Reasoning about concurrent programs: Refining rely-guarantee thinking

I.J. Hayes, C.B. Jones and R.J. Colvin

TECHNICAL REPORT SERIES

No. CS-TR-1395 September 2013

TECHNICAL REPORT SERIES

No. CS-TR-1395 September, 2013

Reasoning about concurrent programs: Refining rely-guarantee thinking

I.J. Hayes, C.B. Jones and R.J. Colvin

Abstract

Interference is the essence of concurrency and it is what makes reasoning about concurrent programs difficult. The fundamental insight of rely-guarantee thinking is that stepwise design of concurrent programs can only be compositional in development methods that offer ways to record and reason about interference. In this way of thinking, a rely relation records assumptions about the behaviour of the environment, and a guarantee relation records commitments about the behaviour of the process. The standard application of these ideas is in the extension of Hoare-like inference rules to quintuples that accommodate rely and guarantee conditions. In contrast, in this paper, we embed rely-guarantee thinking into a refinement calculus for concurrent programs, in which programs are developed in (small) steps from an abstract specification. As is usual, we extend the implementation language with specification constructs (the extended language is sometimes called a wide-spectrum language), in this case adding (in addition to pre and postconditions) two new commands: a guarantee command guar(g)(c) whose valid behaviours are in accord with the command c but all of whose atomic steps also satisfy the relation g, and a rely command rely(r)(c) whose behaviours are like c provided any interference steps from the environment satisfy the relation r. The theory of concurrent program refinement is based on a theory of atomic program steps and more powerful refinement laws, most notably, a law for decomposing a specification into a parallel composition, are developed from a small set of more primitive lemmas, which are proved sound with respect to an operational semantics.

Bibliographical details

HAYES, I.J., JONES, C.B., COLVIN, R.J.

Reasoning about concurrent programs: Refining rely-guarantee thinking
[By] I.J. Hayes, C.B. Jones, R.J. Colvin

Newcastle upon Tyne: Newcastle University: Computing Science, 2013.

(Newcastle University, Computing Science, Technical Report Series, No. CS-TR-1395)

Added entries

NEWCASTLE UNIVERSITY
Computing Science. Technical Report Series. CS-TR-1395

Abstract

Interference is the essence of concurrency and it is what makes reasoning about concurrent programs difficult. The fundamental insight of rely-guarantee thinking is that stepwise design of concurrent programs can only be compositional in development methods that offer ways to record and reason about interference. In this way of thinking, a rely relation records assumptions about the behaviour of the environment, and a guarantee relation records commitments about the behaviour of the process. The standard application of these ideas is in the extension of Hoare-like inference rules to quintuples that accommodate rely and guarantee conditions. In contrast, in this paper, we embed rely-guarantee thinking into a refinement calculus for concurrent programs, in which programs are developed in (small) steps from an abstract specification. As is usual, we extend the implementation language with specification constructs (the extended language is sometimes called a wide-spectrum language), in this case adding (in addition to pre and postconditions) two new commands: a guarantee command guar(g)(c) whose valid behaviours are in accord with the command c but all of whose atomic steps also satisfy the relation g, and a rely command rely(r)(c) whose behaviours are like c provided any interference steps from the environment satisfy the relation r. The theory of concurrent program refinement is based on a theory of atomic program steps and more powerful refinement laws, most notably, a law for decomposing a specification into a parallel composition, are developed from a small set of more primitive lemmas, which are proved sound with respect to an operational semantics.

About the authors

Prof. Ian J. Hayes has spent several periods at the School of Computing Science and these have resulted in joint publications (including several with Prof. Cliff Jones). He is now pursuing joint research which will result in further papers. Prof. Hayes also gives seminars at the School. He has given keynote presentations at five conferences in the last ten years including: the International Conference on Theoretical Aspects of Computing 2010 and the International Symposium on Unifying Theories of Programming 2006. He also won the best paper (joint with Dr. R. Colvin) at the 7th International Conference on Integrated Formal Methods in 2009. Prof. Hayes's interests include: Software engineering; formal specification of computing systems; software development based on mathematical principles; real-time systems; fault-tolerant systems; concurrent systems.

Cliff B. Jones is currently Professor of Computing Science at Newcastle University. As well as his academic career, Cliff has spent over 20 years in industry. His 15 years in IBM saw among other things the creation –with colleagues in Vienna– of VDM which is one of the better known "formal methods". Under Tony Hoare, Cliff wrote his doctoral thesis in two years. From Oxford, he moved directly to a chair at Manchester University where he built a world-class Formal Methods group which –among other projects– was the academic lead in the largest Software Engineering project funded by the Alvey programme (IPSE 2.5 created the "mural" (Formal Method) Support Systems theorem proving assistant). He is now applying research on formal methods to wider issues of dependability. He is PI of two EPSRC-funded responsive mode projects "AI4FM" and "Taming Concurrency", CI on the Platform Grant TrAmS-2 and also leads the ICT research in the ITRC Program Grant. Cliff is a Fellow of the Royal Academy of Engineering (FREng), ACM, BCS, and IET. He has been a member of IFIP Working Group 2.3 (Programming Methodology) since 1973.

Dr Robert Colvin completed a PhD in theoretical computer science in 2002, followed by a postdoc position researching analysis techniques for highly parallel algorithms, before taking a postdoc position at the ARC Centre for Complex Systems, using mathematically-based approaches to modelling large software and hardware systems. In addition, he began collaborating with neuroscientists in applying modelling techniques to the behaviour of the brain, looking specifically at fear conditioning in rats. In 2009 he started work for the Queensland Brain Institute, building on his experience in cross-disciplinary and neuroscience research: connecting neuroscience, psychology, and education, towards the new international research initiative, The Science of Learning.

Suggested keywords

REFINEMENT CALCULUS
RELY-GUARANTEE REASONING

Reasoning about concurrent programs: Refining rely-guarantee thinking

Ian J. Hayes[*], Cliff B. Jones[†], Robert J. Colvin[‡]

5 October 2013

Abstract

Interference is the essence of concurrency and it is what makes reasoning about concurrent programs difficult. The fundamental insight of rely-guarantee thinking is that stepwise design of concurrent programs can only be compositional in development methods that offer ways to record and reason about interference. In this way of thinking, a *rely* relation records assumptions about the behaviour of the *environment*, and a *guarantee* relation records commitments about the behaviour of the *process*. The standard application of these ideas is in the extension of Hoare-like inference rules to quintuples that accommodate rely and guarantee conditions. In contrast, in this paper, we embed rely-guarantee thinking into a refinement calculus for concurrent programs, in which programs are developed in (small) steps from an abstract specification. As is usual, we extend the implementation language with specification constructs (the extended language is sometimes called a wide-spectrum language), in this case adding (in addition to pre and postconditions) two new commands: a guarantee command ($\mathbf{guar}\, g \bullet c$) whose valid behaviours are in accord with the command c but all of whose atomic steps also satisfy the relation g, and a rely command ($\mathbf{rely}\, r \bullet c$) whose behaviours are like c provided any interference steps from the environment satisfy the relation r. The theory of concurrent program refinement is based on a theory of atomic program steps and more powerful refinement laws, most notably, a law for decomposing a specification into a parallel composition, are developed from a small set of more primitive lemmas, which are proved sound with respect to an operational semantics.

Keywords Refinement calculus, rely-guarantee reasoning

[*]School of Information Technology and Electrical Engineering, The University of Queensland, Australia
[†]School of Computing Science, Newcastle University, UK
[‡]Queensland Brain Institute, The University of Queensland, Australia

Contents

1 Introduction

The rely-guarantee rules of Jones (1981, 1983) provide a compositional approach to reasoning about concurrent processes (the most accessible reference is (Jones 1996); an exhaustive analysis of various compositional and non-compositional approaches can be found in (de Roever 2001)). Based on many other contributions such as (Stølen 1990; Collette and Jones 2000; Coleman and Jones 2007) these rules have been absorbed into a more general rely-guarantee "thinking" as exemplified in (Jones and Pierce 2011).

The basic rely-guarantee idea is simple: in order to develop a concurrent process c separately from its surrounding components, one needs to take into account interference from the processes which form the (parallel) environment of c. This is done by assuming that any interleaving step of the environment of c satisfies a rely condition r. The rely condition records assumptions the developer is invited to make about possible interference from the environment. Conversely, each process is associated with a guarantee condition, which must be proved to be the limit of interference that it can inflict on the other processes in its environment. Both rely and guarantee are expressed as binary relations over states.

When an (abstract) specification is refined into a parallel composition of specified components, each component specification, s, has an associated rely condition, r, and guarantee condition, g. A specification s is refined assuming the interference steps of the environment are bounded by r, but the refinement must also ensure every atomic step taken by the implementation is bounded by g. From a developer's point of view, the stronger the rely and the weaker the guarantee of s the easier it is to implement; the trade off is that s "works" in fewer contexts, and can cause more interference to other processes, each of which reduces its usefulness.

The aim of this paper is to develop a theory of concurrent program refinement based on a theory of atomic program steps. To support rely-guarantee reasoning, separate specification language constructs are introduced to support guarantee ($\mathbf{guar}\, g \bullet c$) and rely conditions ($\mathbf{rely}\, r \bullet c$). These constructs are defined in terms of the theory of atomic program steps and a set of program refinement laws are proven in terms of the basic algebraic laws of the underlying theory.

To simplify reasoning about types and invariants in the refinement calculus, Morgan and Vickers (1994) introduced an "invariant command": ($\mathbf{inv}\, p \bullet c$). A step of the execution of c is allowed by ($\mathbf{inv}\, p \bullet c$) only if the step maintains the invariant p (a predicate of a single state). If in a particular state the only steps available to c would all break p, then ($\mathbf{inv}\, p \bullet c$) is infeasible (in refinement calculus terms; also known as unsatisfiable in VDM).

The way in which an invariant constrains a program is similar to the way in which a guarantee constrains a program: an invariant constrains each state, while a guarantee constrains each atomic transition between states. This was already noted in (Collette and Jones 2000) but here the similarity is taken further in that a novel command of the form ($\mathbf{guar}\, g \bullet c$) is introduced. The idea behind this new command was motivated by the analogy with the invariant command of Morgan and Vickers. The command ($\mathbf{guar}\, g \bullet c$) behaves as c but it only allows atomic program steps which either stutter or satisfy the relation g between their before-state and after-state. Any step that c alone could take that does not stutter or satisfy g is not a valid step for ($\mathbf{guar}\, g \bullet c$). If in a particular state σ the only steps available to c would neither stutter nor satisfy g, then ($\mathbf{guar}\, g \bullet c$) is not feasible in σ. The stuttering steps allow a process to perform internal steps that do not affect the shared state. The definition of the guarantee command is given in terms of a more primitive strict conjunction operator ($c \Cap d$) in which every atomic step of $c \Cap d$ must be a valid atomic step of both c and d.

The sequential refinement calculus makes use of a specification command of the form $[p,\ q]$, in which p is a predicate giving its precondition and q is a relation (expressed as a two-state predicate) giving its postcondition (Morgan 1988). When started in a state satisfying p any implementation of $[p,\ q]$ must terminate in a state such that the initial and final states are related by q. For any initial state that satisfies p, the command ($\mathbf{guar}\, g \bullet [p,\ q]$) not only satisfies the postcondition q but also only uses atomic steps which each satisfy the relation g or stutter. At this level there is no particular notion of granularity of atomicity; all that is required is that the atomic program steps (whatever they turn out to be) of any implementation of ($\mathbf{guar}\, g \bullet [p,\ q]$) all individually satisfy g or stutter, while the complete sequence of steps satisfies q.

The main advantage in introducing the guarantee command is that it facilitates the separation of the concern of refining a command from that of showing that the refined code adheres to a guarantee. Because the guarantee command is monotonic with respect to refinement of the command in its body, the body can be refined and then a separate set of refinement laws can be used to distribute and eliminate the guarantee.[1] There is one caveat though: a valid refinement of the body may introduce steps that become infeasible when constrained by the guarantee. This means that in doing the refinement one needs to be aware of the enclosing guarantee context in order to ensure that the refinement respects it. The guarantee command also provides a novel simple definition of framing for a command c, i.e. specifying the set of variables c may modify. Section 3 explores guarantee commands in detail.

Standard sequential refinements are only valid if the interference from the environment is restricted to stuttering steps and hence in order to preserve sequential refinements in our concurrent theory, a specification is defined to abort if the environment does a non-stuttering step. In order to specify a construct that meets a pre-post specification in the context of interference from the environment bounded of a relation r, we make use of a novel command of the form $(\textbf{rely}\, r \bullet c)$, which is an "implementation" of c provided interference from the environment respects the rely condition r. The relation r records an assumption about every atomic step of the environment of the command: either the step satisfies r or it stutters. The command $\langle r \vee \mathrm{id}\rangle^*$ represents any finite number, zero or more, of atomic steps that satisfy the relation r or the identity relation id (i.e. stuttering). For a specification $[p,\ q]$, the command $(\textbf{rely}\, r \bullet [p,\ q])$ if run in parallel with interference bounded by r refines ($\sqsubseteq$) the specification $[p,\ q]$, i.e.

$$[p,\ q] \sqsubseteq (\textbf{rely}\, r \bullet [p,\ q]) \parallel \langle r \vee \mathrm{id}\rangle^* \ .$$

The stronger the rely condition, the more constrained the acceptable environments and hence the easier it is to implement $[p,\ q]$. The empty relation is the strongest rely condition: it represents only stuttering interference from the environment. Common rely conditions are those that require certain variables are not modified, or those that restrict the way in which variables may change (e.g. only increase). Section 4 explores rely commands in detail as well as combining rely and guarantee commands.

Earlier justifications of the proof obligation generation for rely/guarantee conditions had to confront the complete set of assumptions and commitments at once. Prensa Nieto (2001, 2003) limits the task both by making rather draconian limits on the nesting of parallelism and by simplifying assumptions about atomicity; she does however succeed in providing a complete Isabelle-checked proof. In contrast, (Jones 1981; Coleman and Jones 2007) only offer proof outlines but do make minimal atomicity assumptions and use a language with arbitrarily nested parallelism. In this paper, it is only necessary to justify that about two dozen basic lemmas respect the semantic model: all of the laws are derived from this basic set. Section 5 gives derivations of the laws for introducing parallel compositions from the properties of the guarantee and rely commands and Section 6 extends these laws to allow trading of conditions between the postcondition of a specification and rely and guarantee conditions. These proofs give further insight into the way in which rely and guarantee combine to enable reasoning about concurrent programs.

As well as introducing parallel compositions, our laws need to handle refining the individual processes in the presence of interference. Refining specifications to sequential compositions, atomic steps and assignments is treated in Section 7. In situations in which the environment does not modify any of the variables used (read or written) by a process, sequential refinement laws can be used. To accommodate this in the theory another command, $(\textbf{uses}\, X \bullet c)$, is introduced; it restricts c to only access variables in the set X. Reasoning about control structures in the presence of interference requires accepting that interference can affect test evaluation; furthermore, showing termination of loops requires reasoning about the effect of the interference on a well-founded relation. These issues are addressed in Section 8. Section 9 applies the laws to a concurrent example. The semantics of the language is given in Appendix A and proofs of the basic lemmas are contained in Appendix B.

[1]In saying this, there is no suggestion that the advantages of "rely-guarantee thinking" in providing a top-down development method should be forgotten.

Our motivation is to provide a theory for reasoning about concurrent programs based on a set of basic laws for primitives that allow more complex laws to be derived as needed. Hence –to illustrate that the theory does allow this– we have given the proofs of the derived laws in terms of the basic laws and other derived laws. To get a flavour for the theory, the reader could skip the proofs on first reading.

2 Programming language and refinement

Expressions Let v be a value, x be a variable, "$\oplus$" stand for a binary operator and "$\ominus$" stand for a unary operator.

$$e \quad ::= \quad v \mid x \mid e_1 \oplus e_2 \mid \ominus e$$

Basic commands Let p be a predicate, q be a relation, C a set of commands, b a boolean expression, X a set of variables, x a variable, and v a value.

$$\begin{aligned} c \quad ::= \quad & \mathbf{nil} \mid \mathbf{abort} \mid \langle p, q\rangle \mid \lceil q \rceil \mid \{p\}c \mid \bigsqcap C \mid c_1 \Cap c_2 \mid c_1 \mathbin{;} c_2 \mid c_1 \parallel c_2 \mid [[b]] \mid \\ & c^* \mid c^\infty \mid c^\omega \mid \mathbf{uses}\, X \bullet c \mid \mathbf{state}\, x \mapsto v \bullet c \mid \mathbf{env}\, r \bullet c \end{aligned}$$

Figure 1: Syntax of expressions and basic commands

2.1 Syntax

Assume a set of variables *Var* and values *Val*. The syntax of expressions and commands is given in Figure 1. The primitive **nil** represents a terminated command and **abort** a command that has gone astray. The command $\langle p, q\rangle$, where p is a single state precondition and q is a relation, may abort if p does not hold but otherwise performs a single atomic step that satisfies relation q between its before and after states. The specification command $\lceil q \rceil$ may take any finite number (zero or more) of atomic steps to achieve the relation q between its before and after states. The preconditioned command $\{p\}c$ behaves as c if p holds initially, otherwise it aborts. The operator "$\bigsqcap$" is used for (demonic) nondeterministic choice from a set of commands, "$\Cap$" is used for strict conjunction of commands (a specification rather than implementation construct — see Section 3), and ";" and "$\parallel$" are used for sequential and parallel composition, respectively. Sequential composition has a higher precedence than the other binary operators. The command $[[b]]$ tests condition b and may either succeed or fail depending on whether b evaluates to true or false, or it may abort if evaluation of the expression is undefined, (e.g. division by zero); tests are used to define the "if" and "while" commands. A command may be iterated a finite number of times (c^* for zero or more and c^+ for one or more), an infinite number of times (c^∞), and either a finite or infinite number of times (c^ω). The "uses" command restricts its body to only use (read and write) variables within the set X. A local state command ($\mathbf{state}\, x \mapsto v \bullet c$) behaves as c but encapsulates x as a local variable with initial value v. The command ($\mathbf{env}\, r \bullet c$) behaves as c if all the environment steps satisfy r but otherwise aborts; it is used to simplify some proofs in the theory.

Figure 2 gives a set of derived commands that are defined in terms of the basic commands. Note that assignment is not a primitive but is defined in terms of a nondeterministic choice over all possible values, v, such that the test $[[e = v]]$ succeeds, followed by an atomic update of x to be v. The nondeterminism is needed because the evaluation of e may take place in an environment in which the variables in e are

[2]The notation $\{v \in V \bullet f\}$ is often written $\{f \mid v \in V\}$ but it is preferable not to use the latter because it is ambiguous as to whether v is bound within the set comprehension or a free variable being tested for membership of V.

Let p be a predicate, q be a relation, c, c_0 and c_1 be commands, b a boolean expression, e an expression, x a variable, and *Val* the set of values. For a set of variables X, the relation $\mathrm{id}(X)$ is the identity relation on X. For a variable x, $\bar{x}$ is the set of all variables other than x.

$$\begin{array}{rcll}
\langle q \rangle & \mathrel{\widehat=} & \langle \mathsf{true}, q \rangle & (1)\\
[p,\ q] & \mathrel{\widehat=} & \{p\}\,[q] & (2)\\
\mathbf{magic} & \mathrel{\widehat=} & \bigsqcap\{\} & (3)\\
c_0 \sqcap c_1 & \mathrel{\widehat=} & \bigsqcap\{c_0, c_1\} & (4)\\
c^{+} & \mathrel{\widehat=} & c^{*} \mathbin{;} c & (5)\\
c^{\omega+} & \mathrel{\widehat=} & c^{\omega} \mathbin{;} c & (6)\\
\mathbf{if}\, b\ \mathbf{then}\ c_0\ \mathbf{else}\ c_1 & \mathrel{\widehat=} & ([[b]] \mathbin{;} c_0) \sqcap ([[\neg\, b]] \mathbin{;} c_1) & (7)\\
\mathbf{while}\, b\, \mathbf{do}\, c & \mathrel{\widehat=} & ([[b]] \mathbin{;} c)^{\omega} \mathbin{;} [[\neg\, b]] & (8)\\
x := e & \mathrel{\widehat=} & \bigsqcap\{v \in \mathit{Val} \bullet [[e = v]] \mathbin{;} \langle x' = v \wedge \mathrm{id}(\bar{x})\rangle\} & (9)\\
\mathbf{var}\, x \bullet c & \mathrel{\widehat=} & \bigsqcap\{v \in \mathit{Val} \bullet (\mathbf{state}\ x \mapsto v \bullet c)\} & (10)
\end{array}$$

The notation $\{v \in V \bullet f\}$ stands for the set of values of the expression f for v in the set V.[2]

Figure 2: Derived commands

being (concurrently) modified, and so there may be many possible final values v which can be assigned to x. A local variable block ($\mathbf{var}\, x \bullet c$) encapsulates x as a local variable with an arbitrarily chosen initial value.

The language does not include procedures but (as usual) the main concern with adding procedures is the possibility of introducing variable aliasing via the parameter passing mechanism. To simplify the presentation, our language does not allow aliasing but we note below any places that would be impacted by aliasing.

The syntax of predicates and relations is not given in detail here. A *relation* is expressed as a predicate over a pair of states: the before state is represented by unprimed variables and the after state by primed variables. The notation $p_0 \Rrightarrow p_1$ means p_0 implies p_1 for all states, and $q_0 \Rrightarrow q_1$ means q_0 implies q_1 for pairs of before and after states. The composition of two relations q_0 and q_1 is defined by

$$q_0 \fatsemi q_1 \mathrel{\widehat=} (\exists\, \mathit{Var}'' \bullet q_0[\mathit{Var}''/\mathit{Var}'] \wedge q_1[\mathit{Var}''/\mathit{Var}]) \qquad (11)$$

in which the final state variables of the first relation (Var') and the initial state variables of the second relation (Var) are identified by replacing them both with fresh intermediate state variables Var''. The reflexive, transitive closure of a relation r is denoted by r^*.

