Algebraic Principles for Rely-Guarantee Style Concurrency Verification Tools

Alasdair Armstrong, Victor B.F. Gomes, and Georg Struth

Department of Computer Science, University of Sheffield, UK
{a.armstrong,v.gomes,g.struth}@dcs.shef.ac.uk

Abstract. We provide simple equational principles for deriving rely-guarantee-style inference rules and refinement laws based on idempotent semirings. We link the algebraic layer with concrete models of programs based on languages and execution traces. We have implemented the approach in Isabelle/HOL as a lightweight concurrency verification tool that supports reasoning about the control and data flow of concurrent programs with shared variables at different levels of abstraction. This is illustrated on a simple verification example.

1 Introduction

Extensions of Hoare logics are becoming increasingly important for the verification and development of concurrent and multiprocessor programs. One of the most popular extensions is Jones' rely-guarantee method [17]. A main benefit of this method is compositionality: the verification of large concurrent programs can be reduced to the independent verification of individual subprograms. The effect of interactions or interference between subprograms is captured by *rely* and *guarantee* conditions. Rely conditions describe the effect of the environment on an individual subprogram. Guarantee conditions, in turn, describe the effect of an individual subprogram on the environment. By constraining a subprogram by a rely condition, the global effect of interactions is captured locally.

To make this method applicable to concrete program development and verification tasks, its integration into tools is essential. To capture the flexibility of the method, a number of features seem desirable. First, we need to implement solid mathematical models for fine-grained program behaviour. Second, we would like an abstract layer at which inference rules and refinement laws can be derived easily. Third, a high degree of proof automation is mandatory for the analysis of concrete programs. In the context of the rely-guarantee method, tools with these important features are currently missing.

This paper presents a novel approach for providing such a tool integration in the interactive theorem proving environment Isabelle/HOL. At the most abstract level, we use algebras to reason about the control flow of programs as well as for deriving inference rules and refinement laws. At the most concrete level, detailed models of program stores support fine-grained reasoning about program data flow and interference. These models are then linked with the algebras. Isabelle

C. Jones, P. Pihlajasaari, and J. Sun (Eds.): FM 2014, LNCS 8442, pp. 78–93, 2014.

allows us to implement these layers in a modular way and relate them formally with one another. It not only provides us with a high degree of confidence in the correctness of our development, it also supports the construction of custom proof tactics and procedures for program verification and refinement tasks.

For sequential programs, the applicability of algebra, and Kleene algebra in particular, has been known for decades. Kleene algebra provides operations for non-deterministic choice, sequential composition and finite iteration, in addition to skip and abort. With appropriate extensions, Kleene algebras support Hoare-style verification of sequential programs, and allow the derivation of program equivalences and refinement rules [20,16]. Kleene algebras have been used in applications including compiler optimisation, program construction, transformation and termination analysis, and static analysis. Formalisations and tools are available in interactive theorem provers such as Coq [26] and Isabelle [2,3,1]. A first step towards an algebraic description of rely-guarantee based reasoning has recently been undertaken [16].

The main contributions of this paper are as follows. First, we investigate algebraic principles for rely-guarantee style reasoning. Starting from [16] we extract a basic minimal set of axioms for rely and guarantee conditions which suffice to derive the standard rely-guarantee inference rules. These axioms provide valuable insights into the conceptual and operational role of these constraints. However, algebra is inherently compositional, so it turns out that these axioms do not fully capture the semantics of interference in execution traces. We therefore explore how the compositionality of these axioms can be broken in the right way, so as to capture the intended trace semantics.

Second, we link our rely-guarantee algebras with a simple trace based semantics which so far is restricted to finite executions and disregards termination and synchronisation. Despite the simplicity of this model, we demonstrate and evaluate our prototypical verification tool implemented in Isabelle by verifying a simple example from the literature. Beyond that our approach provides a coherent framework from which more complex and detailed models can be implemented in the future.

Third, we derive the usual inference rules of the rely-guarantee method with the exception of assignment axioms directly from the algebra, and obtain assignment axioms from our models. Our formalisation in Isabelle allows us to reason seamlessly across these layers, which includes the data flow and the control flow of concurrent programs.

Taken together, our Isabelle implementation constitutes a tool prototype for the verification and construction of concurrent programs. We illustrate the tool with a simple example from the literature. The complete Isabelle code can be found online[1]. A previous Isabelle implementation of rely-guarantee reasoning is due to Prensa Nieto [24]. Our implementation differs both by making the link between concrete programs and algebras explicit, which increases modularity, and by allowing arbitrary nested parallelism.

