DOI 10.1007/s00165-016-0384-0
BCS © 2016
Formal Aspects of Computing (2016) 28: 1057–1078

Formal Aspects of Computing

Generalised rely-guarantee concurrency: an algebraic foundation

Ian J. Hayes

School of Information Technology and Electrical Engineering, The University of Queensland, Brisbane, Australia

Abstract. The rely-guarantee technique allows one to reason compositionally about concurrent programs. To handle interference the technique makes use of rely and guarantee conditions, both of which are binary relations on states. A rely condition is an assumption that the environment performs only atomic steps satisfying the rely relation and a guarantee is a commitment that every atomic step the program makes satisfies the guarantee relation. In order to investigate rely-guarantee reasoning more generally, in this paper we allow interference to be represented by a process rather than a relation and hence derive more general rely-guarantee laws. The paper makes use of a weak conjunction operator between processes, which generalises a guarantee relation to a guarantee process, and introduces a rely quotient operator, which generalises a rely relation to a process. The paper focuses on the algebraic properties of the general rely-guarantee theory. The Jones-style rely-guarantee theory can be interpreted as a model of the general algebraic theory and hence the general laws presented here hold for that theory.

Keywords: Concurrent programming, rely-guarantee concurrency, program verification, program algebra, concurrent Kleene algebra.

1. Introduction

Rely and guarantee conditions. The rely-guarantee technique of Jones [Jon81, Jon83, Jon96] provides a compositional approach to reasoning about concurrent programs. With hindsight, it is obvious that to achieve compositional handling of concurrency, it is necessary to have some way of recording information about interference. This paper generalises the way that interference is recorded. To allow reasoning about a process c in isolation, Jones used a *rely* condition r, that is a binary relation on states. Every atomic step of the environment of c is assumed to satisfy the rely condition r between its before and after states. Any process running in parallel with c also has a rely condition and hence process c will need to ensure every atomic program step it makes satisfies the rely conditions of all processes in its environment. To represent this Jones uses a *guarantee* condition g, that is also a binary relation on states. Every atomic step of c must satisfy g and the relation g should be contained in the rely condition of every process in the environment of c. Jones records a rely-guarantee specification by generalising the judgements of Hoare logic [Hoa69] to a quintuple of the form,

$$\{p, r\}\ c\ \{g, q\}. \tag{1}$$

Correspondence and offprint requests to: Ian J. Hayes, e-mail: Ian.Hayes@itee.uq.edu.au

The process c satisfies the quintuple if, under the assumption that the initial state satisfies p and every atomic step made by the environment satisfies r between its before and after states, every possible execution of c ensures that every atomic program step made by c satisfies g, and the initial and final states of the overall execution of c satisfy the relational postcondition q.

Refinement calculus. This paper uses a refinement calculus approach [Bac81, BvW98, Mor88, Mor94, Mor87] rather than Hoare logic because it allows for simpler presentation of algebraic laws of programming [HHH+87]. Refinement of one command c by another d is written "$c \sqsubseteq d$" and is read "c is refined (implemented) by d". The refinement calculus introduces a postcondition specification command $\lceil q \rceil$ in which the postcondition q is a binary relation on states, and a precondition command $\{p\}$ in which the precondition p is a set of states. The refinement $\{p\}\,;\,\lceil q \rceil \sqsubseteq d$ means d achieves the postcondition q between its before-state and after-state, provided its before-state satisfies p. As an abbreviation the sequential composition operator "; " may be elided so that the above may be written $\{p\}\,\lceil q \rceil$.

Generalised rely-guarantee. The main contribution of this paper is to generalise a rely condition r to a process i specifying the assumed behaviour of interference from the environment. The actual environment should satisfy (i.e. refine) the process specification i. Similarly, the guarantee condition g is generalised to a process j to be "guaranteed" by the implementation. The process that behaves as a process c as well as respecting the guarantee process j is represented by their weak conjunction $j \Cap c$, which is the process that behaves as both j and c unless one of them aborts.[1] A Jones-style guarantee condition g on a terminating command c is represented by the process $\langle g \rangle^\circledast \Cap c$, where $\langle g \rangle$ represents a command that can perform a single atomic program step for which the before and after states satisfy g and $\langle g \rangle^\circledast$ is the process that iterates the atomic step $\langle g \rangle$ any finite number of times, zero or more. An example of a guarantee process that cannot be expressed as a guarantee condition is the sequential composition $\langle \mathsf{id} \rangle^\circledast \langle g \rangle \langle \mathsf{id} \rangle^\circledast$, in which id is the identity relation. It guarantees that a step satisfying g occurs exactly once but allows stuttering steps before and after. The closest guarantee condition is $g \cup \mathsf{id}$ but that allows any number, zero of more, of steps satisfying $g \cup \mathsf{id}$. Section 3 explores the weak conjunction operator and its relationship to Jones-style guarantee conditions [JHC15].

Rely quotients. To specify a process that refines (implements) c, while relying on its environment refining process i, a rely quotient operator $c \mathbin{/\!\!/} i$ is introduced. The rely quotient $c \mathbin{/\!\!/} i$ when run in parallel with i implements c,

$$c \sqsubseteq (c \mathbin{/\!\!/} i) \parallel i.$$

The operator "$/\!\!/$" is chosen to be similar in appearance to the division operator, where in this context "$\parallel$" takes on a role similar to multiplication. Taking "$x \mathbin{/\!\!/} y$" as the ceiling of their integer division $\lceil x/y \rceil$ gives the best analogy: $x \leq \lceil x/y \rceil \times y$. A terminating process specification c with a Jones-style rely condition r is represented by the quotient $c \mathbin{/\!\!/} \langle r \rangle^\circledast$, where $\langle r \rangle^\circledast$ represents the environment process, all atomic steps of which satisfy r. Section 4 explores the properties of the rely quotient operator. Given the weak conjunction and rely quotient operators, the Jones quintuple (1) is equivalent to the following refinement.

$$\{p\}\,(\langle g \rangle^\circledast \Cap (\lceil q \rceil \mathbin{/\!\!/} \langle r \rangle^\circledast)) \quad \sqsubseteq \quad c \tag{2}$$

Concurrency. The parallel introduction law of Jones makes use of both rely and guarantee conditions. In the more general theory presented here, weak conjunction takes on the role of a guarantee and the rely quotient takes on the role of a rely condition. Both generalised operators are used to give a general version of a law for introducing a parallel composition, which has a surprisingly simple and elegant proof (see Sect. 5).

Distribution laws. Section 6 examines the distribution properties of the rely quotient operator over the other operators. In some cases the general distribution laws for weak conjunction and rely quotient require provisos. However, in the relational rely-guarantee model the provisos are all valid and hence the distribution properties hold without proviso. In the general theory the provisos are explicit and hence it is possible to explore alternatives to Jones-style rely-guarantee that allow more expressive rely conditions.

Relationship to relational rely-guarantee. Exploring the theory more generally leads to simpler laws that can be specialised to the relational model. As an example consider the nesting of two rely processes i and j, i.e. $(c \mathbin{/\!\!/} j) \mathbin{/\!\!/} i$.

[1] Earlier publications referred to weak conjunction as *strict* conjunction but the new name is preferred because the operator is weaker than the (strong) conjunction operator that requires both its operands to abort for it to abort.

Let c and d be commands, C be a set of commands and f a monotonic function on commands. The following are the primitive operators and commands used in the algebra.

$$c \sqcap d,\ \ c \sqcup d,\ \ c \parallel d,\ \ c \Cap d,\ \ c /\!\!/ d,\ \ c\,;d,\ \ \mu f,\ \ \nu f,\ \ \bigsqcap C,\ \ \bigsqcup C,\ \ \bot,\ \ \top,\ \ \mathbf{nil},\ \ \mathbf{skip},\ \ \mathbf{chaos}$$

The precedence of binary operators ranges from "$\sqcap$" on the left having the lowest precedence to ";" on the right having the highest precedence, although "$\sqcap$" and "$\sqcup$" have equal precedence. Unary operators have precedence over binary operators. The sequential composition $c\,;d$ is abbreviated as $c\,d$.

Fig. 1. Operators and primitive commands

That corresponds to handling concurrent interference from both i and j and is equivalent to $c /\!\!/ (i \parallel j)$, i.e. an effective rely process of $i \parallel j$. A relational rely condition of r corresponds to a rely process of $\langle r\rangle^\circledast$ and the nesting of two such processes for rely conditions of r_0 and r_1 corresponds to the rely process of $\langle r_0\rangle^\circledast \parallel \langle r_1\rangle^\circledast$, however, this process is equivalent to $\langle r_0 \vee r_1\rangle^\circledast$, corresponding to a relational rely of $r_0 \vee r_1$. This shows how the well known relational rely-guarantee rule, that the effective rely of nested relational rely conditions is their disjunction, can be derived from the more general view that the effective rely process of nested rely processes is their parallel composition.

Section 7 explores the relationship of the more general theory to the Jones-style relational guarantee and rely conditions. The relational rely-guarantee theory of Jones [Jon96] is a model of the general algebraic theory presented in this paper and hence the laws developed in the general theory are also valid for Jones' theory.

Section 8 examines fair parallel and its impact on the rely quotient operator.

Contributions. The main contribution of this paper is to generalise rely and guarantee conditions from relations to arbitrary processes. In order to make our results as widely applicable as possible, we have based our theory on a relatively small set of definitions and axioms. Any model, such as the relational rely-guarantee model, that satisfies the axioms can then make use of all the laws proved here.

Our core theory adds two specification operators, weak conjunction and rely quotient, to the operators of a simple parallel programming language. The weak conjunction operator allows guarantees to be imposed on a process [HJC14]. The rely quotient operator introduced in this paper allows rely conditions to be generalised to processes. There are a number of advantages of exploring the more general operators. Both weak conjunction and rely quotient have simple algebraic properties and this leads to simple and elegant proofs of laws involving these operators. The approach leads to a nice separation of concerns because properties of weak conjunction (guarantees) and rely quotient can be developed separately and then combined to give generalised equivalents of the main laws used for standard rely-guarantee refinements, which are more simply expressed and proven in the general theory. Further, it is much simpler to devise new rely-guarantee refinement laws because the algebra gives a rich theory of properties which simplify discovering proofs.

As an example of the way in which the theory generalises rely and guarantee conditions, in the relational model, as well as being able to express a relational rely condition via the process $\langle r\rangle^\circledast$, one can express rely processes, such as the sequence $\langle r_0\rangle^\circledast\,\langle r_1\rangle^\circledast$, which cannot be expressed via a relational rely condition. The closest rely condition is $r_0 \vee r_1$ but that does not represent the fact that the rely transitions from r_0 to r_1 just once.

2. Basic commands and refinement

Our presentation separates a core algebraic theory of processes from an instantiation of that theory as a relational model similar to that used by Jones [CJ07]. Section 2.1 introduces the operators in our language. Section 2.2 covers the theory of lattices on which the theory for the language is built. Section 2.3 gives the algebraic properties of basic commands. Section 2.4 gives the relational model to provide an intuition for the behaviour of basic commands.

2.1. Operators and primitive commands

The operators and primitive commands of the core language are given in Fig. 1. Typical commands are represented by c, d, i and j; sets of commands by C and D; and monotonic functions from commands to commands by f. The language includes non-deterministic choice, both binary ($c \sqcap d$) and over a set of commands ($\bigsqcap C$), which form infima with respect to the refinement ordering, and their duals $c \sqcup d$ and ($\bigsqcup C$), which form suprema.

Lattice

$$c_0 \sqcap (c_1 \sqcap c_2) = (c_0 \sqcap c_1) \sqcap c_2 \quad (3)$$
$$c_0 \sqcap c_1 = c_1 \sqcap c_0 \quad (4)$$
$$c \sqcap c = c \quad (5)$$
$$c_0 \sqcup (c_1 \sqcup c_2) = (c_0 \sqcup c_1) \sqcup c_2 \quad (6)$$
$$c_0 \sqcup c_1 = c_1 \sqcup c_0 \quad (7)$$
$$c \sqcup c = c \quad (8)$$
$$c_0 \sqcap (c_0 \sqcup c_1) = c_0 \quad (9)$$
$$c_0 \sqcup (c_0 \sqcap c_1) = c_0 \quad (10)$$

Complete distributive lattice

$$c \in C \;\Rightarrow\; \textstyle\bigsqcap C \sqsubseteq c \quad (11)$$
$$(\forall c \in C \cdot d \sqsubseteq c) \;\Rightarrow\; d \sqsubseteq \textstyle\bigsqcap C \quad (12)$$
$$c \in C \;\Rightarrow\; c \sqsubseteq \textstyle\bigsqcup C \quad (13)$$
$$(\forall c \in C \cdot c \sqsubseteq d) \;\Rightarrow\; \textstyle\bigsqcup C \sqsubseteq d \quad (14)$$
$$c \sqcap (\textstyle\bigsqcup D) \;=\; \textstyle\bigsqcup \{d \in D \cdot c \sqcap d\} \quad (15)$$

Fixed point axioms

$$\mu f \;=\; f(\mu f) \quad (16)$$
$$f(x) \sqsubseteq x \Rightarrow \mu f \sqsubseteq x \quad (17)$$
$$\nu f \;=\; f(\nu f) \quad (18)$$
$$x \sqsubseteq f(x) \Rightarrow x \sqsubseteq \nu f \quad (19)$$

Fig. 2. Axioms for lattices and fixed points

Additional binary operators are parallel composition ($c \parallel d$), sequential composition ($c\,;d$), a weak conjunction operator ($c \Cap d$) explained in Sect. 3, and a rely quotient operator ($c /\!\!/ d$) explained in Sect. 4. Commands include least (μf) and greatest (νf) fixed points of monotonic functions over commands. Primitive commands include: the top element in the refinement lattice $\top$ (called *magic* in the refinement calculus); the bottom element $\bot$ (called *abort*); the command that terminates immediately, **nil**, which is the identity of sequential composition; the command that does nothing but doesn't constrain its environment, **skip**, which is the identity of parallel composition; and the command that can do any non-aborting behaviour, **chaos**, which is the identity of weak conjunction.

