition have a similar overarching structure due to the multiple levels of the definition of process fusion. A corresponding property for trace fusion is proved, then lifted to the process level. Finally, the restrictions on valid processes are considered on top of the set definition to complete the proof for the axiom.

6.3.1 Magic as a Left Annihilator

The axiom describing **magic** as a left annihilator of sequential composition is a succinct representation of this general method. A single trace with an empty sequence of steps is fused with a generic trace, which trivially results in just the first trace as it does not terminate.

```
lemma Tr_fusion_magic: "Tr_fusion (σ, Abs_Tseq []) t = (σ, Abs_Tseq [])"
  by (metis Rep_Tseq_inverse Tr_fusion_anni empty_tseq non_empty_terminate snd_conv
steps_def)
```

Lifting this property to the set level is more involved. The *auto* simplifier expands out the definitions of *Tr_set_fusion* and **magic**. This leaves two subgoals, the first of which requires the *tr_fusion_magic* lemma.

```
lemma Tr_set_fusion_magic [simp]: "(Rep_Pr magic) ⌢ (Rep_Pr c) = (Rep_Pr magic)"
  apply (auto simp add: Tr_set_fusion_def magic_empty pc_empty Tr_fusion_anni)
  apply (smt2 Tr_fusion_def empty_nterminate snd_conv steps_def Tr_fusion_magic)
    by (smt2 Pr_pc Tr_fusion_anni mem_Collect_eq pc_def snd_conv steps_def terminates_def)
```

Finally, **magic** is shown to be a left annihilator of sequential composition, which effectively is the previous lemma with the abort closure included. This has no effect in this situation but does complicate the proofs for other axioms.

```
theorem seq_anni_magic [simp]: "magic;;c = magic"
  by (metis Pr_ac Pr_fusion_def Rep_Pr_inverse Tr_set_fusion_magic)
```

While each stage of the proof here is relatively straightforward, the proofs for certain axioms, such as associativity, are very involved. This results in each of the stages needing numerous individual lemmas to build up to a usable property of each level of the definition.

6.3.2 Associativity

Due to the complex structure of the associativity proof for sequential composition, multiple dependency diagrams are useful for illustrating the relationships and dependencies between lemmas. Although associativity itself is a simple property, the complexity of the model and sequential composition operator dictate that the proof is quite involved. Each of the dependency diagrams combines properties of traces, trace sets, trace fusion and processes as each construct is built off the other lower level definitions.

The top level diagram (Figure 6.1) deals primarily with combining the associativity of *Tr_set_fusion* with closure properties about processes and abort closure.

The subtree found in Figure 6.2 illustrates the variety of properties required to prove the associativity of the *Tr_set_fusion* operator. These range from lemmas about *laststate*, *startstate* and their interactions with *Tr_fusion*, to lifting the associativity of *Tr_fusion* to the set level, with consideration of terminating and non-terminating subsets of sets of traces. Most of the lemmas in this subtree are specific to associativity and have minimal utility in other proofs.

Contrastingly, the subtree in Figure 6.3 is a general closure property of sequential composition. This property is used in the majority of proofs for sequential composition. This can be logically broken down into the requirement that the *Tr_set_fusion* of processes is already prefix closed and that the abort closure in the definition of sequential composition fulfils the other healthiness condition. These two subsections of the proof can be seen in the left and right subtrees in the diagram, respectively.

The dependency diagrams are structured with the dashed lines indicating a subtree exists in a separate diagram and similar lemmas grouped visually using coloured boxes. Each diagram is followed by a complete listing of the lemmas included in that diagram.