[1] `www.dcs.shef.ac.uk/~alasdair/rg`

2 Algebraic Preliminaries

Rely-guarantee algebras, which are introduced in the following section, are based on dioids and Kleene algebras. A *semiring* is a structure $(S, +, \cdot, 0, 1)$ such that $(S, +, 0)$ is a commutative monoid, $(S, \cdot, 1)$ is a monoid and the distributivity laws $x \cdot (y + z) = x \cdot z + y \cdot z$ and $(x + y) \cdot z = x \cdot z + y \cdot z$ as well as the annihilation laws $x \cdot 0 = 0$ and $0 \cdot x = 0$ hold. A *dioid* is a semiring in which addition is idempotent: $x + x = x$. Hence $(S, +, 0)$ forms a join semilattice with least element 0 and partial order defined, as usual, as $x \leq y \Leftrightarrow x + y = y$. The operations of addition and multiplication are isotone with respect to the order, that is, $x \leq y$ implies $z + x \leq z + y$, $z \cdot x \leq z \cdot y$ and $x \cdot z \leq y \cdot z$. A dioid is *commutative* if multiplication is: $x \cdot y = y \cdot x$.

In the context of sequential programs, one typically thinks of $\cdot$ as sequential composition, + as nondeterministic choice, 0 as the abortive action and 1 as skip. In this context it is essential that multiplication is not commutative. Often we use ; for sequential composition when discussing programs. More formally, it is well known that (regular) languages with language union as +, language product as $\cdot$, the empty language as 0 and the empty word language $\{\varepsilon\}$ as 1 form dioids. Another model is formed by binary relations with the union of relations as +, the product of relations as $\cdot$, the empty relation as 0 and the identity relation as 1. A model of commutative dioids is formed by sets of (finite) multisets or Parikh vectors with multiset addition as multiplication.

It is well known that commutative dioids can be used for modelling the interaction between concurrent composition and nondeterministic choice. The following definition serves as a basis for models of concurrency in which sequential and concurrent composition interact.

A *trioid* is a structure $(S, +, \cdot, ||, 0, 1)$ such that $(S, +, \cdot, 0, 1)$ is a dioid and $(S, +, ||, 0, 1)$ a commutative dioid. In a trioid there is no interaction between the sequential composition $\cdot$ and the parallel composition $||$. On the one hand, Gischer has shown that trioids are sound and complete for the equational theory of series-parallel pomset languages [13], which form a well studied model of true concurrency. On the other hand, he has also obtained a completeness result with respect to a notion of pomset subsumption for trioids with the additional *interchange axiom* $(w\|x) \cdot (y\|z) \leq (w \cdot y)\|(x \cdot z)$ and it is well known that this additional axiom also holds for (regular) languages in which $||$ is interpreted as the shuffle or interleaving operation [12].

Formally, the *shuffle* $\|$ of two finite words is defined inductively as $\epsilon\|s = \{s\}$, $s\|\epsilon = \{s\}$, and $as\|bt = a(s\|bt) \cup b(as\|t)$, which is then lifted to the shuffle product of languages X and Y as $X\|Y = \{x\|y : \ x \in X \wedge x \in Y\}$.

For programming, notions of iteration are essential. A *Kleene algebra* is a dioid expanded with a star operation which satisfies both the *left unfold axiom* $1 + x \cdot x^\star \leq x^\star$ and *left* and *right induction axioms* $z + x \cdot y \leq y \Rightarrow x^\star \cdot z \leq y$ and $z + y \cdot x \leq y \Rightarrow z \cdot x^\star \leq y$. It follows that $1 + x \cdot x^\star = x^\star$ and that the right unfold axiom $1 + x^\star \cdot x \leq x^\star$ is derivable as well. Thus iteration x^* is modelled as the least fixpoint of the function $\lambda y. 1 + x \cdot y$, which is the same as the least fixpoint of $\lambda y.1 + y \cdot x$. A *commutative Kleene algebra* is a Kleene algebra in which multiplication is commutative.

It is well known that (regular) languages form Kleene algebras and that (regular) sets of multisets form commutative Kleene algebras. In fact, Kleene algebras are complete with respect to the equational theory of regular languages as well as the equational theory of binary relations with the reflexive transitive closure operation as the star [19]. Moreover, commutative Kleene algebras are complete with respect to the equational theory of regular languages over multisets [7]. It follows that equations in (commutative) Kleene algebras are decidable.

A *bi-Kleene algebra* is a structure $(K, +, \cdot, ||, 0, 1, ^\star, ^{(\star)})$ where $(K, +, \cdot, 0, 1, ^\star)$ is a Kleene algebra and $(K, +, ||, 0, 1, ^{(\star)})$ is a commutative Kleene algebra. Bi-Kleene algebras are sound and complete with respect to the equational theory of regular series-parallel pomset languages, and the equational theory is again decidable [21]. A *concurrent Kleene algebra* is a bi-Kleene algebra which satisfies the interchange law [16]. It can be shown that shuffle languages and regular series-parallel pomset languages with a suitable notion of pomset subsumption form concurrent Kleene algebras.