2.2. Lattices and fixed points

The theory for the language is built on a lattice of commands ordered by *refinement*. The refinement relation "$\sqsubseteq$" is defined in terms of the infimum operator "$\sqcap$"; refinement is reflexive, anti-symmetric and transitive (a partial order).

Definition 1 (*refinement*) For any c, d, $\quad c \sqsubseteq d \;\widehat{=}\; (c \sqcap d) = c$. Equivalently $c \sqsubseteq d \;\Leftrightarrow\; (c \sqcup d) = d$.

The lattice-theoretic axioms of the language are given in Fig. 2. *Com* is the set of all commands and lattice infimum, "$\sqcap$", corresponds to nondeterministic choice.

- $(Com, \sqcap, \sqcup)$ forms a lattice with infimum (greatest lower bound) "$\sqcap$" and supremum (least upper bound) "$\sqcup$", i.e. axioms (3–10) hold.
- The lattice is *complete*, i.e. the infimum $\bigsqcap C$ and the supremum $\bigsqcup C$ exist for all sets of commands C, including empty or infinite C. The infima and suprema satisfy axioms by (11–14).
- The infimum (i.e. nondeterministic choice) distributes over arbitrary suprema (15).
- The bottom element of the lattice is $\bot$. It is the identity of "$\sqcup$" and an annihilator for "$\sqcap$".

$$\bot \;\widehat{=}\; \textstyle\bigsqcup\{\} = \bigsqcap Com \quad (20)$$
$$c \sqcup \bot = c = \bot \sqcup c \quad (21)$$
$$c \sqcap \bot = \bot = \bot \sqcap c \quad (22)$$

- The top element of the lattice is $\top$. It is the identity of "$\sqcap$" and an annihilator for "$\sqcup$".

$$\top \;\widehat{=}\; \textstyle\bigsqcap\{\} = \bigsqcup Com \quad (23)$$
$$c \sqcap \top = c = \top \sqcap c \quad (24)$$
$$c \sqcup \top = \top = \top \sqcup c \quad (25)$$

The following law can be used to handle refinement to or from a nondeterministic choice [BvW98]. A common special case is if C (or D) is a singleton set, i.e. $\bigsqcap\{c\} = c$ (or $\bigsqcap\{d\} = d$).

Lemma 1 (non-deterministic-choice) *For any sets C and D over a complete lattice,*

$$(\forall\, d \in D \cdot (\exists\, c \in C \cdot c \sqsubseteq d)) \quad \Rightarrow \quad (\textstyle\bigsqcap C) \sqsubseteq (\bigsqcap D).$$

The reverse implication does not hold in general, e.g. for $C = \{c_0, c_1\}$ and $D = \{c_0 \sqcap c_1\}$.

The notation $\{c \in C \cdot f\}$ stands for the set of values of the expression f for c an element of C.

Sequential

$$c_0\,(c_1\,c_2) = (c_0\,c_1)\,c_2 \quad (30)$$
$$c\,\mathbf{nil} = c \quad (31)$$
$$\mathbf{nil}\,c = c \quad (32)$$
$$c\,(d_0 \sqcap d_1) = (c\,d_0) \sqcap (c\,d_1) \quad (33)$$
$$(\bigsqcap C)\,d = \bigsqcap\{c \in C \cdot c\,d\} \quad (34)$$
$$\bot\,c = \bot \quad (35)$$

Parallel

$$c_0 \parallel (c_1 \parallel c_2) = (c_0 \parallel c_1) \parallel c_2 \quad (36)$$
$$c_0 \parallel c_1 = c_1 \parallel c_0 \quad (37)$$
$$c \parallel \mathbf{skip} = c \quad (38)$$
$$(\bigsqcap C) \parallel d = \bigsqcap\{c \in C \cdot c \parallel d\} \quad (39)$$

Identities

$$\mathbf{skip}\,\mathbf{skip} = \mathbf{skip} \quad (40)$$
$$\mathbf{skip} \sqsubseteq \mathbf{nil} \quad (41)$$

Weak conjunction

$$c_0 \Cap (c_1 \Cap c_2) = (c_0 \Cap c_1) \Cap c_2 \quad (42)$$
$$c_0 \Cap c_1 = c_1 \Cap c_0 \quad (43)$$
$$c \Cap c = c \quad (44)$$
$$c \Cap \mathbf{chaos} = c \quad (45)$$
$$\mathbf{chaos} \sqsubseteq \mathbf{skip} \quad (46)$$
$$\mathbf{chaos} \parallel \mathbf{chaos} = \mathbf{chaos} \quad (47)$$
$$D \neq \emptyset \Rightarrow c \Cap (\bigsqcap D) = \bigsqcap\{d \in D \cdot c \Cap d\} \quad (48)$$
$$c \Cap (\bigsqcup D) = \bigsqcup\{d \in D \cdot c \Cap d\} \quad (49)$$

Weak exchange axioms

$$(c_0 \parallel c_1) \Cap (d_0 \parallel d_1) \sqsubseteq (c_0 \Cap d_0) \parallel (c_1 \Cap d_1) \quad (50)$$
$$(c_0\,c_1) \Cap (d_0\,d_1) \sqsubseteq (c_0 \Cap d_0)\,(c_1 \Cap d_1) \quad (51)$$

Fig. 3. Axioms for core language of commands

Lemma 2 (operator-monotonic) *If a binary operator "$\circ$" distributes over non-deterministic choice in both arguments then,* $c_0 \sqsubseteq c_1 \wedge d_0 \sqsubseteq d_1 \;\Rightarrow\; c_0 \circ d_0 \sqsubseteq c_1 \circ d_1$.

For a monotonic function f on a complete lattice, the least and greatest fixed points of f, $\mu\,f$ and νf, respectively, satisfy axioms (16–19). As usual, $\mu(\lambda\,x \cdot f(x))$ is abbreviated $\mu\,x \cdot f(x)$. The following lemma allows reasoning about fixed points [ABB+95, BvW98].

Lemma 3 (fusion) *For any monotonic functions F, G and H on complete lattices with order $\sqsubseteq$,*

$$F(\mu\,G) \sqsubseteq \mu\,H \quad \textit{provided } F \circ G \sqsubseteq H \circ F \textit{ and } F \textit{ distributes over arbitrary suprema} \quad (26)$$
$$F(\mu\,G) = \mu\,H \quad \textit{provided } F \circ G = H \circ F \textit{ and } F \textit{ distributes over arbitrary suprema} \quad (27)$$
$$F(\nu\,G) \sqsupseteq \nu\,H \quad \textit{provided } F \circ G \sqsupseteq H \circ F \textit{ and } F \textit{ distributes over arbitrary infima} \quad (28)$$
$$F(\nu\,G) = \nu\,H \quad \textit{provided } F \circ G = H \circ F \textit{ and } F \textit{ distributes over arbitrary infima} \quad (29)$$

where F distributes over arbitrary suprema if $F(\bigsqcup C) = \bigsqcup\{c \in C \cdot F(c)\}$ for all sets of commands C, and F distributes over arbitrary infima if $F(\bigsqcap C) = \bigsqcap\{c \in C \cdot F(c)\}$ for all sets of commands C.

2.3. An algebra for concurrency

The properties of the operators in Fig. 1 are given in terms of a set of axioms given in Definition 2. The axioms have been split into groups which are discussed below. The main results of the paper depend only on these axioms. The majority of the axioms are taken from existing algebraic theories of programs (such as [vW04, HMSW11]), the main exceptions being the axioms for weak conjunction, including the exchange axioms. The axioms hold for the relational model introduced in Sect. 2.4.

Definition 2 (*concurrent-algebra*) The set of commands Com satisfies the axioms given in Fig. 3 in addition to the axioms of lattices from Fig. 2.

- $(Com, ;\ , \mathbf{nil})$ forms a monoid with identity $\mathbf{nil}$, i.e. axioms (30–32). Note that the operator ";" is elided, so that "$c\,;\,d$" is written "$c\,d$".
- Sequential composition distributes over finite non-deterministic choices on the left (33) and arbitrary infima on the right (34) and hence it has a left annihilator of $\top$ (52); $\bot$ is a left annihilator of sequential composition (35).

$$\top\,c = \top \quad (52)$$

- $(Com, \parallel, \mathbf{skip})$ forms a monoid with identity **skip** in which "$\parallel$" is commutative, i.e. axioms (36–38). Note that the identity of parallel composition is different to the identity of sequential composition; that allows a wider range of models, included the relational model introduced in Sect. 2.4.
- Parallel distributes over non-deterministic choice of any set of commands (39), and hence has an annihilator of $\top$.

$$\top \parallel c = \top \tag{53}$$

- The identity of parallel composition, **skip**, sequentially composed with itself is equivalent to **skip** (40) and is refined by the identity of sequential composition, **nil** (41).
- $(Com, \Cap, \mathbf{chaos})$ forms a monoid with identity **chaos** in which "$\Cap$" is commutative and idempotent, i.e. axioms (42–45).
- **chaos** allows any non-aborting behaviour including **skip** (46) and **chaos** in parallel with itself doesn't make it any more (or less) chaotic (47).
- Weak conjunction distributes over the non-deterministic choice of non-empty sets of commands by axiom (48) and hence it distributes over binary choices.

$$c \Cap (d_0 \sqcap d_1) = (c \Cap d_0) \sqcap (c \Cap d_1) \tag{54}$$

- Weak conjunction distributes over arbitrary suprema axiom (49) and hence it has an annihilator of $\bot$.

$$c \Cap \bot = \bot = \bot \Cap c \tag{55}$$

- Weak conjunction does *not* distribute through either parallel or sequential composition, instead it satisfies the weak exchange axioms (50) and (51). Note that axiom (50) is a refinement rather than an equality because, on the left, behaviour of c_0 may synchronise with behaviour of either d_0 or d_1, whereas, on the right, it can only synchronise with behaviour of d_0; axiom (51) is similar; see Sect. 3 for more details.

Note that the set of all commands that refine **chaos** forms a sub-lattice of all non-aborting commands.

The iteration operators are based on von Wright's refinement algebra [vW04]. Kleene algebra provides the finite iteration operator $c^\star$, which iterates c zero or more times but only a finite number of times [Con71, Bli78, Koz97]. A generalisation of this more appropriate for modelling programs is the iteration operator, c°, that iterates c zero or more times, including the possibility of an infinite number of iterations [vW04]. For both these operators the number of iterations they take is non-deterministic.

Definition 3 (*iteration*) The iteration operators are defined via least (μ) and greatest (ν) fixed point operators.

$$c^\star \mathrel{\widehat{=}} (\nu x \cdot \mathbf{nil} \sqcap c\, x) \tag{56} \qquad c^\circ \mathrel{\widehat{=}} (\mu x \cdot \mathbf{nil} \sqcap c\, x) \tag{57}$$

The iteration operators have corresponding induction and folding/unfolding lemmas [BvW98, BvW99, vW04].

Lemma 4 (fold/unfold) *The iteration unfolding properties follow from fixed point unfolding* (18) *and* (16).

$$c^\star = \mathbf{nil} \sqcap c\, c^\star \tag{58} \qquad c^\circ = \mathbf{nil} \sqcap c\, c^\circ \tag{59}$$

Lemma 5 (induction) *The iteration induction properties follow from Lemma 3 and fixed point induction* (19) *and* (17).

$$x \sqsubseteq d \sqcap c\, x \;\Rightarrow\; x \sqsubseteq c^\star\, d \tag{60} \qquad d \sqcap c\, x \sqsubseteq x \;\Rightarrow\; c^\circ\, d \sqsubseteq x \tag{61}$$

We use the term "law" for theorems about our new operators and "lemma" for existing theorems from standard theory. Laws and lemmas share their numbering sequence.