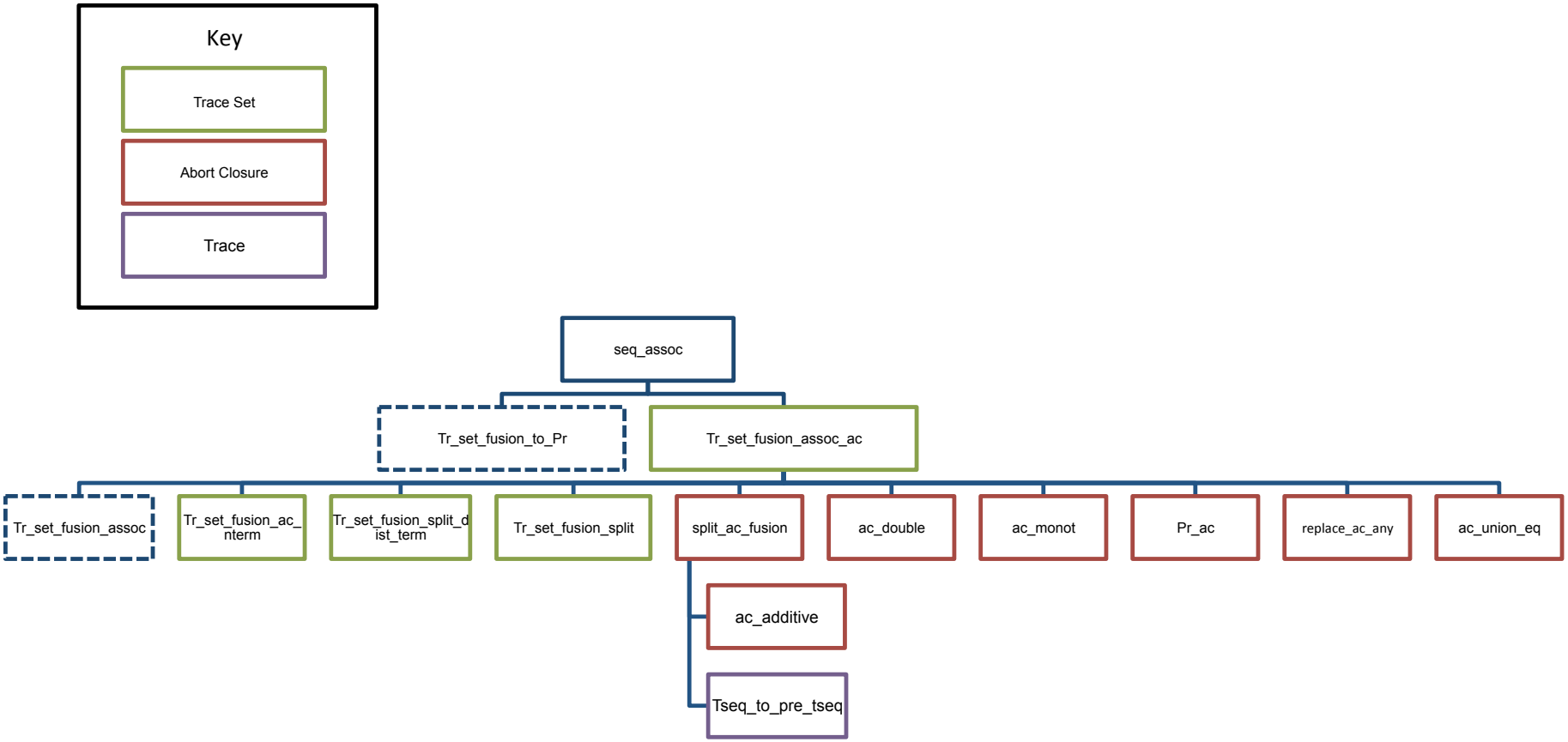

Figure 6.1: Dependency Diagram for associativity property of sequential composition operator

Lemmas Used In Figure 6.1

theorem *seq_assoc:* "$c_0;;(c_1;;c_2) = (c_0;;c_1);;c_2$"

lemma *Tr_set_fusion_to_Pr:* "$\exists$ C. ac ((Rep_Pr f) $\frown$ (Rep_Pr g)) = Rep_Pr C"

lemma *Tr_set_fusion_assoc_ac:* "ac ((ac ((Rep_Pr a) $\frown$ (Rep_Pr b))) $\frown$ (Rep_Pr c))
= ac ((Rep_Pr a) $\frown$ (ac ((Rep_Pr b) $\frown$ (Rep_Pr c))))"

lemma *Tr_set_fusion_assoc:*
"((Rep_Pr a) $\frown$ (Rep_Pr b)) $\frown$ (Rep_Pr c)
= (Rep_Pr a) $\frown$ ((Rep_Pr b) $\frown$ (Rep_Pr c))"

lemma *Tr_set_fusion_ac_nterm:* "g $\neq$ {} $\Longrightarrow$ ac (n$\mathcal{T}$ f) $\supseteq$ (ac (n$\mathcal{T}$ f)) $\frown$ g"

lemma *Tr_set_fusion_split_dist_term:* "g $\neq$ {} $\Longrightarrow$ f $\frown$ g = ($\mathcal{T}$ f) $\frown$ g $\cup$ (n$\mathcal{T}$ f)"

lemma *Tr_set_fusion_split:*
"((Rep_Pr a) $\frown$ (Rep_Pr b)) $\frown$ (Rep_Pr c)
= ((Rep_Pr a) $\frown$ ($\mathcal{T}$ (Rep_Pr b))) $\frown$ (Rep_Pr c)
$\cup$ ((Rep_Pr a) $\frown$ (n$\mathcal{T}$ (Rep_Pr b))) $\frown$ (Rep_Pr c)"

lemma *split_ac_fusion:*
"ac (Rep_Pr f $\frown$ Rep_Pr g)
= Rep_Pr f $\frown$ Rep_Pr g $\cup$ ac (n$\mathcal{T}$ (Rep_Pr f))
$\cup$ $\mathcal{T}$ (Rep_Pr f) $\frown$ ac ((Rep_Pr g))"