In some contexts, it is also useful to add a meet operation $\sqcap$ to a bi-Kleene algebra, such that $(K, +, \sqcap)$ is a distributive lattice. This is particularly needed in the context of refinement, where we typically want to represent specifications as well as programs.

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

$$x(\Sigma Y) = \Sigma\{xy | y \in Y\}, \qquad (\Sigma X)y = \Sigma\{xy | x \in X\}.$$

In a quantale the star is the sum of all powers x^n. Therefore, all quantales are also Kleene algebras.

3 Generalised Hoare Logics in Kleene Algebra

It is well known that the inference rules of sequential Hoare logic (except the assignment axiom) can be derived in expansions of Kleene algebras. One approach is as follows [23]. Suppose a suitable Boolean algebra B of *tests* has been embedded into a Kleene algebra K such that 0 and 1 are the minimal and maximal element of B, $+$ corresponds to join and $\cdot$ to meet. Complements $-$ are defined only on B. Suppose further that a *backward diamond operator* $\langle x|p$ has been defined for each $x \in K$ and $p \in B$, which models the set of all states to which each terminating execution of program x may lead from states p. Finally suppose that a *forward box operator* $|x]p$ has been defined which models the (largest) set of states from which every terminating execution of x must end in states p and that boxes and diamonds are adjoints of the Galois connection $\langle x|p \leq q \Leftrightarrow p \leq |x]q$, for all $x \in K$ and $p, q \in B$. It is then evident from the above explanations that validity of a Hoare triple $\vdash \{p\}x\{q\}$ can be encoded as $\langle x|p \leq q$ and the weakest liberal precondition operator $\mathsf{wlp}(x, q)$ as $|x]p$. Hence the relationship between the proof theory and the semantics of Hoare logic is captured by the Galois connection $\vdash \{p\}x\{q\} \Leftrightarrow p \leq \mathsf{wlp}(x, q)$. It has been shown that the relational semantics of sequential while-programs can be encoded in these *modal Kleene algebras* and that the inference rules of Hoare logic can be derived [23].

In the context of concurrency, this relational approach is no longer appropriate; the following approach by Tarlecki [28] can be used instead. One can now encode validity of a Hoare triple as

$$\vdash \{x\}y\{z\} \Leftrightarrow x \cdot y \leq z$$

for arbitrary elements of a Kleene algebra. Nevertheless all the rules of sequential Hoare logic except the assignment axiom can still be derived [16]. Tarlecki's motivating explanations carry over to the algebraic approach.

As an example we show the derivation of a generalised while rule. Suppose $x \cdot t \cdot y \leq x$. Then $x \cdot (t \cdot y)^* \leq x$ by the right induction axiom of Kleene algebra and therefore $x \cdot (t \cdot y)^* \cdot t' \leq x \cdot t'$ for arbitrary element t' by isotonicity of multiplication. This derives the while rule

$$\frac{\vdash \{x \cdot t\}y\{x\}}{\vdash \{x\}(t \cdot y)^* \cdot t'\{x \cdot t'\}}$$

for a generalised while loop $(t \cdot y)^* \cdot t'$, which specialises to the conventional rule when t and t' are, in some sense, complements.

The correspondence to a wlp-style semantics, as in modal Kleene algebra, now requires a generalisation of the Galois connection for boxes and diamonds to multiplication and an upper adjoint in the form of residuation. This can be achieved in the context of *action algebras* [27], which expand Kleene algebras by operations of left and right residuation defined by the Galois connections

$$x \cdot y \leq z \Leftrightarrow x \leq z \leftarrow y, \qquad x \cdot y \leq z \Leftrightarrow y \leq x \rightarrow z.$$

These residuals, and now even the Kleene star, can be axiomatised equationally in action algebras. For a comprehensive list of the properties of action algebras and their most important models see [2], including the language and the relational model. In analogy to the development in modal Kleene algebra we can now stipulate $\mathsf{wlp}(x, y) = y \leftarrow x$ and obtain the Galois connection

$$\vdash \{x\}y\{z\} \Leftrightarrow x \leq \mathsf{wlp}(y, z)$$

with $\vdash \{\mathsf{wlp}(y, z)\}y\{z\}$ and $x \leq \mathsf{wlp}(y, z) \Rightarrow \vdash \{x\}y\{z\}$ as characteristic properties. Moreover, if the action algebra is also a quantale, and infinite sums exist, it follows that $\mathsf{wlp}(y, z) = \sum\{x : \vdash \{x\}y\{z\}\}$. It is obvious that this definition makes sense in all models of action algebras and quantales. Intuitively, suppose p stands for the set of all behaviours of a system, for instance the set of all execution traces, that end in state p, and likewise for q. Then $\{p\}x\{q\}$ states that all executions ending in p can be extended by x to executions ending in q. $\mathsf{wlp}(x, q)$ is the most general behaviour, that is the set of all executions p after which all executions of x must end in q.