Law 6 (monotonic) If $c \sqsubseteq d$ and $c_0 \sqsubseteq d_0$ and $c_1 \sqsubseteq d_1$, all of the following hold.

$$c_0 \sqcap c_1 \sqsubseteq d_0 \sqcap d_1 \tag{62} \qquad c_0 \Cap c_1 \sqsubseteq d_0 \Cap d_1 \tag{65}$$
$$c_0 \parallel c_1 \sqsubseteq d_0 \parallel d_1 \tag{63} \qquad c^\star \sqsubseteq d^\star \tag{66}$$
$$c_0\, c_1 \sqsubseteq d_0\, d_1 \tag{64} \qquad c^\circ \sqsubseteq d^\circ \tag{67}$$

Proof. Property (62) holds because non-deterministic choice is associative, commutative and idempotent. The proofs of (63–65) follow from Lemma 2 because ";", "$\parallel$" and "$\Cap$" distribute non-deterministic choice in both their left and right arguments. Properties (66) and (67) can be shown by induction, respectively, (60) and (61), using (58) and (59) (see [vW04]). □

2.4. A relational model

In this paper we focus on the algebraic laws satisfied by commands but it is useful to have a model to gain intuitions and ensure the algebra is consistent. The model used corresponds to the rely-guarantee theory of Jones based on Aczel traces [Acz83, dBHdR99, dR01, HJC14]. Typical single-state predicates are represented by p and binary relations on states by g, q and r. The additional commands in the relational model are

$$\pi(r),\ \ \epsilon(r),\ \ \tau(p),\ \ \{p\},\ \ \langle q\rangle,\ \ \lceil q\rceil\,.$$

This set of commands is left open and may be extended with other commands, for example, tests, assignments, conditionals and loops are added in [HJC14].

States (Σ) are modelled by a mapping from variable names to values. The set of program states $\Sigma_\perp$ is extended to include the undefined state $\perp$, which is used to denote that the process has aborted.[2] An *Aczel trace* consists of an initial state $\sigma \in \Sigma$ and a sequence of steps, each of which is either a program step labelled $\Pi(\sigma')$ or an environment step labelled $\mathcal{E}(\sigma')$, where $\sigma' \in \Sigma_\perp$ is the program state after the step. In this paper the term "step" always means an atomic step (either of a program or its environment). A *terminating* Aczel trace ends with a step labelled $\checkmark$. The step $\Pi(\perp)$ is an aborting step of the program and the step $\mathcal{E}(\perp)$ allows an aborting step by the environment. The special steps $\checkmark$, $\Pi(\perp)$ and $\mathcal{E}(\perp)$ can appear only as the last step of a (finite) trace. The set $Trace$ is the set of all valid Aczel traces. The notation $[v_1, v_2, \ldots]$ stands for the sequence containing $v_1, v_2, \ldots$.

A set of traces T is *prefix closed* if $(\sigma, [\,]) \in T$ for all $\sigma \in \Sigma$ and whenever $(\sigma, t) \in T$ and t' is a prefix of t, $(\sigma, t') \in T$. A set of traces T is *abort closed* if whenever $(\sigma, t \frown [\Pi(\perp)]) \in T$, then for any valid trace $(\sigma, t \frown t') \in Trace$, $(\sigma, t \frown t') \in T$. The set of all commands, Com, consists of all the prefix and abort closed subsets of $Trace$.

The command $\pi(r)$ performs a single program step with its before and after states related by r and terminates (68), $\epsilon(r)$ is similar but performs an environment step (69), $\epsilon_\perp(r)$ represents an environment step that satisfies r or allows a parallel process to abort (70), $\tau(p)$ terminates from states satisfying p only (71), $\perp$ aborts immediately and hence can do any behaviour whatsoever (72), $\top$ can make no steps whatsoever (73), and **nil** terminates immediately from any state (74). Recall that $\{x \in S \cdot e\}$ stands for the set of values of e for all values of x in the set S.

$$\pi(r) = prefixes(\{(\sigma,\sigma') \in r \cdot (\sigma, [\Pi(\sigma'), \checkmark])\}) \quad (68)$$
$$\epsilon(r) = prefixes(\{(\sigma,\sigma') \in r \cdot (\sigma, [\mathcal{E}(\sigma'), \checkmark])\}) \quad (69)$$
$$\epsilon_\perp(r) = \epsilon(r) \cup prefixes(\{\sigma \in \Sigma \cdot (\sigma, [\mathcal{E}(\perp)])\}) \quad (70)$$
$$\tau(p) = prefixes(\{\sigma \in p \cdot (\sigma, [\checkmark])\}) \quad (71)$$

$$\perp = Trace \quad (72)$$
$$\top = \{\sigma \in \Sigma \cdot (\sigma, [\,])\} \quad (73)$$
$$\mathbf{nil} = \tau(\Sigma) \quad (74)$$

The set of traces of a non-deterministic choice $\bigsqcap C$ is the union $\bigcup C$ and the supremum $\bigsqcup C$ is the intersection $\bigcap C$. A trace of a sequential composition $(c\ d)$ is any unterminated trace of c or a terminating trace t of c (minus the $\checkmark$ step) followed by a trace of d that starts in the final state of t. Note that an unterminated trace may be infinite or it may be a finite trace that does not end in $\checkmark$.

The traces of $c \parallel d$ are formed by matching traces of c and d. A program step sc of c matches an environment step sd of d if their states are the same, in which case the program step is the step taken by their parallel composition. Identical environment steps of both c and d match to give an environment step of their parallel composition. The following predicate defines matching a step sc of c with a step sd of d to give a step st of $c \parallel d$.

$$\begin{aligned} match\_step(sc, sd, st) \ \widehat{=}\ & \exists \sigma \in \Sigma_\perp \cdot\ sc = \Pi(\sigma) \wedge sd = \mathcal{E}(\sigma) \wedge st = \Pi(\sigma) \vee \\ & sc = \mathcal{E}(\sigma) \wedge sd = \Pi(\sigma) \wedge st = \Pi(\sigma) \vee \\ & sc = \mathcal{E}(\sigma) \wedge sd = \mathcal{E}(\sigma) \wedge st = \mathcal{E}(\sigma) \vee \\ & sc = \checkmark \wedge sd = \checkmark \wedge st = \checkmark \end{aligned}$$

$$\begin{aligned} match\_trace((\sigma_c, t_c), (\sigma_d, t_d), (\sigma, t)) \ \widehat{=}\ & \sigma_c = \sigma_d = \sigma \wedge dom(t_c) = dom(t_d) = dom(t) \wedge \\ & (\forall i \in dom(t) \cdot match\_step(t_c(i), t_d(i), t(i)) \end{aligned}$$

$$c \parallel d \ \widehat{=}\ abort\_close(\{t \in Trace \mid \exists tc \in c, td \in d \cdot match\_trace(tc, td, t)\})$$

Two traces match if they have the same initial state and are the same length (including both being infinite) and all their corresponding steps match. The parallel composition of c and d consists of all their matching traces. The abort closure ensures aborting traces can be refined by any other behaviour.

[2] The symbol $\perp$ is overloaded between the undefined state and the bottom of the lattice of commands, which corresponds to the aborted process. As usual their meaning is resolved by context.

A weak conjunction $c \Cap d$ represents synchronised step-by-step execution of c and d unless one of them aborts. Hence if both c and d can make a step $\Pi(\sigma)$ then so can $c \Cap d$, if both c and d can make a step $\mathcal{E}(\sigma)$ then so can $c \Cap d$, if both c and d can make a step $\checkmark$ then so can $c \Cap d$, but if either c or d can make an aborting step $\Pi(\bot)$ then so can $c \Cap d$. The properties of weak conjunction in the relational model are discussed in more detail in Sect. 3.2.

Other commands in the relational model are defined as follows, where univ stands for the universal relation $\Sigma \times \Sigma$ on states.

$$\mathbf{skip} \mathrel{\widehat{=}} (\epsilon_\bot(\mathsf{univ}))^\circ \tag{75}$$

$$\langle r \rangle \mathrel{\widehat{=}} \mathbf{skip}\, \pi(r)\, \mathbf{skip} \tag{76}$$

$$\{p\} \mathrel{\widehat{=}} \tau(p) \sqcap (\tau(\neg\, p)\, \bot) \tag{77}$$

$$(\mathbf{env}\ r) \mathrel{\widehat{=}} (\pi(\mathsf{univ}) \sqcap \epsilon_\bot(r))^\circ\, (\mathbf{nil} \sqcap \epsilon(\bar{r})\, \bot) \tag{78}$$

The command **skip** does no program steps but allows its environment to do any steps, including abort. The atomic step command $\langle r \rangle$ performs a single program step satisfying r (if possible) and allows its environment to do any steps. The precondition command $\{p\}$ characterises an assumption about the initial state — it terminates immediately if the initial state satisfies p, otherwise it aborts immediately. The command $(\mathbf{env}\ r)$ characterises an assumption that all steps of its environment satisfy the relation r; it aborts if its environment performs a step that does not satisfy r. The relational commands satisfy the following laws [HJC14].

$$p_0 \subseteq p_1 \;\Leftrightarrow\; \{p_0\} \sqsubseteq \{p_1\} \tag{79}$$

$$r_0 \subseteq r_1 \;\Leftrightarrow\; (\mathbf{env}\ r_0) \sqsubseteq (\mathbf{env}\ r_1) \tag{80}$$

$$q_1 \subseteq q_0 \;\Leftrightarrow\; \langle q_0 \rangle \sqsubseteq \langle q_1 \rangle \tag{81}$$

Whereas **nil** terminates immediately allowing no program or environment steps, **skip** allows any number of environment steps, including allowing the environment to abort. That ensures that $c \parallel \mathbf{skip} = c$ because any trace tc of program, environment or termination steps of c is matched by a trace of **skip** to give the same trace tc. Note that $c \parallel \mathbf{nil}$ either terminates immediately if c can, otherwise the trace becomes infeasible. Because **nil** terminates immediately with no intervening environment steps, $\{p\}\,\mathbf{nil}\,\{p\} = \{p\}$, but if **nil** is replaced by **skip**, environment steps allowed by **skip** may change the state thus invalidating p and hence $\{p\}\,\mathbf{skip}\,\{p\} = \{p\}$ does not hold in general.

3. Weak conjunction

A weak conjunction of commands $c \Cap d$ behaves as both c and d provided neither aborts but aborts as soon as either c or d aborts. If neither process aborts, $c \Cap d$ is the same as their supremum $c \sqcup d$ (which in the relational model forms the intersection of traces). Weak conjunction was introduced as part of a relational model in [HJC14] but here it is viewed more abstractly via its axioms in Definition 2. In Sect. 3.1 a set of laws based only on the axioms of weak conjunction are derived. Weak conjunction in the relational model is examined in Sect. 3.2, while Sect. 3.3 looks at its use for representing relational guarantees and Sect. 3.4 presents a set of laws about relational guarantees.

3.1. Laws for weak conjunction

This section presents a number of laws about weak conjunction that can be derived from the axioms presented in Sect. 2.3.

Law 7 (refine-conjunction) If $c_0 \sqsubseteq d$ and $c_1 \sqsubseteq d$, $\quad c_0 \Cap c_1 \sqsubseteq d$.

Proof. The proof follows by Law 6 (monotonic) part (65) and because weak conjunction is idempotent (44): $c_0 \Cap c_1 \sqsubseteq d \Cap d = d$. □

Law 8 (refine-to-conjunction) If $c \sqsubseteq d_0$ and $c \sqsubseteq d_1$, $\quad c \sqsubseteq d_0 \Cap d_1$.

Proof. The proof follows because weak conjunction is idempotent (44) and by Law 6 (monotonic) part (65): $c = c \Cap c \sqsubseteq d_0 \Cap d_1$. □

It is *not* the case that $c \sqsubseteq c \Cap d$ in general, e.g. take d to be $\bot$, however, if d refines the identity of weak conjunction, **chaos**, it does hold.

Law 9 (conjoin-non-aborting) If $\mathbf{chaos} \sqsubseteq d$, $\quad c \sqsubseteq c \Cap d$.

Proof. The proof follows because **chaos** is the identity of weak conjunction (45) and by Law 6 (monotonic) part (65): $c = c \Cap \mathbf{chaos} \sqsubseteq c \Cap d$. □

The following two laws highlight the difference between "$\Cap$" and "$\sqcup$". In general, $c \Cap d \sqsubseteq c \sqcup d$ but they coincide if both arguments are non-aborting.

Law 10 (conjunction-supremum) $c \Cap d \sqsubseteq c \sqcup d$.

Proof. By axiom (13), both $c \sqsubseteq c \sqcup d$ and $d \sqsubseteq c \sqcup d$, and hence by Law 7 (refine-conjunction), $c \Cap d \sqsubseteq c \sqcup d$. □

Law 11 (conjunction-supremum-nonaborting) If $\mathbf{chaos} \sqsubseteq c$ and $\mathbf{chaos} \sqsubseteq d$, $\quad c \Cap d = c \sqcup d$.