lemma *ac_additive:* "ac s $\supseteq$ s"

lemma *pre_tseq_to_tseq:* "($\exists$ t'. (Rep_Pre_Tseq t) @@ [a] = (Rep_Tseq t'))"

lemma *ac_double:* "ac (ac s) = ac s"

lemma *ac_monot:* "s $\supseteq$ t $\Longrightarrow$ ac s $\supseteq$ ac t"

lemma *Pr_ac:* "ac (Rep_Pr c) = Rep_Pr c"

lemma *replace_ac_any:*
"lfinite (Rep_Pre_Tseq b)
$\Longrightarrow$ (a, Abs_Tseq (Rep_Pre_Tseq b @@ [pgm abortState])) $\in$ s
$\Longrightarrow$ lprefix (Rep_Pre_Tseq b) (Rep_Tseq c)
$\Longrightarrow$ (a, c) $\in$ ac (s)"

lemma *ac_union_eq:* "ac (s $\cup$ t) = ac s $\cup$ ac t"

Figure 6.2: Dependency Diagram for associativity property of trace set fusion operator

Lemmas Used In Figure 6.2

lemma *Tr_set_fusion_assoc:*
```
  "((Rep_Pr a) ⌢ (Rep_Pr b)) ⌢ (Rep_Pr c)
  = (Rep_Pr a) ⌢ ((Rep_Pr b) ⌢ (Rep_Pr c))"
```

lemma *laststate_Tr_fusion [simp]:*
```
   "⟦ lfinite (steps y); terminates y; terminates x ⟧
⟹ laststate x = state (startstate y)
⟹ laststate (Tr_fusion x y) = laststate y"
```

lemma *laststate_Tr_fusion_stop:*
```
   "terminates x ⟹ laststate x = state σ
⟹ laststate (Tr_fusion x (σ, Abs_Tseq [stop])) = state σ"
```

lemma *laststate_forwards:* "(lfinite (Rep_Pre_Tseq t)) ⟹ a ≠ stop ⟹ laststate (σ, Abs_Tseq ((Rep_Pre_Tseq t) @@ [a])) = laststate (σ, Abs_Tseq [a])"

lemma *laststate_env:* "(lfinite (Rep_Pre_Tseq t)) ⟹ laststate (σ, Abs_Tseq ((Rep_Pre_Tseq t) @@ [env σ'])) = σ'"

lemma *pre_tseq_to_tseq:* "(∃ t'. (Rep_Pre_Tseq t) @@ [a] = (Rep_Tseq t'))"

lemma *laststate_pgm:* "(lfinite (Rep_Pre_Tseq t)) ⟹ laststate (σ, Abs_Tseq ((Rep_Pre_Tseq t) @@ [pgm σ'])) = σ'"

lemma *startstate_Tr_fusion [simp]:* "laststate x = state (startstate y) ⟹ startstate (Tr_fusion x y) = startstate x"

lemma *lift_Tr_fusion_assoc:*
```
  "{ Tr_fusion (Tr_fusion a b) c | a b c . terminates a ∧ terminates b
   ∧ (a ∈ Rep_Pr A) ∧ (b ∈ Rep_Pr B) ∧ (c ∈ Rep_Pr C)
   ∧ (laststate a = state (startstate b))
   ∧ (laststate b = state (startstate c))  }
 = { Tr_fusion a (Tr_fusion b c) | a b c . terminates a ∧ terminates b
   ∧ (a ∈ Rep_Pr A) ∧ (b ∈ Rep_Pr B) ∧ (c ∈ Rep_Pr C)
   ∧ (laststate b = state (startstate c))
   ∧ (laststate a = state (startstate b)) }"
```

lemma *Tr_fusion_assoc:* "laststate c_0 = state (startstate c_1) ⟹ laststate c_1 = state (startstate c_2) ⟹ Tr_fusion c_0 (Tr_fusion c_1 c_2) = Tr_fusion (Tr_fusion c_0 c_1) c_2"

lemma *tseq_lappend:* "(∃ t'. (lbutlast (Rep_Tseq a)) @@ (Rep_Tseq b) = Rep_Tseq t')"

lemma *laststate_stop:* "(lfinite (Rep_Pre_Tseq t)) ⟹ laststate (σ, Abs_Tseq ((Rep_Pre_Tseq t) @@ [stop])) = laststate (σ, Abs_Tseq (Rep_Pre_Tseq t))"

lemma *Tr_fusion_anni:* "¬(terminates t) ⟹ Tr_fusion t g = t"

lemma *Tr_fusion_terminates:*
```
  "⟦ terminates x; terminates y; laststate x = state (startstate y) ⟧
⟹ terminates (Tr_fusion x y)"
```