A residuation for concurrent composition can be considered as well:

$$x\|y \leq z \Leftrightarrow y \leq x/z.$$

The residual x/z represents the weakest program such that when placed in parallel with x, the parallel composition behaves as z.

4 A Rely-Guarantee Algebra

We now show how bi-Kleene algebras can be expanded into a simple algebra that supports the derivation of rely-guarantee style inference rules. This development does *not* use the interchange law for several reasons. First, this law fails for fair parallel composition $x \|_f y$ in models with possibly infinite, or non-terminating programs. In this model, $x \cdot y \not\leq x \|_f y$ whenever x is non-terminating. Secondly, it is not needed for deriving the usual rules of rely-guarantee.

A rely-guarantee algebra is a structure $(K, I, +, \sqcap, \cdot, \|, ^\star, 0, 1)$, where $(K, +, \sqcap)$ is a distributive lattice, $(K, +, \cdot, \|, 0, 1)$ is a trioid and $(K, +, \sqcap, \cdot, \|, ^\star, 0, 1)$ is a bi-Kleene algebra where we do not consider the parallel star. I is a distinguished subset of rely and guarantee conditions or *interference constraints* which satisfy the following axioms

$$r\|r \leq r, \tag{1}$$

$$r \leq r\|r', \tag{2}$$

$$r\|(x \cdot y) = (r\|x) \cdot (r\|y), \tag{3}$$

$$r\|x^+ \leq (r\|x)^+. \tag{4}$$

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

The general idea is to constrain a program by a rely condition by executing the two in parallel. Axiom (1) states that interference from a constraint being run twice in parallel is no different from just the interference from that constraint begin run once in parallel. Axiom (2) states that interference from a single constraint is less than interference from itself and another interference constraint. Axiom (3) allows an interference constraint to be split across sequential programs. Axiom (4) is similar to Axiom (3) in intent, except it deals with finite iteration.

Some elementary consequences of these rules are

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

Theorem 1. *Axioms (1), (2) and (3) are independent.*

Proof. We have used Isabelle's *Nitpick* [4] counterexample generator to construct models which violate each particular axiom while satisfying all others. □

Theorem 2. *Axiom (3) implies (4) in a quantale where $\|$ distributes over arbitrary suprema.*

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

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

□

In first-order Kleene algebras (3) and (4) are independent, but it is impossible to find a counterexample with Nitpick because it generates only finite counterexamples, and all finite Kleene algebras are a forteriori quantales.

Jones quintuples can be encoded in this setting as

$$r, g \vdash \{p\}x\{q\} \iff p \cdot (r\|x) \leq q \wedge x \leq g. \tag{5}$$

This means that program x when constrained by a rely r, and executed after p, behaves as q. Moreover, all behaviours of x are included in its guarantee q. Note that this encoding is stronger than in traditional rely-guarantee, as x is required to unconditionally implement g. The algebra could easily be extended with an additional operator f such that $f(r, x) \leq q$ would encode that x implements q only under interference of at most r. For more complex examples than what we present in section 8 such an encoding may prove neccessary.

Theorem 3. *The standard rely-guarantee inference rules can be derived with the above encoding, as shown in Figure 1.*

Thus (1) to (4), which are all necessary to derive these rules, represent a minimal set of axioms from which these inference rules can be derived.

If we add residuals to our algebra quintuples can be encoded in the following way, which is equivalent to the encoding in Equation (5).

$$r, g \vdash \{p\}x\{q\} \iff x \leq r/(p \to q) \sqcap g. \tag{6}$$

This encoding allows us to think in terms of program refinement, as in [14], since $r/(p \to q) \sqcap g$ defines the weakest program that when placed in parallel with interference from r, and guaranteeing interference at most g, goes from p to q—a generic specification for a concurrent program.

5 Breaking Compositionality

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

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

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

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

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

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

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

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

Fig. 1. Rely-guarantee inference rules

This can be read as follows; if x and y are equal, then they must be equal under all possible interferences from an arbitrary z. At first, this might seem to preclude any fine-grained reasoning about interference using purely algebra. This is not the case, but breaking such inherent compositionality in just the right way to capture interesting properties of interference requires extra work.

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

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

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

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

we can break compositionality in just the right way, provided we chose appropriate properties for π. These properties are extracted from properties of the trace model, which will be explained in detail in the next section. Many of those can be derived from the fact that, in our model, $\pi = \lambda x.\ x \sqcap c$, where c is healthiness condition which filters out ill-defined traces. We do not list these properties here. In addition π must satisfy the properties

$$x^\star \leq_\pi \pi(x)^\star, \tag{7}$$

$$x \cdot y \leq_\pi \pi(x) \cdot \pi(y), \tag{8}$$

$$z + x \cdot y \leq_\pi y \implies x^\star \cdot z \leq_\pi y, \tag{9}$$

$$z + y \cdot x \leq_\pi y \implies z \cdot x^\star \leq_\pi y. \tag{10}$$

For any operator $\bullet$, we write $x \bullet_\pi y$ for the operator $\pi(x \bullet y)$, and we write x^π for $\pi(x^\star)$.