Proof. By Law 10 (conjunction-supremum) $c \Cap d \sqsubseteq c \sqcup d$. By Law 9 (conjoin-non-aborting) because both c and d refine **chaos**, both $c \sqsubseteq c \Cap d$ and $d \sqsubseteq c \Cap d$, and hence by axiom (14), $c \sqcup d \sqsubseteq c \Cap d$. □

Law 12 (conjunction-distribute)

$$c \Cap (d_0 \Cap d_1) = (c \Cap d_0) \Cap (c \Cap d_1) \tag{82}$$

$$c \Cap (d_0 \parallel d_1) \sqsubseteq (c \Cap d_0) \parallel (c \Cap d_1) \quad \text{if } c \sqsubseteq c \parallel c \tag{83}$$

$$c \Cap (d_0\ d_1) \sqsubseteq (c \Cap d_0)(c \Cap d_1) \quad \text{if } c \sqsubseteq c\ c \tag{84}$$

$$c^\star \Cap d^\star \sqsubseteq (c \Cap d)^\star \tag{85}$$

$$c^\circ \Cap d^\circ \sqsubseteq (c \Cap d)^\circ \tag{86}$$

Proof. Property (82) follows because weak conjunction is idempotent (44), commutative (43) and associative (42). For (83), assuming $c \sqsubseteq c \parallel c$,

$$\begin{array}{ll} & c \Cap (d_0 \parallel d_1) \\ \sqsubseteq & \text{by Law 6 (monotonic) part (65) assuming } c \sqsubseteq c \parallel c \\ & (c \parallel c) \Cap (d_0 \parallel d_1) \\ \sqsubseteq & \text{exchanging weak conjunction and parallel by axiom (50)} \\ & (c \Cap d_0) \parallel (c \Cap d_1) \end{array}$$

and for (84), assuming $c \sqsubseteq c\ c$,

$$\begin{array}{ll} & c \Cap (d_0\ d_1) \\ \sqsubseteq & \text{by Law 6 (monotonic) part (65) assuming } c \sqsubseteq c\ c \\ & (c\ c) \Cap (d_0\ d_1) \\ \sqsubseteq & \text{exchanging weak conjunction and sequential by axiom (51)} \\ & (c \Cap d_0)(c \Cap d_1) \end{array}$$

Property (85) holds by Lemma 5 for finite iteration (60), if

$$c^\star \Cap d^\star \sqsubseteq \mathbf{nil} \sqcap (c \Cap d)(c^\star \Cap d^\star),$$

which can be shown as follows.

$$\begin{array}{ll} & c^\star \Cap d^\star \\ = & \text{by Lemma 4 part (58)} \\ & (\mathbf{nil} \sqcap c\ c^\star) \Cap d^\star \\ \sqsubseteq & \text{as weak conjunction distributes over non-deterministic choice (48)} \\ & (\mathbf{nil} \Cap d^\star) \sqcap (c\ c^\star \Cap d^\star) \\ \sqsubseteq & \text{by Law 7 (refine-conjunction) as by (58) } d^\star = \mathbf{nil} \sqcap d\ d^\star \text{ and hence } d^\star \sqsubseteq \mathbf{nil} \text{ and } d^\star \sqsubseteq d\ d^\star \\ & \mathbf{nil} \sqcap (c\ c^\star \Cap d\ d^\star) \\ \sqsubseteq & \text{exchanging weak conjunction and sequential by axiom (51)} \\ & \mathbf{nil} \sqcap (c \Cap d)(c^\star \Cap d^\star) \end{array}$$

For (86) the proof uses Lemma 3 part (26) with function $F = (\lambda\, x \cdot c^\circ \Cap x)$, $G = (\lambda\, x \cdot \mathbf{nil} \sqcap d\ x)$ and hence $\mu\, G = d^\circ$, and $H = (\lambda\, x \cdot \mathbf{nil} \sqcap (c \Cap d)\ x)$ and hence $\mu\, H = (c \Cap d)^\circ$. F, G and H are monotonic because "$\sqcap$", ";" and "$\Cap$" are. Property (86) corresponds to $F(\mu\, G) \sqsubseteq \mu\, H$, and Lemma 3 states that this holds if $F \circ G \sqsubseteq H \circ F$, i.e. for any x,

$$c^\circ \Cap (\mathbf{nil} \sqcap d\ x) \sqsubseteq \mathbf{nil} \sqcap (c \Cap d)(c^\circ \Cap x) \tag{87}$$

which holds as follows.

$$
\begin{array}{ll}
& c^\circ \mathbin{⩓} (\mathbf{nil} \sqcap d\ x) \\
= & \quad \text{distributing conjunction over nondeterministic choice (48)} \\
& (c^\circ \mathbin{⩓} \mathbf{nil}) \sqcap (c^\circ \mathbin{⩓} d\ x) \\
\sqsubseteq & \quad \text{by Law 7 (refine-conjunction) as by (59) } c^\circ = \mathbf{nil} \sqcap c\ c^\circ \text{ and hence } c^\circ \sqsubseteq \mathbf{nil} \text{ and } c^\circ \sqsubseteq c\ c^\circ \\
& \mathbf{nil} \sqcap (c\ c^\circ \mathbin{⩓} d\ x) \\
\sqsubseteq & \quad \text{exchanging weak conjunction and sequential by axiom (51)} \\
& \mathbf{nil} \sqcap (c \mathbin{⩓} d)(c^\circ \mathbin{⩓} x)
\end{array}
$$

Lemma 3 also requires that F distributes over arbitrary suprema, which holds because weak conjunction distributes over arbitrary suprema (49). □

The iterations $c^\star$ and c° iterating zero times, are equivalent to **nil**, which in the relational model allows no steps at all, not even environment steps, but for use in guarantees, zero iterations should allow environment steps and hence the iteration operators $c^⊛$ and $c^⊙$ are introduced.

Definition 4 (*guarantee-iteration*)

$$c^⊛ \mathrel{\widehat{=}} c^\star\ \mathbf{skip} \qquad (88) \qquad\qquad c^⊙ \mathrel{\widehat{=}} c^\circ\ \mathbf{skip} \qquad (89)$$

Lemma 13 (iteration) *The following properties follow from Lemmas 4 and 5.*

$$c^⊙ \sqsubseteq c^⊛ \qquad (90) \qquad\qquad c^⊙ \sqsubseteq c^⊙\ c^⊙ \quad \textit{if } c \sqsubseteq \mathbf{skip}\ c \qquad (92)$$

$$c^⊙ \sqsubseteq \mathbf{skip} \qquad (91) \qquad\qquad c^⊙ \sqsubseteq (c^⊙)^\star \quad \textit{if } c \sqsubseteq \mathbf{skip}\ c \qquad (93)$$

Law 14 (conjunction-distribute-guarantee) If $c \sqsubseteq \mathbf{skip}\ c$,

$$c^⊙ \mathbin{⩓} d^\circ \sqsubseteq (c^⊙ \mathbin{⩓} d)^\circ \qquad (94)$$

Proof. The proof can be shown using Lemma 3 part (26) with $G = (\lambda\, x \cdot \mathbf{nil} \sqcap d\ x)$ and hence $\mu\ G = d^\circ$, $H = (\lambda\, x \cdot \mathbf{nil} \sqcap (c^⊙ \mathbin{⩓} d)\ x)$ and hence $\mu\ H = (c^⊙ \mathbin{⩓} d)^\circ$, and $F = (\lambda\, x \cdot c^⊙ \mathbin{⩓} x)$ and hence $F(\mu\ G) = c^⊙ \mathbin{⩓} d^\circ$. Note that F distributes over arbitrary suprema because weak conjunction distributes over arbitrary suprema (49). The proviso for Lemma 3 part (26) requires $c^⊙ \mathbin{⩓} (\mathbf{nil} \sqcap d\ x) \sqsubseteq \mathbf{nil} \sqcap (c^⊙ \mathbin{⩓} d)(c^⊙ \mathbin{⩓} x)$ which holds as follows.

$$
\begin{array}{ll}
& c^⊙ \mathbin{⩓} (\mathbf{nil} \sqcap d\ x) \\
= & \quad \text{distributing weak conjunction over non-deterministic choice (48)} \\
& (c^⊙ \mathbin{⩓} \mathbf{nil}) \sqcap (c^⊙ \mathbin{⩓} d\ x) \\
\sqsubseteq & \quad \text{by Law 7 (refine-conjunction) as } c^⊙ \sqsubseteq \mathbf{skip} \sqsubseteq \mathbf{nil} \text{ by (91) and (41) and } c^⊙ \sqsubseteq c^⊙\ c^⊙ \text{ by (92) as } c \sqsubseteq \mathbf{skip}\ c \\
& \mathbf{nil} \sqcap (c^⊙\ c^⊙ \mathbin{⩓} d\ x) \\
\sqsubseteq & \quad \text{exchanging weak conjunction and sequential composition by axiom (51)} \\
& \mathbf{nil} \sqcap (c^⊙ \mathbin{⩓} d)(c^⊙ \mathbin{⩓} x)
\end{array}
$$

□

3.2. Weak conjunction in the relational model

In the relational model weak conjunction corresponds to synchronised execution of atomic steps by both processes unless either process aborts, i.e. every non-aborting step taken by $c \mathbin{⩓} d$ must be a step allowed by both c and d. If either process aborts, the conjunction aborts (55). The weak conjunction of two atomic step commands $\langle g \rangle$ and $\langle r \rangle$ can perform a program step that satisfies both g and r (95). An atomic step $\langle g \rangle$ allows any environment step whatsoever and hence two atomic step commands synchronise trivially on environment steps. More generally, the first program steps of conjoined commands synchronise followed by the weak conjunction of the remainder of both commands (96). If one command in a weak conjunction must do a program step but the other cannot, their conjunction never terminates and does no program steps (97).

$$\langle g \rangle \mathbin{⩓} \langle r \rangle = \langle g \cap r \rangle \qquad (95)$$

$$(\langle g \rangle\ c) \mathbin{⩓} (\langle r \rangle\ d) = \langle g \cap r \rangle\ (c \mathbin{⩓} d) \qquad (96)$$

$$\mathbf{skip} \mathbin{⩓} (\langle g \rangle\ c) = \mathbf{skip}\ \top \qquad (97)$$

The command **chaos** performs any sequence of non-aborting program steps and allows any environment steps, while **term** allows only a finite sequence of non-aborting program steps and any environment steps. Both are defined in terms of the iteration operators that allow environment steps for zero iterations.

$$\mathbf{chaos} \mathrel{\widehat{=}} \langle \mathsf{univ} \rangle^{\circledcirc} \quad (98) \qquad \mathbf{term} \mathrel{\widehat{=}} \langle \mathsf{univ} \rangle^{\circledast} \quad (99)$$

Iterations of atomic steps satisfy the following properties [HJC14].

$$r_1 \subseteq r_0 \;\Rightarrow\; \langle r_0 \rangle^{\circledast} \sqsubseteq \langle r_1 \rangle^{\circledast} \quad (100) \qquad \langle r_0 \cup r_1 \rangle^{\circledast} = \langle r_0 \rangle^{\circledast} \parallel \langle r_1 \rangle^{\circledast} \quad (102)$$

$$r_1 \subseteq r_0 \;\Rightarrow\; \langle r_0 \rangle^{\circledcirc} \sqsubseteq \langle r_1 \rangle^{\circledcirc} \quad (101) \qquad \langle r_0 \cup r_1 \rangle^{\circledcirc} \sqsubseteq \langle r_0 \rangle^{\circledcirc} \parallel \langle r_1 \rangle^{\circledcirc} \quad (103)$$

$$\langle r \rangle^{\circledcirc} = \langle r \rangle^{\circledcirc} \parallel \langle r \rangle^{\circledcirc} \quad (104)$$

Properties (100) and (101) follow using (81) from (66) and (67), respectively.

In the relational model a command c preconditioned by the state predicate p is represented by $(\{p\}\, c)$. If p holds initially, $\{p\}$ behaves as **nil** and hence $(\{p\}\, c)$ behaves as c but if p does not hold initially, the preconditioned command aborts. A precondition distributes into both a weak conjunction and into a parallel composition. These laws follow from the definition of a precondition command (77) and distribution properties in the relational semantics.

Law 15 (precondition-conjunction) $\{p\}(c \mathbin{⩓} d) = (\{p\}\, c) \mathbin{⩓} (\{p\}\, d)$.

Law 16 (precondition-parallel) $\{p\}(c \parallel d) = (\{p\}\, c) \parallel (\{p\}\, d)$.