lemma *Tr_set_fusion_split_dist_term:* "g ≠ {} ⟹ f ⌢ g = (𝒯 f) ⌢ g ∪ (n𝒯 f)"

lemma *Tr_set_fusion_dist_r:* "(f ∪ g) ⌢ h = (f ⌢ h) ∪ (g ⌢ h)"

lemma *Tr_set_fusion_dist_l:* "h ⌢ (f ∪ g) = (h ⌢ f) ∪ (h ⌢ g)"

lemma *Tr_set_fusion_anni:* "g ≠ {} ⟹ (n𝒯 f) ⌢ g = n𝒯 f"

lemma *Tr_set_term_split:* "s = (𝒯 s) ∪ (n𝒯 s)"

lemma *Tr_set_fusion_split2:*
```
  "((𝒯 (Rep_Pr a)) ⌢ (n𝒯 (Rep_Pr b))) ⌢ (Rep_Pr c)
  = (𝒯 (Rep_Pr a)) ⌢ (n𝒯 (Rep_Pr b))"
```

lemma *Tr_set_fusion_nterm2:* "⟦ (n𝒯 g) ≠ {}; h ≠ {} ⟧
```
                        ⟹ (f ⌢ (n𝒯 g)) ⌢ h = f ⌢ (n𝒯 g)"
```