2.2 Program refinement

The refinement calculus (Back 1981; Morris 1987; Morgan 1988; Morgan 1994; Morgan and Vickers 1994; Back and von Wright 1998) provides a systematic approach to program development based on step-wise refinement from a specification to code. A key concept is that the development takes place using a *wide-spectrum language*, which includes implementation constructs as well as specification constructs that are not directly executable. The process of refinement is to transform the specification into code in small steps, some of which may generate proof obligations.

The sequential refinement calculus introduced a specification as a command $[p,\ q]$ in the language, encapsulating pre and post conditions. In this paper as well as the specification command we introduce two new commands ($\mathbf{guar}\, g \bullet c$) and ($\mathbf{rely}\, r \bullet c$) which extend the expressive power of specifications to allow reasoning about interference between concurrently executing commands. The most general form of specification is thus $(\mathbf{guar}\, g \bullet (\mathbf{rely}\, r \bullet [p,\ q]))$. The statement that this specification is refined by a command c is written

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

and is equivalent to the Jones-style five tuple $\{p, r\}\ c\ \{g, q\}$.

In the standard sequential refinement calculus, refinement of a command c by a command d, written $c \sqsubseteq d$, is defined in terms of weakest precondition predicate transformers but because we wish to deal with concurrent execution of processes, this form of semantics is inadequate. Instead we use an operational semantics based on traces formed from atomic steps (detailed in Appendix A). In order to handle specifications with rely conditions, we use a form a trace invented by Aczel (1983), which distinguishes program steps and environment steps (de Boer, Hannemann, and de Roever 1999; de Roever 2001). We use the word "step" to always mean atomic step.

A state is a total mapping from program variable names (*Var*) to values (*Val*). Each atomic step of a program, $\alpha \in \mathcal{L}$, is either a program step, $\pi(\sigma, \sigma')$, or an environment step, $\epsilon(\sigma, \sigma')$, where σ and σ' represent the pre (before) and post (after) states of a step.

$$\begin{array}{rcl@{\qquad}rcl}
\Sigma & \mathrel{\widehat{=}} & \mathit{Var} \rightarrow \mathit{Val} & & & \\
\mathit{pre}(\pi(\sigma, \sigma')) & \mathrel{\widehat{=}} & \sigma & \mathit{post}(\pi(\sigma, \sigma')) & \mathrel{\widehat{=}} & \sigma' \\
\mathit{pre}(\epsilon(\sigma, \sigma')) & \mathrel{\widehat{=}} & \sigma & \mathit{post}(\epsilon(\sigma, \sigma')) & \mathrel{\widehat{=}} & \sigma'
\end{array}$$

A trace t is a sequence of steps, which may be finite or infinite in length (i.e. $t \in \mathcal{L}^\omega$), and which must be consistent, that is, the post state of any step in t always matches the pre state of the next step in t. Indices of sequences start from zero.

$$\mathit{consistent}(t) \mathrel{\widehat{=}} \forall\, i \in \mathrm{dom}(t) - \{0\} \bullet \mathit{post}(t(i-1)) = \mathit{pre}(t(i)) \tag{12}$$

$$\mathit{Trace} \mathrel{\widehat{=}} \{t : \mathcal{L}^\omega \mid \mathit{consistent}(t)\} \tag{13}$$

The domain, $\mathrm{dom}(t)$, of a sequence is the set of valid indices of t and its range, $\mathrm{ran}(t)$, is the set of elements in the sequence. The semantics of a command c is represented by a set of traces, $[\![c]\!]$, which is determined by the operational semantics given in Appendix A. Often we wish to restrict ourselves to traces of c in a particular environment; given a trace t, its environment may be extracted by collecting the pairs of states in each environment step.

$$\mathit{env}(t) \mathrel{\widehat{=}} \{(\sigma, \sigma') \mid \epsilon(\sigma, \sigma') \in \mathrm{ran}(t)\} \tag{14}$$

The traces of c for which the environment steps are restricted to satisfy r or stutter (i.e. satisfy id) are defined as follows.

$$[\![c]\!]_r \mathrel{\widehat{=}} \{t \in [\![c]\!] \mid \mathit{env}(t) \subseteq r \vee \mathrm{id}\} \tag{15}$$

Refinement of a command c by a command d in environment r requires that all traces of d in environment r are traces of c. Refinement is a pre-order.

Definition 1 (refinement-in-context) *For any relation r and commands c and d,*

$$c \sqsubseteq_r d \mathrel{\widehat{=}} [\![d]\!]_r \subseteq [\![c]\!] \tag{16}$$

$$c =_r d \mathrel{\widehat{=}} (c \sqsubseteq_r d) \wedge (d \sqsubseteq_r c) \tag{17}$$

Note that (16) is also equivalent to $[\![d]\!]_r \subseteq [\![c]\!]_r$. Sequential refinement corresponds to refinement in a context in which the environment may only stutter: $c \sqsubseteq_{\mathrm{id}} d$. Refinement in any context corresponds to choosing r to be the universal relation true because this allows any environment steps.[3]

[3]Because our specification constructs allow for stuttering steps in their traces and we are concerned with refinement from a specification to program code, rather than program equivalence, our refinement relation does not need to be complicated by issues of stuttering and mumbling equivalence (Brookes 2007; Dingel 2002).

Definition 2 (refinement) *For and commands c and d,*

$$\begin{array}{lll} c \sqsubseteq d & \mathrel{\widehat{=}} & c \sqsubseteq_{\mathsf{true}} d \\ c = d & \mathrel{\widehat{=}} & (c \sqsubseteq d) \wedge (d \sqsubseteq c) \end{array}$$

Strengthening the environment assumption preserves refinement. In particular, if $c \sqsubseteq d$ then $c \sqsubseteq_r d$, for any r.

Law 3 (refinement-monotonic) *For any commands c and d and relations r_0 and r_1, if $r_0 \Rrightarrow r_1 \vee \mathrm{id}$ and $c \sqsubseteq_{r_1} d$, then $c \sqsubseteq_{r_0} d$.*

Proof. As $r_0 \Rrightarrow r_1 \vee \mathrm{id}$ by (15), $[\![d]\!]_{r_0} \subseteq [\![d]\!]_{r_1}$ and by Definition 1 (refinement-in-context) $[\![d]\!]_{r_1} \subseteq [\![c]\!]$ and hence the result follows by transitivity. □

A trace t has only a finite number of program steps if an index in t may be found after which all steps are environment steps.

$$\mathit{finite\_pgm\_steps}(t) \mathrel{\widehat{=}} (\exists\, i \in \mathrm{dom}(t) \bullet (\forall j \in \mathrm{dom}(t) \bullet i < j \Rightarrow (\exists\, \sigma, \sigma' \bullet t(j) = \epsilon(\sigma, \sigma'))))$$

We say a command c stops from an initial state σ in environment r if all traces of c in environment r that start from σ have only a finite number of program steps. We use the abbreviations $\mathit{pre}(t)$ for $\mathit{pre}(t(0))$ and $\mathit{post}(t)$ for $\mathit{post}(t(\#t - 1))$.

Definition 4 (stops) *For any relation r and command c,*

$$\mathit{stps}(c, r)(\sigma) \mathrel{\widehat{=}} (\forall\, t \in [\![c]\!]_r \bullet (\sigma = \mathit{pre}(t)) \Rightarrow \mathit{finite\_pgm\_steps}(t))$$

Note that we do not make any fairness assumptions in the semantics, and hence the set of traces of c may include traces where c is interrupted forever by the environment. Such traces are "stopped" by the above definition. However the presence of these traces does not change the set of states $\mathit{stps}(c, r)$, as the corresponding uninterrupted traces must also be considered in determining which states are in the set.

2.3 Basic properties of the refinement calculus

We use the term "lemma" to refer to a basic property of a language construct for which the proof is dependent on the operational semantics of that construct, and the term "law" for more general properties of all constructs or properties that are proven from the basic lemmas and other laws. Proofs of most of the basic lemmas may be found in Appendix B.

A preconditioned command $\{p\}c$ aborts if p is false in the initial state (i.e. before any execution steps, even environment steps, have taken place); if p holds in the initial state, the behaviour of $\{p\}c$ is identical to that of c.[4]

Lemma 5 (precondition-traces) *For any predicate p and command c,*

$$\begin{array}{rcl} [\![\{p\}c]\!] & = & \{t \in \mathit{Trace} \mid \mathit{pre}(t) \in p \Rightarrow t \in [\![c]\!]\} \\ \{p\}c \sqsubseteq_r d & \Leftrightarrow & (\forall\, t \in [\![d]\!]_r \bullet \mathit{pre}(t) \in p \Rightarrow t \in [\![c]\!]) \end{array}$$

Hence preconditioned commands satisfy the properties in Law 6 (precondition). As these properties are so basic, we often make use of them without explicit reference to this lemma.

[4]In the sequential refinement calculus, often $\{p\}$ is defined as a separate command and $\{p\}c$ as a sequential composition of $\{p\}$ and c. Here that is not the case: $\{p\}c$ is a preconditioned command. The difference is that there may be (environment) execution steps associated with $\{p\}$ if it is treated as a separate command; that is not the case with the preconditioned command form used here.

Law 6 (precondition) *For any predicates p, p_0 and p_1, relation r and commands c and d,*

$$\begin{array}{rcll}
\{\mathsf{true}\}c & = & c & (18)\\
\{\mathsf{false}\}c & = & \mathbf{abort} & (19)\\
\{p_0\}\{p_1\}c & = & \{p_0 \wedge p_1\}c & (20)\\
(p_0 \Rrightarrow p_1) & \Rightarrow & (\{p_0\}c \sqsubseteq \{p_1\}c) & (21)\\
(\{p\}c \sqsubseteq_r \{p\}d) & \Leftrightarrow & (\{p\}c \sqsubseteq_r d) & (22)\\
stps(\{p\}c, r) & \equiv & p \wedge stps(c, r) & (23)
\end{array}$$

Lemma 7 (parallel-precondition) *For any predicate p, and commands c and d,*

$$\{p\}(c \parallel d) = (\{p\}c) \parallel (\{p\}d)$$

A specification $[p,\ q]$ is defined to achieve q between its before and after states provided p holds initially and the environment only stutters; if the environment does a non-stuttering step the specification aborts. To handle an environment other than stuttering, a specification needs to be enclosed in a rely command ($\mathbf{rely}\ r \bullet [p,\ q]$), where the relation r bounds the interference from the environment. The only interesting case for refinement of a specification without a rely condition is refinement in an environment of id, because a specification command is defined to abort in any other environment.

Lemma 8 (refine-specification) *For any predicate p, relation q, and command c,*

$$([p,\ q] \sqsubseteq c) \quad \Leftrightarrow \quad ([p,\ q] \sqsubseteq_{\mathrm{id}} c)$$

As a result of this property one can use "$\sqsubseteq$" in place of "$\sqsubseteq_{\mathrm{id}}$" whenever the left side of a refinement is a specification.

Lemma 9 (specification-term) *For any predicate p and relations q and r,*

$$\begin{array}{rcll}
stps([p,\ q], r) & = & p, & \textit{if } r \Rrightarrow \mathrm{id}\\
stps([p,\ q], r) & = & \mathsf{false}, & \textit{if } r \not\Rrightarrow \mathrm{id}
\end{array}$$

Lemma 10 (make-atomic) *For any predicate p and relation q,* $\quad [p,\ q] \sqsubseteq \langle p, q\rangle$.

Some basic laws from the sequential refinement calculus are still valid in our extended language.

Lemma 11 (consequence) *For any predicates p_0 and p_1, and relations q_0 and q_1, provided $p_0 \Rrightarrow p_1$ and $p_0 \wedge q_1 \Rrightarrow q_0$,*

$$\begin{array}{rcll}
\langle p_0, q_0\rangle & \sqsubseteq & \langle p_1, q_1\rangle & (24)\\
[p_0,\ q_0] & \sqsubseteq & [p_1,\ q_1] & (25)
\end{array}$$

Lemma 12 (sequential) *For any predicates p_0 and p_1, and relations q, q_0 and q_1, such that $p_0 \wedge ((q_0 \wedge p_1') \fatsemi q_1) \Rrightarrow q$,*

$$[p_0,\ q] \quad \sqsubseteq \quad [p_0,\ q_0 \wedge p_1'] \ ; [p_1,\ q_1]$$

The traces of a nondeterministic choice over a set of commands consists of the union of their traces.

Lemma 13 (nondeterminism-traces) *For a set of commands C,*

$$[\![\textstyle\bigsqcap C]\!] \quad \widehat{=} \quad \bigcup\{c \in C \bullet [\![c]\!]\}$$

Nondeterministic choice has a unit of **magic** and a zero of **abort**. Nondeterministic choice is the greatest lower bound with respect to the refinement ordering and hence satisfies the following laws.

Law 14 (nondeterministic-choice) *For any relation r, command c and set of commands C,*

$$(\forall\, d \in D \bullet (\exists\, c \in C \bullet c \sqsubseteq_r d)) \;\Rightarrow\; (\textstyle\bigsqcap C) \sqsubseteq_r (\textstyle\bigsqcap D) \tag{26}$$

$$c \in C \;\Rightarrow\; (\textstyle\bigsqcap C) \sqsubseteq_r c \tag{27}$$

$$(\forall\, d \in D \bullet c \sqsubseteq_r d) \;\Rightarrow\; c \sqsubseteq_r (\textstyle\bigsqcap D) \tag{28}$$

Proof. The proof of (26) follows from Lemma 13 (nondeterminism-traces) and Definition 1 (refinement-in-context). The other two parts are special cases of (26) given that $c = \bigsqcap \{c\}$. □

In the sequential refinement calculus the specification $\left[\mathit{def}(b),\ b \wedge \mathrm{id}\right]$, where predicate $\mathit{def}(b)$ holds if and only if b is defined (e.g. not the result of a division by zero), can be used to represent test evaluation in the definitions of conditional and loop commands. Such a definition is inadequate in the context of concurrent interference and hence here test evaluation is represented by the separate command $[[b]]$, which is defined operationally in Appendix A. A test $[[b]]$ refines $\left[\mathit{def}(b),\ b \wedge \mathrm{id}\right]$ in an environment consisting of only stuttering. If b evaluates to false the test is infeasible.

Lemma 15 (introduce-test) *For any boolean expression b,* $\left[\mathit{def}(b),\ b \wedge \mathrm{id}\right] \sqsubseteq [[b]]$.

Finite iteration (c^*), infinite iteration (c^∞) and possibly infinite iteration (c^ω) of commands are defined via greatest (ν) and least (μ) fixed points with respect to the refinement ordering (Cohen 2000; von Wright 2004).

Definition 16 (iteration) *For any command c,*

$$c^* \;\widehat{=}\; \nu x \bullet \mathbf{nil} \sqcap c\,;x \tag{29}$$

$$c^\infty \;\widehat{=}\; \mu x \bullet c\,;x \tag{30}$$

$$c^\omega \;\widehat{=}\; \mu x \bullet \mathbf{nil} \sqcap c\,;x \tag{31}$$

The following laws can be derived from Definition 16 (iteration) (von Wright 2004).

Law 17 (fold/unfold-iteration) *For any command c,*

$$c^* = \mathbf{nil} \sqcap c\,;c^* \tag{32}$$

$$c^\infty = c\,;c^\infty \tag{33}$$

$$c^\omega = \mathbf{nil} \sqcap c\,;c^\omega \tag{34}$$

Law 18 (iteration-induction) *For any relation r and commands c, d and x,*

$$x \sqsubseteq_r d \sqcap c\,;x \;\Rightarrow\; x \sqsubseteq_r c^*\,;d \tag{35}$$

$$c\,;x \sqsubseteq_r x \;\Rightarrow\; c^\infty \sqsubseteq_r x \tag{36}$$

$$d \sqcap c\,;x \sqsubseteq_r x \;\Rightarrow\; c^\omega\,;d \sqsubseteq_r x \tag{37}$$

Proof. Standard fixed point theory gives

$$y \sqsubseteq d \sqcap c\,;y \;\Rightarrow\; y \sqsubseteq c^*\,;d\,. \tag{38}$$

To show (35), note that using (38)

$$(x \sqsubseteq_r c^*\,;d) \Leftarrow (\exists\, y \bullet (x \sqsubseteq_r y) \wedge (y \sqsubseteq c^*\,;d)) \Leftarrow (\exists\, y \bullet (x \sqsubseteq_r y) \wedge (y \sqsubseteq d \sqcap c\,;y))$$

As a witness for y choose $(\mathbf{env}\, r \bullet x)$, where $[\![\mathbf{env}\, r \bullet c]\!] = \{t \in \mathit{Trace} \mid \mathit{env}(t) \subseteq r \Rightarrow t \in [\![c]\!]\}$, which gives $x \sqsubseteq_r (\mathbf{env}\, r \bullet x)$ by Definition 1 (refinement-in-context). The refinement $y \sqsubseteq d \sqcap c\,;y$ also holds as follows.

$$([\![d \sqcap c\,;(\mathbf{env}\, r \bullet x)]\!] \subseteq [\![\mathbf{env}\, r \bullet x]\!]) \Leftarrow ([\![d \sqcap c\,;x)]\!]_r \subseteq [\![x]\!]_r) \Leftarrow (x \sqsubseteq_r d \sqcap c\,;x)$$

Hence (35) holds. The proofs of (36) and (37) are similar, except for the witness y for the existential quantifier they use a process with traces equal to $[\![x]\!]_r$. □

Law 19 (iteration-monotonic) *For any relation r and commands c and d, if* $c \sqsubseteq_r d$ *then both* $c^* \sqsubseteq_r d^*$ *and* $c^\omega \sqsubseteq_r d^\omega$ *.*

All our commands are conjunctive (i.e. $c\,;(d_0 \sqcap d_1) = c\,;d_0 \sqcap c\,;d_1$) and hence c^ω can be decomposed into a choice between finite and infinite iteration (von Wright 2004).

Law 20 (isolation) *For any command c,* $\quad c^\omega = c^* \sqcap c^\infty$ *.*

The follow lemma allows reasoning about fixed points (Back and von Wright 1998).

Law 21 (iteration-fusion) *For monotonic functions f and k on complete lattices and a function h, if* $h \circ f = k \circ h$ *then*

$$\begin{array}{ll} h(\mu f) = \mu\, k & \textit{provided } h \textit{ is continuous and} \\ h(\nu f) = \nu k & \textit{provided } h \textit{ is co-continuous.} \end{array}$$

3 The guarantee command

The guarantee command ($\mathbf{guar}\, g \bullet c$) constrains the possible implementations of the command c such that each atomic step must either satisfy the relation g or stutter. Intuitions for its semantics are provided by some examples in Section 3.1 before its definition in terms of a strict conjunction operator and iteration of atomic steps is given in Section 3.2. Sections 3.3 and 3.4 give some properties of atomic steps and strict conjunction. Sections 3.5, 3.6, 3.7 and 3.8 give laws for manipulating guarantees. Section 3.9 introduces a special case when the guarantee preserves an invariant. Section 3.10 applies guarantees and guarantee invariants to a novel development of a simple search algorithm (which is revisited when we consider concurrency in Section 9).

3.1 Examples

The laws referred to in the following examples are given later in Section 3.

1. Refine a specification enclosed in a guarantee by an assignment which respects the guarantee and implements the specification.

$$\begin{array}{ll} & \mathbf{guar}\, x < x' \bullet \left[x' = x + 1\right] \\ \sqsubseteq & \text{by Law 60 (guarantee-assignment) as } x' = x + 1 \Rrightarrow x < x' \\ & x := x + 1 \end{array}$$

2. Use two atomic steps that both satisfy the guarantee.

$$\begin{array}{ll} & \mathbf{guar}\, x < x' \bullet \left[x' = x + 2\right] \\ \sqsubseteq & \text{by Law 41 (guarantee-monotonic) as } \left[x' = x + 2\right] \sqsubseteq \left[x' = x + 1\right] \,;\, \left[x' = x + 1\right] \\ & \mathbf{guar}\, x < x' \bullet (\left[x' = x + 1\right] \,;\, \left[x' = x + 1\right]) \\ = & \text{by Law 47 (distribute-guarantee) over sequential (55)} \\ & (\mathbf{guar}\, x < x' \bullet \left[x' = x + 1\right]) \,;\, (\mathbf{guar}\, x < x' \bullet \left[x' = x + 1\right]) \\ \sqsubseteq & \text{by example (1) above (twice)} \\ & x := x + 1 \,;\, x := x + 1 \end{array}$$

3. A guarantee may restrict a choice. As every atomic step must satisfy the relation $x < x'$ or stutter, the whole must satisfy the reflexive transitive closure of this relation: $(x < x')^*$.

$$\begin{array}{ll} & \mathbf{guar}\, x < x' \bullet \left[x' = x + 1 \vee x' = x - 1\right] \\ = & \text{by Law 52 (trading-post-guarantee) and as } (x < x')^* = (x \leq x') \\ & \mathbf{guar}\, x < x' \bullet \left[(x' = x + 1 \vee x' = x - 1) \wedge x \leq x'\right] \\ = & \mathbf{guar}\, x < x' \bullet \left[x' = x + 1\right] \end{array}$$

4. A specification constrained by a guarantee g cannot be implemented if there is no sequence of atomic steps satisfying g that satisfy the postcondition of the specification overall.

$$
\begin{aligned}
& \quad \mathbf{guar}\, x < x' \bullet \left[x' = x - 1\right] \\
= & \quad \text{by Law 52 (trading-post-guarantee) and as } (x < x')^* = (x \le x') \\
& \quad \mathbf{guar}\, x < x' \bullet \left[x' = x - 1 \wedge (x \le x')\right] \\
= & \ \mathbf{guar}\, x < x' \bullet \left[\mathsf{false}\right] \\
= & \ \left[\mathsf{false}\right]
\end{aligned}
$$

3.2 Defining the guarantee command

The definition of the guarantee command makes use of the strict conjunction operator "$\Cap$", where $c \Cap d$ defines a command that behaves as both c and d in the sense that each atomic step of taken by $c \Cap d$ must be an atomic step that both c and d can make. The conjunction is strict in the sense that if either c or d can abort, then $c \Cap d$ can also abort.