Morgan's specification command, $\lceil q \rceil$, is refined by any program that terminates with its initial and final states related by q provided there is no interference from the environment [Mor88].

$$\lceil q \rceil \mathrel{\widehat{=}} \bigsqcap \{\sigma \in \Sigma \cdot \tau(\{\sigma\})\, \mathbf{term}\, \tau(\{\sigma' \mid (\sigma, \sigma') \in q\})\} \mathbin{⩓} (\mathbf{env}\ \mathsf{id}) \quad (105)$$

The behaviour of $\lceil q \rceil$ consists of terminating traces that start in some state σ and terminate in a state σ' such that $(\sigma, \sigma') \in q$. It assumes all steps of its environment do not modify the state (i.e. satisfy the identity relation id). Its behaviour includes finite infeasible traces starting from any state and traces ending in an infinite sequence of environment steps. Conjoining two specifications achieves the conjunction of their postconditions.

$$\lceil q_0 \rceil \mathbin{⩓} \lceil q_1 \rceil = \lceil q_0 \cap q_1 \rceil \quad (106)$$

3.3. Relationship to Jones-style guarantee

Jones introduced the idea of using a guarantee condition g, a binary relation between states, to express the fact that every atomic program step a process makes is guaranteed to satisfy g between its before-state and after-state [Jon83]. The relation g is required to be reflexive so that stuttering steps are allowed. A guarantee g on a terminating command c can be defined in terms of a weak conjunction as $\langle g \rangle^{\circledast} \mathbin{⩓} c$. The weak conjunction with $\langle g \rangle^{\circledast}$ restricts the behaviour of c so that every atomic program step satisfies g. The command $\langle g \rangle^{\circledast}$ is used rather than $\langle g \rangle^{\star}$ so that zero iterations corresponds to **skip** rather than **nil** and hence does not constrain environment steps in this case. More generally, if c is not assumed to be terminating, a guarantee is represented by $\langle g \rangle^{\circledcirc} \mathbin{⩓} c$. Possibly infinite iteration is used rather than finite iteration because weak conjunction with finite iteration forces termination and hence is too strong [HJC14]. Termination of $\langle g \rangle^{\circledcirc} \mathbin{⩓} c$ depends only on whether c terminates if its traces are restricted to program steps satisfying g. The guarantee component $\langle g \rangle^{\circledcirc}$ is non-aborting and hence any aborting behaviour can only arise from c. Using the supremum operator $\langle g \rangle^{\circledcirc} \sqcup c$ would be too strong a guarantee because $\langle g \rangle^{\circledcirc}$ has only non-aborting traces and hence would mask any aborting behaviour of c.

A guarantee relation g in the style of Jones is represented here by an iterated atomic step satisfying the relation, either $\langle g \rangle^{\circledast}$ or $\langle g \rangle^{\circledcirc}$. By treating guarantees as processes more expressive guarantee conditions can be expressed, for example, the process $\langle g_0 \rangle^{\circledast} \langle g_1 \rangle^{\circledast}$ represents a guarantee of g_0 initially, followed at some point by a switch to a guarantee of g_1. As another example, the process $\langle \mathsf{id} \rangle^{\circledast} \langle g \rangle \langle \mathsf{id} \rangle^{\circledast}$ represents a guarantee to perform a single step satisfying g surrounded by any finite number of steps that don't modify any variables. Neither of these guarantee processes can be represented as a single guarantee relation unless additional variables that distinguish the phases of the guarantees are used. It is possible to encode a sequence such as $\langle g_0 \rangle^{\circledast} \langle g_1 \rangle^{\circledast}$ via the use of an additional boolean variable b which is initially false: $(\neg\, b \wedge g_0) \vee (b \wedge g_1 \wedge b')$, where it is assumed b is set to true for the transition from a guarantee of g_0 to g_1.

3.4. Laws for guarantees

If $g_0 \subseteq g_1$, then a guarantee of g_0 is stronger than a guarantee of g_1.

Law 17 (guarantee-strengthen) For any command c and relations g_0 and g_1 such that $g_0 \subseteq g_1$,

$$\langle g_1 \rangle^\circledcirc \mathbin{⩓} c \;\sqsubseteq\; \langle g_0 \rangle^\circledcirc \mathbin{⩓} c.$$

Proof. By (101), $\langle g_1 \rangle^\circledcirc \sqsubseteq \langle g_0 \rangle^\circledcirc$, and hence the law follows by Law 6 (monotonic) part (65). □

Law 18 (guarantee-introduce) $\quad c \;\sqsubseteq\; \langle g \rangle^\circledcirc \mathbin{⩓} c.$

Proof. The proof follows by Law 9 (conjoin-non-aborting) because by (98) $\mathbf{chaos} = \langle \mathsf{univ} \rangle^\circledcirc \sqsubseteq \langle g \rangle^\circledcirc$ by (101). □

Law 19 (conjunction-atomic-iterated) $\quad \langle g_0 \rangle^\circledcirc \mathbin{⩓} \langle g_1 \rangle^\circledcirc \;=\; \langle g_0 \cap g_1 \rangle^\circledcirc$

Proof. The refinement from left to right follows by Law 7 (refine-conjunction) because by (101) both $\langle g_0 \rangle^\circledcirc$ and $\langle g_1 \rangle^\circledcirc$ are refined by $\langle g_0 \cap g_1 \rangle^\circledcirc$. The refinement from right to left can be proved using Lemma 5 part (61) using (96) and (97). □

Law 20 (guarantee-nested) $\quad \langle g_0 \rangle^\circledcirc \mathbin{⩓} \langle g_1 \rangle^\circledcirc \mathbin{⩓} c \;=\; \langle g_0 \cap g_1 \rangle^\circledcirc \mathbin{⩓} c$

Proof. By Law 19 (conjunction-atomic-iterated), $\langle g_0 \rangle^\circledcirc \mathbin{⩓} \langle g_1 \rangle^\circledcirc = \langle g_0 \cap g_1 \rangle^\circledcirc$. □

A guarantee distributes through non-deterministic choice, weak conjunction, parallel and sequential composition, and finite and infinite iterations.

Law 21 (guarantee-distribute)

$$\langle g \rangle^\circledcirc \mathbin{⩓} (c \sqcap d) \;=\; (\langle g \rangle^\circledcirc \mathbin{⩓} c) \sqcap (\langle g \rangle^\circledcirc \mathbin{⩓} d) \tag{107}$$
$$\langle g \rangle^\circledcirc \mathbin{⩓} (c \mathbin{⩓} d) \;=\; (\langle g \rangle^\circledcirc \mathbin{⩓} c) \mathbin{⩓} (\langle g \rangle^\circledcirc \mathbin{⩓} d) \tag{108}$$
$$\langle g \rangle^\circledcirc \mathbin{⩓} (c \parallel d) \;\sqsubseteq\; (\langle g \rangle^\circledcirc \mathbin{⩓} c) \parallel (\langle g \rangle^\circledcirc \mathbin{⩓} d) \tag{109}$$
$$\langle g \rangle^\circledcirc \mathbin{⩓} (c\; d) \;\sqsubseteq\; (\langle g \rangle^\circledcirc \mathbin{⩓} c)(\langle g \rangle^\circledcirc \mathbin{⩓} d) \tag{110}$$
$$\langle g \rangle^\circledcirc \mathbin{⩓} c^\star \;\sqsubseteq\; (\langle g \rangle^\circledcirc \mathbin{⩓} c)^\star \tag{111}$$
$$\langle g \rangle^\circledcirc \mathbin{⩓} c^\circ \;\sqsubseteq\; (\langle g \rangle^\circledcirc \mathbin{⩓} c)^\circ \tag{112}$$

Proof. Property (107) holds because weak conjunction distributes over non-deterministic choice (48), and (108–111) hold by the corresponding properties (82–85) of Law 12 (conjunction-distribute). For property (109) the proviso holds because $\langle g \rangle^\circledcirc = \langle g \rangle^\circledcirc \parallel \langle g \rangle^\circledcirc$ by (104); and for property (110) the proviso holds because $\langle g \rangle^\circledcirc \;\sqsubseteq\; \langle g \rangle^\circledcirc \langle g \rangle^\circledcirc$ by (92). Property (111) holds by (85) because $\langle g \rangle^\circledcirc \sqsubseteq (\langle g \rangle^\circledcirc)^\star$ by (93). Both (92) and (93) require the side condition $\langle g \rangle \sqsubseteq \mathbf{skip}\, \langle g \rangle$, which holds by (76). Property (112) follows from Law 14 (conjunction-distribute-guarantee). □

4. The rely quotient command

Jones introduced the idea of a rely condition, a reflexive relation assumed to be satisfied by every atomic step of the interference from the environment of a process [Jon83]. In essence it abstracts the environment by a process $\langle r \rangle^\circledast$ that executes steps satisfying the rely condition r. In the general algebra the environment is represented by an arbitrary process i. The rules of Jones then become a special case when $i = \langle r \rangle^\circledast$ (see Sect. 7). To handle relies in the general algebra, a rely quotient operator "$/\!\!/$" is introduced. It is defined so that $c /\!\!/ i$ in parallel with i implements c, i.e.,

$$c \;\sqsubseteq\; (c /\!\!/ i) \parallel i, \tag{113}$$

and furthermore for any process d, if $c \sqsubseteq d \parallel i$ then $c /\!\!/ i \sqsubseteq d$. For example, because $\langle r_0 \vee r_1 \rangle^\circledast \sqsubseteq \langle r_0 \rangle^\circledast \parallel \langle r_1 \rangle^\circledast$ holds in the relational model, one refinement of the quotient $\langle r_0 \vee r_1 \rangle^\circledast /\!\!/ \langle r_1 \rangle^\circledast$ is $\langle r_0 \rangle^\circledast$.

The motivation for the rely quotient is similar to that for the weakest pre- and post-specifications of Hoare and He [HH86], although they deal with residuals of sequential composition rather than parallel composition, and weakest environment of Zhou and Hoare [ZH81, Zho82]. The rely quotient $c /\!\!/ i$ is defined as the non-deterministic choice over all commands d satisfying the defining property of the rely quotient: $c \;\sqsubseteq\; d \parallel i$.

Definition 5 (*rely-quotient*) $c \mathbin{/\!\!/} i \;\widehat{=}\; \bigsqcap\{d \mid (c \sqsubseteq d \parallel i)\}$.

This definition is similar to defining division over the positive integers in terms of multiplication and minimum ($\bigsqcap$).

$$\lceil c/i \rceil \mathrel{\widehat{=}} \bigsqcap\{d \mid (c \leq d \times i)\}$$

The only command d satisfying $c \sqsubseteq d \parallel i$ might be the infeasible command $\top$, in which case $c \mathbin{/\!\!/} i$ is infeasible. In particular, taking the interference i to be the aborting process $\bot$ gives, $c \mathbin{/\!\!/} \bot = \bigsqcap\{d \mid (c \sqsubseteq d \parallel \bot)\} = \top$, unless $c = \bot$, in which case $\bot \mathbin{/\!\!/} \bot = \bot$.

Because the rely quotient operation is defined in terms of nondeterministic choice and parallel composition, its instantiation in the relational model follows directly from its definition. For completeness, an expansion of its definition in the relational model is given below, in which $/\!\!/_r$ and $\parallel_r$ stand for the interpretations of these operators in the relational model; recall that nondeterministic choice corresponds to set union and refinement to set containment.

$$\begin{aligned} c \mathbin{/\!\!/_r} i &= \bigcup\{d \in Com \mid c \supseteq d \parallel_r i\} \\ &= \bigcup\{d \in Com \mid c \supseteq abort\_close(\{t \in Trace \mid \exists\, td \in d, ti \in i \cdot match\_trace(td, ti, t)\})\} \end{aligned}$$

A full appreciation of the utility of the rely quotient operator flows from its use in introducing a parallel composition in Sect. 5 but first we examine a set of basic laws that it satisfies.

4.1. Laws for rely quotients

The following law shows that the rely quotient command satisfies its motivating property (113). The law corresponds to $c \leq \lceil c/i \rceil \times i$ for positive integer division.

Law 22 (rely-quotient) $\quad c \sqsubseteq (c \mathbin{/\!\!/} i) \parallel i.$

Proof. The notation $\{x \mid p \cdot e\}$ used below represents the set of values of the expression e for x ranging over values that satisfy the predicate p.

$$\begin{array}{ll} & c \sqsubseteq (c \mathbin{/\!\!/} i) \parallel i \\ \Leftrightarrow & \text{by Definition 5} \\ & c \sqsubseteq \bigsqcap\{d \mid (c \sqsubseteq d \parallel i)\} \parallel i \\ \Leftrightarrow & \text{distributing parallel over non-deterministic choice (39)} \\ & c \sqsubseteq \bigsqcap\{d \mid (c \sqsubseteq d \parallel i) \cdot (d \parallel i)\} \\ \Leftarrow & \text{by Lemma 1} \\ & \forall\, d \in \{d \mid (c \sqsubseteq d \parallel i)\} \cdot c \sqsubseteq (d \parallel i) \end{array}$$

□

The following fundamental law shows that the rely quotient is the least command satisfying its defining property. It provides the basis for the proof of many of the laws that follow and shows the Galois connection between rely quotient and parallel composition [Aar92, BCG02]. It corresponds to $\lceil c/i \rceil \leq d \Leftrightarrow c \leq d \times i$ for positive integer division.