Theorem 4. $(\pi(K), +_\pi, \cdot_\pi, ^\pi, 0, 1)$ *is a Kleene algebra.*

Proof. It can be shown that π is a retraction, that is, $\pi^2 = \pi$. Therefore, $x \in \pi(K)$ iff $\pi(x) = x$. This condition can then be used to check the closure conditions for all operations. □

We redefine our rely-guarantee algebra as a structure $(K, I, +, \sqcap, \cdot, \|, ^\star, \pi, 0, 1)$ which, in addition to the rules in Section 4, satisfies (7) to (10).

Theorem 5. *All rules in Figure 1 can be derived in this algebra.*

Moreover their proofs remain the same, mutatis mutandis.

6 Finite Language Model

We now construct a finite language model satisfying the axioms in Section 4 and 5. Restricting our attention to finite languages means we do not need to concern ourselves with termination side-conditions, nor do we need to worry about additional restrictions on parallel composition, e.g. fairness. However, all the results in this section can be adapted to potentially infinite languages, and our Isabelle/HOL formalisation already includes general definitions by using coinductively defined lazy lists to represent words, and having a weakly-fair shuffle operator for such infinite languages.

We consider languages where the alphabet contains state pairs of the form $(\sigma_1, \sigma_2) \in \Sigma^2$. A word in such a language is *consistent* if every such pair in a word has the same first state as the previous transition's second state. For example, $(\sigma_1, \sigma_2)(\sigma_2, \sigma_3)$ is consistent, while $(\sigma_1, \sigma_2)(\sigma_3, \sigma_3)$ is consistent only if $\sigma_2 = \sigma_3$. Sets of consistent words are essentially *Aczel traces* [9], but lack the usual process labels. We denote the set of all consistent words by C and define the function π from the previous section as $\lambda X.\ X \cap C$ in our model.

Sequential composition in this model is language product, as per usual. Concurrent composition is the shuffle product defined in Section 2. The shuffle product is associative, commutative, and distributes over arbitrary joins. Both products share the same unit, $\{\epsilon\}$ and zero, $\emptyset$. In Isabelle proving properties of shuffle is surprisingly tricky (especially if one considers infinite words). For a in-depth treatment of the shuffle product see [22].

Theorem 6. $(\mathcal{P}((\Sigma^2)^\star), \cup, \cdot, \|, \emptyset, \{\epsilon\})$ *forms a trioid.*

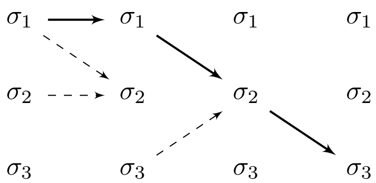
The rely-guarantee elements in this model are sets containing all the words which can be built from some set of state pairs in Σ^2. We define a function $\langle R \rangle$ which lifts a relation R to a language containing words of length one for each pair in R. The set of rely-guarantee conditions I is then defined as $\{r.\ \exists R. r = \langle R \rangle^\star\}$.

Theorem 7. $(\mathcal{P}((\Sigma^2)^\star), I, \cup, \cdot, \|, ^\star, \pi, \emptyset, \{\epsilon\})$ *is a rely-guarantee algebra.*

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

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

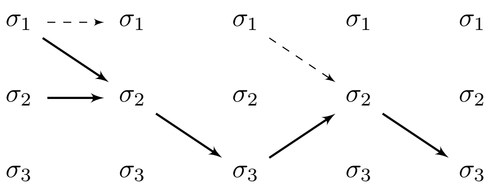
To demonstrate how this model works, consider the graphical representation of a language shown below.



The language contains the following six words

$$\begin{array}{ll} (\sigma_1,\sigma_1)(\sigma_1,\sigma_2)(\sigma_2,\sigma_3), & (\sigma_1,\sigma_2)(\sigma_1,\sigma_2)(\sigma_2,\sigma_3), \\ (\sigma_2,\sigma_2)(\sigma_1,\sigma_2)(\sigma_2,\sigma_3), & (\sigma_1,\sigma_1)(\sigma_3,\sigma_2)(\sigma_2,\sigma_3), \\ (\sigma_1,\sigma_2)(\sigma_3,\sigma_2)(\sigma_2,\sigma_3), & (\sigma_2,\sigma_2)(\sigma_3,\sigma_2)(\sigma_2,\sigma_3), \end{array}$$

where only the first, $(\sigma_1,\sigma_1)(\sigma_1,\sigma_2)(\sigma_2,\sigma_3)$ is consistent. This word is highlighted with solid arrows in the diagram above. Now if we shuffle the single state pair (σ_2,σ_3) into the above language, would end up with a language containing the words represented in the diagram below:



By performing this shuffle action, we no longer have a consistent word from σ_1 to σ_3, but instead a consistent word from σ_2 to σ_3 and σ_1 to σ_3. These new consistent words were constructed from previously inconsistent words—the shuffle operator can generate many consistent words from two inconsistent words. If we only considered consistent words, à la Aczel traces, we would be unable to define such a shuffle operator directly on the traces themselves, and would instead have to rely on some operational semantics to generate traces.