Every atomic step of a guarantee command, ($\mathbf{guar}\, g \bullet c$), must satisfy g or stutter. The command $\langle g \vee \mathrm{id} \rangle$ represents a single atomic step that satisfies g or the identity relation (i.e. a stuttering step), and $\langle g \vee \mathrm{id} \rangle^\omega$ represents the iteration of $\langle g \vee \mathrm{id} \rangle$ any number of times, zero or more, including infinitely many times. The strict conjunction operator "$\Cap$" is used to conjoin $\langle g \vee \mathrm{id} \rangle^\omega$ with c to define the guarantee command.

Definition 22 (guarantee) *For any relation g and command c,*

$$\mathbf{guar}\, g \bullet c \mathrel{\widehat{=}} \langle g \vee \mathrm{id} \rangle^\omega \Cap c \ .$$

3.3 Properties of atomic steps and iterations

Law 23 (strengthen-iterated-atomic) *For any relations g_0 and g_1, if $g_0 \Rrightarrow g_1$ then both $\langle g_1 \rangle^* \sqsubseteq \langle g_0 \rangle^*$ and $\langle g_1 \rangle^\omega \sqsubseteq \langle g_0 \rangle^\omega$.*

Proof. The proof follows by combining Lemma 11 (consequence) for atomic steps with Law 19 (iteration-monotonic). □

Law 24 (refine-iterated-relation) *For any relation g,* $\left[g^*\right] \sqsubseteq \langle g \rangle^*$.

Proof. The proof follows using Law 18 (iteration-induction) for finite iteration (35) provided, $\left[g^*\right] \sqsubseteq \mathbf{nil} \sqcap \langle g \rangle \ ; \left[g^*\right]$, which as holds as follows.

$$
\begin{aligned}
& \left[g^*\right] \\
= & \quad \text{as } g^* = \mathrm{id} \vee (g \fatsemi g^*) \\
& \left[\mathrm{id} \vee (g \fatsemi g^*)\right] \\
\sqsubseteq & \quad \text{by Law 14 (nondeterministic-choice) and Lemma 12 (sequential)} \\
& \left[\mathrm{id}\right] \sqcap \left[g\right] \ ; \left[g^*\right] \\
\sqsubseteq & \quad \text{by Lemma 10 (make-atomic)} \\
& \mathbf{nil} \sqcap \langle g \rangle \ ; \left[g^*\right] \ .
\end{aligned}
$$

□

A terminating command can only perform a finite number of atomic steps. As in the sequential refinement calculus, a command may be guaranteed to terminate only from starting states satisfying a precondition p. Further, in the context of concurrency, it may be guaranteed to stop only if every interference step of the environment satisfies a relation r or stutters.

Law 25 (stops) *For any predicate p, command c and relation r,*

$$(p \Rrightarrow stps(c, r)) \quad \Leftrightarrow (\{p\}\langle \mathsf{true} \rangle^* \sqsubseteq_r c) \ .$$

Proof. The proof follows directly from Definition 4 (stops). First assume $p \Rrightarrow stps(c, r)$. If p holds in state σ, c will stop from σ in environment r, and hence it will perform a finite number of program steps, each of which refines $\langle \mathsf{true} \rangle$. If $\{p\}\langle \mathsf{true} \rangle^* \sqsubseteq_r c$ then from any state satisfying p, c performs only a finite number of program steps in environment r and hence $p \Rrightarrow stps(c, r)$. □

Law 26 (term-monotonic) *For any commands c and d and relations r, r_0 and r_1,*

$$(r_0 \Rrightarrow r_1 \vee \mathrm{id}) \quad \Rightarrow \quad (stps(c, r_1) \Rrightarrow stps(c, r_0)) \tag{39}$$

$$(c \sqsubseteq_r d) \quad \Rightarrow \quad (stps(c, r) \Rrightarrow stps(d, r)) \tag{40}$$

Proof. For (39) by Law 25 (stops),

$$\begin{array}{ll} & \{stps(c, r_1)\}\langle \mathsf{true} \rangle^* \sqsubseteq_{r_1} c \\ \Rightarrow & \text{by Law 3 (refinement-monotonic) as } r_0 \Rrightarrow r_1 \vee \mathrm{id} \\ & \{stps(c, r_1)\}\langle \mathsf{true} \rangle^* \sqsubseteq_{r_0} c \end{array}$$

and hence by Law 25 (stops), $stps(c, r_1) \Rrightarrow stps(c, r_0)$. For (40) from Law 25 (stops)

$$\{stps(c, r)\}\langle \mathsf{true} \rangle^* \sqsubseteq_r c \sqsubseteq_r d$$

and hence by Law 25 (stops), $stps(c, r) \Rrightarrow stps(d, r)$. □

3.4 Properties of strict conjunction

Strict conjunction "$\Cap$" is an associative, commutative and idempotent operator with identity $\langle \mathsf{true} \rangle^\omega$ and zero **abort**. It is monotonic with respect to refinement in each of its arguments.

Lemma 27 (conjunction-monotonic) *For any relation r and commands c_0, c_1, d_0 and d_1,*

$$(c_0 \sqsubseteq_r d_0) \wedge (c_1 \sqsubseteq_r d_1) \quad \Rightarrow \quad (c_0 \Cap c_1) \sqsubseteq_r (d_0 \Cap d_1)$$

Law 28 (refine-conjunction) *For any commands c_0, c_1 and d, if $c_0 \sqsubseteq_r d$ and $c_1 \sqsubseteq_r d$, then $c_0 \Cap c_1 \sqsubseteq_r d$.*

Proof. Any trace of d in environment r must also be a trace of both c_0 and c_1, and hence it is a trace of $c_0 \Cap c_1$, noting that if either c_0 or c_1 can abort their conjunction also can. □

Law 29 (simplify-conjunction) *For commands c and d, if $c \sqsubseteq_r d$, $\quad c \Cap d \sqsubseteq_r d$.*

Proof. The proof follows from Law 28 (refine-conjunction) as $c \sqsubseteq_r d$ and $d \sqsubseteq_r d$. □

Lemma 30 (conjunction-strict) *For any predicate p and commands c and d,*

$$\{p\}(c \Cap d) = c \Cap (\{p\}d) \quad = \quad c \Cap (\{p\}d) = (\{p\}c) \Cap (\{p\}d) \ .$$

Law 31 (conjunction-term) *For any relation r and commands c_0 and c_1,*

$$stps(c_0, r) \wedge stps(c_1, r) \quad \Rrightarrow \quad stps((c_0 \Cap c_1), r) \ .$$

Proof. Let $p \equiv stps(c_0, r) \wedge stps(c_1, r)$, then by Law 25 (stops) both $\{p\}\langle \mathsf{true} \rangle^* \sqsubseteq_r c_0$ and $\{p\}\langle \mathsf{true} \rangle^* \sqsubseteq_r c_1$ and hence by Lemma 27 (conjunction-monotonic)

$$\{p\}\langle \mathsf{true} \rangle^* \Cap \{p\}\langle \mathsf{true} \rangle^* \sqsubseteq_r c_0 \Cap c_1.$$

and hence as strict conjunction is idempotent by Law 25 (stops), $p \Rrightarrow stps((c_0 \Cap c_1), r)$. □

Lemma 32 (conjunction-atomic) *For any predicates p_0 and p_1, relations q_0 and q_1, and commands c_0 and c_1,*

$$\mathbf{nil} \Cap \mathbf{nil} = \mathbf{nil} \tag{41}$$
$$(\langle p_0, q_0\rangle \,;\, c_0) \Cap \mathbf{nil} = \langle p, \mathsf{false}\rangle \tag{42}$$
$$\langle p_0, q_0\rangle \Cap \langle p_1, q_1\rangle = \langle p_0 \wedge p_1, q_0 \wedge q_1\rangle \tag{43}$$
$$(\langle p_0, q_0\rangle \,;\, c_0) \Cap (\langle p_1, q_1\rangle \,;\, c_1) = (\langle p_0, q_0\rangle \Cap \langle p_1, q_1\rangle) \,;\, (c_0 \Cap c_1) \tag{44}$$

A relation g only depends on a set of variables X, if the effect of g in a state is independent of all variables other than X. Recall that $\mathrm{id}(X)$ is the identity relation on X that allows chaos for variables other than X.

Definition 33 (depends-only) *For any relation g and set of variables X,*

$$\mathit{depends\_only}(g, X) \ \widehat{=}\ (\mathrm{id}(X) \fatsemi g \fatsemi \mathrm{id}(X)) \equiv g\ .$$

Note that the above is equivalent to $(\mathrm{id}(X) \fatsemi g \fatsemi \mathrm{id}(X)) \Rrightarrow g$ because the reverse implication holds any any g and X. Furthermore $\mathit{depends\_only}(g, X)$ implies $(\mathrm{id}(X) \fatsemi g) \equiv g \equiv (g \fatsemi \mathrm{id}(X))$.

Strict conjunction distributes through both itself and nondeterministic choice, and conjunction of an iterated atomic step $\langle g\rangle^\omega$ distributes through both sequential and parallel composition, as well as through local variable blocks and iterations.

Lemma 34 (distribute-conjunction) *For any relations g and g_1, commands c, d, d_0 and d_1, nonempty set of commands D, and variable x, such that g_1 does not depend on x, i.e. $\mathit{depends\_only}(g_1, \bar{x})$,*

$$c \Cap (d_0 \Cap d_1) = (c \Cap d_0) \Cap (c \Cap d_1) \tag{45}$$
$$c \Cap (\textstyle\bigsqcap D) = \textstyle\bigsqcap \{d \in D \bullet (c \Cap d)\} \tag{46}$$
$$\langle g\rangle^\omega \Cap (c \,;\, d) = (\langle g\rangle^\omega \Cap c) \,;\, (\langle g\rangle^\omega \Cap d) \tag{47}$$
$$\langle g\rangle^\omega \Cap (c \parallel d) = (\langle g\rangle^\omega \Cap c) \parallel (\langle g\rangle^\omega \Cap d) \tag{48}$$
$$\langle g_1\rangle^\omega \Cap (\mathbf{var}\, x \bullet c) = \mathbf{var}\, x \bullet (\langle g_1\rangle^\omega \Cap c) \tag{49}$$
$$\langle g\rangle^\omega \Cap (c^\omega) = (\langle g\rangle^\omega \Cap c)^\omega \tag{50}$$

Proof. Property (45) holds since strict conjunction is associative, commutative and idempotent.

For (46), a nonaborting trace of the left side is both a trace of c and a trace of $\bigsqcap D$, and hence for some $d \in D$ a trace of d. Hence it is a trace of $c \Cap d$ and hence a trace of the right side. Note that if D is empty and hence $\bigsqcap D = \mathbf{magic}$, the left side becomes $c \Cap \mathbf{magic}$ and the right $\mathbf{magic}$ and hence one only gets a refinement (e.g. when c is $\mathbf{abort}$). The reverse inclusion of traces is similar, as is the argument for aborting traces.

For (47) a nonaborting trace of the left side is a trace t of $(c\,;d)$ in which every program step satisfies g. Either t is an infinite trace of c or $t = t_c \frown t_d$, where t_c is a finite trace of c and t_d is a trace of d. If t is an infinite trace of c, as every program step of t satisfies g, it is an infinite trace of $(\langle g\rangle^\omega \Cap c)$, and hence a trace of the right side. Otherwise, t_c is a finite trace of c for which every program step satisfies g and hence t_c is a trace of $(\langle g\rangle^\omega \Cap c)$; similarly t_d is a trace of $(\langle g\rangle^\omega \Cap d)$. Hence t is a trace of the right side.

For (48) a nonaborting trace of the left side is a trace t of $(c \parallel d)$ for which every program step satisfies g. Hence there must exist traces t_c of c and t_d of d such that t is an "interleaving" of t_c and t_d. As every program step of t satisfies g, so does every program step of t_c and t_d and hence t_c is a trace of $(\langle g\rangle^\omega \Cap c)$ and t_d is a trace of $(\langle g\rangle^\omega \Cap d)$. Hence t is a trace of the right side.

For (49) $(\langle g_1\rangle^\omega \Cap c)$ can do a program step $\pi(\sigma, \sigma')$ if and only if c can do $\pi(\sigma, \sigma')$ and $(\sigma, \sigma') \in g_1$. In which case $(\mathbf{var}\, x \bullet (\langle g_1\rangle^\omega \Cap c))$ can do a step $\pi(\sigma[v/x], \sigma'[v/x])$ for any v. If c can do a step $\pi(\sigma, \sigma')$ then $(\mathbf{var}\, x \bullet c)$ can do a step $\pi(\sigma[v/x], \sigma'[v/x])$ for any v and as g_1 is independent of x, $(\sigma[v/x], \sigma'[v/x]) \in g_1 \Leftrightarrow (\sigma, \sigma') \in g_1$. Hence the traces of the left and right sides are the same.

Law 21 (iteration-fusion) can be applied to prove (50) by choosing $f = (\lambda\, x \bullet \mathbf{nil} \sqcap c \,;\, x)$ and hence $\mu f = c^\omega$, choosing $k = (\lambda\, x \bullet \mathbf{nil} \sqcap (\langle g\rangle^\omega \Cap c) \,;\, x)$, and hence $\mu\, k = (\langle g\rangle^\omega \Cap c)^\omega$, and choosing

$h = (\lambda x \bullet \langle g\rangle^{\omega} \Cap x)$, and hence $h(\mu f) = \langle g\rangle^{\omega} \Cap c^{\omega}$. By Law 21 (iteration-fusion) $\langle g\rangle^{\omega} \Cap c^{\omega} = (\langle g\rangle^{\omega} \Cap c)^{\omega}$ if,

$$\langle g\rangle^{\omega} \Cap (\mathbf{nil} \sqcap c \mathbin{;} x) = \mathbf{nil} \sqcap (\langle g\rangle^{\omega} \Cap c) \mathbin{;} (\langle g\rangle^{\omega} \Cap x)$$

which holds by parts (46) and (47), Law 17 (fold/unfold-iteration) and Lemma 32 (conjunction-atomic). The function h is continuous because strict conjunction is continuous. □

Law 35 (conjunction-with-atomic) *For any relations g and q,*

$$\langle g\rangle^{\omega} \Cap \langle p, q\rangle \quad = \quad \langle p, g \wedge q\rangle \ .$$

Proof.

$\langle g\rangle^{\omega} \Cap \langle p, q\rangle$
$=$ by Law 17 (fold/unfold-iteration)
$(\mathbf{nil} \sqcap \langle g\rangle \mathbin{;} \langle g\rangle^{\omega}) \Cap \langle p, q\rangle$
$=$ by Lemma 34 (distribute-conjunction) over choice (46) and $\langle p, q\rangle = \langle p, q\rangle \mathbin{;} \mathbf{nil}$
$(\mathbf{nil} \Cap \langle p, q\rangle) \sqcap (\langle g\rangle \mathbin{;} \langle g\rangle^{\omega} \Cap \langle p, q\rangle \mathbin{;} \mathbf{nil})$
$=$ by (42), (44) and (43) of Lemma 32 (conjunction-atomic); Law 17 (fold/unfold-iteration)
$\langle p, \mathsf{false}\rangle \sqcap (\langle p, g \wedge q\rangle \mathbin{;} ((\mathbf{nil} \sqcap \langle g\rangle \mathbin{;} \langle g\rangle^{\omega}) \Cap \mathbf{nil}))$
$=$ by Lemma 34 (distribute-conjunction) over choice (46); (41)
$\langle p, \mathsf{false}\rangle \sqcap (\langle p, g \wedge q\rangle \mathbin{;} (\mathbf{nil} \sqcap (\langle g\rangle \mathbin{;} \langle g\rangle^{\omega} \Cap \mathbf{nil}))$
$=$ by Lemma 32 (conjunction-atomic) part (42)
$\langle p, \mathsf{false}\rangle \sqcap (\langle p, g \wedge q\rangle \mathbin{;} (\mathbf{nil} \sqcap \langle \mathsf{false}\rangle)$
$=$ as $\mathbf{nil} \sqcap \langle \mathsf{false}\rangle = \mathbf{nil}$ and $\langle p, \mathsf{false}\rangle \sqsubseteq \langle p, g \wedge q\rangle$
$\langle p, g \wedge q\rangle$

□

Law 36 (terminating-iteration) *For any relation q,*

$$\langle \mathsf{true}\rangle^{*} \Cap \langle q\rangle^{\omega} = \langle q\rangle^{*} \ .$$

Proof. Law 21 (iteration-fusion) can be applied by choosing $f = (\lambda x \bullet \mathbf{nil} \sqcap \langle \mathsf{true}\rangle \mathbin{;} x)$ and hence $\nu f = \langle \mathsf{true}\rangle^{*}$, and choosing $k = (\lambda x \bullet \mathbf{nil} \sqcap \langle q\rangle \mathbin{;} x)$, and hence $\nu k = \langle q\rangle^{*}$, and choosing $h = (\lambda x \bullet x \Cap \langle q\rangle^{\omega})$, and hence $h(\nu f) = \langle \mathsf{true}\rangle^{*} \Cap \langle q\rangle^{\omega}$. By Law 21 (iteration-fusion) $\langle \mathsf{true}\rangle^{*} \Cap \langle q\rangle^{\omega} = \langle q\rangle^{*}$ if,

$$(\mathbf{nil} \sqcap \langle \mathsf{true}\rangle \mathbin{;} x) \Cap \langle q\rangle^{\omega} = \mathbf{nil} \sqcap \langle q\rangle \mathbin{;} (x \Cap \langle q\rangle^{\omega})$$

which we show as follows starting from the left side.

$(\mathbf{nil} \sqcap \langle \mathsf{true}\rangle \mathbin{;} x) \Cap \langle q\rangle^{\omega}$
$=$ by Law 17 (fold/unfold-iteration)
$(\mathbf{nil} \sqcap \langle \mathsf{true}\rangle \mathbin{;} x) \Cap (\mathbf{nil} \sqcap \langle q\rangle \mathbin{;} \langle q\rangle^{\omega})$
$=$ by Lemma 34 (distribute-conjunction) over choice (46)
$(\mathbf{nil} \Cap \mathbf{nil}) \sqcap ((\langle \mathsf{true}\rangle \mathbin{;} x) \Cap \mathbf{nil}) \sqcap (\mathbf{nil} \Cap (\langle q\rangle \mathbin{;} \langle q\rangle^{\omega})) \sqcap ((\langle \mathsf{true}\rangle \mathbin{;} x) \Cap (\langle q\rangle \mathbin{;} \langle q\rangle^{\omega}))$
$=$ by Lemma 32 (conjunction-atomic) parts (41), (42), (44) and (43)
$\mathbf{nil} \sqcap \langle \mathsf{false}\rangle \sqcap \langle \mathsf{false}\rangle \sqcap (\langle q\rangle \mathbin{;} (x \Cap \langle q\rangle^{\omega}))$
$=$ as $\mathbf{nil} \sqsubseteq \langle \mathsf{false}\rangle$
$\mathbf{nil} \sqcap (\langle q\rangle \mathbin{;} (x \Cap \langle q\rangle^{\omega}))$

The function h is co-continuous because strict conjunction with $\langle q\rangle^{\omega}$ is co-continuous. □

Law 37 (conjunction-atomic-iterated) *For any relations g_0 and g_1 both the following hold.*

$$\langle g_0\rangle^{\omega} \Cap \langle g_1\rangle^{\omega} \quad = \quad \langle g_0 \wedge g_1\rangle^{\omega} \tag{51}$$

$$\langle g_0\rangle^{*} \Cap \langle g_1\rangle^{*} \quad = \quad \langle g_0 \wedge g_1\rangle^{*} \tag{52}$$

Proof. By Law 23 (strengthen-iterated-atomic) both the refinements $\langle g_0\rangle^\omega \sqsubseteq \langle g_0 \wedge g_1\rangle^\omega$ and $\langle g_1\rangle^\omega \sqsubseteq \langle g_0 \wedge g_1\rangle^\omega$ hold and hence by Law 28 (refine-conjunction) $\langle g_0\rangle^\omega \Cap \langle g_1\rangle^\omega \sqsubseteq \langle g_0 \wedge g_1\rangle^\omega$. The reverse refinement for (51) holds by Law 18 (iteration-induction) provided

$$\mathbf{nil} \sqcap \langle g_0 \wedge g_1\rangle \mathbin{;} (\langle g_0\rangle^\omega \Cap \langle g_1\rangle^\omega) \sqsubseteq \langle g_0\rangle^\omega \Cap \langle g_1\rangle^\omega$$

which we show by expanding the right side as follows.

$\langle g_0\rangle^\omega \Cap \langle g_1\rangle^\omega$
$=$ by Law 17 (fold/unfold-iteration) unfolding
$(\mathbf{nil} \sqcap \langle g_0\rangle \mathbin{;} \langle g_0\rangle^\omega) \Cap (\mathbf{nil} \sqcap \langle g_1\rangle \mathbin{;} \langle g_1\rangle^\omega)$
$=$ by Lemma 34 (distribute-conjunction) over choice (46)
$(\mathbf{nil} \Cap \mathbf{nil}) \sqcap (\mathbf{nil} \Cap \langle g_1\rangle \mathbin{;} \langle g_1\rangle^\omega) \sqcap (\langle g_0\rangle \mathbin{;} \langle g_0\rangle^\omega \Cap \mathbf{nil}) \sqcap ((\langle g_0\rangle \mathbin{;} \langle g_0\rangle^\omega) \Cap (\langle g_1\rangle \mathbin{;} \langle g_1\rangle^\omega))$
$=$ by Lemma 32 (conjunction-atomic) parts (41), (42), (44) and (43)
$\mathbf{nil} \sqcap \langle \mathsf{false}\rangle \sqcap \langle \mathsf{false}\rangle \sqcap \langle g_0 \wedge g_1\rangle \mathbin{;} (\langle g_0\rangle^\omega \Cap \langle g_1\rangle^\omega)$
$=$ as $\mathbf{nil} \sqsubseteq \langle \mathsf{false}\rangle$
$\mathbf{nil} \sqcap \langle g_0 \wedge g_1\rangle \mathbin{;} (\langle g_0\rangle^\omega \Cap \langle g_1\rangle^\omega)$

To prove (52) we start by using (51).