Law 23 (rely-refinement) $\quad c \mathbin{/\!\!/} i \sqsubseteq d \quad \Leftrightarrow \quad c \sqsubseteq d \parallel i.$

Proof. For the proof from right to left assume $c \sqsubseteq d \parallel i$.

$$\begin{array}{ll} & c \mathbin{/\!\!/} i \sqsubseteq d \\ \Leftrightarrow & \text{by Definition 5} \\ & \bigsqcap\{d_1 \mid (c \sqsubseteq d_1 \parallel i)\} \sqsubseteq d \\ \Leftarrow & \text{by Lemma 1} \\ & \exists\, d_0 \in \{d_1 \mid (c \sqsubseteq d_1 \parallel i)\} \cdot d_0 \sqsubseteq d \\ \Leftarrow & \text{by assumption } d \in \{d_1 \mid (c \sqsubseteq d_1 \parallel i)\} \\ & d \sqsubseteq d \end{array}$$

The proof from left to right assumes $c \mathbin{/\!\!/} i \sqsubseteq d$ and starts with Law 22 (rely-quotient).

$$
\begin{array}{ll}
 & c \sqsubseteq (c \mathbin{/\!\!/} i) \parallel i \\
\Rightarrow & \text{by Law 6 (monotonic) part (63) as } c \mathbin{/\!\!/} i \sqsubseteq d \\
 & c \sqsubseteq d \parallel i
\end{array}
$$

□

The property in Law 23 (rely-refinement) could be used as an alternative definition of the rely quotient operator. From Galois theory, the rely quotient (lower adjoint) is uniquely defined by the Galois connection provided parallel distributes over non-deterministic choice (39).

Because **skip** is the identity of parallel composition, it is also the right identity of the rely quotient. This is similar to 1 being the right identity of integer division ($c/1 = c$).

Law 24 (rely-identity-right) $\quad c \mathbin{/\!\!/} \mathbf{skip} = c$

Proof. The law holds by indirect equality if for all x, $c \mathbin{/\!\!/} \mathbf{skip} \sqsubseteq x \Leftrightarrow c \sqsubseteq x$, which holds by Law 23 (rely-refinement) as follows: $c \mathbin{/\!\!/} \mathbf{skip} \sqsubseteq x \;\Leftrightarrow\; c \sqsubseteq x \parallel \mathbf{skip} \;\Leftrightarrow\; c \sqsubseteq x$. □

The following two laws correspond to $c \leq d \Rightarrow \lceil c/i \rceil \leq \lceil d/i \rceil$ and $i \leq j \Rightarrow \lceil c/j \rceil \leq \lceil c/i \rceil$ for positive integer division.

Law 25 (rely-monotonic) $\quad c \sqsubseteq d \;\Rightarrow\; (c \mathbin{/\!\!/} i) \sqsubseteq (d \mathbin{/\!\!/} i)$.

Proof. By Law 23 (rely-refinement), $(c \mathbin{/\!\!/} i) \sqsubseteq (d \mathbin{/\!\!/} i)$ holds if $c \sqsubseteq (d \mathbin{/\!\!/} i) \parallel i$, which holds by the assumption $c \sqsubseteq d$ and Law 22 (rely-quotient) because $c \sqsubseteq d \sqsubseteq (d \mathbin{/\!\!/} i) \parallel i$. □

Law 26 (rely-weaken) $\quad i \sqsubseteq j \;\Rightarrow\; (c \mathbin{/\!\!/} j) \sqsubseteq (c \mathbin{/\!\!/} i)$.

Proof. By Law 23 (rely-refinement) $(c \mathbin{/\!\!/} j) \sqsubseteq (c \mathbin{/\!\!/} i)$ holds if $c \sqsubseteq (c \mathbin{/\!\!/} i) \parallel j$, which holds as follows.

$$
\begin{array}{ll}
 & c \\
\sqsubseteq & \text{by Law 22 (rely-quotient)} \\
 & (c \mathbin{/\!\!/} i) \parallel i \\
\sqsubseteq & \text{by Law 6 (monotonic) part (63) as } i \sqsubseteq j \\
 & (c \mathbin{/\!\!/} i) \parallel j
\end{array}
$$

□

[Italic text between horizontal lines partitions out material that applies only to the relational model.]

For relational rely conditions, if $r_1 \subseteq r_0$, then by (100), $\langle r_0 \rangle^{\circledast} \sqsubseteq \langle r_1 \rangle^{\circledast}$, and applying Law 26 (rely-weaken) gives $(c \mathbin{/\!\!/} \langle r_1 \rangle^{\circledast}) \sqsubseteq (c \mathbin{/\!\!/} \langle r_0 \rangle^{\circledast})$, i.e. the relational rely condition can be weakened in a refinement.

A nested rely $(c \mathbin{/\!\!/} j) \mathbin{/\!\!/} i$ corresponds to implementing c within environment j, all within in environment i, i.e. c is implemented in environment $i \parallel j$. The next law corresponds to $\lceil \lceil c/i \rceil / j \rceil = \lceil c/(i \times j) \rceil$ for positive integer division.

Law 27 (rely-nested) $\quad (c \mathbin{/\!\!/} j) \mathbin{/\!\!/} i \;=\; c \mathbin{/\!\!/} (i \parallel j)$.

Proof. The law follows by indirect equality if for all x, $(c \mathbin{/\!\!/} j) \mathbin{/\!\!/} i \sqsubseteq x \;\Leftrightarrow\; c \mathbin{/\!\!/} (i \parallel j) \sqsubseteq x$, which is shown as follows.

$$
\begin{array}{ll}
 & (c \mathbin{/\!\!/} j) \mathbin{/\!\!/} i \sqsubseteq x \\
\Leftrightarrow & \text{by Law 23 (rely-refinement)} \\
 & c \mathbin{/\!\!/} j \sqsubseteq x \parallel i \\
\Leftrightarrow & \text{by Law 23 (rely-refinement)} \\
 & c \sqsubseteq x \parallel i \parallel j \\
\Leftrightarrow & \text{by Law 23 (rely-refinement)} \\
 & c \mathbin{/\!\!/} (i \parallel j) \sqsubseteq x
\end{array}
$$

□

Because parallel is commutative, it follows that $(c /\!\!/ j) /\!\!/ i \;=\; c /\!\!/ (i \parallel j) \;=\; c /\!\!/ (j \parallel i) \;=\; (c /\!\!/ i) /\!\!/ j$.

For relational rely conditions by property (102), $\langle r_0\rangle^{\circledast} \parallel \langle r_1\rangle^{\circledast} \;=\; \langle r_0 \cup r_1\rangle^{\circledast}$, *and hence by Law 27 (rely-nested) nested relational relies of* r_0 *and* r_1 *give an effective rely of* $r_0 \cup r_1$.

$$(c /\!\!/ \langle r_1\rangle^{\circledast}) /\!\!/ \langle r_0\rangle^{\circledast} \;=\; c /\!\!/ (\langle r_0\rangle^{\circledast} \parallel \langle r_1\rangle^{\circledast}) \;=\; c /\!\!/ \langle r_0 \cup r_1\rangle^{\circledast}. \tag{114}$$

5. Parallel-introduction law

The prime motivation of Jones [Jon83] for introducing rely and guarantee conditions was to support reasoning about parallel compositions. In the current paper a guarantee condition is generalised to a weak conjunction with a process, and a rely condition by a rely quotient by a process. Law 28 (parallel-introduce) provides an general law for introducing a parallel composition. The guarantee j of the first branch of the parallel corresponds to the rely of the second branch and vice versa for i.

Law 28 (parallel-introduce) $\quad c \mathbin{⫛} d \;\sqsubseteq\; (j \mathbin{⫛} (c /\!\!/ i)) \;\parallel\; (i \mathbin{⫛} (d /\!\!/ j))$

Proof. By Law 22 (rely-quotient) both $c \;\sqsubseteq\; (c /\!\!/ i) \parallel i$ and $d \;\sqsubseteq\; (d /\!\!/ j) \parallel j$ and hence the proof follows using these two properties in the first step.

$$\begin{array}{ll} & c \mathbin{⫛} d \\ \sqsubseteq & \text{by Law 6 (monotonic) part (65) and parallel is commutative (37)} \\ & ((c /\!\!/ i) \parallel i) \mathbin{⫛} (j \parallel (d /\!\!/ j)) \\ \sqsubseteq & \text{exchanging weak conjunction and parallel by axiom (50)} \\ & ((c /\!\!/ i) \mathbin{⫛} j) \;\parallel\; (i \mathbin{⫛} (d /\!\!/ j)) \end{array}$$

□

The simplicity and elegance of the proof of this fundamental law for handling rely-guarantee concurrency is an indication that weak conjunction and rely quotient are well chosen abstractions. The relationship to the parallel law of Jones is explored in Sect. 7 but first distribution properties of rely quotients need to be explored.

6. Distribution of rely quotients

Law 28 (parallel-introduce) introduces rely quotients of the form $c /\!\!/ i$ for some specification c. One way of refining such a quotient is to refine c, for example, c may be refined to a sequential composition $c_0\, c_1$. Law 25 (rely-monotonic) then gives that $c /\!\!/ i \sqsubseteq (c_0\, c_1) /\!\!/ i$. To further refine this it is useful to have a distribution law that allows the rely quotient to be distributed over the sequential composition, i.e. $(c_0\, c_1) /\!\!/ i \;\sqsubseteq\; (c_0 /\!\!/ i)(c_1 /\!\!/ i)$. A proviso is needed for this refinement to be valid (see Law 32 below). This section investigates laws for distributing rely quotients over the other operators. A rely quotient distributes straightforwardly over both weak conjunction and non-deterministic choice.

Law 29 (rely-distribute-conjunction) $\quad (c \mathbin{⫛} d) /\!\!/ i \;\sqsubseteq\; (c /\!\!/ i) \mathbin{⫛} (d /\!\!/ i)$

Proof. By Law 23 (rely-refinement) the law is equivalent to $c \mathbin{⫛} d \;\sqsubseteq\; ((c /\!\!/ i) \mathbin{⫛} (d /\!\!/ i)) \parallel i$.

$$\begin{array}{ll} & c \mathbin{⫛} d \\ \sqsubseteq & \text{by Law 22 (rely-quotient) twice} \\ & ((c /\!\!/ i) \parallel i) \mathbin{⫛} ((d /\!\!/ i) \parallel i) \\ \sqsubseteq & \text{exchanging weak conjunction and parallel by axiom (50)} \\ & ((c /\!\!/ i) \mathbin{⫛} (d /\!\!/ i)) \;\parallel\; (i \mathbin{⫛} i) \\ = & \text{as “⫛” is idempotent (44)} \\ & ((c /\!\!/ i) \mathbin{⫛} (d /\!\!/ i)) \;\parallel\; i \end{array}$$

□

Law 30 (rely-distribute-choice) $(c \sqcap d) /\!\!/ i \sqsubseteq (c /\!\!/ i) \sqcap (d /\!\!/ i)$

Proof. By Law 23 (rely-refinement) the law is equivalent to $c \sqcap d \sqsubseteq ((c /\!\!/ i) \sqcap (d /\!\!/ i)) \parallel i$.

$$
\begin{array}{ll}
& c \sqcap d \\
\sqsubseteq & \text{by Law 22 (rely-quotient) twice} \\
& ((c /\!\!/ i) \parallel i) \sqcap ((d /\!\!/ i) \parallel i) \\
= & \text{distributing parallel over non-deterministic choice (39)} \\
& ((c /\!\!/ i) \sqcap (d /\!\!/ i)) \parallel i
\end{array}
$$

□

Distribution of the rely quotient over parallel requires a proviso on the interference i that $i \parallel i \sqsubseteq i$. That distribution law follows from a more general law with a parallel in both arguments of the quotient.

Law 31 (rely-distribute-parallel)

$$(c \parallel d) /\!\!/ (i \parallel j) \sqsubseteq (c /\!\!/ i) \parallel (d /\!\!/ j) \tag{115}$$

$$(c \parallel d) /\!\!/ i \sqsubseteq (c /\!\!/ i) \parallel (d /\!\!/ i) \quad \text{if } i \parallel i \sqsubseteq i \tag{116}$$

Proof. By Law 23 (rely-refinement), (115) holds if $c \parallel d \sqsubseteq (c /\!\!/ i) \parallel (d /\!\!/ j) \parallel i \parallel j$, which holds as follows.

$$
\begin{array}{ll}
& c \parallel d \\
\sqsubseteq & \text{by Law 22 (rely-quotient) twice} \\
& ((c /\!\!/ i) \parallel i) \parallel ((d /\!\!/ j) \parallel j) \\
= & \text{by associativity (36) and commutativity (37) of parallel} \\
& (c /\!\!/ i) \parallel (d /\!\!/ j) \parallel i \parallel j
\end{array}
$$

The proof of (116) uses (115) with $j = i$ as follows.

$$
\begin{array}{ll}
& (c \parallel d) /\!\!/ i \\
\sqsubseteq & \text{by Law 26 (rely-weaken) using assumption } i \parallel i \sqsubseteq i \\
& (c \parallel d) /\!\!/ (i \parallel i) \\
\sqsubseteq & \text{by part (115) with } j = i \\
& (c /\!\!/ i) \parallel (d /\!\!/ i)
\end{array}
$$

□

For a relational rely condition, if $i = \langle r \rangle^\circledast$ then by (102), $\langle r \rangle^\circledast \parallel \langle r \rangle^\circledast = \langle r \cup r \rangle^\circledast = \langle r \rangle^\circledast$, and hence the proviso for (116) holds in this case. The fact that the proviso for a relational rely condition holds allows rely conditions to be distributed into any parallel composition.

Distribution of a rely quotient of a process i over a sequential composition requires that separate occurrences of i running in parallel with each command in the sequence can be refined to a single occurrence of i run in parallel with the sequence as given by condition (117).