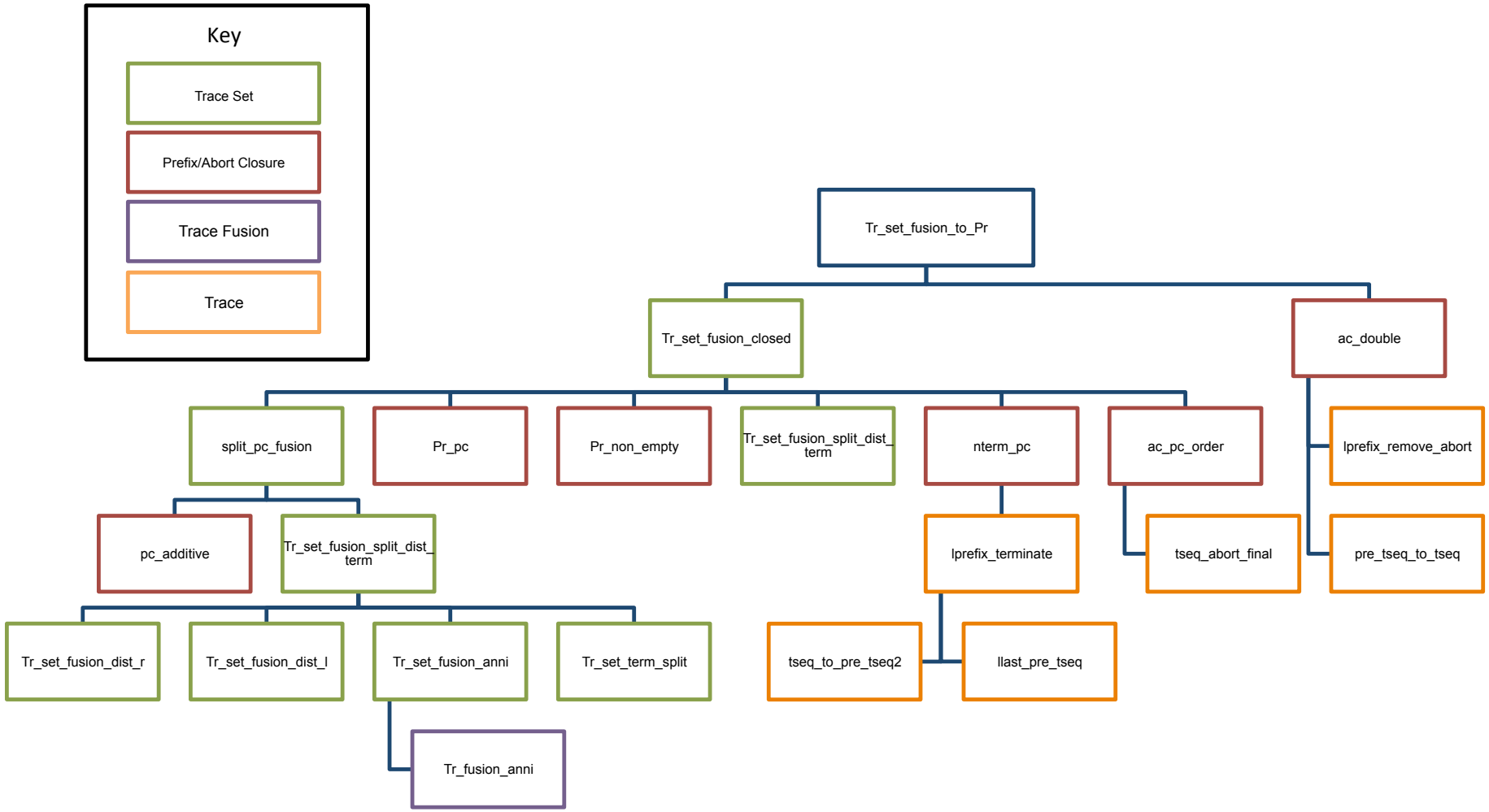

Figure 6.3: Dependency Diagram for closure property of sequential composition

Lemmas Used In Figure 6.3

lemma *Tr_set_fusion_to_Pr: "∃ C. ac ((Rep_Pr f) ⌢ (Rep_Pr g)) = Rep_Pr C"*

lemma *Tr_set_fusion_closed: "pc (ac ((Rep_Pr f) ⌢ (Rep_Pr g)))*
= ac ((Rep_Pr f) ⌢ (Rep_Pr g))"

lemma *split_pc_fusion: "pc ((Rep_Pr s) ⌢ (Rep_Pr t))*
= pc (n𝒯 (Rep_Pr s)) ∪ (𝒯 (Rep_Pr s)) ⌢ (pc (Rep_Pr t))"

lemma *pc_additive: "pc s ⊇ s"*

lemma *Tr_set_fusion_split_dist_term: "g ≠ {} ⟹ f ⌢ g = (𝒯 f) ⌢ g ∪ (n𝒯 f)"*

lemma *Tr_set_fusion_dist_r: "(f ∪ g) ⌢ h = (f ⌢ h) ∪ (g ⌢ h)"*

lemma *Tr_set_fusion_dist_l: "h ⌢ (f ∪ g) = (h ⌢ f) ∪ (h ⌢ g)"*

lemma *Tr_set_fusion_anni: "g ≠ {} ⟹ (n𝒯 f) ⌢ g = n𝒯 f"*

lemma *Tr_fusion_anni: "¬(terminates t) ⟹ Tr_fusion t g = t"*

lemma *Tr_set_term_split: "s = (𝒯 s) ∪ (n𝒯 s)"*

lemma *Pr_pc: "pc (Rep_Pr c) = Rep_Pr c"*

lemma *Pr_non_empty: "Rep_Pr c ≠ {}"*

lemma *nterm_pc: "pc (n𝒯 (Rep_Pr s)) = n𝒯 (Rep_Pr s)"*

lemma *lprefix_terminate: "¬ terminates (σ, t') ⟹ lprefix (Rep_Tseq t) (Rep_Tseq t') ⟹ ¬ terminates (σ, t)"*

lemma *llast_pre_tseq: "lfinite (Rep_Pre_Tseq t) ⟹ (Rep_Pre_Tseq t) ≠ [] ⟹ llast (Rep_Pre_Tseq t) ≠ stop"*

lemma *tseq_to_pre_tseq2: "lprefix (Rep_Tseq t) (Rep_Tseq t') ⟹ (Rep_Tseq t') ≠ (Rep_Tseq t) ⟹ (∃ t''. Rep_Pre_Tseq t'' = Rep_Tseq t)"*

lemma *ac_pc_order: "ac (pc s) = pc (ac s)"*

lemma *tseq_abort_final:*
"lfinite (Rep_Pre_Tseq t')
⟹ lprefix (Rep_Tseq (Abs_Tseq (Rep_Pre_Tseq t' @@ [pgm abortState]))) (Rep_Tseq t'')
⟹ Rep_Tseq (Abs_Tseq (Rep_Pre_Tseq t' @@ [pgm abortState])) = Rep_Tseq t''"

lemma *ac_double: "ac (ac s) = ac s"*

lemma *lprefix_remove_abort:*
"lprefix (Rep_Pre_Tseq t'') ((Rep_Pre_Tseq t') @@ [pgm abortState])
⟹ lprefix (Rep_Pre_Tseq t'') (Rep_Pre_Tseq t')"

lemma *pre_tseq_to_tseq: "(∃ t'. (Rep_Pre_Tseq t) @@ [a] = (Rep_Tseq t'))"*

6.3.3 Incomplete Proofs

There are a small number of proofs not completed for the sequential composition operator. The most notable of these is the axiom for sequential composition distributing across the generalised angelic choice operator to the right.

theorem *seq_dist_sup: "(⨆⨆ C);;d = ⨆⨆{ c;;d | c. c ∈ C}"*

In addition to this axiom, there are two simple properties that are not proved at this point. Intuitively, both of these properties seem straightforward to prove but there appear to be subtle issues preventing the automated theorem provers from completing the proofs.