$\langle g_0\rangle^\omega \Cap \langle g_1\rangle^\omega = \langle g_0 \wedge g_1\rangle^\omega$
$\Rightarrow$ by Lemma 27 (conjunction-monotonic)
$\langle \mathsf{true}\rangle^* \Cap \langle g_0\rangle^\omega \Cap \langle g_1\rangle^\omega = \langle \mathsf{true}\rangle^* \Cap \langle g_0 \wedge g_1\rangle^\omega$
$\Rightarrow$ by Law 36 (terminating-iteration) as conjunction is idempotent and associative
$\langle g_0\rangle^* \Cap \langle g_1\rangle^* = \langle g_0 \wedge g_1\rangle^*$

□

Law 38 (conjunction-with-terminating) *For any predicate p, relations r and g, and command c, such that $p \Rrightarrow stps(c, r)$,*

$$\{p\}\langle g\rangle^* \quad \sqsubseteq_r \quad \langle g\rangle^\omega \Cap (\{p\}c) \ .$$

Proof. Law 25 (stops) gives $\{p\}\langle \mathsf{true}\rangle^* \sqsubseteq_r \{p\}c$, which is used in the last step.

$\{p\}\langle g\rangle^*$
$=$ by Law 36 (terminating-iteration) and Lemma 30 (conjunction-strict)
$\langle g\rangle^\omega \Cap (\{p\}\langle \mathsf{true}\rangle^*)$
$\sqsubseteq_r$ by Lemma 27 (conjunction-monotonic)
$\langle g\rangle^\omega \Cap (\{p\}c)$

□

Law 39 (conjoined-specifications) *For any relations q_0 and q_1,*

$$\left[q_0 \wedge q_1\right] \quad = \quad \left[q_0\right] \Cap \left[q_1\right] \ .$$

Proof. In any environment other than id both sides may abort and hence by Lemma 8 (refine-specification) one only needs to consider equivalence in environment id. A trace of $\left[q_0\right] \Cap \left[q_1\right]$ must be a trace of both $\left[q_0\right]$ and a trace of $\left[q_1\right]$, and hence it satisfies both q_0 and q_1 end-to-end and hence it satisfies $q_0 \wedge q_1$ end-to-end, and thus it is a trace of $\left[q_0 \wedge q_1\right]$. By Lemma 11 (consequence) both $\left[q_0\right] \sqsubseteq \left[q_0 \wedge q_1\right]$ and $\left[q_1\right] \sqsubseteq \left[q_0 \wedge q_1\right]$ and hence by Law 28 (refine-conjunction) $\left[q_0\right] \Cap \left[q_1\right] \sqsubseteq \left[q_0 \wedge q_1\right]$. □

3.5 Laws for refining guarantee commands

Law 40 (guarantee-true) *For any command c,* $(\mathbf{guar}\,\mathsf{true} \bullet c) = c$.

Proof. The proof relies on the fact that $\langle\mathsf{true}\rangle^\omega$ is the identity of "$\Cap$".
$(\mathbf{guar}\,\mathsf{true} \bullet c) = \langle\mathsf{true} \lor \mathrm{id}\rangle^\omega \Cap c = \langle\mathsf{true}\rangle^\omega \Cap c = c$ □

Law 41 (guarantee-monotonic) *For any commands c and d, and relations g and r,*

$$c \sqsubseteq_r d \qquad \Rrightarrow \qquad (\mathbf{guar}\,g \bullet c) \sqsubseteq_r (\mathbf{guar}\,g \bullet d)\ .$$

Proof. The proof relies on Lemma 27 (conjunction-monotonic). If $c \sqsubseteq_r d$,
$(\mathbf{guar}\,g \bullet c) = \langle g \lor \mathrm{id}\rangle^\omega \Cap c \sqsubseteq_r \langle g \lor \mathrm{id}\rangle^\omega \Cap d = (\mathbf{guar}\,g \bullet d)$. □

Law 42 (strengthen-guarantee) *For any command c and relations g_0 and g_1,*

$$(g_0 \Rrightarrow g_1 \lor \mathrm{id}) \qquad \Rrightarrow \qquad (\mathbf{guar}\,g_1 \bullet c) \sqsubseteq (\mathbf{guar}\,g_0 \bullet c)\ .$$

Proof. The proof relies on Lemma 27 (conjunction-monotonic) and Law 23 (strengthen-iterated-atomic). Assume $g_0 \Rrightarrow g_1 \lor \mathrm{id}$,
$(\mathbf{guar}\,g_1 \bullet c) = \langle g_1 \lor \mathrm{id}\rangle^\omega \Cap c \sqsubseteq \langle g_0 \lor \mathrm{id}\rangle^\omega \Cap c = (\mathbf{guar}\,g_0 \bullet c)$. □

Law 43 (guarantee-precondition) *For any predicate p, relation g and command c,*

$$(\mathbf{guar}\,g \bullet \{p\}c) \quad = \quad \{p\}(\mathbf{guar}\,g \bullet c)\ .$$

Proof. From Lemma 30 a strict conjunction aborts if either of its operands aborts,
$(\mathbf{guar}\,g \bullet \{p\}c) = \langle g \lor \mathrm{id}\rangle^\omega \Cap \{p\}c = \{p\}(\langle g \lor \mathrm{id}\rangle^\omega \Cap c) = \{p\}(\mathbf{guar}\,g \bullet c)$. □

Law 44 (introduce-guarantee) *For any command c and relation g,* $c \sqsubseteq (\mathbf{guar}\,g \bullet c)$.

Proof. The proof follows from Law 40 (guarantee-true) and Law 42 (strengthen-guarantee). $c = (\mathbf{guar}\,\mathsf{true} \bullet c) \sqsubseteq (\mathbf{guar}\,g \bullet c)$ □

The traces of $(\mathbf{guar}\,g \bullet c)$ are a subset of the traces of c and hence if c stops from some state σ, $(\mathbf{guar}\,g \bullet c)$ must also stop (or it is infeasible).

Law 45 (guarantee-term) *For any command c, and relations g and r,*

$$stps(c,r) \quad \Rrightarrow \quad stps((\mathbf{guar}\,g \bullet c), r)$$

Proof. By Law 25 (stops) it is sufficient to show $\{stps(c,r)\}\langle\mathsf{true}\rangle^* \sqsubseteq_r (\mathbf{guar}\,g \bullet c)$, which holds using Law 25 (stops) and Law 44 (introduce-guarantee) as follows.
$\{stps(c,r)\}\langle\mathsf{true}\rangle^* \sqsubseteq_r c \sqsubseteq (\mathbf{guar}\,g \bullet c)$. □

Law 46 (nested-guarantees) *For a command c and relations g_0 and g_1,*

$$(\mathbf{guar}\,g_0 \bullet (\mathbf{guar}\,g_1 \bullet c)) \quad = \quad (\mathbf{guar}\,g_0 \land g_1 \bullet c)\ .$$

Proof. The proof relies on the property of relations: $(g_0 \lor \mathrm{id}) \land (g_1 \lor \mathrm{id}) = (g_0 \land g_1) \lor \mathrm{id}$.

$$\begin{aligned}
&(\mathbf{guar}\,g_0 \bullet (\mathbf{guar}\,g_1 \bullet c))\\
=&\quad \text{by Definition 22 (guarantee) twice, "}\Cap\text{" is associative}\\
&\langle g_0 \lor \mathrm{id}\rangle^\omega \Cap \langle g_1 \lor \mathrm{id}\rangle^\omega \Cap c\\
=&\quad \text{by Law 37 (conjunction-atomic-iterated) part (51) as property of relations}\\
&\langle (g_0 \land g_1) \lor \mathrm{id}\rangle^\omega \Cap c\\
=&\ (\mathbf{guar}\,g_0 \land g_1 \bullet c)
\end{aligned}$$

□ A guarantee on a composite command may be distributed to its component commands.

Law 47 (distribute-guarantee) *For any relations g and g_1, commands c and d, set of command C, and variable x such that g_1 does not depend on x, i.e. depend_only$(g, \bar{x})$, the following hold.*

$$\begin{aligned}
\mathbf{guar}\, g \bullet (c \Cap d) &= (\mathbf{guar}\, g \bullet c) \Cap (\mathbf{guar}\, g \bullet d) && (53)\\
\mathbf{guar}\, g \bullet (\textstyle\bigsqcap C) &= \textstyle\bigsqcap\{c \in C \bullet (\mathbf{guar}\, g \bullet c)\} && (54)\\
\mathbf{guar}\, g \bullet (c \,;\, d) &= (\mathbf{guar}\, g \bullet c) \,;\, (\mathbf{guar}\, g \bullet d) && (55)\\
\mathbf{guar}\, g \bullet (c \parallel d) &= (\mathbf{guar}\, g \bullet c) \parallel (\mathbf{guar}\, g \bullet d) && (56)\\
\mathbf{guar}\, g_1 \bullet (\mathbf{var}\, x \bullet c) &= \mathbf{var}\, x \bullet (\mathbf{guar}\, g_1 \bullet c) && (57)\\
\mathbf{guar}\, g \bullet (c^{\omega}) &= (\mathbf{guar}\, g \bullet c)^{\omega} && (58)
\end{aligned}$$

Proof. The proofs of (53)-(58) follow directly from Lemma 34 (distribute-conjunction) properties (45)-(50) respectively. □ A guarantee g on an atomic step that satisfies q must satisfy both q and the guarantee.

Law 48 (guarantee-atomic) *For any predicate p and relations g and q,*

$$(\mathbf{guar}\, g \bullet \langle p, q\rangle) \quad = \quad \langle p, (g \vee \mathrm{id}) \wedge q\rangle\ .$$

Proof. The proof follows using Law 35 (conjunction-with-atomic).
$(\mathbf{guar}\, g \bullet \langle p, q\rangle) = \langle g \vee \mathrm{id}\rangle^{\omega} \Cap \langle p, q\rangle = \langle p, (g \vee \mathrm{id}) \wedge q\rangle$ □

Law 49 (refine-to-guarantee) *For any predicate p, relation g, and command c such that $p \Rrightarrow stps(c, r)$,*

$$\{p\}\langle g \vee \mathrm{id}\rangle^{*} \quad \sqsubseteq_r \quad (\mathbf{guar}\, g \bullet \{p\}c)\ .$$

Proof.

$\{p\}\langle g \vee \mathrm{id}\rangle^{*}$
$\sqsubseteq_r$ by Law 38 (conjunction-with-terminating) as $p \Rrightarrow stps(c, r)$
$\langle g \vee \mathrm{id}\rangle^{\omega} \Cap (\{p\}c)$
$=$ by Definition 22 (guarantee)
$(\mathbf{guar}\, g \bullet \{p\}c)$

□

The following law is a fundamental property of guarantees used in a parallel composition and is used in proving the main parallel introduction laws in Section 5.

Lemma 50 (refine-in-guarantee-context) *For any relations g and r and commands c_0, c_1 and d, such that $c_0 \sqsubseteq_{g \vee r} c_1$,*

$$c_0 \parallel (\mathbf{guar}\, g \bullet d) \quad \sqsubseteq_r \quad c_1 \parallel (\mathbf{guar}\, g \bullet d)\ .$$

Law 51 (refine-in-context) *For any relations r_0 and r_1 and commands c_0 and c_1, if $c_0 \sqsubseteq_{r_0 \vee r_1} c_1$,*

$$c_0 \parallel \langle r_0 \vee \mathrm{id}\rangle^{*} \quad \sqsubseteq_{r_0 \vee r_1} \quad c_1 \parallel \langle r_0 \vee \mathrm{id}\rangle^{*}\ .$$

Proof. Using Lemma 50 (refine-in-guarantee-context) taking d to be $\langle \mathsf{true}\rangle^{*}$, g to be r_0 and r to be $r_0 \vee r_1$ gives the following.

$c_0 \parallel (\mathbf{guar}\, r_0 \bullet \langle \mathsf{true}\rangle^{*}) \ \sqsubseteq_{r_0 \vee r_1} \ c_1 \parallel (\mathbf{guar}\, r_0 \bullet \langle \mathsf{true}\rangle^{*})$
$\equiv$ by Definition 22 (guarantee)
$c_0 \parallel (\langle r_0 \vee \mathrm{id}\rangle^{\omega} \Cap \langle \mathsf{true}\rangle^{*}) \ \sqsubseteq_{r_0 \vee r_1} \ c_1 \parallel (\langle r_0 \vee \mathrm{id}\rangle^{\omega} \Cap \langle \mathsf{true}\rangle^{*})$
$\equiv$ by Law 36 (terminating-iteration) twice
$c_0 \parallel \langle r_0 \vee \mathrm{id}\rangle^{*} \ \sqsubseteq_{r_0 \vee r_1} \ c_1 \parallel \langle r_0 \vee \mathrm{id}\rangle^{*}$

□

The execution of any command enclosed in a guarantee consists of zero or more atomic steps each of which must satisfy g or stutter and hence any such finite execution sequence satisfies the reflexive, transitive closure of g. The following law allows a postcondition ensuring g^* to be traded for a guarantee of g on every atomic step.

Law 52 (trading-post-guarantee) *For any relations g and q,*

$$(\mathbf{guar}\, g \bullet \left[g^* \wedge q\right]) = (\mathbf{guar}\, g \bullet \left[q\right]) \ .$$

Proof. By Lemma 11 (consequence) $\left[q\right] \sqsubseteq \left[g^* \wedge q\right]$ and hence by Law 41 (guarantee-monotonic) the refinement holds from right to left. The refinement from left to right below uses the fact that $\langle g \vee \mathrm{id}\rangle^* \sqsubseteq_{\mathrm{id}} \langle g \vee \mathrm{id}\rangle^\omega \Cap \left[q\right]$ by Law 38 (conjunction-with-terminating).

$\mathbf{guar}\, g \bullet \left[g^* \wedge q\right]$

$=$ by Definition 22 (guarantee), Law 39 (conjoined-specifications)

$\langle g \vee \mathrm{id}\rangle^\omega \Cap \left[g^*\right] \Cap \left[q\right]$

$=$ by Law 24 (refine-iterated-relation) and $(g \vee \mathrm{id})^* = g^*$

$\langle g \vee \mathrm{id}\rangle^\omega \Cap \langle g \vee \mathrm{id}\rangle^* \Cap \left[q\right]$

$\sqsubseteq_{\mathrm{id}}$ by Law 29 (simplify-conjunction) as $\langle g \vee \mathrm{id}\rangle^* \sqsubseteq_{\mathrm{id}} \langle g \vee \mathrm{id}\rangle^\omega \Cap \left[q\right]$

$\langle g \vee \mathrm{id}\rangle^\omega \Cap \left[q\right]$

$=$ Definition 22 (guarantee)

$\mathbf{guar}\, g \bullet \left[q\right]$

□

Definition 53 (feasible-guarantee-specification) *A specification command within a guarantee* $(\mathbf{guar}\, g \bullet \left[p,\ q\right])$ *is feasible if and only if for all states that satisfy the precondition, there exists some final state that satisfies the postcondition and the reflexive transitive closure of the guarantee, i.e.* $p \Rrightarrow (\exists\, Var' \bullet g^* \wedge q)$.

When g is the universal relation true, $(\mathbf{guar}\, \mathsf{true} \bullet \left[p,\ q\right]) = \left[p,\ q\right]$, and the feasibility condition becomes $p \Rrightarrow (\exists\, Var' \bullet q)$, which is the same as that used by Morgan (1988).

3.6 Tests and control structures

In order to define conditionals and loops, a primitive test command, $[[b]]$, is used. It evaluates the boolean expression b and if it evaluates to true the test terminates normally, however if it evaluates to false the test is equivalent to magic and hence eliminates that trace of behaviour. We assume that expressions are evaluated in any order as opposed to a strict left-to-right evaluation. One may enforce the order of evaluation by defining boolean operators, such as "conditional and" (**cand**)[5] and "conditional or" (**cor**), that evaluate their second operand depending on the evaluation of the first. Tests satisfy the following laws.

Lemma 54 (tests) *For any boolean expressions a and b*

$$\begin{array}{lclclcl} [[a \wedge b]] & = & [[a]] \parallel [[b]] & \sqsubseteq & [[a]] \,;\, [[b]] & = & [[a\ \mathbf{cand}\ b]] \\ [[a \vee b]] & = & [[a]] \sqcap [[b]] & \sqsubseteq & [[a]] \sqcap [[\neg a]] \,;\, [[b]] & = & [[a\ \mathbf{cor}\ b]] \end{array}$$

and for variables x and y assuming atomic access to a single variable,

$$[[x < y]] = \bigsqcap \{v \in Val, w \in Val \mid v < w \bullet \langle x = v \wedge \mathrm{id}\rangle \parallel \langle y = w \wedge \mathrm{id}\rangle\}$$

and other relational operators are treated similarly.

[5]In languages deriving their syntax from C, **cand** and **cor** are written "&&" and "||" but we reserve "||" for the parallel operator here.

Because a test does not modify any variables, it ensures any guarantee. Note that because tests only stutter they do not have to be atomic in order to satisfy a guarantee.

Law 55 (test-guarantee) *For any relation g and test predicate b,*

$$(\mathbf{guar}\, g \bullet [[b]]) \quad = \quad [[b]].$$

Proof. Refinement from right to left holds by Law 44 (introduce-guarantee). Because a test does not modify any variables $(\mathbf{guar}\,\mathrm{id} \bullet [[b]]) = [[b]]$, which is used in the following refinement.

$$\begin{array}{l} \quad (\mathbf{guar}\, g \bullet [[b]]) \\ \sqsubseteq \quad \text{by Law 42 (strengthen-guarantee) as } \mathrm{id} \Rrightarrow g \vee \mathrm{id} \\ \quad (\mathbf{guar}\,\mathrm{id} \bullet [[b]]) \\ = [[b]] \end{array}$$

□

For the conditional and loop control structures, because guarantees distribute over program combinators, they distribute into conditionals and loops.

Law 56 (guarantee-conditional) *For any relation g, boolean expression b, and commands c and d,*

$$(\mathbf{guar}\, g \bullet \mathbf{if}\, b\, \mathbf{then}\, c\, \mathbf{else}\, d) \quad = \quad \mathbf{if}\, b\, \mathbf{then}\, (\mathbf{guar}\, g \bullet c)\, \mathbf{else}\, (\mathbf{guar}\, g \bullet d)\ .$$

Proof. The proof is mechanical: it expands the left side conditional using its definition (7), applies Law 47 (distribute-guarantee) to distribute the guarantee over the nondeterministic choice and sequential compositions, then applies Law 55 (test-guarantee) twice to eliminate the guarantee around the tests, and finally rewrites the result as a conditional using definition (7). □

Law 57 (guarantee-loop) *For any relation g, boolean expression b, and command c,*

$$(\mathbf{guar}\, g \bullet \mathbf{while}\, b\, \mathbf{do}\, c) \quad = \quad \mathbf{while}\, b\, \mathbf{do} (\mathbf{guar}\, g \bullet c)\ .$$

Proof. The proof is mechanical: it expands the left side loop using its definition (8), then applies Law 47 (distribute-guarantee) to distribute the guarantee over the sequential composition and iteration, then applies Law 55 (test-guarantee) to eliminate the guarantees around the tests, and finally rewrites the result as a while loop using definition (8). □

3.7 Frames and assignment

The refinement calculus as described by Morgan (1988) includes a version of a specification command $x\colon [q]$ with an explicit frame x representing the set of variables that may be changed by it. Using the notation $\bar{x}$ to stand for all variables other than x, and $\mathrm{id}(\bar{x})$ to stand for the identity relation on all variables other than x, in Morgan's sequential refinement calculus $x\colon [q]$ is defined as $[q \wedge \mathrm{id}(\bar{x})]$, where this latter specification implicitly has all variables in its frame. In the context of concurrent programs this definition is not strong enough as it allows the following refinement,

$$x\colon [x' = y + 1] \sqsubseteq y := y + 1;\ x := y;\ y := y - 1$$

which, although it modifies y, leaves its final value the same as its initial value and hence satisfies the specification on the left. If a concurrent process accesses y during the execution this may lead to unexpected results. However, if the specification is strengthened by making $\mathrm{id}(\bar{x})$ a guarantee, i.e. $(\mathbf{guar}\,\mathrm{id}(\bar{x}) \bullet [x' = y + 1])$, the above refinement is no longer valid. Hence, rather than Morgan's definition above, a frame on a specification satisfies the following.

$$x\colon [q] = (\mathbf{guar}\,\mathrm{id}(\bar{x}) \bullet [q])\ .$$

It turns out to be simple to allow a frame on any command, not just a specification.

Definition 58 (frame) *For any set of variables x and command c,*

$$x : c \mathrel{\widehat{=}} (\mathbf{guar}\, \mathrm{id}(\bar{x}) \bullet c)\ .$$

The importance of framing in reasoning about concurrency is highlighted here by the fact that framing is defined in terms of a guarantee which is central to our approach to concurrency.

Note that one needs to avoid nested frames because by Law 46 (nested-guarantees)

$$x : (y : c) = (\mathbf{guar}\, \mathrm{id}(\bar{x}) \wedge \mathrm{id}(\bar{y}) \bullet c) = (\mathbf{guar}\, \mathrm{id} \bullet c) \neq (\mathbf{guar}\, \mathrm{id}(\overline{x,y}) \bullet c) = (x, y : c)\ .$$

Law 59 (guarantee-frame) *For any set of variables x, relation g and command c,*

$$x : (\mathbf{guar}\, g \bullet c) \quad = \quad \mathbf{guar}\, g \bullet (x : c)\ .$$

Proof. The proof follows from Definition 58 (frame) and Law 46 (nested-guarantees).

$$x : (\mathbf{guar}\, g \bullet c) = (\mathbf{guar}\, \mathrm{id}(\bar{x}) \wedge g \bullet c) = \mathbf{guar}\, g \bullet (x : c)$$

□ An assignment is defined (9) in terms of a test $[[e = v]]$ representing the expression evaluation, followed by an atomic update of x.

$$x := e \quad \widehat{=} \quad \textstyle\bigsqcap\{v \in \mathit{Val} \bullet [[e = v]]\ ;\ \langle x' = v \wedge \mathrm{id}(\bar{x})\rangle\}$$

If the expression e is undefined (e.g. via a division by zero) the test $[[e = v]]$ aborts and hence so does the assignment. No variables other than x may be modified and the only modification allowed to x is to set it to v; this rules out an implementation that sets x to some intermediate value before assigning x its final value v. This interpretation is required in the context of concurrency because if x can be set to an intermediate value, a parallel process may observe the intermediate value and alter its behaviour. The update made by the assignment $x := e$ satisfies the relation $x' = e \wedge \mathrm{id}(\bar{x})$ and hence this relation must imply $g \vee \mathrm{id}$ in order for the assignment to implement the guarantee.