7 Enriching the Model

To model and verify programs we need additional concepts such as tests and assignment axioms. A *test* is any language P where $P \leq \langle \mathsf{Id} \rangle$. We write $\mathsf{test}(P)$ for $\langle \mathsf{Id}_P \rangle$. In Kleene algebra the sequential composition of two tests should be equal to their intersection. However, the traces $\mathsf{test}(P);\mathsf{test}(Q)$ and $\mathsf{test}(P \cap Q)$ are incomparable, as all words in the former have length two, while all the words in the latter have length one. To overcome this problem, we use the concepts of *stuttering* and *mumbling*, following [5] and [11]. We inductively generate the *mumble language* $w^\dagger$ for a word w in a language over Σ^2 as follows: Assume $\sigma_1, \sigma_2, \sigma_3 \in \Sigma$ and $u, v, w \in (\Sigma^2)^\star$. First, $w \in w^\dagger$. Secondly, if $u(\sigma_1, \sigma_2)(\sigma_2, \sigma_3)v \in w^\dagger$ then $u(\sigma_1, \sigma_3)v \in w^\dagger$. This operation is lifted to languages in the obvious way as

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

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

Assuming we apply mumbling to both sides of the following equation, we have that

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

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

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

We henceforth assume that all languages are implicitly mumble closed.

Using tests, we can encode if statements and while loops

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

Next, we define the operator $\mathsf{end}(P)$ which contains all the words which end in a state satisfying P. Some useful properties of end include

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

In this model, assignment is defined as

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

where $x \leftarrow v$ denotes the atomic command which assigns the value v to x. The eval function atomically evaluates an expression e in the state σ. Using this definition we derive the assignment rule

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

The rely condition states the following: First, the environment is not allowed to modify any of the variables used when evaluating e, i.e. those variables must remain unchanged. Second, the environment must preserve the precondition. Third, the postcondition of the assignment statement is also preserved. In turn, the assignment statement itself guarantees that it leaves every variable other than x unchanged. Preserves and unchanged are defined as

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

We also defined two futher rely conditions, increasing and decreasing, which are defined much like unchanged except they only require that variables increase or decrease, rather than stay the same. We can easily define other useful assignment rules—if we know properties about P and e, we can make stronger guarantees about what $x := e$ can do. For example the assignment $x := x - 2$ can also guarantee that x will always decrease.

8 Examples

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

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

The environment $\langle\mathsf{Id}\rangle^\star$ is only giving us stuttering interference. Since we are considering this program in isolation, we make no guarantees about how this affects the environment. To apply the parallel rule from Figure 1, we weaken or strengthen the interference constrains and pre/postcondition as needed to fit the form of the parallel rule.

First, we weaken the rely condition to $\mathsf{unchanged}\{x\} \sqcap \mathsf{unchanged}\{y,z\}$. Second we strengthen the guarantee condition to $\mathsf{unchanged}\{y,z\} \parallel \mathsf{unchanged}\{x\}$. When we apply the parallel rule each assignment's rely will become the other assignment's guarantee. Finally, we split the precondition and postcondition into $\mathsf{end}(x = 2) \sqcap \mathsf{end}(y = 2 \wedge z = 5)$ and $\mathsf{end}(x = 4) \sqcap \mathsf{end}(y = 5 \wedge z = 5)$ respectively. Upon applying the parallel rule, we obtain two trivial goals

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

Figure 2 shows the FINDP program, which has been used by numerous authors e.g. [25,17,10,14]. The program finds the least element of an array satisfying a predicate P. The index of the first element satisfying p is placed in the variable f. If no element of the array satisfies P then f will be set to the length of the array. The program has two subprograms, A and B, running in parallel, one of which searches the even indices while the other searches the odd indices. A speedup over a sequential implementation is achieved as A will terminate when B finds an element of the array satisfying P which is less than i_A.