Law 32 (rely-distribute-sequential) If for process i,

$$\forall\, c_0, c_1 \cdot (c_0 \parallel i)(c_1 \parallel i) \sqsubseteq (c_0\, c_1) \parallel i, \tag{117}$$

then

$$(c\, d) /\!\!/ i \sqsubseteq (c /\!\!/ i)(d /\!\!/ i). \tag{118}$$

Proof. By Law 23 (rely-refinement), (118) is equivalent to $c\, d \sqsubseteq ((c /\!\!/ i)(d /\!\!/ i)) \parallel i$.

$$
\begin{array}{ll}
& c\, d \\
\sqsubseteq & \text{by Law 22 (rely-quotient) twice} \\
& ((c /\!\!/ i) \parallel i)((d /\!\!/ i) \parallel i) \\
\sqsubseteq & \text{by assumption (117) with } c_0 = c /\!\!/ i \text{ and } c_1 = d /\!\!/ i \\
& ((c /\!\!/ i)(d /\!\!/ i)) \parallel i
\end{array}
$$

□

For a relational rely condition, if $i = \langle r \rangle^\circledast$ then $(c \parallel \langle r \rangle^\circledast)(d \parallel \langle r \rangle^\circledast) = (c\ d) \parallel \langle r \rangle^\circledast$ holds for any c, d and r and hence proviso (117) holds. As with parallel, the use of a relational rely condition allows the rely to be distributed into any sequential composition. In the general case, if proviso (117) does not hold the question arises as to what alternative approaches could be used – as with Law 31 (rely-distribute-parallel) these are likely to depend on the form of the interference.

Distribution of the rely quotient over an iteration requires the same side condition (117) on distribution of the interference i over a sequential composition as for Law 32. The law uses the more general form $c^\circ\ d = \mu\, x \cdot d \sqcap c\, x$. This allows the law to be applied to a while loop $\mathbf{while}\ b\ \mathbf{do}\ c$, which can be defined in the form $(bc)^\circ \bar{b}$ where b stands for the test of the while loop succeeding and $\bar{b}$ for it failing. Just developing a law for c° is problematic for the zero iterations case because this corresponds to $\mathbf{nil} /\!\!/ i$ and $\mathbf{nil} /\!\!/ i \sqsubseteq d$ holds if and only if $\mathbf{nil} \sqsubseteq d \parallel i$, which only holds if i behaves as either $\mathbf{nil}$ or $\top$.

Law 33 (rely-distribute-iteration) If

$$\forall\, c_0, c_1 \cdot (c_0 \parallel i)(c_1 \parallel i) \sqsubseteq (c_0\ c_1) \parallel i, \tag{119}$$

holds for i, $(c^\circ\ d) /\!\!/ i \sqsubseteq (c /\!\!/ i)^\circ (d /\!\!/ i)$.

Proof. By Law 23 (rely-refinement) the law is equivalent to $c^\circ\ d \sqsubseteq ((c /\!\!/ i)^\circ (d /\!\!/ i)) \parallel i$ and by Lemma 5 it is sufficient to show,

$$d \sqcap c\,(((c /\!\!/ i)^\circ (d /\!\!/ i)) \parallel i) \sqsubseteq ((c /\!\!/ i)^\circ (d /\!\!/ i)) \parallel i,$$

which can be shown as follows.

$d \sqcap c\,(((c /\!\!/ i)^\circ (d /\!\!/ i)) \parallel i)$
$\sqsubseteq$ by Law 22 (rely-quotient) applied to each of the first d and c
$((d /\!\!/ i) \parallel i) \sqcap ((c /\!\!/ i) \parallel i)(((c /\!\!/ i)^\circ (d /\!\!/ i)) \parallel i)$
$\sqsubseteq$ by assumption (119) with $c_0 = c /\!\!/ i$ and $c_1 = (c /\!\!/ i)^\circ (d /\!\!/ i)$
$((d /\!\!/ i) \parallel i) \sqcap (((c /\!\!/ i)(c /\!\!/ i)^\circ (d /\!\!/ i)) \parallel i)$
$=$ distributing parallel over non-deterministic choice (39)
$((d /\!\!/ i) \sqcap (c /\!\!/ i)(c /\!\!/ i)^\circ (d /\!\!/ i)) \parallel i$
$=$ factoring out $d /\!\!/ i$ using (34)
$((\mathbf{nil} \sqcap (c /\!\!/ i)(c /\!\!/ i)^\circ)(d /\!\!/ i)) \parallel i$
$=$ folding using (16)
$((c /\!\!/ i)^\circ (d /\!\!/ i)) \parallel i$

□

The proviso (119) holds for a relational rely $i = \langle r \rangle^\circledast$ and hence Law 33 (rely-distribute-iteration) holds in this case.

The following laws combine distribution properties with the introduction of a parallel composition.

Law 34 (parallel-introduce-with-rely) $(c \Cap d) /\!\!/ i \sqsubseteq (j_1 \Cap (c /\!\!/ (j_0 \parallel i))) \parallel (j_0 \Cap (d /\!\!/ (j_1 \parallel i)))$

Proof.

$(c \Cap d) /\!\!/ i$
$\sqsubseteq$ by Law 29 (rely-distribute-conjunction)
$(c /\!\!/ i) \Cap (d /\!\!/ i)$
$\sqsubseteq$ by Law 28 (parallel-introduce)
$(j_1 \Cap ((c /\!\!/ i) /\!\!/ j_0)) \parallel (j_0 \Cap ((d /\!\!/ i) /\!\!/ j_1))$
$=$ by Law 27 (rely-nested) twice
$(j_1 \Cap (c /\!\!/ (j_0 \parallel i))) \parallel (j_0 \Cap (d /\!\!/ (j_1 \parallel i)))$

□

In the right side of the above law one branch of the parallel guarantees j_1 and the other guarantees j_0, and hence their parallel combination guarantees $j_1 \parallel j_0$.

Law 35 (parallel-introduce-with-rely-guarantee)

$$(j_1 \parallel j_0) \mathbin{⩓} (c \mathbin{⩓} d) /\!\!/ i \quad \sqsubseteq \quad (j_1 \mathbin{⩓} (c /\!\!/ (j_0 \parallel i))) \parallel (j_0 \mathbin{⩓} (d /\!\!/ (j_1 \parallel i))).$$

Proof.

$$\begin{array}{ll} & (j_1 \parallel j_0) \mathbin{⩓} ((c \mathbin{⩓} d) /\!\!/ i) \\ \sqsubseteq & \text{by Law 34 (parallel-introduce-with-rely)} \\ & (j_1 \parallel j_0) \mathbin{⩓} ((j_1 \mathbin{⩓} (c /\!\!/ (j_0 \parallel i))) \parallel (j_0 \mathbin{⩓} (d /\!\!/ (j_1 \parallel i)))) \\ \sqsubseteq & \text{exchanging weak conjunction and parallel by axiom (50)} \\ & (j_1 \mathbin{⩓} j_1 \mathbin{⩓} (c /\!\!/ (j_0 \parallel i))) \parallel (j_0 \mathbin{⩓} j_0 \mathbin{⩓} (d /\!\!/ (j_1 \parallel i))) \\ = & \text{as weak conjunction is idempotent (44)} \\ & (j_1 \mathbin{⩓} (c /\!\!/ (j_0 \parallel i))) \parallel (j_0 \mathbin{⩓} (d /\!\!/ (j_1 \parallel i))) \end{array}$$

□

In the relational model by (103), $\langle g \cup r \rangle^\circledcirc \sqsubseteq \langle g \rangle^\circledcirc \parallel \langle r \rangle^\circledcirc$ and hence if $j_1 = \langle g \rangle^\circledcirc$ and $j_0 = \langle r \rangle^\circledcirc$ the effective guarantee for Law 35 is $g \cup r$.

7. Relationship to relational rely

This section explores the relationship to the Jones-style rely condition. Jones considered total correctness rules for handling the implementation of a pre-post specification in a context satisfying a rely condition [CJ07]. To instantiate the general theory presented here for Jones-style rely-guarantee rules, termination needs to be handled. For a terminating command, such as a specification $\lceil q \rceil$, using a rely quotient of $\lceil q \rceil /\!\!/ \langle r \rangle^\circledcirc$ leads to an infeasible specification because by Law 22 (rely-quotient) this requires

$$\lceil q \rceil \quad \sqsubseteq \quad (\lceil q \rceil /\!\!/ \langle r \rangle^\circledcirc) \parallel \langle r \rangle^\circledcirc$$

but $\lceil q \rceil$ is terminating and $\langle r \rangle^\circledcirc$ has non-terminating behaviours and hence $\lceil q \rceil /\!\!/ \langle r \rangle^\circledcirc$ must rule out such infinite behaviours of its environment. However, executable code cannot rule out behaviours of its environment and hence using $\langle r \rangle^\circledcirc$ for a rely quotient for a terminating command is not a feasible approach. Therefore the terminating iteration $\langle r \rangle^\circledast$ must be used. Choosing i and j be the processes $\langle r \rangle^\circledast$ and $\langle g \rangle^\circledast$, respectively, in Law 28 (parallel-introduce) gives the following.

$$c \mathbin{⩓} d \quad \sqsubseteq \quad (\langle g \rangle^\circledast \mathbin{⩓} (c /\!\!/ \langle r \rangle^\circledast)) \parallel (\langle r \rangle^\circledast \mathbin{⩓} (d /\!\!/ \langle g \rangle^\circledast)) \tag{120}$$

Note that due to the use of a weak conjunction to enforce a guarantee, the first branch of the parallel composition is only required to maintain its guarantee condition g as long as its environment maintains its rely condition r. If its environment does not maintain r the rely quotient can abort, at which point the whole branch of the parallel is considered to have aborted and hence the guarantee no longer needs to be maintained.

The parallel introduction rule of Jones [Jon83] takes a postcondition of the form $q_0 \cap q_1$ and introduces a parallel composition in which the two branches ensure q_0 and q_1 respectively.

Law 36 (parallel-specification)

$$\{p\}\ \lceil q_0 \cap q_1 \rceil \quad \sqsubseteq\ (\{p\}\,(\langle g \rangle^\circledast \mathbin{⩓} (\lceil q_0 \rceil /\!\!/ \langle r \rangle^\circledast))) \parallel (\{p\}\,(\langle r \rangle^\circledast \mathbin{⩓} (\lceil q_1 \rceil /\!\!/ \langle g \rangle^\circledast)))$$

Proof. Note that by (106) a specification $\lceil q_0 \cap q_1 \rceil$ is equivalent to $\lceil q_0 \rceil \mathbin{⩓} \lceil q_1 \rceil$.

$$\begin{array}{ll} & \{p\}\ \lceil q_0 \cap q_1 \rceil \\ = & \text{by (106)} \\ & \{p\}\,(\lceil q_0 \rceil \mathbin{⩓} \lceil q_1 \rceil) \\ \sqsubseteq & \text{by Law 28 (parallel-introduce)} \\ & \{p\}\,((\langle g \rangle^\circledast \mathbin{⩓} (\lceil q_0 \rceil /\!\!/ \langle r \rangle^\circledast)) \parallel (\langle r \rangle^\circledast \mathbin{⩓} (\lceil q_1 \rceil /\!\!/ \langle g \rangle^\circledast))) \\ = & \text{by Law 16 (precondition-parallel)} \\ & (\{p\}\,(\langle g \rangle^\circledast \mathbin{⩓} (\lceil q_0 \rceil /\!\!/ \langle r \rangle^\circledast))) \parallel (\{p\}\,(\langle r \rangle^\circledast \mathbin{⩓} (\lceil q_1 \rceil /\!\!/ \langle g \rangle^\circledast))) \end{array}$$

□

The above corresponds to the Jones-style proof rule for introducing a parallel composition although phrased in refinement calculus form rather than as a quintuple.

8. Fair parallelism

This section highlights the parts of the theory that are influenced by the choice as to whether or not parallelism is assumed to be fair. The semantics for parallel does not require fairness. A fair semantics would rule out traces ending in an infinite sequence of program steps of one process, if the other process could make a program step. Most algebraic properties are independent of whether or not parallel is assumed to be fair. Fair parallel is denoted by $c \parallel_f d$. It refines the parallel operator used so far, which does not assume fairness.

$$c \parallel d \quad \sqsubseteq \quad c \parallel_f d \tag{121}$$

If no fairness assumption is made about the parallel operator, the notion of termination of a process is weak as it means a process terminates provided it is not permanently interrupted by its environment. For the program

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

the while loop will not terminate unless the $x := 0$ is given a chance to set x to 0. If parallelism is not assumed to be fair, the loop is not guaranteed to terminate even if it is not permanently interrupted; in fact the problem comes if it is never interrupted by $x := 0$. However, if parallel is assumed to be fair, the right process will eventually set x to 0 and the loop will terminate.

Because the definition of the rely quotient operator depends on the parallel operator there is different quotient operator corresponding to fair parallel.