Law 60 (guarantee-assignment) *For any precondition p, relation g, variable x and expression e, such that $p \Rrightarrow \mathit{def}(e)$ and such that $p \wedge x' = e \wedge \mathrm{id}(\bar{x}) \Rrightarrow q \wedge (g \vee \mathrm{id})$,*

$$\mathbf{guar}\, g \bullet \big[p,\ q\big] \quad \sqsubseteq \quad x := e\ .$$

Proof.

$$\begin{array}{ll}
& \mathbf{guar}\, g \bullet \big[p,\ q\big] \\
\sqsubseteq & \text{by Lemma 11 (consequence) as } p \wedge x' = e \wedge \mathrm{id}(\bar{x}) \Rrightarrow q \\
& \mathbf{guar}\, g \bullet \big[p,\ x' = e \wedge \mathrm{id}(\bar{x})\big] \\
\sqsubseteq & \text{by Law 14 (nondeterministic-choice) and Lemma 12 (sequential)} \\
& \mathbf{guar}\, g \bullet \bigsqcap\big\{v \in \mathit{Val} \bullet \big[p,\ p \wedge v = e \wedge \mathrm{id}\big]\ ;\ \big[p \wedge v = e,\ x' = v \wedge \mathrm{id}(\bar{x})\big]\big\} \\
\sqsubseteq & \text{by Law 47 (distribute-guarantee) into choice (54) and sequential (55)} \\
& \bigsqcap\big\{v \in \mathit{Val} \bullet (\mathbf{guar}\, g \bullet \big[p,\ p \wedge v = e \wedge \mathrm{id}\big])\ ;\ (\mathbf{guar}\, g \bullet \big[p \wedge v = e,\ x' = v \wedge \mathrm{id}(\bar{x})\big])\big\} \\
\sqsubseteq & \text{by Lemma 15 (introduce-test) as } p \Rrightarrow \mathit{def}(e)\text{; Lemma 10 (make-atomic)} \\
& \bigsqcap\{v \in \mathit{Val} \bullet (\mathbf{guar}\, g \bullet [[e = v]])\ ;\ (\mathbf{guar}\, g \bullet \langle p \wedge v = e, x' = v \wedge \mathrm{id}(\bar{x})\rangle)\} \\
\sqsubseteq & \text{by Law 55 (test-guarantee); Law 48 (guarantee-atomic) as } p \wedge x' = e \wedge \mathrm{id}(\bar{x}) \Rrightarrow (g \vee \mathrm{id}) \\
& \bigsqcap\{v \in \mathit{Val} \bullet [[e = v]]\ ;\ \langle x' = v \wedge \mathrm{id}(\bar{x})\rangle\} \\
= & \text{by the definition of an assignment (9)} \\
& x := e
\end{array}$$

□ In the standard sequential refinement calculus $\big[\mathit{def}(e),\ x' = e \wedge \mathrm{id}(\bar{x})\big]$ is equivalent to $x := e$ but here it is a strict refinement. The standard refinement calculus gives the following equivalences.

$$\begin{array}{rcl}
x := x + 2 & = & x\colon \big[x' = x + 2\big] \\
& = & x\colon \big[x' = x + 1\big]\ ; x\colon \big[x' = x + 1\big] \\
& = & x := x + 1\ ; x := x + 1
\end{array}$$

However, if x is initially even, a process running in parallel with $x := x + 1\ ; x := x + 1$ may observe an odd value of x, whereas a process running in parallel with $x := x + 2$ will not, and hence they are not equivalent in a concurrent programming context.

3.8 Local variables

A local variable declaration introduces a new local variable for its scope and adds the variable to the frame.

Lemma 61 (introduce-variable) *For any variable x, set of variables y, and command c, such that x is not in y and does not occur free in c,*

$$y : c \quad \sqsubseteq \quad (\mathbf{var}\, x \bullet x, y : c) \ .$$

Commands within the scope of a local variable declaration guarantee not to change any non-local variable with the same name.[6]

Law 62 (guarantee-variable) *For any command c, and local variable x,*

$$(\mathbf{var}\, x \bullet c) \quad = \quad (\mathbf{guar}\, \mathrm{id}(x) \bullet (\mathbf{var}\, x \bullet c)) \ .$$

Proof. Because $(\mathrm{id}(x) \vee \mathrm{id}) = \mathrm{id}(x)$, from Definition 22 (guarantee),

$$(\mathbf{guar}\, \mathrm{id}(x) \bullet (\mathbf{var}\, x \bullet c)) \quad = \quad \langle \mathrm{id}(x) \rangle^{\omega} \Cap (\mathbf{var}\, x \bullet c)$$

but because x is local to $(\mathbf{var}\, x \bullet c)$ every program step of $(\mathbf{var}\, x \bullet c)$ maintains $\mathrm{id}(x)$ on the global variable x, and hence the conjunction with $\langle \mathrm{id}(x) \rangle^{\omega}$ has no effect. □

3.9 Guarantee invariants

A special case of a guarantee relation is that a single-state predicate is an invariant of every atomic step. The following notation is used as an abbreviation in this case.

Definition 63 (guarantee-invariant) *For a single-state predicate p and command c,*

$$(\mathbf{guar\text{-}inv}\, p \bullet c) \quad \widehat{=} \quad (\mathbf{guar}(p \Rightarrow p') \bullet c) \ .$$

where by convention p' stands for the predicate p with all program variables replaced by their primed counterparts, i.e. p in the after state.

An interesting aspect of an invariant predicate is that the relation $(p \Rightarrow p')$ is both reflexive and transitive and hence

$$(p \Rightarrow p')^* \equiv (p \Rightarrow p') \tag{59}$$

that is, for any number of steps (zero or more) if each step maintains the invariant, then the whole does. This fact can be combined with Law 52 (trading-post-guarantee) to give the following law.

Law 64 (trade-guarantee-invariant) *For any predicate p and relation q,*

$$[p,\ p' \wedge q] \quad \sqsubseteq \quad (\mathbf{guar\text{-}inv}\, p \bullet [p,\ q])$$

The effect of this law is to take a property p that is required to be invariant overall and require it to be invariant for every atomic step of the computation – a stronger requirement. Proof.

$[p,\ p' \wedge q]$

$\sqsubseteq$ by Lemma 11 (consequence) using (59); Law 44 (introduce-guarantee)

$\mathbf{guar}(p \Rightarrow p') \bullet [p,\ (p \Rightarrow p')^* \wedge q]$

$=$ by Law 52 (trading-post-guarantee); Definition 63 (guarantee-invariant)

$\mathbf{guar\text{-}inv}\, p \bullet [p,\ q]$

[6]Additional care would be needed with a language that allowed variable aliasing because the non-local variable may be accessible via an alias. Aliasing is excluded throughout this paper.

$\square$

Part of the motivation for inventing a guarantee command came from the invariant command of Morgan and Vickers (1994), which is closely related to a guarantee-invariant command. The invariant command was originally defined within the sequential refinement calculus in terms of weakest precondition semantics. In our concurrent programming context it can be defined as follows:

$$(\mathbf{inv}\, p \bullet c) \quad \widehat{=} \quad \langle p, p' \rangle^{\omega} \Cap c\ .$$

Note that $(\mathbf{inv}\, p \bullet c) \sqsubseteq (\mathbf{guar\text{-}inv}\, p \bullet c)$.

3.10 Extended example (sequential version)

Sect. 9 presents, in a new style, the development of a well-known concurrent algorithm. To cement the material so far, this section offers an unconventional development of a sequential program from the same specification. The objective is, given an array v with indices starting from one, to find the least index t for which a predicate p holds,[7] or if p does not hold for any element of v, to set t to $len(v) + 1$. As shown in Sect. 9, the unconventional approach using guarantee invariants is useful when deriving a concurrent implementation from the same specification. The specification of the *findp* program is

$$findp \mathrel{\widehat{=}} t\colon \big[(t' = len(v) + 1 \lor satp(v, t')) \land notp(v, \mathrm{dom}(v), t')\big] \qquad \lhd$$

where

$$\begin{aligned} satp(v,t) \quad &\widehat{=} \quad t \in \mathrm{dom}(v) \land p(v(t)) \\ notp(v,s,t) \quad &\widehat{=} \quad (\forall\, i \in s \bullet i < t \Rightarrow \neg\, p(v(i))) \end{aligned}$$

No explicit laws have been given above to handle frames on a specification but any construct involving a frame can be converted to an equivalent guarantee command and the corresponding refinement law used. In the development below such steps are elided. The first refinement step introduces an invariant $(t = len(v) + 1 \lor satp(v,t))$ and an initialisation of t that establishes the invariant, using Lemma 12 (sequential) and Law 60 (guarantee-assignment) to handle the frame. We follow the convention that a refinement applies to the most recent command marked with "$\lhd$".

$$\begin{aligned} &\sqsubseteq t := len(v) + 1; \\ &\quad t\colon \big[t = len(v) + 1 \lor satp(v,t),\ (t' = len(v) + 1 \lor satp(v,t')) \land notp(v, \mathrm{dom}(v), t')\big] \qquad \lhd \end{aligned}$$

The second specification can be refined using Law 64 (trade-guarantee-invariant).

$$\begin{aligned} &\sqsubseteq \mathbf{guar\text{-}inv}\, t = len(v) + 1 \lor satp(v,t) \bullet \\ &\qquad\quad t\colon \big[notp(v, \mathrm{dom}(v), t')\big] \qquad \lhd \end{aligned}$$

The body of this involves a quantification within *notp* which can be refined using a loop with fresh control variable c and loop invariant $notp(v, \mathrm{dom}(v), c)$. The invariant is also strengthened by bounds on c, where

$$bnd(c,v) \mathrel{\widehat{=}} 1 \le c \le len(v) + 1$$

The invariant is trivially established if c is set to one. We use Lemma 61 (introduce-variable) for c and initialise it using Lemma 12 (sequential), and Law 60 (guarantee-assignment).

$$\begin{aligned} &\sqsubseteq \mathbf{var}\, c \bullet c := 1; \\ &\quad c, t\colon \big[notp(v, \mathrm{dom}(v), c) \land bnd(c,v),\ notp(v, \mathrm{dom}(v), c') \land bnd(c',v) \land t' = c'\big] \qquad \lhd \end{aligned}$$

[7]For brevity, it is assumed here that $p(x)$ is always defined (undefinedness is considered in (Coleman and Jones 2007) but it has little bearing on the actual design).

This can be refined using Law 64 (trade-guarantee-invariant);

$$\begin{array}{l} \sqsubseteq \textbf{guar-inv}\, notp(v, \mathrm{dom}(v), c) \wedge bnd(c, v) \bullet \\ \qquad c, t \colon \left[t' = c'\right] \end{array} \qquad \lhd$$

This would appear to say that all that is required is that the final values of t and c are equal, however, this is in the context of an accumulated guarantee invariant $(t = len(v) + 1 \vee satp(v, t)) \wedge notp(v, \mathrm{dom}(v), c) \wedge bnd(c, v)$, which must be preserved by every atomic step. The body can be refined to a loop with a well founded relation that reduces $t - c$.

$$\begin{array}{l} \sqsubseteq \textbf{while}\, c < t\, \textbf{do} \\ \qquad c, t \colon \left[c < t,\ 0 \le t' - c' < t - c\right] \end{array} \qquad \lhd$$

Note that this is in the context of the accumulated guarantee invariant, which also acts as an invariant of the loop. The well-founded relation can be achieved either by increasing c or by decreasing t. If c is increased the invariant $notp(v, \mathrm{dom}(v), c)$ must be maintained. To increase c by one this requires $\neg\, p(v(c))$. If t is decreased the invariant $t' = len(v) + 1 \vee satp(v, t')$ must be maintained. To decrease t to c this requires $p(v(c))$. Hence the body of the loop is refined by

$$\sqsubseteq \textbf{if}\, p(v(c))\ \textbf{then}\ t \colon \left[p(v(c)),\ t' = c\right]\ \textbf{else}\ c \colon \left[\neg\, p(v(c)),\ c' = c + 1\right]$$

If the guarantee invariants are distributed into the program this becomes.

```
if $p(v(c))$ then
    guar-inv$(t = len(v) + 1 \vee satp(v,t)) \wedge notp(v, \mathrm{dom}(v), c) \wedge bnd(c, v) \bullet$
        $t \colon [p(v(c)),\ t' = c]$
else
    guar-inv$(t = len(v) + 1 \vee satp(v,t)) \wedge notp(v, \mathrm{dom}(v), c) \wedge bnd(c, v) \bullet$
        $c \colon [\neg\, p(v(c)),\ c' = c + 1]$
```

The branches of the conditional can be refined using Law 60 (guarantee-assignment) to give the following final program.

```
$t := len(v) + 1\ ;\ c := 1$;
while $c < t$ do
    if $p(v(c))$ then $t := c$ else $c := c + 1$
```

4 The rely command

Given a parallel composition $(c \parallel d)$, it is not possible to use the sequential refinement laws to refine one branch, say c, because d may interfere with execution of c by modifying variables shared between c and d. Jones (1981, 1983) addressed this issue by introducing the notion of a *rely* condition, which is a relation r that bounds the tolerable interference acceptable from the environment (either d or the wider environment of both c and d). Every atomic step of the environment is assumed to satisfy the relation r or stutter.

In this paper a new command $(\textbf{rely}\, r \bullet c)$ is introduced; when it is put in an environment in which every atomic interference step satisfies the relation r or stutters, the composite behaviour implements c from states in which c terminates with no interference. Finite interference can be represented by the process $\langle r \vee \mathrm{id}\rangle^*$, that is, the process that can do any finite number, zero or more, of atomic steps, each of which satisfies the relation r or stutters. Hence, for a command c that terminates from states satisfying p (i.e. $p \Rrightarrow stps(c, \mathrm{id})$) the rely command satisfies the following property.

$$\{p\}c \sqsubseteq_{\mathrm{id}} (\textbf{rely}\, r \bullet c) \parallel \langle r \vee \mathrm{id}\rangle^*$$

A rely command should be thought of as giving a designer permission to assume that the environment in which an implementation will run is at least as benign as r records. The motivation for the rely command is similar to that for the weakest pre-specification of Hoare and He (1986), although they deal with sequential composition rather than parallel composition.

4.1 Examples

To understand better the rely command and whether it is feasible, a few examples are examined.

1. $(\textbf{rely}\, x < x' \bullet \big[x + 1 \leq x'\big])$ guarantees that, when it is put in an environment that may increase x, the value of x is increased by at least one. A possible implementation of this rely command is to increment x by one. The environment may further increase x but together they ensure that it is increased by at least one.

2. $(\textbf{rely}\, x < x' \bullet \big[x' = x\big])$ guarantees that, when it is put in an environment that may increase x, the value of x is unchanged. There is no possible implementation of this. Even the "obvious" implementation, **nil**, which does nothing is not a valid implementation because when put in an environment that may increase x, the overall effect may be to increase x.

3. $(\textbf{rely}\, x = x' \bullet \big[x' = x + 1\big])$ guarantees that, when put in an environment in which each interference step does not modify x, x is incremented by exactly one. It may be implemented by the assignment $x := x + 1$. This assignment does not have to be performed atomically, i.e. it may be interleaved with interference that satisfies $x = x'$, which guarantees not to modify x but may arbitrarily modify any variables other than x. Because none of the interference steps modify x, the evaluation of $x + 1$ and its assignment to x are not affected.

4. $(\textbf{rely}\, x = x' \wedge y = y' \bullet \big[x' = x + 1\big])$ guarantees that, when put in an environment in which each interference step does not modify either x or y (although it may modify variables other than x and y), x is incremented by one. It puts no constraints on the final value of y. It may be implemented by the (non-atomic) assignments $(y := x + 1;\ x := y)$, but note that this sequence of assignments does not implement example (3) above because the environment in (3) may arbitrarily modify y between the two assignments.

4.2 Properties of interference

Before delving into the rely command in detail, we look at some basic properties of interference as represented by iteration of atomic steps. Two sets of interference in parallel corresponds to the disjunction of the interferences.

Lemma 65 (parallel-interference) *For any relations r_0 and r_1,*

$$\langle r_0\rangle^* \parallel \langle r_1\rangle^* \quad = \quad \langle r_0 \vee r_1\rangle^* .$$

Interference on an atomic command can only precede or follow it.

Lemma 66 (interference-atomic) *For any predicate p and relations q and r,*

$$\langle p, q\rangle \parallel \langle r\rangle^* \qquad = \qquad \langle r\rangle^* \,;\, \langle p, q\rangle \,;\, \langle r\rangle^* .$$

Law 67 (term-in-context) *For any relations r_0 and r_1, and command c,*

$$stps(c, r_0 \vee r_1) \quad \Rrightarrow \quad stps(c \parallel \langle r_1 \vee \mathrm{id}\rangle^*, r_0 \vee r_1)$$

Proof. By Law 25 (stops) it is sufficient to show

$$\{stps(c, r_0 \vee r_1)\}\langle \mathsf{true}\rangle^* \sqsubseteq_{r_0 \vee r_1} c \parallel \langle r_1 \vee \mathrm{id}\rangle^*$$

which holds as follows.

$$
\begin{array}{ll}
 & \{stps(c, r_0 \vee r_1)\}\langle \mathsf{true}\rangle^* \\
\sqsubseteq & \text{by Lemma 65 (parallel-interference) and Lemma 7 (parallel-precondition)} \\
 & \{stps(c, r_0 \vee r_1)\}\langle \mathsf{true}\rangle^* \parallel \langle r_1 \vee \mathrm{id}\rangle^* \\
\sqsubseteq_{r_0 \vee r_1} & \text{by Law 51 (refine-in-context) as } \{stps(c, r_0 \vee r_1)\}\langle \mathsf{true}\rangle^* \sqsubseteq_{r_0 \vee r_1} c \\
 & c \parallel \langle r_1 \vee \mathrm{id}\rangle^*
\end{array}
$$

□

During design, the inherited interference must be passed on to sub-components. Thus, parallel must satisfy the following distribution properties. Note that except for nondeterministic choice and conjunction one can only distribute interference (rather than an arbitrary command).

Lemma 68 (distribute-parallel) *For any commands c, c_0, c_1, d, d_0, and d_1, set of commands C and relation r,*

$$\begin{aligned}
(\bigsqcap C) \parallel d &= \bigsqcap\{c \in C \bullet (c \parallel d)\} && (60)\\
(c_0 \parallel d_0) \Cap (c_1 \parallel d_1) &\sqsubseteq (c_0 \Cap c_1) \parallel (d_0 \Cap d_1) && (61)\\
(c_0 \mathbin{;} c_1) \parallel \langle r\rangle^* &= (c_0 \parallel \langle r\rangle^*) \mathbin{;} (c_1 \parallel \langle r\rangle^*) && (62)\\
(c_0 \parallel c_1) \parallel \langle r\rangle^* &= (c_0 \parallel \langle r\rangle^*) \parallel (c_1 \parallel \langle r\rangle^*) && (63)
\end{aligned}$$

For (61), distribution is valid only in one direction because, on the left, a step of c_0 may synchronise with either c_1 or d_1, whereas on the right a step of c_0 always synchronises with a step of c_1.

4.3 Fundamental properties of rely

Our theory makes use of a generalisation of the rely command discussed above. The more general command $(\mathbf{rely}\, r \bullet c_z)$ adds an extra parameter relation z which represents the rely context within c. When run in parallel with finite interference that is bounded by the relation r, $(\mathbf{rely}\, r \bullet c_z)$ implements c in an environment bounded by z if started in a state from which c terminates in environment z, i.e.

$$\{stps(c,z)\}c \quad \sqsubseteq_z \quad (\mathbf{rely}\, r \bullet c_z) \parallel \langle r \vee \mathrm{id}\rangle^* \,. \tag{64}$$

The additional parameter z is necessary to determine the states from which the rely command is required to terminate, which corresponds to $stps(c,z)$. The default value for the relation z is id, i.e.

$$(\mathbf{rely}\, r \bullet c) \quad \widehat{=} \quad (\mathbf{rely}\, r \bullet c_{\mathrm{id}}) \tag{65}$$

Condition (64) does not cover the case of $(\mathbf{rely}\, r \bullet c_z)$ failing to terminate in the presence of infinite interference from its environment. For example, the left loop in

$$(\mathbf{while}\, 0 < i\, \mathbf{do}\, i := i-1) \parallel (\mathbf{while}\, i < 10\, \mathbf{do}\, i := i+1)$$

is guaranteed to terminate in the presence of finite interference satisfying $(i' = i+1 \vee \mathrm{id})$ but not in the presence of potentially infinite interference as represented by the right loop. Hence condition (64) is not sufficient to characterise the rely command and we need a further termination condition, which we now investigate.

The specification command $[p,\ q]$ is guaranteed to terminate from states satisfying p in an environment that satisfies id, i.e. only stuttering interference. Hence $(\mathbf{rely}\, r \bullet [p,\ q])$ must be guaranteed to terminate from states satisfying p in an environment that satisfies r. This is captured by the following termination condition.

$$stps((\mathbf{rely}\, r \bullet [p,\ q]), r) \equiv stps([p,\ q], \mathrm{id}) \equiv p$$

If the specification is included in nested relies, this leads to the following.

$$\begin{aligned}
p &\equiv stps([p,\ q], \mathrm{id})\\
&\equiv stps((\mathbf{rely}\, r_1 \bullet [p,\ q]), r_1)\\
&\equiv stps((\mathbf{rely}\, r_0 \bullet (\mathbf{rely}\, r_1 \bullet [p,\ q])_{r_1}), r_0 \vee r_1)
\end{aligned}$$

The rely command $(\mathbf{rely}\, r \bullet c_z)$ is the weakest command such that $(\mathbf{rely}\, r \bullet c_z)$ in parallel with interference $\langle r \vee \mathrm{id}\rangle^*$ refines c in an environment z from initial states from which c terminates in environment z.