lemma *tseq_pre_tseq_append4:*
"lfinite (Rep_Tseq xs) ⟹ (Rep_Tseq xs) ≠ []
⟹ (Rep_Pre_Tseq ys) ≠ (Rep_Tseq xs)
⟹ lprefix (Rep_Pre_Tseq ys) ((lbutlast (Rep_Tseq xs)) @@ [llast (Rep_Tseq xs)])
⟹ ∃ t'. (Rep_Pre_Tseq ys) @@ ((lbutlast (Rep_Tseq t')) @@ [llast (Rep_Tseq t')])
= (lbutlast (Rep_Tseq xs)) @@ [llast (Rep_Tseq xs)]"

lemma *lprefix_lappend_split_tseq:*
"lprefix (Rep_Tseq xs) ((lbutlast (Rep_Tseq ys)) @@ (Rep_Tseq zs))
⟹ lprefix (Rep_Tseq xs) (lbutlast (Rep_Tseq ys))
∨ (∃ pz. lprefix (Rep_Tseq pz) (Rep_Tseq zs)
∧ (Rep_Tseq xs) = (lbutlast (Rep_Tseq ys)) @@ (Rep_Tseq pz))"

With the exception of these three lemmas, the entirety of the lattice and sequential composition axioms are proved to hold for the model.

7 Discussion

7.1 Reflection on the Overall Approach

The approach taken can be broken down into three major parts. The first two sections concern formalising the trace model in Isabelle and defining the operators on this model. The final section involves proving that the model formalisation is an instance of the rely/guarantee algebra.

7.1.1 Trace Model

The trace model effectively models real-world programs, making it quite expressive but this comes at the cost of complexity as it leads to complex proofs. This is important as the success of future use depends on the practicality of the results. The tools that improve the development of concurrent programs must be practical and simple to use, which is a difficult problem as this is a necessarily complex topic. The semantic description of programming languages is far from a simple problem, especially taking parallel programs into account.

The original definition of the model [6] is thorough and accurate. Despite the well-defined nature of the model, small changes were required to simplify proofs. For example, when combining a set of processes using the generalised non-deterministic choice operator (or its dual), the union of the empty set is the empty set, which is not a valid process. The model solves this issue by applying the prefix closure, which complicates cases where the set is not empty. This is simplified by taking the union of **magic** (the identity of non-deterministic choice) as this will achieve the same result. Non-deterministic choice distributes over all other operators, so this extra element is the definition is easy to manipulate in proofs.

The operators acting on the model have dense, expressive definitions which although correct, do not offer simple comprehension. These definitions are split up into smaller functional parts and then combined into full definitions of the operators. Adding extra levels of abstraction allows key properties of the sections to be considered separately and then combined into useful properties about the complete definition. The most useful abstraction for the compositional operators was considering the behaviour of traces individually then raising the definition to the set level. For example, the sequential composition operator is defined in terms of trace fusion (concatenation of traces), which is then lifted to the set level. This allows properties of trace fusion to be considered in isolation before applying these intermediate lemmas to prove the properties of trace set fusion.

7.1.2 Instantiation Process

As the model is efficient and expressive, the complexity of reasoning about it is elevated. The success of the model in modelling a wide range of computer programs causes the instantiation process to be complex and drawn out. The number of cases for each trace in each process requires that every proof be broken down into a variety of special cases, each needing key properties of datatypes and functions.

Despite the difficulty in mechanising proofs, there is every indication that the model is an accurate instantiation of the algebra. The remaining proofs for sequential composition are quite simple (see Section 6.3.3), while the parallel composition proofs are still underdeveloped. With the lattice operators completely instantiated, only the strict conjunction operator remains for the algebra to be proved to be an interpretation of the rely/guarantee algebra.

7.2 Comparison to Other Approaches

A simplified trace model has been shown to be an instantiation of the Kleene Algebra [2]. The sequential composition operator is far less complex in this model and is therefore less expressive. This is effective for proving the consistency of the axioms of the Kleene Algebra, but has limited use past this point.

As only finite traces with generic step types are considered, the simplified definition of trace fusion is straightforward.

```
definition t_fusion :: "('p, 'a) trace ⇒ ('p, 'a) trace ⇒ ('p, 'a) trace" where
  "t_fusion x y ≡ if last x = first y then (fst x, snd x @ snd y) else undefined"
```

Associativity for this operator is proved trivially in contrast to the extensive proofs required for the corresponding property in this thesis.

```
lemma t_fusion_assoc [simp]:
  "⟦ last x = first y; last y = first z ⟧
⟹ t_fusion x (t_fusion y z) = t_fusion (t_fusion x y) z"
  by (cases x, cases y, cases z, simp add: t_fusion_def)
```

Trace fusion is lifted to the set level with no extra constraints, so the set level associativity follows directly from this definition and the trace fusion associativity lemma.

```
definition t_prod :: "('p, 'a) trace set ⇒ ('p, 'a) trace set ⇒ ('p, 'a) trace set"
where "X · Y = {t_fusion u v| u v. u ∈ X ∧ v ∈ Y ∧ last u = first v}"
```

Clearly, this is not an accurate model for real-world program semantics and demonstrates the complexity introduced with a more complete model. While the fundamental concepts and methodologies are similar, the two approaches have different goals. The simple model is concise with only 130 lines in Isabelle/HOL for the entire model formalisation and instantiation and demonstrates the basic concept effectively. This thesis develops the relationship between a semantic description of a general parallel programming language and the axioms of the rely/guarantee algebra.