$$
\begin{array}{l}
f_A := \mathsf{len}(\mathsf{array});\\
f_B := \mathsf{len}(\mathsf{array});\\
\left(\begin{array}{l\|l}
i_A = 0 & i_B = 1\\
\mathsf{while}\ i_A < f_A \wedge i_A < f_B\ \{ & \mathsf{while}\ i_B < f_A \wedge i_B < f_B\ \{\\
\quad \mathsf{if}\ P(\mathsf{array}[i_A])\ \{ & \quad \mathsf{if}\ P(\mathsf{array}[i_B])\ \{\\
\qquad f_A := i_A & \qquad f_B := i_B\\
\quad \}\ \mathsf{else}\ \{ & \quad \}\ \mathsf{else}\ \{\\
\qquad i_A := i_A + 2 & \qquad i_B := i_B + 2\\
\quad \} & \quad \}\\
\} & \}
\end{array}\right);\\
f = \min(f_A, f_B)
\end{array}
$$

Fig. 2. FINDP Program

Here, we only sketch the correctness proof, and comment on its implementation in Isabelle. We do not attempt to give a detailed proof, as this has been done many times previously.

To prove the correctness of FINDP, we must show that

$$\text{FINDP} \leq_\pi \mathsf{end}(\mathsf{leastP}(f)) + \mathsf{end}(f = \mathsf{len}(\text{array})),$$

where $\mathsf{leastP(f)}$ is the set of states where f is the least index satisfying P, and $f = len(array)$ is the set of states where f is the length of the array. In other words, either we find the least element, or f remains the same as the length of the array, in which case no elements in the array satisfy P.

To prove the parallel part of the program, subprogram A guarantees that it does not modify any of the variables used by subprogram B, except for f_A, which it guarantees will only ever decrease. Subprogram B makes effectively the same guarantee to A. Under these interference constraints we then prove that A or B will find the lowest even or odd index which satisfies P respectively—or they do not find it, in which case f_A or f_B will remain equal to the length of the array.

Despite the seemingly straightforward nature of this proof, it turns out to be surprisingly difficult in Isabelle. Each atomic step needs to be shown to satisfy the guarantee of its containing subprogram, as well as any goals relating to its pre and post conditions. This invariably leads to a proliferation of many small proof goals, even for such a simple program. More work must be done to manage the complexity of such proofs within interactive theorem provers.

9 Conclusion

We have introduced variants of semirings and Kleene algebras intended to model rely-guarantee and interference based reasoning. We have developed an interleaving model for these algebras which uses familiar concepts from traces and language theory. This theory has been implemented in the Isabelle/HOL theorem prover, providing a solid mathematical basis on which to build a tool for mechanised refinement and verification tasks. In line with this aim, we have applied our formalisation to a simple example program.

This implementation serves as a basis from which further interesting aspects of concurrent programs, such as non-termination and synchronisation can be explored. As mentioned in Section 6, some of the work needed to implement this we have already done in Isabelle.

Algebra plays an important role in our development. First, it allowed us to derive inference rules rapidly and with little proof effort. Second, it yields an abstract layer at which many properties that would be difficult to prove in concrete models can be verified with relative ease by equational reasoning. Third, as pointed out in Section 2, some fragments of the algebras considered are decidable. Therefore, decision procedures for some aspects of rely-guarantee reasoning can be implemented in interactive theorem proving tools such as Isabelle. However, we have not yet investigated the extent to which such decision procedures would benefit our approach.

The examples from Section 8 confirm previous evidence [24] that even seemingly straightforward concurrency verification tasks can be tedious and complex. It is too early to draw informed conclusions, but while part of this complexity may be unavoidable, more advanced models and proof automation are needed to overcome such difficulties. Existing work on combining rely-guarantee with separation logic [29] may prove useful here. Our language model is sufficiently generic such that arbitrary models of stores may be used, including those common in separation logic, which have already been implemented in Isabelle [18].

In addition, algebraic approaches to separation logic have already been introduced. Examples are the separation algebras in [6], and algebraic separation logic [8]. More recently, concurrent Kleene algebras have given an algebraic account of some aspects of concurrent separation logic [16,15].

Acknowledgements. The authors would like to thank Brijesh Dongol and Ian Hayes for inspiring discussions on concurrency verification and the rely-guarantee method. The first author acknowledges funding from an EPSRC doctoral fellowship. The second author is supported by CNPq Brazil. The third author acknowledges funding by EPSRC grant EP/J003727/1.