Definition 69 (rely) *For any command c and relations r and z,*

$$\textbf{rely}\, r \bullet c_z \;\;\widehat{=}\;\; \textstyle\bigsqcap\{d \mid (\{stps(c,z)\}c \sqsubseteq_z d \parallel \langle r \vee \mathrm{id}\rangle^*) \wedge (stps(c,z) \Rrightarrow stps(d, z \vee r))\}$$

For particular r, z and c, the command $(\textbf{rely}\, r \bullet c_z)$ may not be feasible because all d satisfying the conditions in the nondeterministic choice in Definition 69 (rely) are infeasible, as in example (2) in Section 4.1. In fact, any "code" consisting of control structures and assignments becomes infeasible within any rely context.

The first basic law for the rely command determines when it terminates.

Law 70 (rely-stops) *For any relations z and r and command c,*

$$stps((\textbf{rely}\, r \bullet c_z), z \vee r) \;\;\equiv\;\; stps(c,z)\;.$$

Proof.

$$\begin{array}{ll}
 & stps((\textbf{rely}\, r \bullet c_z), z \vee r) \\
\equiv & \text{by Definition 69 (rely)} \\
 & stps((\bigsqcap\{d \mid (\{stps(c,z)\}c \sqsubseteq_z d \parallel \langle r \vee \mathrm{id}\rangle^*) \wedge (stps(c,z) \Rrightarrow stps(d, z \vee r))\}), z \vee r) \\
\equiv & \text{termination of non-deterministic choice} \\
 & \forall\, d \bullet (\{stps(c,z)\}c \sqsubseteq_z d \parallel \langle r \vee \mathrm{id}\rangle^*) \wedge (stps(c,z) \Rrightarrow stps(d, z \vee r)) \Rightarrow stps(d, z \vee r) \\
\equiv & stps(c,z)
\end{array}$$

For the last step the reverse direction is straightforward. In the forward direction, take d to be "$\{stps(c,z)\}$**magic**" and hence $stps(d, z \vee r) \equiv stps(c,z)$ and d satisfies the conditions on the left of the implication in the quantification and hence $stps(d, z \vee r)$ and hence $stps(c,z)$. □

The following law provides the fundamental property of a rely command. It is used as the basis for the remaining laws involving rely commands. It transforms refinement of a rely command into a refinement to a parallel composition with the interference corresponding to the rely condition. Finite interference, as represented by $\langle r \vee \mathrm{id}\rangle^*$, only handles terminating constructs and hence the rely command in the context of the interference only needs to refine c from initial states for which c terminates in environment z.

Law 71 (rely-refinement) *For any relations z and r, and commands c and d,*

$$((\textbf{rely}\, r \bullet c_z) \sqsubseteq d) \;\;\Leftrightarrow\;\; (\{stps(c,z)\}c \sqsubseteq_z d \parallel \langle r \vee \mathrm{id}\rangle^*) \wedge (stps(c,z) \Rrightarrow stps(d, z \vee r)).$$

Proof. The reverse implication holds as follows.

$$\begin{array}{ll}
 & (\textbf{rely}\, r \bullet c_z) \sqsubseteq d \\
\equiv & \text{by Definition 69 (rely)} \\
 & \bigsqcap\{d_0 \mid (\{stps(c,z)\}c \sqsubseteq_z d_0 \parallel \langle r \vee \mathrm{id}\rangle^*) \wedge (stps(c,z) \Rrightarrow stps(d_0, z \vee r))\} \sqsubseteq d \\
\Leftarrow & \text{by Law 14 (nondeterministic-choice) part (27)} \\
 & \exists\, d_0 \bullet (\{stps(c,z)\}c \sqsubseteq_z d_0 \parallel \langle r \vee \mathrm{id}\rangle^*) \wedge (stps(c,z) \Rrightarrow stps(d_0, z \vee r)) \wedge (d_0 \sqsubseteq d) \\
\Leftarrow & \text{a witness for the existential quantifier is } d \\
 & (\{stps(c,z)\}c \sqsubseteq_z d \parallel \langle r \vee \mathrm{id}\rangle^*) \wedge (stps(c,z) \Rrightarrow stps(d, z \vee r))
\end{array}$$

For the forward implication by Law 3 (refinement-monotonic)

$$((\textbf{rely}\, r \bullet c_z) \sqsubseteq d) \quad\Rightarrow\quad ((\textbf{rely}\, r \bullet c_z) \sqsubseteq_{z \vee r} d) \tag{66}$$

and hence it is sufficient to show the following.

$$((\textbf{rely}\, r \bullet c_z) \sqsubseteq_{z \vee r} d) \quad\Rightarrow\quad (\{stps(c,z)\}c \sqsubseteq_z d \parallel \langle r \vee \mathrm{id}\rangle^*) \wedge (stps(c,z) \Rrightarrow stps(d, z \vee r)) \tag{67}$$

We show each of the conjuncts on the right holds separately. To make the steps more readable we abbreviate $stps(c,z)$ by p.

$$
\begin{array}{ll}
 & \{p\}c \sqsubseteq_z d \parallel \langle r \vee \mathrm{id}\rangle^* \\
\Leftarrow & \text{by Law 51 (refine-in-context) as } (\textbf{rely}\, r \bullet c_z) \sqsubseteq_{z \vee r} d \\
 & \{p\}c \sqsubseteq_z (\textbf{rely}\, r \bullet c_z) \parallel \langle r \vee \mathrm{id}\rangle^* \\
\equiv & \text{by Definition 69 (rely)} \\
 & \{p\}c \sqsubseteq_z \bigsqcap\{d \mid (\{p\}c \sqsubseteq_z d \parallel \langle r \vee \mathrm{id}\rangle^*) \wedge (p \Rrightarrow stps(d, z \vee r))\} \parallel \langle r \vee \mathrm{id}\rangle^* \\
\equiv & \text{by Lemma 68 (distribute-parallel) part (60)} \\
 & \{p\}c \sqsubseteq_z \bigsqcap\{d \mid (\{p\}c \sqsubseteq_z d \parallel \langle r \vee \mathrm{id}\rangle^*) \wedge (p \Rrightarrow stps(d, z \vee r)) \bullet d \parallel \langle r \vee \mathrm{id}\rangle^*\} \\
\Leftarrow & \text{by Law 14 (nondeterministic-choice)} \\
 & \forall d \bullet (\{p\}c \sqsubseteq_z d \parallel \langle r \vee \mathrm{id}\rangle^*) \wedge (p \Rrightarrow stps(d, z \vee r)) \Rightarrow (\{p\}c \sqsubseteq_z d \parallel \langle r \vee \mathrm{id}\rangle^*)
\end{array}
$$

The last condition trivially holds. The termination condition is shown as follows.

$$
\begin{array}{ll}
 & stps(d, z \vee r) \\
\Leftarrow & \text{by Law 26 (term-monotonic) as } (\textbf{rely}\, r \bullet c_z) \sqsubseteq_{z \vee r} d \\
 & stps((\textbf{rely}\, r \bullet c_z), z \vee r) \\
\equiv & \text{by Law 70 (rely-stops)} \\
 & stps(c, z)
\end{array}
$$

□

Law 72 (rely-environment) *For any relations z and r, and commands c and d,*

$$(\textbf{rely}\, r \bullet c_z) \sqsubseteq d \quad \Leftrightarrow \quad (\textbf{rely}\, r \bullet c_z) \sqsubseteq_{z \vee r} d$$

Proof. The forward implication holds by Law 3 (refinement-monotonic). For the reverse implication by Law 71 (rely-refinement) it is sufficient to show

$$((\textbf{rely}\, r \bullet c_z) \sqsubseteq_{z \vee r} d) \quad \Rightarrow \quad (\{stps(c,z)\}c \sqsubseteq_z d \parallel \langle r \vee \mathrm{id}\rangle^*) \wedge (stps(c,z) \Rrightarrow stps(d, z \vee r))$$

which was shown as property (67) above. □ Law 72 (rely-environment) allows one to use the stronger "$\sqsubseteq$" instead of "$\sqsubseteq_{z \vee r}$" when dealing with rely commands.

Law 73 (rely-refinement-precondition) *For any predicate p, relations z and r, and commands c and d, such that $p \Rrightarrow stps(c,z) \wedge stps(d, z \vee r)$,*

$$(\textbf{rely}\, r \bullet \{p\}c_z) \sqsubseteq d \quad \Leftrightarrow \quad \{p\}c \sqsubseteq_z d \parallel \langle r \vee \mathrm{id}\rangle^* .$$

Proof. Because $stps(\{p\}c, z) \equiv p \wedge stps(c,z) \equiv p \Rrightarrow stps(d, z \vee r)$, the law follows by Law 71 (rely-refinement) . □

Law 74 (rely-specification) *For any predicate p, relations q and r, and command d, such that $p \Rrightarrow stps(d, r)$,*

$$(\textbf{rely}\, r \bullet [p,\ q]) \sqsubseteq d \quad \Leftrightarrow \quad [p,\ q] \sqsubseteq d \parallel \langle r \vee \mathrm{id}\rangle^*$$

Proof. The law follows from Law 73 (rely-refinement-precondition) and Lemma 8 (refine-specification) because $stps([p,\ q]\,, \mathrm{id}) \equiv p$. □

Law 75 (rely) *For any predicate p, relations z and r, and command c, such that $p \Rrightarrow stps(c,z)$,*

$$\{p\}c \quad \sqsubseteq_z \quad (\textbf{rely}\, r \bullet \{p\}c_z) \parallel \langle r \vee \mathrm{id}\rangle^* .$$

Proof. Because by Law 70 (rely-stops), $stps((\mathbf{rely}\, r \bullet \{p\}c_z), z \vee r) \equiv stps(\{p\}c, z) \equiv p$, the law follows from Law 73 (rely-refinement-precondition) by taking d to be $(\mathbf{rely}\, r \bullet \{p\}c_z)$. □

There may be any finite number, zero or more, of interference steps (each of which satisfies r or stutters) between any two program steps. Hence the interference between any two program steps satisfies r^*. For this reason most formulations of the rely-guarantee approach (including (Coleman and Jones 2007)) require r to be reflexive and transitive, so that $r^* = r$. Here we do not require r to be either reflexive or transitive but use its reflexive transitive closure where necessary.

Note that care needs to be taken with the order of guarantee and rely clauses. The form usually required is a rely nested within a guarantee. The problem with using a guarantee command nested within a rely is that to show $(\mathbf{rely}\, r \bullet (\mathbf{guar}\, g \bullet [p,\ q])) \sqsubseteq d$, Law 71 (rely-refinement) requires one to show $(\mathbf{guar}\, g \bullet [p,\ q]) \sqsubseteq d \parallel \langle r \vee \mathrm{id}\rangle^*$, which requires the interference r to guarantee g as well as d guaranteeing g. However, for a rely nested within a guarantee, only d is required to satisfy the guarantee. A guarantee within a rely should therefore be avoided, although there are situations in which it is allowed: a trivial case is if the rely is id but the more general case is if the rely condition happens to imply the guarantee; this sometimes happens with a set of operations handling a shared data structure (Collette and Jones 2000).

Law 76 (guarantee-plus-rely) *For any relations g, z and r, and commands c and d,*

$$(\mathbf{guar}\, g \bullet (\mathbf{rely}\, r \bullet c_z)) \quad \sqsubseteq \quad d, \tag{68}$$

if both the following hold

$$(\mathbf{rely}\, r \bullet c_z) \quad \sqsubseteq \quad d \tag{69}$$

$$(\mathbf{guar}\, g \bullet d) \quad \sqsubseteq \quad d\,. \tag{70}$$

Proof. Applying Law 41 (guarantee-monotonic) to (69) gives the following.

$$(\mathbf{guar}\, g \bullet (\mathbf{rely}\, r \bullet c_z)) \sqsubseteq (\mathbf{guar}\, g \bullet d)$$

When combined with (70) this implies (68). □

4.4 Laws for refining rely commands

Law 77 (weaken-rely) *For any command c, and any relations z, r_0 and r_1, such that $r_0 \Rrightarrow r_1 \vee \mathrm{id}$,*

$$(\mathbf{rely}\, r_0 \bullet c_z) \quad \sqsubseteq \quad (\mathbf{rely}\, r_1 \bullet c_z)\,.$$

Proof. By Law 71 (rely-refinement) the theorem holds provided

$$\begin{array}{l} (\{stps(c,z)\}c \sqsubseteq_z (\mathbf{rely}\, r_1 \bullet c_z) \parallel \langle r_0 \vee \mathrm{id}\rangle^*) \wedge (stps(c,z) \Rrightarrow stps((\mathbf{rely}\, r_1 \bullet c_z), z \vee r_0)) \\ \Leftarrow \quad \text{Law 23 (strengthen-iterated-atomic) and Law 26 (term-monotonic) as } r_0 \Rrightarrow r_1 \vee \mathrm{id} \\ (\{stps(c,z)\}c \sqsubseteq_z (\mathbf{rely}\, r_1 \bullet c_z) \parallel \langle r_1 \vee \mathrm{id}\rangle^*) \wedge (stps(c,z) \Rrightarrow stps((\mathbf{rely}\, r_1 \bullet c_z), z \vee r_1) \end{array}$$

which follow by Law 75 (rely) and Law 70 (rely-stops). □

Note that for relations q and r we have the following relationships.

$$[q] = (\mathbf{rely}\, \mathrm{id} \bullet [q]) \sqsubseteq (\mathbf{rely}\, r \bullet [q]) \sqsubseteq (\mathbf{rely}\, \mathsf{true} \bullet [q])$$

In particular, $[\mathsf{false}]$ is infeasible in an environment of just stuttering (id) but aborts in any non-id environment, whereas $(\mathbf{rely}\, \mathsf{true} \bullet [\mathsf{false}])$ is infeasible in any environment.

Law 78 (rely-monotonic) *For any relations z and r, and commands c and d, such that $\{stps(c,z)\}c \sqsubseteq_z d$,*

$$(\mathbf{rely}\, r \bullet c_z) \quad \sqsubseteq \quad (\mathbf{rely}\, r \bullet d_z)\,.$$

Proof. The theorem holds by Law 71 (rely-refinement) provided the following holds.

$$(\{stps(c,z)\}c \sqsubseteq_z (\mathbf{rely}\, r \bullet d_z) \parallel \langle r \vee \mathrm{id}\rangle^*) \wedge (stps(c,z) \Rrightarrow stps((\mathbf{rely}\, r \bullet d_z), z \vee r))$$

$\Leftarrow$ by Law 75 (rely), $\{stps(d,z)\}d \sqsubseteq_z (\mathbf{rely}\, r \bullet d_z) \parallel \langle r \vee \mathrm{id}\rangle^*$ and Law 70 (rely-stops)

$$(\{stps(c,z)\}c \sqsubseteq_z \{stps(d,z)\}d) \wedge (stps(c,z) \Rrightarrow stps(d,z))$$

By assumption $\{stps(c,z)\}c \sqsubseteq_z d$ and hence by Law 26 (term-monotonic) $stps(c,z) \Rrightarrow stps(d,z)$. □
Note that refining the body of a rely command does not necessarily preserve feasibility. For example, $[x = 0,\ x < x'] \sqsubseteq [x' = 1]$ is a valid refinement but while $(\mathbf{rely}\, x \le x' \bullet [x = 0,\ x < x'])$ is feasible, $(\mathbf{rely}\, x \le x' \bullet [x' = 1])$ is not feasible because the interference may increase x beyond one.

Law 79 (rely-precondition) *For any predicate p, relations z and r, and command c, such that $p \Rrightarrow stps(c,z)$,*

$$(\mathbf{rely}\, r \bullet \{p\}c_z) = \{p\}(\mathbf{rely}\, r \bullet \{p\}c_z)\ .$$

Proof. Refinement from right to left is simply removing the precondition. Refinement from left to right holds by Law 73 (rely-refinement-precondition) provided both the following hold.

$$\{p\}c \quad \sqsubseteq_z \quad (\{p\}(\mathbf{rely}\, r \bullet \{p\}c_z)) \parallel \langle r \vee \mathrm{id}\rangle^* \tag{71}$$

$$p \quad \Rrightarrow \quad stps(\{p\}(\mathbf{rely}\, r \bullet \{p\}c_z), z \vee r) \tag{72}$$

Because by Law 6 (precondition) part (20), $\{p\}c = \{p\}\{p\}c$, (71) can be shown as follows.

$\{p\}\{p\}c$

$\sqsubseteq_z$ by Law 75 (rely) using assumption $p \Rrightarrow stps(c,z)$

$\{p\}((\mathbf{rely}\, r \bullet \{p\}c_z) \parallel \langle r \vee \mathrm{id}\rangle^*)$

$\sqsubseteq$ by Lemma 7 (parallel-precondition) and Law 6 (precondition)

$(\{p\}(\mathbf{rely}\, r \bullet \{p\}c_z)) \parallel \langle r \vee \mathrm{id}\rangle^*$

One can use Law 70 (rely-stops) to show (72) as follows.

$$stps(\{p\}(\mathbf{rely}\, r \bullet \{p\}c_z), z \vee r) \equiv p \wedge stps((\mathbf{rely}\, r \bullet \{p\}c_z), z \vee r) \equiv p \wedge stps(\{p\}c, z) \equiv p$$

□

Law 80 (nested-rely) *For any command c and relations z, r_0 and r_1,*

$$(\mathbf{rely}\, r_0 \bullet (\mathbf{rely}\, r_1 \bullet c_z)_{z \vee r_1}) \quad = \quad (\mathbf{rely}\, r_0 \vee r_1 \bullet c_z)\ .$$

Proof. The proof uses the fact that $stps(d, z \vee r_0 \vee r_1) \Rrightarrow stps(d \parallel \langle r_0 \vee \mathrm{id}\rangle^*, z \vee r_1)$ by Law 67 (term-in-context).

$(\mathbf{rely}\, r_0 \bullet (\mathbf{rely}\, r_1 \bullet c_z)_{z \vee r_1})$

$=$ by Definition 69 (rely)

$\bigsqcap\{d \mid (\{stps((\mathbf{rely}\, r_1 \bullet c_z), z \vee r_1)\}(\mathbf{rely}\, r_1 \bullet c_z) \sqsubseteq_{z \vee r_1} d \parallel \langle r_0 \vee \mathrm{id}\rangle^*) \wedge$
$(stps((\mathbf{rely}\, r_1 \bullet c_z), z \vee r_1) \Rrightarrow stps(d, z \vee r_0 \vee r_1))\}$

$=$ by Law 70 (rely-stops)

$\bigsqcap\{d \mid (\{stps(c,z)\}(\mathbf{rely}\, r_1 \bullet c_z) \sqsubseteq_{z \vee r_1} d \parallel \langle r_0 \vee \mathrm{id}\rangle^*) \wedge (stps(c,z) \Rrightarrow stps(d, z \vee r_0 \vee r_1))\}$

$=$ by Law 71 (rely-refinement)

$\bigsqcap\{d \mid (\{stps(c,z)\}c \sqsubseteq_z d \parallel \langle r_0 \vee \mathrm{id}\rangle^* \parallel \langle r_1 \vee \mathrm{id}\rangle^*) \wedge$
$(stps(c,z) \Rrightarrow stps(d \parallel \langle r_0 \vee \mathrm{id}\rangle^*, z \vee r_1)) \wedge (stps(c,z) \Rrightarrow stps(d, z \vee r_0 \vee r_1))\}$

$=$ by Lemma 65 (parallel-interference) and Law 67 (term-in-context)

$\bigsqcap\{d \mid (\{stps(c,z)\}c \sqsubseteq_z d \parallel \langle r_0 \vee r_1 \vee \mathrm{id}\rangle^*) \wedge (stps(c,z) \Rrightarrow stps(d, z \vee r_0 \vee r_1))\}$

$=$ by Definition 69 (rely)

$(\mathbf{rely}\, r_0 \vee r_1 \bullet c_z)$

□

In a development process, the permission to make assumptions must be passed on to sub-components, so a rely on a composite command may be distributed to its component commands.

Law 81 (distribute-rely-choice) *For any relations z and r, and set of commands C,*

$$\textbf{rely}\, r \bullet (\textstyle\bigsqcap C)_z \quad \sqsubseteq \quad \bigsqcap\{c \in C \bullet (\textbf{rely}\, r \bullet c_z)\} \ .$$

Proof. By Law 71 (rely-refinement) we must show both the following.

$$\{stps(\textstyle\bigsqcap C, z)\}(\bigsqcap C) \quad \sqsubseteq_z \quad (\bigsqcap\{c \in C \bullet (\textbf{rely}\, r \bullet c_z)\}) \parallel \langle r \vee \mathrm{id}\rangle^* \qquad (73)$$

$$stps(\textstyle\bigsqcap C, z) \quad \Rrightarrow \quad stps(\bigsqcap\{c \in C \bullet (\textbf{rely}\, r \bullet c_z)\}, z \vee r) \qquad (74)$$

The proof of (73) follows starting from the right side.

$(\bigsqcap\{c \in C \bullet (\textbf{rely}\, r \bullet c_z)\}) \parallel \langle r \vee \mathrm{id}\rangle^*$

$=$ by Lemma 68 (distribute-parallel) over nondeterministic choice (60)

$\bigsqcap\{c \in C \bullet (\textbf{rely}\, r \bullet c_z) \parallel \langle r \vee \mathrm{id}\rangle^*\}$

$\sqsupseteq_z$ by Law 14 (nondeterministic-choice) part (26) and Law 75 (rely) using assumption

$\bigsqcap\{c \in C \bullet \{stps(c,z)\}c\}$

$= \{stps(\bigsqcap C, z)\}(\bigsqcap C)$

The proof of (74) follows.