7.3 Reflection on Isabelle/HOL

Isabelle/HOL is a popular theorem prover in computer science and abstract algebra, and it has many features that improve the process of mechanising proofs in this context. The syntax is very readable, especially with the choice of fine-grained, explicit Isar proofs or more concise apply-style proofs. In addition to this, the locales and structuring features make the use of the *interpretation* command simple. This is particularly impressive, as expressing the instantiation of a model as an algebra is a complex task.

The major downside to the use of Isabelle/HOL for expressing the trace model is that functions in HOL are total. This is an issue in particular here, as co-inductively defined lists are used to model infinite lists and functions are needed to reason about the final states of traces. This necessarily requires powerful partial functions support. While there are mechanisms to deal with this issue, they are complicated and unintuitive.

It became more apparent throughout the process that sensible structure and naming is important to keep theories manageable and usable. It is also important to take the time to plan out proofs ahead of time and consider what the key properties of the datatypes are. As the expressions involved in proofs become more complex, apply-style proofs are usually more appropriate and readable than step-wise Isar proofs. The mechanical proof document is structured logically taking these principles into account. This makes the work easily extensible and searchable for future use. The positive impacts of logical structure and naming is clear from the dependency diagrams for sequential composition associativity (Diagrams 6.1 - 6.3) as breaking up the complexities of a proof makes each level of abstraction easier to reason about. In addition to this, similarly named proofs contain similar content, allowing properties to be located based on the level of abstraction they deal with.

7.4 Future Work

Despite not completely instantiating the algebra, significant progress has been made on supporting the thesis. Firstly, the model is completely formalised in Isabelle/HOL, which represents a large contribution towards further work on the instantiation.

The approach to proving the axioms for the sequential composition operator is also shown to be effective. The parallel composition and strict conjunction operators will require a similar approach, part of which has been sketched out. In addition to this, as iteration is expressed using the least and greatest fixed points with sequential composition and non-deterministic choice, no axioms are needed to be proved for iteration to apply to the model. Similarly, the rely-quotient operator is defined in terms of parallel composition and non-deterministic choice.

The partial results achieved alone are useful in developing tools for reasoning about sequential programs. The instantiated part of the algebra is effectively an Omega Algebra. Having a mechanised expressive semantic model for the Omega Algebra is an interesting result in its own right, as this enables complex reasoning about potentially infinite, aborting and terminating sequential programs.

The number of lines in each theory give a rough indication of the amount of work to prove each module of the overall model. Generally shorter proofs are desirable, so the more refined sections can manage the complexity of the proofs in a more concise way. Therefore the longer sections are not necessarily more complex, although the Pr_Fusion and Lattice_Sem theory files do contain the most content. Table 7.1 is a breakdown of the lines in each theory file.

Table 7.1: Lines per section

Section	**Theory**	**Lines**
Misc	Llist_Fix.thy	137
Model	Traces.thy	251
	Processes.thy	190
	Commands.thy	84
	Termination.thy	92
Lattice Operators	Lattice_Sem.thy	258
Sequential Composition	Tr_Fusion.thy	193
	Pr_Fusion.thy	402
Parallel Composition	Parallel_Comp.thy	132
	Total	1817

8 Conclusion

The core idea behind this thesis is bridging the gap between the rely/guarantee algebra [12] and the semantic trace model [6], which focus on different levels of abstraction on the same overarching topic. The generic proof assistant Isabelle/HOL is used to formalise the model and verify the instantiation. Building on top of earlier work to formalise the algebra in Isabelle/HOL, this thesis is broken down into three major sections; formalising the model, formalising the operators and completing the instantiation proofs.

The trace model aims to give a semantics of real-world programs in a wide-spectrum language. This achieves considerable expressiveness for describing the behaviour of programs and program specifications in the same language. The expressiveness of the model causes the formalisation and instantiation process to be very involved, due to the complexity involved in programming language semantics. The extent of the complexity is clear as the Isabelle/HOL theory files required to prove 21 of the 22 axioms for the lattice and sequential composition operators consist of 1817 lines of definitions and proof.