References

1. Armstrong, A., Gomes, V.B.F., Struth, G.: Algebras for program correctness in isabelle/HOL. In: Kahl, W. (ed.) RAMiCS 2014. LNCS, vol. 8428, pp. 49–64. Springer, Heidelberg (2014)

2. Armstrong, A., Struth, G., Weber, T.: Kleene algebra. In: Archive of Formal Proofs (2013)
3. Armstrong, A., Struth, G., Weber, T.: Program analysis and verification based on Kleene algebra in Isabelle/HOL. In: Blazy, S., Paulin-Mohring, C., Pichardie, D. (eds.) ITP 2013. LNCS, vol. 7998, pp. 197–212. Springer, Heidelberg (2013)
4. Blanchette, J.C., Nipkow, T.: Nitpick: A counterexample generator for higher-order logic based on a relational model finder. In: Kaufmann, M., Paulson, L.C. (eds.) ITP 2010. LNCS, vol. 6172, pp. 131–146. Springer, Heidelberg (2010)
5. Brookes, S.: Full abstraction for a shared variable parallel language. In: Okada, M., Panangaden, P. (eds.) LICS, 1993, pp. 98–109 (1993)
6. Calcagno, C., O'Hearn, P.W., Yang, H.: Local action and abstract separation logic. In: Ong, L. (ed.) LICS 2007, pp. 366–378 (2007)
7. Conway, J.H.: Regular algebra and finite machines. Chapman and Hall (1971)
8. Dang, H.-H., Höfner, P., Möller, B.: Algebraic separation logic. J. Log. Algebr. Program. 80(6), 221–247 (2011)
9. de Boer, F.S., Hannemann, U., de Roever, W.-P.: Formal justification of the rely-guarantee paradigm for shared-variable concurrency: A semantic approach. In: Wing, J.M., Woodcock, J., Davies, J. (eds.) FM 1999. LNCS, vol. 1709, pp. 1245–1265. Springer, Heidelberg (1999)
10. de Roever, W.-P., de Boer, F., Hanneman, U., Hooman, J., Lakhnech, Y., Poel, M., Zwiers, J.: Concurrency verification: an introduction to state-based methods. Cambridge University Press, Cambridge (2001)
11. Dingel, J.: A refinement calculus for shared-variable parallel and distributed programming. Formal Aspects of Computing 14(2), 123–197 (2002)
12. Gischer, J.L.: Shuffle languages, Petri nets, and context-sensitive grammars. Commun. ACM 24(9), 597–605 (1981)
13. Gischer, J.L.: The equational theory of pomsets. Theoretical Computer Science 61(2-3), 199–224 (1988)
14. Hayes, I.J., Jones, C.B., Colvin, R.J.: Refining rely-guarantee thinking (2013) (unpublished)
15. Hoare, C.A.R., Hussain, A., Möller, B., O'Hearn, P.W., Petersen, R.L., Struth, G.: On locality and the exchange law for concurrent processes. In: Katoen, J.-P., König, B. (eds.) CONCUR 2011. LNCS, vol. 6901, pp. 250–264. Springer, Heidelberg (2011)
16. Hoare, T., Möller, B., Struth, G., Wehrman, I.: Concurrent Kleene algebra and its foundations. J. Log. Algebr. Program. 80(6), 266–296 (2011)
17. Jones, C.B.: Development methods for computer programs including a notion of interference. PhD thesis, Oxford University (1981)
18. Klein, G., Kolanski, R., Boyton, A.: Mechanised separation algebra. In: Beringer, L., Felty, A. (eds.) ITP 2012. LNCS, vol. 7406, pp. 332–337. Springer, Heidelberg (2012)
19. Kozen, D.: A completeness theorem for Kleene algebras and the algebra of regular events. Information and Computation 110(2), 366–390 (1994)
20. Kozen, D.: Kleene algebra with tests. ACM Trans. Program. Lang. Syst. 19(3), 427–443 (1997)
21. Laurence, M.R., Struth, G.: Completeness results for bi-Kleene algebras and regular pomset languages (2013) (submitted)
22. Mateescu, A., Mateescu, G.D., Rozenberg, G., Salomaa, A.: Shuffle-like operations on ω-words. In: Păun, G., Salomaa, A. (eds.) New Trends in Formal Languages. LNCS, vol. 1218, pp. 395–411. Springer, Heidelberg (1997)

23. Möller, B., Struth, G.: Algebras of modal operators and partial correctness. Theoretical Computer Science 351(2), 221–239 (2006)
24. Nieto, L.P.: The rely-guarantee method in Isabelle/HOL. In: Degano, P. (ed.) ESOP 2003. LNCS, vol. 2618, pp. 348–362. Springer, Heidelberg (2003)
25. Owicki, S.: Axiomatic Proof Techniques for Parallel Programs. PhD thesis, Cornell University (1975)
26. Pous, D.: Kleene algebra with tests and Coq tools for while programs. In: Blazy, S., Paulin-Mohring, C., Pichardie, D. (eds.) ITP 2013. LNCS, vol. 7998, pp. 180–196. Springer, Heidelberg (2013)
27. Pratt, V.R.: Action logic and pure induction. In: van Eijck, J. (ed.) JELIA 1990. LNCS, vol. 478, pp. 97–120. Springer, Heidelberg (1991)
28. Tarlecki, A.: A language of specified programs. Science of Computer Programming 5, 59–81 (1985)
29. Vafeiadis, V.: Modular fine-grained concurrency verification. PhD thesis, University of Cambridge (2008)