$stps(\bigsqcap\{c \in C \bullet (\textbf{rely}\, r \bullet c_z)\}, z \vee r)$

$\equiv$ termination of a nondeterministic choice

$(\forall c \in C \bullet stps((\textbf{rely}\, r \bullet c_z), z \vee r))$

$\equiv$ by Law 70 (rely-stops)

$(\forall c \in C \bullet stps(c,z))$

$\equiv$ termination of nondeterministic choice

$stps(\bigsqcap C, z)$

□

Law 82 (distribute-rely-sequential) *For any predicate p, relations z and r, commands c_0 and c_1, such that $(p \Rrightarrow stps((c_0\,;\,c_1), z)$*

$$\textbf{rely}\, r \bullet (\{p\}c_0\,;\,c_1)_z \quad \sqsubseteq \quad (\textbf{rely}\, r \bullet (\{p\}c_0)_z)\,;\,(\textbf{rely}\, r \bullet (c_1)_z) \ .$$

Proof. By Law 73 (rely-refinement-precondition) we must show both the following.

$$\{p\}c_0\,;\,c_1 \quad \sqsubseteq_z \quad ((\textbf{rely}\, r \bullet (\{p\}c_0)_z)\,;\,(\textbf{rely}\, r \bullet (c_1)_z)) \parallel \langle r \vee \mathrm{id}\rangle^* \qquad (75)$$

$$p \quad \Rrightarrow \quad stps(((\textbf{rely}\, r \bullet (\{p\}c_0)_z)\,;\,(\textbf{rely}\, r \bullet (c_1)_z)), z \vee r) \qquad (76)$$

Note that $p \Rrightarrow stps((c_0\,;\,c_1), z) \Rrightarrow stps(c_0, z)$. The proof of (75) follows starting from its right side.

$((\textbf{rely}\, r \bullet (\{p\}c_0)_z)\,;\,(\textbf{rely}\, r \bullet (c_1)_z)) \parallel \langle r \vee \mathrm{id}\rangle^*$

$=$ by Lemma 68 (distribute-parallel) over sequential (62)

$((\textbf{rely}\, r \bullet (\{p\}c_0)_z) \parallel \langle r \vee \mathrm{id}\rangle^*)\,;\,((\textbf{rely}\, r \bullet (c_1)_z) \parallel \langle r \vee \mathrm{id}\rangle^*)$

$\sqsupseteq_z$ by Law 75 (rely) twice as $p \Rrightarrow stps(c_0, z)$

$\{p\}c_0\,;\,\{stps(c_1, z)\}c_1$

$=_z$ as $p \Rrightarrow stps((c_0\,;\,c_1), z)$

$\{p\}c_0\,;\,c_1$

For the proof of (76) we note because $p \Rrightarrow stps((c_0 \,;\, c_1), z)$ that $\{p\}c_0 \,;\, c_1 =_z \{p\}c_0 \,;\, \{stps(c_1, z)\}c_1$ and hence by Law 75 (rely)

$$\begin{array}{ll}
 & \{p\}c_0 \,;\, c_1 \sqsubseteq_z ((\textbf{rely}\, r \bullet (\{p\}c_0)_z) \parallel \langle r \vee \mathrm{id}\rangle^*) \,;\, \{stps(c_1, z)\}((\textbf{rely}\, r \bullet (c_1)_z) \parallel \langle r \vee \mathrm{id}\rangle^*) \\
\Rightarrow & \text{by Law 26 (term-monotonic) and assumption } p \Rrightarrow stps((c_0 \,;\, c_1), z) \\
 & p \Rrightarrow stps(((\textbf{rely}\, r \bullet (\{p\}c_0)_z) \parallel \langle r \vee \mathrm{id}\rangle^*) \,;\, \{stps(c_1, z)\}((\textbf{rely}\, r \bullet (c_1)_z) \parallel \langle r \vee \mathrm{id}\rangle^*), z) \\
\Rightarrow & \text{by Lemma 83 (term-sequential)} \\
 & p \Rrightarrow stps(((\textbf{rely}\, r \bullet (\{p\}c_0)_z) \,;\, \{stps(c_1, z)\}(\textbf{rely}\, r \bullet (c_1)_z)), z \vee r)
\end{array}$$

For the last step Law 70 (rely-stops) gives $p \Rrightarrow stps((\textbf{rely}\, r \bullet (\{p\}c_0)_z), z \vee r)$ and $stps(c_1, z) \Rrightarrow stps((\textbf{rely}\, r \bullet (c_1)_z), z \vee r)$. □

Lemma 83 (term-sequential) *For any predicate p, relations r and z, and commands c_0 and c_1, such that $p \Rrightarrow stps(c_0, z \vee r)$ and $p_1 \Rrightarrow stps(c_1, z \vee r)$,*

$$stps((\{p\}(c_0 \parallel \langle r \vee \mathrm{id}\rangle^*) \,;\, \{p_1\}(c_1 \parallel \langle r \vee \mathrm{id}\rangle^*)), z) \Rrightarrow stps((\{p\}c_0 \,;\, \{p_1\}c_1), z \vee r) \,.$$

Law 84 (distribute-rely-conjunction) *For any predicate p, relations z and r, and commands c_0 and c_1, such that $p \Rrightarrow stps(c_0, z) \wedge stps(c_1, z)$,*

$$\textbf{rely}\, r \bullet \{p\}(c_0 \Cap c_1)_z \quad \sqsubseteq \quad (\textbf{rely}\, r \bullet (\{p\}c_0)_z) \Cap (\textbf{rely}\, r \bullet (\{p\}c_1)_z) \,.$$

Proof. By Law 73 (rely-refinement-precondition) we must show both the following.

$$\{p\}(c_0 \Cap c_1) \quad \sqsubseteq_z \quad ((\textbf{rely}\, r \bullet (\{p\}c_0)_z) \Cap (\textbf{rely}\, r \bullet (\{p\}c_1)_z)) \parallel \langle r \vee \mathrm{id}\rangle^* \qquad (77)$$

$$p \quad \Rrightarrow \quad stps(((\textbf{rely}\, r \bullet (\{p\}c_0)_z) \Cap (\textbf{rely}\, r \bullet (\{p\}c_1)_z)), z \vee r)) \qquad (78)$$

The proof of (77) follows starting from the right side.

$$\begin{array}{ll}
 & ((\textbf{rely}\, r \bullet (\{p\}c_0)_z) \Cap (\textbf{rely}\, r \bullet (\{p\}c_1)_z)) \parallel \langle r \vee \mathrm{id}\rangle^* \\
\sqsupseteq & \text{by Lemma 68 (distribute-parallel) over conjunction (61) and conjunction is idempotent} \\
 & ((\textbf{rely}\, r \bullet (\{p\}c_0)_z) \parallel \langle r \vee \mathrm{id}\rangle^*) \Cap ((\textbf{rely}\, r \bullet (\{p\}c_1)_z) \parallel \langle r \vee \mathrm{id}\rangle^*) \\
\sqsupseteq_z & \text{by Law 75 (rely) twice using termination assumptions} \\
 & (\{p\}c_0) \Cap (\{p\}c_1) \\
= & \text{by Lemma 30 (conjunction-strict)} \\
 & \{p\}(c_0 \Cap c_1)
\end{array}$$

For the proof of termination property (78), using Law 70 (rely-stops) one can deduce $p \Rrightarrow stps((\textbf{rely}\, r \bullet (\{p\}c_0)_z), z \vee r) \wedge stps((\textbf{rely}\, r \bullet (\{p\}c_1)_z), z \vee r)$ and hence that (78) holds by Law 31 (conjunction-term). □

Law 85 (distribute-rely-iteration) *For any predicate p, relations z and r, and command c, such that $p \Rrightarrow stps((\{p\}c)^{\omega+}, z)$,*

$$\textbf{rely}\, r \bullet ((\{p\}c)^{\omega+})_z \quad \sqsubseteq \quad (\textbf{rely}\, r \bullet \{p\}c_z)^{\omega+}$$

Proof. Because $p \Rrightarrow stps((\{p\}c)^{\omega+}, z)$, we have $p \Rrightarrow stps(\{p\}c \,;\, (\{p\}c)^{\omega}, z) \Rrightarrow stps(c, z)$. Using Law 73 (rely-refinement-precondition) one must show both the following.

$$(\{p\}c)^{\omega+} \quad \sqsubseteq_z \quad (\textbf{rely}\, r \bullet \{p\}c_z)^{\omega+} \parallel \langle r \vee \mathrm{id}\rangle^* \qquad (79)$$

$$p \quad \Rrightarrow \quad stps((\textbf{rely}\, r \bullet \{p\}c_z)^{\omega+}, z \vee r) \qquad (80)$$

For $c^{\omega+}$ iteration we use Law 18 (iteration-induction) for ω-iteration (37). In general,

$$c \sqcap (c \,;\, d) \sqsubseteq_{r_x} d \qquad \Rightarrow \qquad c^{\omega+} \sqsubseteq_{r_x} d \qquad (81)$$

and hence (79) holds if

$$\{p\}c \sqcap (\{p\}c\,;((\textbf{rely}\,r \bullet \{p\}c_z)^{\omega+} \parallel \langle r \vee \mathrm{id}\rangle^*)) \quad \sqsubseteq_z \quad (\textbf{rely}\,r \bullet \{p\}c_z)^{\omega+} \parallel \langle r \vee \mathrm{id}\rangle^*$$

which we show as follows.

$\{p\}c \sqcap (\{p\}c\,;((\textbf{rely}\,r \bullet \{p\}c_z)^{\omega+} \parallel \langle r \vee \mathrm{id}\rangle^*))$
$\sqsubseteq_z$ by Law 75 (rely) twice as $p \Rrightarrow \mathit{stps}(c,z)$
$((\textbf{rely}\,r \bullet \{p\}c_z) \parallel \langle r \vee \mathrm{id}\rangle^*) \sqcap$
$(((\textbf{rely}\,r \bullet \{p\}c_z) \parallel \langle r \vee \mathrm{id}\rangle^*);((\textbf{rely}\,r \bullet \{p\}c_z)^{\omega+} \parallel \langle r \vee \mathrm{id}\rangle^*))$
$=$ by Lemma 68 (distribute-parallel) over choice (60) and sequential (62)
$((\textbf{rely}\,r \bullet \{p\}c_z) \sqcap ((\textbf{rely}\,r \bullet \{p\}c_z)\,;(\textbf{rely}\,r \bullet \{p\}c_z)^{\omega+})) \parallel \langle r \vee \mathrm{id}\rangle^*$
$=$ by Law 17 (fold/unfold-iteration)
$(\textbf{rely}\,r \bullet \{p\}c_z)^{\omega+} \parallel \langle r \vee \mathrm{id}\rangle^*$

For termination condition (80) by Law 75 (rely)

$\{p\}c \sqsubseteq_z (\textbf{rely}\,r \bullet \{p\}c_z) \parallel \langle r \vee \mathrm{id}\rangle^*$
$\Rightarrow$ by Law 19 (iteration-monotonic)
$(\{p\}c)^{\omega+} \sqsubseteq_z ((\textbf{rely}\,r \bullet \{p\}c_z) \parallel \langle r \vee \mathrm{id}\rangle^*)^{\omega+}$
$\Rightarrow$ by Law 26 (term-monotonic) and assumption
$p \Rrightarrow \mathit{stps}((\{p\}c)^{\omega+},z) \Rrightarrow \mathit{stps}(((\textbf{rely}\,r \bullet \{p\}c_z) \parallel \langle r \vee \mathrm{id}\rangle^*)^{\omega+},z)$
$\Rightarrow$ by Lemma 86 (term-iteration) as $p \Rrightarrow \mathit{stps}((\textbf{rely}\,r \bullet \{p\}c_z), z \vee r)$ by Law 70 (rely-stops)
$p \Rrightarrow \mathit{stps}((\textbf{rely}\,r \bullet \{p\}c_z)^{\omega+}, z \vee r)$

□

Lemma 86 (term-iteration) *For any predicate p, relations r and z, and command c, such that $p \Rrightarrow \mathit{stps}(c, z \vee r)$,*

$$\mathit{stps}((\{p\}c \parallel \langle r \vee \mathrm{id}\rangle^*)^{\omega+},z) \Rrightarrow \mathit{stps}((\{p\}c)^{\omega+}, z \vee r)\ .$$

Note that Law 85 (distribute-rely-iteration) applies to $c^{\omega+}$ but not c^{ω} because $(\textbf{rely}\,r \bullet \textbf{nil})$ is not refined by **nil**.

5 Parallel refinement

This section develops laws for refining to a parallel composition making use of rely and guarantee commands to handle interference between the parallel processes. For command conjunction one has the identity

$$\left[x' = 1\right] \Cap \left[y' = 2\right] = \left[x' = 1 \wedge y' = 2\right]\ .$$

However, for parallel composition $\left[x' = 1\right] \parallel \left[y' = 2\right]$ aborts because each of the specifications implicitly has a rely condition of id but each also must modify either x or y, thus breaking the rely of the other. Hence in refining from a conjunction of commands to a parallel composition one must introduce rely and guarantee conditions to control the interference.

A guarantee on a command bounds the interference the command can impose on its environment (Law 87). This allows a command to be refined by a parallel combination with any command that respects the rely (Law 88) and hence a conjunction of commands to be implemented by a parallel combination of commands provided each respects the rely of the other (Law 89). Law 90 extends that to handle a conjunction of commands within a rely context, and Law 91 specialises this to refining a specification (with a conjunction of postconditions) by a parallel composition.

Law 87 (guarantee-bounds-interference) *For any predicate p_1, relations z, g_0 and g_1 and command c_1 such that $p_1 \Rrightarrow stps(c_1, z)$,*

$$\{p_1\}\langle g_1 \vee \mathrm{id}\rangle^* \quad \sqsubseteq_{z \vee g_0} \quad (\mathbf{guar}\, g_1 \bullet (\mathbf{rely}\, g_0 \bullet (\{p_1\}c_1)_z)) .$$

Proof. By Law 70 (rely-stops), $p_1 \Rrightarrow stps((\mathbf{rely}\, g_0 \bullet (\{p_1\}c_1)_z), z \vee g_0)$.

$$\begin{array}{ll} & \{p_1\}\langle g_1 \vee \mathrm{id}\rangle^* \\ \sqsubseteq_{z \vee g_0} & \text{by Law 49 (refine-to-guarantee) as } p_1 \Rrightarrow stps((\mathbf{rely}\, g_0 \bullet (\{p_1\}c_1)_z), z \vee g_0) \\ & (\mathbf{guar}\, g_1 \bullet (\mathbf{rely}\, g_0 \bullet (\{p_1\}c_1)_z)) \end{array}$$

□

The following law makes use of a fundamental property of rely commands (Law 75 (rely)) to introduce a parallel composition and a fundamental property of guarantee commands (Lemma 50 (refine-in-guarantee-context)) to develop a symmetric parallel refinement law.

Law 88 (refine-by-parallel) *For any predicates p and p_1, relations z, g_0 and g_1, such that $p \Rrightarrow p_1$, and commands c_0 and c_1 such that $p \Rrightarrow stps(c_0, z)$ and $p_1 \Rrightarrow stps(c_1, z)$,*

$$\{p\}c_0 \quad \sqsubseteq_z \quad (\mathbf{guar}\, g_0 \bullet (\mathbf{rely}\, g_1 \bullet (\{p\}c_0)_z)) \parallel (\mathbf{guar}\, g_1 \bullet (\mathbf{rely}\, g_0 \bullet (\{p_1\}c_1)_z)) .$$

Proof.

$$\begin{array}{ll} & \{p\}c_0 \\ \sqsubseteq_z & \text{by Law 75 (rely) using assumption } p \Rrightarrow stps(c_0, z) \\ & \{p\}((\mathbf{rely}\, g_1 \bullet (\{p\}c_0)_z) \parallel \langle g_1 \vee \mathrm{id}\rangle^*) \\ \sqsubseteq & \text{by Law 44 (introduce-guarantee) and Lemma 7 (parallel-precondition) and } p \Rrightarrow p_1 \\ & (\mathbf{guar}\, g_0 \bullet (\mathbf{rely}\, g_1 \bullet (\{p\}c_0)_z)) \parallel \{p_1\}\langle g_1 \vee \mathrm{id}\rangle^* \\ \sqsubseteq_z & \text{by Law 87 (guarantee-bounds-interference) and Lemma 50 (refine-in-guarantee-context)} \\ & (\mathbf{guar}\, g_0 \bullet (\mathbf{rely}\, g_1 \bullet (\{p\}c_0)_z)) \parallel (\mathbf{guar}\, g_1 \bullet (\mathbf{rely}\, g_0 \bullet (\{p_1\}c_1)_z)) \end{array}$$

□ The following law captures the core of the rely-guarantee approach to developing a parallel program.

Law 89 (introduce-parallel) *For any predicates p, p_0 and p_1, relations z, g_0 and g_1, such that $p \Rrightarrow p_0 \wedge p_1$, and commands c_0 and c_1, such that $p_0 \Rrightarrow stps(c_0, z)$ and $p_1 \Rrightarrow stps(c_1, z)$,*

$$\{p\}(c_0 \Cap c_1) \quad \sqsubseteq_z \quad (\mathbf{guar}\, g_0 \bullet (\mathbf{rely}\, g_1 \bullet (\{p_0\}c_0)_z)) \parallel (\mathbf{guar}\, g_1 \bullet (\mathbf{rely}\, g_0 \bullet (\{p_1\}c_1)_z)) .$$

Proof. First one can distribute the precondition using Lemma 30 (conjunction-strict).

$$\{p\}(c_0 \Cap c_1) = (\{p\}c_0) \Cap (\{p\}c_1)$$

Using Law 88 (refine-by-parallel) as $p \Rrightarrow p_0 \wedge p_1$ and the termination assumptions, one can deduce both the following.

$$\begin{array}{lll} \{p\}c_0 & \sqsubseteq_z & (\mathbf{guar}\, g_0 \bullet (\mathbf{rely}\, g_1 \bullet (\{p_0\}c_0)_z)) \parallel (\mathbf{guar}\, g_1 \bullet (\mathbf{rely}\, g_0 \bullet (\{p_1\}c_1)_z)) \\ \{p\}c_1 & \sqsubseteq_z & (\mathbf{guar}\, g_1 \bullet (\mathbf{rely}\, g_0 \bullet (\{p_1\}c_1)_z)) \parallel (\mathbf{guar}\, g_0 \bullet (\mathbf{rely}\, g_1 \bullet (\{p_0\}c_0)_z)) \end{array}$$

Because parallel is commutative the right sides of both these refinements are the same and hence by Law 28 (refine-conjunction) the right side refines the conjoined left sides.

$$(\{p\}c_0) \Cap (\{p\}c_1) \quad \sqsubseteq_z \quad (\mathbf{guar}\, g_0 \bullet (\mathbf{rely}\, g_1 \bullet (\{p_0\}c_0)_z)) \parallel (\mathbf{guar}\, g_1 \bullet (\mathbf{rely}\, g_0 \bullet (\{p_1\}c_1)_z))$$

□

Law 90 (introduce-parallel-with-rely) *For any predicates p, p_0 and p_1, relations z, r, g_0 and g_1, such that $p \Rrightarrow p_0 \wedge p_1$, and commands c_0 and c_1 such that $p_0 \Rrightarrow stps(c_0, z)$ and $p_1 \Rrightarrow stps(c_1, z)$*

$$\begin{array}{ll} & (\mathbf{rely}\, r \bullet (\{p\}(c_0 \Cap c_1))_z) \\ \sqsubseteq & (\mathbf{guar}\, g_0 \bullet (\mathbf{rely}\, g_1 \vee r \bullet (\{p_0\}c_0)_z)) \parallel (\mathbf{guar}\, g_1 \bullet (\mathbf{rely}\, g_0 \vee r \bullet (\{p_1\}c_1)_z)) \end{array} \qquad (82)$$

Proof. From the assumptions one can deduce both $p_0 \Rrightarrow stps((\mathbf{rely}\, r \bullet (\{p_0\}c_0)_z), z \vee r)$ and $p_1 \Rrightarrow stps((\mathbf{rely}\, r \bullet (\{p_1\}c_1)_z), z \vee r)$. These are used in the application of Law 89 (introduce-parallel) below. By Law 72 (rely-environment) it is sufficient to show (82) in context $z \vee r$.

$$\begin{array}{ll} & (\mathbf{rely}\, r \bullet \{p\}(c_0 \Cap c_1)_z) \\ \sqsubseteq & \text{by Lemma 30 (conjunction-strict) and as } p \Rrightarrow p_0 \text{ and } p \Rrightarrow p_1 \\ & (\mathbf{rely}\, r \bullet (\{p_0\}c_0 \Cap \{p_1\}c_1)_z) \\ \sqsubseteq & \text{Law 84 (distribute-rely-conjunction) using termination conditions} \\ & (\mathbf{rely}\, r \bullet (\{p_0\}c_0)_z) \Cap (\mathbf{rely}\, r \bullet (\{p_1\}c_1)_z) \\ \sqsubseteq & \text{by Law 79 (rely-precondition) twice using termination conditions} \\ & (\{p_0\}(\mathbf{rely}\, r \bullet (\{p_0\}c_0)_z)) \Cap (\{p_1\}(\mathbf{rely}\, r \bullet (\{p_1\}c_1)_z)) \\ = & \text{by Lemma 30 (conjunction-strict)} \\ & \{p_0 \wedge p_1\}((\mathbf{rely}\, r \bullet (\{p_0\}c_0)_z) \Cap (\mathbf{rely}\, r \bullet (\{p_1\}c_1)_z) \\ \sqsubseteq_{z \vee r} & \text{by Law 89 (introduce-parallel) using termination conditions} \\ & (\mathbf{guar}\, g_0 \bullet (\mathbf{rely}\, g1 \bullet (\mathbf{rely}\, r \bullet (\{p_0\}c_0)_z)_{z \vee r})) \parallel (\mathbf{guar}\, g_1 \bullet (\mathbf{rely}\, g_0 \bullet (\mathbf{rely}\, r \bullet (\{p_1\}c_1)_z)_{z \vee r})) \\ = & \text{by Law 80 (nested-rely) twice} \\ & (\mathbf{guar}\, g_0 \bullet (\mathbf{rely}\, g1 \vee r \bullet (\{p_0\}c_0)_z)) \parallel (\mathbf{guar}\, g_1 \bullet (\mathbf{rely}\, g_0 \vee r \bullet (\{p_1\}c_1)_z)) \end{array}$$

□

The following law applies Law 90 to specifications. This corresponds to the parallel introduction law of Jones (1983).