Definition 6 (*fair-quotient*) $c \mathbin{/\!\!/_f} i \ \widehat{=}\ \bigsqcap\{d \mid (c \sqsubseteq d \parallel_f i)\}$

From (121) it follows that $c \mathbin{/\!\!/_f} i \sqsubseteq c \mathbin{/\!\!/} i$, that is, any implementation that handles any interference from process i also handles fair interference from process i.

In the relational model, the property

$$\langle r_0 \cup r_1\rangle^{\circledcirc} \quad = \quad \langle r_0\rangle^{\circledcirc} \parallel \langle r_1\rangle^{\circledcirc} \tag{122}$$

holds, but if parallel is fair (122) becomes a refinement because the left command allows an infinite sequence of steps satisfying r_0 (that do not satisfy $r_0 \cap r_1$), while the right command does not allow such a sequence if parallel is fair. In proving the laws in this paper, we have relied on (122) only being a refinement, i.e. property (103), and hence our laws also apply for fair parallel and fair quotient.

9. Related work

Dingel developed a refinement calculus for rely-guarantee concurrency [Din00, Din02]. Like [HJC14] it is based on relational rely and guarantee conditions but unlike [HJC14] and here, it makes use of a monolithic specification which is a four-tuple of pre, rely, guarantee and post conditions, rather than our separate commands and operators. The approach used here has the benefit of separating the different concepts and providing laws for each operator as well as combinations of operators. The laws given here can be combined to derive laws similar to those of Dingel as well as many other laws. The other major advance over Dingel is the generalisation to use processes for relies and guarantees.

Hoare et al. [HMSW11] have developed a *Concurrent Kleene Algebra (CKA)* and investigated its extension to a *rely/guarantee CKA*. Their algebra includes the axiom $(c\,\top) = \top$, which is not satisfied if c is either a non-terminating process or $\bot$ and hence they only consider partial correctness. Their rely/guarantee CKA includes a sub-algebra of commands called *invariants*, in which an invariant j satisfies

$$j \sqsubseteq \mathbf{nil} \tag{123}$$

$$j \sqsubseteq j \parallel j \tag{124}$$

$$j \sqsubseteq j\,j \tag{125}$$

because in their algebra $c \parallel d \quad \sqsubseteq \quad c\,d$ and hence (124) implies (125). Properties (124) and (125) match the properties used in Law 12 (conjunction-distribute) parts (83) and (84). Properties (123) and (125) together ensure that $j = j^\star$ and hence that $j \Cap d^\star = j^\star \Cap d^\star \sqsubseteq (j \Cap d)^\star$ matching Law 12 (conjunction-distribute) part (85). In a rely/guarantee CKA, for any c and d and any invariant j,

$$(c \parallel j)(d \parallel j) \quad \sqsubseteq \quad (c\,d) \parallel j,$$

which matches our property (117). A rely/guarantee CKA does not require our property $j \parallel j \sqsubseteq j$ but [HMSW11] does not consider an equivalent of Law 31 (rely-distribute-parallel) for which this property is required. In a rely/guarantee CKA a Jones-like rely-guarantee quintuple, written $p\, r\{d\}c\, g$ there, is defined in terms of a Hoare triple plus guarantee condition, in which r and g are invariants (rather than relations).

$$p\, r\{d\}c\, g \;\mathrel{\widehat{=}}\; p\{r \parallel d\}c \;\wedge\; d \;\mathbf{guar}\; g, \tag{126}$$

Our "equivalent" of (126) is of the form

$$g \Cap \{p\}\,(c /\!\!/ r) \;\sqsubseteq\; d, \tag{127}$$

although the two differ due to the different approaches taken. Because g is an invariant the requirement d **guar** g in (126) reduces to $g \sqsubseteq d$, which is stronger than the requirement in (127). Firstly, in (127) d is only required to satisfy the guarantee from initial states satisfying the precondition p. Secondly and more subtly, $c /\!\!/ r$ may abort because its environment does not satisfy r and hence the left side of (127) aborts and so d no longer needs to maintain the guarantee. This latter condition corresponds to Jones' requirement that the implementation only needs to maintain the guarantee condition as long as its environment maintains the rely condition [Jon83]. Our ability to use the weaker requirement comes from the use of the weak conjunction operator, which is not available in CKA.

10. Conclusions

The main contribution of this paper is to explore the essence of the rely-guarantee approach to concurrency. Jones' guarantee condition is generalised from a relation to a process by making use of a weak conjunction operator and his rely condition from a relation to a process by introducing a rely quotient operator, which forms a residual with respect to parallel composition (see Law 23 (rely-refinement)). Both weak conjunction and rely quotient have simple algebraic properties. The weak conjunction operator and parallel composition satisfy an exchange property (50) which leads to a simple and elegant proof of Law 28 (parallel-introduce), which is the key law for introducing a parallel composition in the generalised rely-guarantee theory. Because our theory allows non-terminating processes, it can handle total correctness properties as well as reasoning about non-terminating processes.

Generalising rely-guarantee theory so that guarantees and relies are arbitrary processes rather than binary relations has highlighted the important algebraic properties of rely-guarantee theory. In Law 12 (conjunction-distribute), for a weak conjunction of a command to distribute over a parallel composition one needs proviso (128); to distribute over a sequential composition one needs (129); and to distribute over finite iteration one needs (129) and (130).

$$c \;\sqsubseteq\; c \parallel c \tag{128}$$

$$c \;\sqsubseteq\; c\, c \tag{129}$$

$$c \;\sqsubseteq\; \mathbf{nil} \tag{130}$$

Because all these properties hold if c is of the form $\langle g \rangle^{\circledcirc}$ for any relation g, the choice by Jones to represent interference by an (iterated atomic) relation, rather than a general process, means that Law 21 (guarantee-distribute) for the relational model does not require any provisos.

Even within the relational model more expressive guarantees are possible, for example, a guarantee of $\langle g_0 \rangle^{\circledast}\, \langle g_1 \rangle^{\circledast}$ on c may lead to the following refinement, in which c is refined sequentially to match the guarantees.

$$\begin{array}{ll} & \langle g_0 \rangle^{\circledast}\, \langle g_1 \rangle^{\circledast} \Cap c \\ \sqsubseteq & \quad \text{assuming } c \sqsubseteq c_0\, c_1 \\ & \langle g_0 \rangle^{\circledast}\, \langle g_1 \rangle^{\circledast} \Cap c_0\, c_1 \\ \sqsubseteq & \quad \text{exchanging weak conjunction and sequential composition (51)} \\ & (\langle g_0 \rangle^{\circledast} \Cap c_0)\,(\langle g_1 \rangle^{\circledast} \Cap c_1) \end{array}$$

Law 31 (rely-distribute-parallel) has a proviso of (131), and both Law 32 (rely-distribute-sequential) and Law 33 (rely-distribute-iteration) have a proviso of (132).

$$i \parallel i \quad \sqsubseteq \quad i \tag{131}$$

$$\forall\, c_0,\, c_1 \cdot (c_0 \parallel i)(c_1 \parallel i) \quad \sqsubseteq \quad (c_0\; c_1) \parallel i \tag{132}$$

Because both these properties hold for i of the form $\langle r \rangle^{\circledast}$ for any relation r, the laws do not require any provisos for relational rely conditions thus simplifying the process of distributing relational rely conditions. Note that taking c_0 and c_1 to both be **skip** in (132) gives $i\; i \sqsubseteq i$. An interesting question for future research is what other processes satisfy the provisos required for the distribution properties to hold, or what other distribution properties can be used in their place.

In this paper we have considered an example model based on relational rely-guarantee. The model is similar to that used by others [CJ07, dBHdR99, Din02, dR01, HJC14] but even within the relational model, guarantees and relies are treated more generally as processes. Other possible models for future consideration are an event-based model similar to that used with Concurrent Kleene Algebra [HMSW11] or a model that handles concurrency in a hybrid setting.

Acknowledgements

The research reported here was supported by Australian Research Council Grant DP130102901. This paper has benefited from feedback from Robert Colvin, Cliff Jones, João Ferreira, Larissa Meinicke, Carroll Morgan, Kim Solin, Georg Struth, Kirsten Winter and the anonymous referees but the remaining errors are all courtesy of the author. Special thanks go to Julian Fell and Andrius Velykis for mechanising the proofs of the laws in Isabelle/HOL.

References

[Aar92] Aarts CJ (1992) Galois connections presented calculationally. Technical report, Department of Computing Science, Eindhoven University of Technology. Afstudeer verslag (Graduating Dissertation)

[ABB+95] Aarts C, Backhouse R, Boiten E, Doombos H, van Gasteren N, van Geldrop R, Hoogendijk P, Voermans E, van der Woude J (1995) Fixed-point calculus. Inform Process Lett 53:131–136. (**Mathematics of Program Construction Group**)

[Acz83] Aczel PHG (1983) On an inference rule for parallel composition. Private communication to Cliff Jones. http://homepages.cs.ncl.ac.uk/cliff.jones/publications/MSs/PHGA-traces.pdf

[Bac81] Back R-JR (1981) On correct refinement of programs. J Comput Syst Sci 23(1):49–68

[BCG02] Backhouse R, Crole R, Gibbons J (eds) (2002) Algebraic and coalgebraic methods in the mathematics of program construction. Springer, Berlin

[Bli78] Blikle A (1978) Specified programming. In: Blum EK, Paul M, Takasu S (eds) Mathematical studies of information processing, volume 75 of Lecture Notes in Computer Science. Springer, Berlin, pp 228–251

[BvW98] Back R-JR, von Wright J (1998) Refinement calculus: a systematic introduction. Springer, New York

[BvW99] Back R-JR, von Wright J (1999) Reasoning algebraically about loops. Acta Informatica 36:295–334

[CJ07] Coleman JW, Jones CB (2007) A structural proof of the soundness of rely/guarantee rules. J Logic Comput 17(4):807–841

[Con71] Conway JH (1971) Regular algebra and finite machines. Chapman & Hall, London

[dBHdR99] de Boer FS, Hannemann U, de Roever W-P (1999) Formal justification of the rely-guarantee paradigm for shared-variable concurrency: a semantic approach. In: Wing J, Woodcock J, Davies J (eds) FM99 formal methods, volume 1709 of Lecture Notes in Computer Science. Springer, Berlin, pp 1245–1265

[Din00] Dingel J (2000) Systematic parallel programming. PhD thesis, Carnegie Mellon University. CMU-CS-99-172

[Din02] Dingel J (2002) A refinement calculus for shared-variable parallel and distributed programming. Formal Asp Comput 14(2):123–197

[dR01] de Roever W-P (2001) Concurrency verification: introduction to compositional and noncompositional methods. Cambridge University Press, Cambridge

[HH86] Hoare CAR, He J (1986) The weakest prespecification. Fundamenta Informaticae IX:51–84

[HHH+87] Hoare CAR, Hayes IJ, He J, Morgan C, Roscoe AW, Sanders JW, Sørensen IH, Spivey JM, Sufrin BA (1987) Laws of programming. Commun ACM 30(8):672–686. Corrigenda: CACM 30(9):770

[HJC14] Hayes IJ, Jones CB, Colvin RJ (2014) Laws and semantics for rely-guarantee refinement. Technical Report CS-TR-1425, Newcastle University

[HMSW11] Hoare T, Möller B, Struth G, Wehrman I (2011) Concurrent Kleene algebra and its foundations. J Log Algebr Program 80(6):266–296

[Hoa69] Hoare CAR (1969) An axiomatic basis for computer programming. Commun ACM 12(10):576–580, 583

[JHC15] Jones CB, Hayes IJ, Colvin RJ (2015) Balancing expressiveness in formal approaches to concurrency. Formal Asp Comput 27:475–497

[Jon81] Jones CB (1981) Development methods for computer programs including a notion of interference. PhD thesis, Oxford University. Printed as: Programming Research Group, Technical Monograph 25

[Jon83] Jones CB (1983) Tentative steps toward a development method for interfering programs. ACM Trans Program Lang Syst 5(4):596–619
[Jon96] Jones CB (1996) Accommodating interference in the formal design of concurrent object-based programs. Formal Methods Syst Design 8(2):105–122
[Koz97] Kozen D (1997) Kleene algebra with tests. ACM Trans Program Lang Syst 19(3):427–443
[Mor87] Morris JM (1987) A theoretical basis for stepwise refinement and the programming calculus. Sci Comput Program 9(3):287–306
[Mor88] Morgan CC (1988) The specification statement. ACM Trans Program Lang Syst 10(3):403–419
[Mor94] Morgan CC (1994) Programming from specifications, 2nd edn. Prentice Hall, Upper Saddle River
[vW04] von Wright J (2004) Towards a refinement algebra. Sci Comput Program 51:23–45
[ZH81] Zhou C, Hoare CAR (1981) Partial correctness of communication protocols. Technical Monograph PRG-20, Partial Correctness of Communicating Processes and Protocols. Oxford University Computing, Laboratory, pp 13–23
[Zho82] Zhou C (1982) Weakest environment of communicating processes. In: Proc. of the June 7–10, 1982, National Computer Conf., AFIPS '82, pp 679–690, New York, NY, USA. ACM

Received 22 April 2014
Accepted in revised form 17 June 2016 by Jim Woodcock
Published online 29 July 2016