Discovering the properties for each level of abstraction in the formalisation of the model proved to be key in achieving succinct proofs of the properties of the compositional operators. As the model consists of processes built up from sets of traces, where traces are built up from co-inductive lists, reasoning about each construct in the model is central in establishing properties of the high-level definitions that are interesting from a program transformation perspective.

While the entire instantiation is not completed, a few key conclusions can be drawn from the work:

- The model satisfies the axioms of the sequential and choice-based operators, which is interesting in its own right. This forms a realistic semantic model for an omega algebra, enabling reasoning about sequential programs and their specifications.
- The framework for completing the rest of the instantiation is in place, including key properties of the data types and an outline of the parallel composition proofs.

Bibliography

[1] P. H. G. Aczel. On an inference rule for parallel composition. [Online] http://homepages.cs.ncl.ac.uk/cliff.jones/publications/MSs/PHGA-traces.pdf, 1983. Private communication to Cliff Jones.

[2] Alasdair Armstrong, Georg Struth, and Tjark Weber. Kleene algebra. *Archive of Formal Proofs*, Jan 2013.

[3] Ralph-Johan Back and Joakim von Wright. Reasoning algebraically about loops. *Acta Inf.*, 36(4):295–334, 1999.

[4] Clemens Ballarin. Interpretation of locales in isabelle: Theories and proof contexts. In *Mathematical Knowledge Management*, volume 4108 of *Lecture Notes in Computer Science*, pages 31–43. Springer, 2006.

[5] Richard Bird and Oege de Moor. The algebra of programming. In *Proceedings of the NATO Advanced Study Institute on Deductive Program Design*, pages 167–203, Secaucus, NJ, USA, 1996. Springer-Verlag New York, Inc.

[6] Robert Colvin, Ian Hayes, and Larissa Meinicke. Designing a wide-spectrum semantics for concurrency. 2015.

[7] Brian Davey and Hilary Priestley. *Introduction to lattices and order*. Cambridge University Press, 2nd edition, 2002.

[8] W.-P. de Roever. *Concurrency Verification: Introduction to Compositional and Noncompositional Methods*. Cambridge University Press, 2001.

[9] Solomon Feferman. Does mathematics need new axioms? *The American Mathematical Monthly*, 106(2):pp. 99–111, 1999.

[10] Julian Fell. Generalised rely-guarantee concurrency: Proof document. 2015.

[11] A.G. Hamilton. *Numbers, Sets and Axioms*. Cambridge University Press, 1982.

[12] Ian Hayes. Generalised rely-guarantee concurrency: An algebraic foundation. 2015.

[13] Ian Hayes, Cliff Jones, and Robert Colvin. Laws and semantics for rely-guarantee refinement. Technical report, Technical report CS-TR-1425, Newcastle University, 2014.

[14] C.A.R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 26(1):576–580, 1969.

[15] Tony Hoare, Bernhard Möller, Georg Struth, and Ian Wehrman. Concurrent kleene algebra and its foundations. *J. Log. Algebr. Program.*, 80(6):266–296, 2011.

[16] Cliff B. Jones. Tentative steps toward a development method for interfering programs. *ACM Trans. Program. Lang. Syst.*, 5(4):596–619, 1983.

[17] Cliff B. Jones. Accommodating interference in the formal design of concurrent object-based programs. *Formal Methods in System Design*, 8(2):105–122, 1996.

[18] Cliff B. Jones. Operational semantics: Concepts and their expression. In *Information Processing Letters 88*, pages 27–32, 2003.

[19] Dexter Kozen. On kleene algebras and closed semirings. volume 452 of *Lecture Notes in Computer Science*, pages 26–47. Springer, May 1990.

[20] Dexter Kozen and Ra Silva. Practical coinduction. November 2012. Unpublished.

[21] Alexander Krauss. Defining recursive functions in isabelle/hol. [Online] http://isabelle.in.tum.de/website-Isabelle2015-RC4/dist/Isabelle2015-RC4/doc/functions.pdf, 2008. Accessed: 2015-09-8.

[22] Andreas Lochbihler. Coinductive. *Archive of Formal Proofs*, Feb 2010.

[23] Lawrence Paulson. Isabelle. [Online] http://isabelle.in.tum.de, 2015. Accessed: 2015-03-11.

[24] Lawrence C. Paulson and Jasmin Christian Blanchette. Three years of experience with sledgehammer, a practical link between automatic and interactive theorem provers. [Online] https://www21.in.tum.de/ blanchet/iwil2010-sledgehammer.pdf, 2010. Accessed: 2015-09-21.

[25] David Schmidt. *Denotational Semantics: A Methodology For Language Development*. Kansas State University, 1997.

[26] Joakim von Wright. Towards a refinement algebra. *Sci. Comput. Program.*, 51(1-2):23–45, 2004.

[27] Makarius Wenzel. *The Isabelle/Isar Reference Manual*. University of Cambridge, August 2014.