Law 91 (introduce-parallel-spec-with-rely) *For predicates p, p_0 and p_1, and relations r, q, q_0, q_1, g_0 and g_1, such that $p \Rrightarrow p_0 \wedge p_1$ and $p \wedge q_0 \wedge q_1 \Rrightarrow q$,*

$$(\mathbf{rely}\, r \bullet [p,\ q]) \quad \sqsubseteq \quad (\mathbf{guar}\, g_0 \bullet (\mathbf{rely}\, g_1 \vee r \bullet [p_0,\ q_0])) \parallel (\mathbf{guar}\, g_1 \bullet (\mathbf{rely}\, g_0 \vee r \bullet [p_1,\ q_1])) .$$

Proof. The termination assumptions for the application of Law 90 (introduce-parallel-with-rely) below are $p \Rrightarrow stps([q_0], \mathrm{id})$ and $p \Rrightarrow stps([q_1], \mathrm{id})$, which are trivial.

$$\begin{array}{ll} & (\mathbf{rely}\, r \bullet [p,\ q]) \\ \sqsubseteq & \text{by Lemma 11 (consequence) as } p \wedge q_0 \wedge q_1 \Rrightarrow q \text{ and Law 78 (rely-monotonic)} \\ & (\mathbf{rely}\, r \bullet \{p\}\ [q_0 \wedge q_1]) \\ = & \text{by Law 39 (conjoined-specifications)} \\ & (\mathbf{rely}\, r \bullet \{p\}([q_0] \Cap [q_1])) \\ \sqsubseteq & \text{by Law 90 (introduce-parallel-with-rely)} \\ & ((\mathbf{guar}\, g_0 \bullet (\mathbf{rely}\, g_1 \vee r \bullet [p_0,\ q_0])) \parallel (\mathbf{guar}\, g_1 \bullet (\mathbf{rely}\, g_0 \vee r \bullet [p_1,\ q_1]))) \end{array}$$

□

6 Trading postconditions with rely and guarantee

Any command in a guarantee context of g and a rely context of r will only execute atomic program steps satisfying $(g \vee \mathrm{id})$ and assumes the environment only executes atomic steps satisfying $(r \vee \mathrm{id})$

and hence any single step (whether program or environment) satisfies $(g \vee r \vee \mathrm{id})$ and hence any sequence of steps satisfies $(g \vee r)^*$. We first apply this property to a specification and then to augment the law for introducing a parallel composition.

Law 92 (trade-rely-guarantee) *For any predicate p and relations g, r and q,*

$$\mathbf{guar}\, g \bullet (\mathbf{rely}\, r \bullet [p,\ q \wedge (g \vee r)^*]) \quad = \quad \mathbf{guar}\, g \bullet (\mathbf{rely}\, r \bullet [p,\ q])\ .$$

Proof. Refinement from right to left is just a strengthening of the postcondition. The refinement from left to right can be shown using Law 76 (guarantee-plus-rely). The guarantee proviso (70) of Law 76 is trivial; to prove the other proviso (69) one can use Law 74 (rely-specification) which requires one to show both the following.

$$[p,\ q \wedge (g \vee r)^*] \quad \sqsubseteq \quad (\mathbf{guar}\, g \bullet (\mathbf{rely}\, r \bullet [p,\ q])) \parallel \langle r \vee \mathrm{id}\rangle^* \tag{83}$$

$$p \quad \Rrightarrow \quad stps((\mathbf{guar}\, g \bullet (\mathbf{rely}\, r \bullet [p,\ q])), r) \tag{84}$$

The proof for (84) follows by Law 70 (rely-stops) and Law 45 (guarantee-term).

$$p \equiv stps([p,\ q]\ , \mathrm{id}) \equiv stps((\mathbf{rely}\, r \bullet [p,\ q]), r) \Rrightarrow stps((\mathbf{guar}\, g \bullet (\mathbf{rely}\, r \bullet [p,\ q])), r)$$

The proof of the refinement (83) follows.

$[p,\ q \wedge (g \vee r)^*]$

$\sqsubseteq$ by Law 44 (introduce-guarantee) of $g \vee r$ and Law 52 (trading-post-guarantee)

$\mathbf{guar}\, g \vee r \bullet [p,\ q]$

$\sqsubseteq$ by Law 75 (rely)

$\mathbf{guar}\, g \vee r \bullet ((\mathbf{rely}\, r \bullet [p,\ q]) \parallel \langle r \vee \mathrm{id}\rangle^*)$

$=$ by Law 47 (distribute-guarantee) over parallel (56)

$(\mathbf{guar}\, g \vee r \bullet (\mathbf{rely}\, r \bullet [p,\ q])) \parallel (\mathbf{guar}\, g \vee r \bullet \langle r \vee \mathrm{id}\rangle^*)$

$\sqsubseteq$ by Law 42 (strengthen-guarantee) twice

$(\mathbf{guar}\, g \bullet (\mathbf{rely}\, r \bullet [p,\ q])) \parallel (\mathbf{guar}\, r \bullet \langle r \vee \mathrm{id}\rangle^*)$

$=$ by Law 37 (conjunction-atomic-iterated) part(52) and Law 36 (terminating-iteration)

$(\mathbf{guar}\, g \bullet (\mathbf{rely}\, r \bullet [p,\ q])) \parallel \langle r \vee \mathrm{id}\rangle^*$

Because conjunction is idempotent the last step holds as

$$(\mathbf{guar}\, r \bullet \langle r \vee \mathrm{id}\rangle^*) = \langle r \vee \mathrm{id}\rangle^\omega \Cap \langle r \vee \mathrm{id}\rangle^* = \langle r \vee \mathrm{id}\rangle^\omega \Cap \langle \mathsf{true}\rangle^* \Cap \langle r \vee \mathrm{id}\rangle^* = \langle r \vee \mathrm{id}\rangle^*\ .$$

□

Law 93 (introduce-parallel-spec-with-trading) *For any predicates p, p_0 and p_1, and relations r, q, q_0, q_1, g_0 and g_1, such that $p \Rrightarrow p_0 \wedge p_1$ and $p \wedge q_0 \wedge q_1 \Rrightarrow q$,*

$$\begin{aligned} &(\mathbf{rely}\, r \bullet [p,\ q \wedge (g_0 \vee g_1 \vee r)^*]) \\ \sqsubseteq\ &(\mathbf{guar}\, g_0 \bullet (\mathbf{rely}\, g_1 \vee r \bullet [p_0,\ q_0])) \parallel (\mathbf{guar}\, g_1 \bullet (\mathbf{rely}\, g_0 \vee r \bullet [p_1,\ q_1])) \end{aligned}$$

Proof. We use Law 92 (trade-rely-guarantee) and then introduce a parallel composition.

$(\mathbf{rely}\, r \bullet [p,\ q \wedge (g_0 \vee g_1 \vee r)^*])$

$\sqsubseteq$ by Law 44 (introduce-guarantee) of $g_0 \vee g_1$ and Law 92 (trade-rely-guarantee)

$(\mathbf{guar}\, g_0 \vee g_1 \bullet (\mathbf{rely}\, r \bullet [p,\ q]))$

$\sqsubseteq$ by Law 91 (introduce-parallel-spec-with-rely) as $p \Rrightarrow p_0 \wedge p_1$ and $p \wedge q_0 \wedge q_1 \Rrightarrow q$

$(\mathbf{guar}\, g_0 \vee g_1 \bullet ((\mathbf{guar}\, g_0 \bullet (\mathbf{rely}\, g_1 \vee r \bullet [p_0,\ q_0])) \parallel (\mathbf{guar}\, g_1 \bullet (\mathbf{rely}\, g_0 \vee r \bullet [p_1,\ q_1]))))$

$=$ by Law 47 (distribute-guarantee) and Law 46 (nested-guarantees)

$(\mathbf{guar}\, g_0 \bullet (\mathbf{rely}\, g_1 \vee r \bullet [p_0,\ q_0])) \parallel (\mathbf{guar}\, g_1 \bullet (\mathbf{rely}\, g_0 \vee r \bullet [p_1,\ q_1]))$

□

Law 94 (introduce-parallel-spec-weakened-relies) *For any predicates p, p_0 and p_1 and relations r, r_0, r_1, q, q_0, q_1, g_0 and g_1, such that $p \Rrightarrow p_0 \wedge p_1$ and $p \wedge q_0 \wedge q_1 \Rrightarrow q$ and $g_0 \vee r \Rrightarrow r_1 \vee \mathrm{id}$ and $g_1 \vee r \Rrightarrow r_0 \vee \mathrm{id}$,*

$$\begin{array}{ll} & (\mathbf{rely}\, r \bullet [p,\ q \wedge (g_0 \vee g_1 \vee r)^*]) \\ \sqsubseteq & (\mathbf{guar}\, g_0 \bullet (\mathbf{rely}\, r_0 \bullet [p_0,\ q_0])) \parallel (\mathbf{guar}\, g_1 \bullet (\mathbf{rely}\, r_1 \bullet [p_1,\ q_1])) \end{array}$$

Proof. The law follows from Law 93 (introduce-parallel-spec-with-trading) and two applications of Law 77 (weaken-rely). □

Trading can also be applied to a guarantee invariant.

Law 95 (trade-rely-guarantee-invariant) *For predicate p and relations r and q, such that $r \Rrightarrow (p \Rightarrow p')$,*

$$\mathbf{rely}\, r \bullet [p,\ p' \wedge q] \quad \sqsubseteq \quad \mathbf{guar\text{-}inv}\, p \bullet (\mathbf{rely}\, r \bullet [p,\ q])$$

Proof.

$$\begin{array}{ll} & \mathbf{rely}\, r \bullet [p,\ p' \wedge q] \\ \sqsubseteq & \text{by Lemma 11 (consequence) as } r \Rrightarrow (p \Rightarrow p') \\ & \mathbf{rely}\, r \bullet [p,\ ((p \Rightarrow p') \vee r)^* \wedge q] \\ \sqsubseteq & \text{by Law 44 (introduce-guarantee)} \\ & \mathbf{guar}(p \Rightarrow p') \bullet \mathbf{rely}\, r \bullet ([p,\ ((p \Rightarrow p') \vee r)^* \wedge q]) \\ = & \text{by Law 92 (trade-rely-guarantee) and Definition 63 (guarantee-invariant)} \\ & \mathbf{guar\text{-}inv}\, p \bullet (\mathbf{rely}\, r \bullet [p,\ q]) \end{array}$$

□

7 Specifications and rely commands

A specification placed in an environment that can generate interference steps that satisfy r or stutter must *at least* be able to tolerate any finite number of r steps (zero or more) both before and after its execution. Hence we introduce the following definition.

Definition 96 (tolerate-interference) *A specification $[p,\ q]$ tolerates interference r provided*

$$r \Rrightarrow (p \Rightarrow p') \tag{85}$$

$$p \wedge (r \fatsemi q) \Rrightarrow q \tag{86}$$

$$p \wedge (q \fatsemi r) \Rrightarrow q \tag{87}$$

Law 97 (tolerate-interference) *If a specification $[p,\ q]$ tolerates interference r then*

$$[p,\ q] \quad \sqsubseteq \quad \langle r \vee \mathrm{id}\rangle^* \,;\, [p,\ q] \,;\, \langle r \vee \mathrm{id}\rangle^*$$

Proof. Properties (85), (86) and (87) imply both the following.

$$r^* \Rrightarrow (p \Rightarrow p') \tag{88}$$

$$p \wedge (r^* \fatsemi q \fatsemi r^*) \Rrightarrow q \tag{89}$$

$$\begin{array}{ll} & [p,\ q] \\ \sqsubseteq & \text{by Lemma 11 (consequence) as } p \wedge (r^* \fatsemi q \fatsemi r^*) \Rrightarrow q \\ & [p,\ r^* \fatsemi q \fatsemi r^*] \\ \sqsubseteq & \text{by Lemma 12 (sequential) twice as } r^* \Rrightarrow (p \Rightarrow p') \\ & [r^*] \,;\, [p,\ q] \,;\, [r^*] \\ \sqsubseteq & \text{by Law 24 (refine-iterated-relation) twice as } (r \vee \mathrm{id})^* = r^* \\ & \langle r \vee \mathrm{id}\rangle^* \,;\, [p,\ q] \,;\, \langle r \vee \mathrm{id}\rangle^* \end{array}$$

□ Conditions (85), (86) and (87) are slight generalisations of conditions **PR-ident**, **RQ-ident**, and **QR-ident** used by Coleman and Jones (2007, Sect. 3.3) in which r is assumed to be reflexive and transitive. They are also closely related to the the concept of *stability* of p and q in the sense of Wickerson, Dodds, and Parkinson (2010), although that paper limits postconditions to single state predicates rather than relations.

If $\left[p,\ q\right]$ can be implemented by a single atomic step, tolerating interference is sufficient to show overall feasibility but, in general, tolerating interference does not guarantee that $\left[p,\ q\right]$ can be implemented because the conditions do not address interference while $\left[p,\ q\right]$ is executing, only before and after. Interference during $\left[p,\ q\right]$ is handled by distributing the rely. We may have that $\left[p,\ q\right]$ tolerates interference r and

$$\left[p,\ q\right] \quad \sqsubseteq \quad \left[p,\ q_0\right] ; \left[p_1,\ q_1\right]$$

but, when these are placed in a rely context and the rely is distributed, one gets

$$(\mathbf{rely}\, r \bullet \left[p,\ q\right]) \quad \sqsubseteq \quad (\mathbf{rely}\, r \bullet \left[p,\ q_0\right]) ; (\mathbf{rely}\, r \bullet \left[p_1,\ q_1\right])$$

but there is no guarantee that either $\left[p,\ q_0\right]$ or $\left[p_1,\ q_1\right]$ tolerate interference r. Hence, as expected, a feasible refinement in the standard sequential refinement calculus may no longer be feasible in the context of a rely condition.

The following law corresponds to Jones-style sequential introduction (Jones 1983); note that by Law 41 (guarantee-monotonic) both sides of the law may be enclosed in the same guarantee, which may then be distributed to the two components on the right using Law 47 (distribute-guarantee) over sequential (55).

Law 98 (rely-sequential) *For preconditions p_0 and p_1, and relations r, q_0 and q_1, such that $p_0 \wedge ((q_0 \wedge p_1') \fatsemi q_1) \Rrightarrow q$,*

$$(\mathbf{rely}\, r \bullet \left[p_0,\ q\right]) \sqsubseteq (\mathbf{rely}\, r \bullet \left[p_0,\ q_0 \wedge p_1'\right]) ; (\mathbf{rely}\, r \bullet \left[p_1,\ q_1\right])$$

Proof.

$(\mathbf{rely}\, r \bullet \left[p_0,\ q\right])$

$\sqsubseteq$ by Lemma 12 (sequential) and Law 78 (rely-monotonic)

$(\mathbf{rely}\, r \bullet \left[p_0,\ q_0 \wedge p_1'\right] ; \left[p_1,\ q_1\right])$

$\sqsubseteq$ by Law 82 (distribute-rely-sequential)

$(\mathbf{rely}\, r \bullet \left[p_0,\ q_0 \wedge p_1'\right]) ; (\mathbf{rely}\, r \bullet \left[p_1,\ q_1\right])$

□

A specification command within a rely may be refined to an atomic step satisfying the specification provided it can tolerate interference satisfying the rely before and after the atomic step.

Law 99 (rely-to-atomic) *For any predicate p, and relations r and q, such that $\left[p,\ q\right]$ tolerates interference r,*

$$(\mathbf{rely}\, r \bullet \left[p,\ q\right]) \sqsubseteq \langle p, q\rangle\ .$$

Proof. Note that the precondition p in the specification $\left[p,\ q\right]$ must hold in the initial state before any steps, including environment steps, whereas the precondition p in $\langle p, q\rangle$ must hold in the state in which the atomic step is executed, which may be after a number of environment steps. As r preserves p, if p holds initially, it holds after any number of environment steps that respect r, and hence $p \Rrightarrow stps(\langle p, q\rangle, r)$.

$(\mathbf{rely}\, r \bullet \left[p,\ q\right]) \sqsubseteq \langle p, q\rangle$

$\equiv$ by Law 74 (rely-specification) as $p \Rrightarrow stps(\langle p, q\rangle, r)$

$$
\begin{array}{l}
\quad [p,\ q] \sqsubseteq \langle p,q\rangle \parallel \langle r \vee \mathrm{id}\rangle^* \\
\equiv \quad \text{by Lemma 66 (interference-atomic)} \\
\quad [p,\ q] \sqsubseteq \langle r \vee \mathrm{id}\rangle^* \mathbin{;} \langle p,q\rangle \mathbin{;} \langle r \vee \mathrm{id}\rangle^* \\
\Leftarrow \quad \text{by Lemma 10 (make-atomic)} \\
\quad [p,\ q] \sqsubseteq \langle r \vee \mathrm{id}\rangle^* \mathbin{;} [p,\ q] \mathbin{;} \langle r \vee \mathrm{id}\rangle^*
\end{array}
$$

The last refinement holds by Law 97 (tolerate-interference) as $[p,\ q]$ tolerates r. □

If $[p,\ q]$ tolerates interference $\mathrm{id}(X)$, where X contains the free variables of p and q, it is tempting to use a non-atomic specification on the right in Law 99. However, recall that a specification without a rely condition is treated as if the rely is the identity relation, i.e. only stuttering interference is allowed. In this case a refinement of $[p,\ q]$ is free to make use of variables outside X and these variables are not guaranteed to be stable according to the rely condition $\mathrm{id}(X)$. However, if any refinement of $[p,\ q]$ is restricted to use only variables in X, it will be an implementation of the left side. To address this issue we introduce a new language construct $(\mathbf{uses}\, X \bullet c)$ that can be refined by a command d only if $c \sqsubseteq d$ and d exclusively uses (reads or writes) variables in X.

The set of traces of $(\mathbf{uses}\, X \bullet c)$ contains just those traces of c such that each atomic step is dependent on only the variables in X. When c has been refined to code this requirement can be discharged syntactically. The **uses** construct distributes through all language constructs in a straightforward manner, although a little care is required with local variable declarations. The rules are straightforward and hence they are not spelled out here.[8]

Law 100 (uses-atomic-effective) *For any predicate p, relation q, command c, and set of variables X,*

$$[p,\ q] \sqsubseteq (\mathbf{uses}\, X \bullet c) \parallel \langle \mathrm{id}(X)\rangle^* \qquad \Leftrightarrow \qquad [p,\ q] \sqsubseteq \langle \mathrm{id}(X)\rangle^* \mathbin{;} (\mathbf{uses}\, X \bullet c) \mathbin{;} \langle \mathrm{id}(X)\rangle^*$$

Proof. The implication from left to right is straightforward. For the reverse implication, any trace of the $(\mathbf{uses}\, X \bullet c)$ must be a trace of c in which every atomic step does not modify or depend on variables outside X. As there are only a finite number of interference steps each satisfying $\mathrm{id}(X)$ in a trace of $(\mathbf{uses}\, X \bullet c) \parallel \langle \mathrm{id}(X)\rangle^*$, the above property allows all such interference steps to be moved to the left or right to give an equivalent trace (in terms of the overall relation) consisting of a trace of c surrounded by traces of $\langle \mathrm{id}(X)\rangle^*$ on either side. □

Law 101 (rely-uses) *For any predicate p, relation q, and set of variables X, such that* $[p,\ q]$ *tolerates interference* $\mathrm{id}(X)$,

$$(\mathbf{rely}\, \mathrm{id}(X) \bullet [p,\ q]) \sqsubseteq (\mathbf{uses}\, X \bullet [p,\ q])\ .$$

Proof. Noting that $\mathrm{id}(X) \vee \mathrm{id} = \mathrm{id}(X)$ and $p \equiv \mathit{stps}((\mathbf{uses}\, X \bullet [p,\ q]), \mathrm{id}(X))$, by Law 74 (rely-specification) the theorem holds if,

$$
\begin{array}{l}
\quad [p,\ q] \sqsubseteq (\mathbf{uses}\, X \bullet [p,\ q]) \parallel \langle \mathrm{id}(X)\rangle^* \\
\Leftarrow \quad \text{by Law 100 (uses-atomic-effective)} \\
\quad [p,\ q] \sqsubseteq \langle \mathrm{id}(X)\rangle^* \mathbin{;} (\mathbf{uses}\, X \bullet [p,\ q]) \mathbin{;} \langle \mathrm{id}(X)\rangle^* \\
\Leftarrow \quad \text{by Law 97 (tolerate-interference) as } [p,\ q] \text{ tolerates } \mathrm{id}(X) \\
\quad [p,\ q] \sqsubseteq (\mathbf{uses}\, X \bullet [p,\ q])
\end{array}
$$

which holds as $c \sqsubseteq (\mathbf{uses}\, X \bullet c)$ for any command c. □ Because the “uses” construct is monotonic with respect to refinement, if $[p,\ q] \sqsubseteq c$, then the refinement $(\mathbf{uses}\, X \bullet [p,\ q]) \sqsubseteq (\mathbf{uses}\, X \bullet c)$ is also valid; that allows sequential refinement rules to be used to refine $(\mathbf{uses}\, X \bullet [p,\ q])$ provided the final code respects the “uses” clause restriction.

[8]Care would be needed for a language which allowed aliasing of variable names because a **uses** clause involving a variable x would implicitly include any aliases of x.

Law 102 (assignment-rely-guarantee) *For any variable x, expression e, set of variables X, predicate p and relations g and q, such that* $\lceil p,\ q \rceil$ *tolerates interference* $\mathrm{id}(X)$*,* $p \Rrightarrow \mathit{def}(e)$ *and* $p \wedge x' = e \wedge \mathrm{id}(\bar{x}) \Rrightarrow q \wedge (g \vee \mathrm{id})$ *and* $\mathit{vars}(e) \cup \{x\} \subseteq X$

$$x : (\mathbf{guar}\, g \bullet (\mathbf{rely}\, \mathrm{id}(X) \bullet \lceil p,\ q \rceil)) \sqsubseteq x := e\ .$$

Proof. Many of the steps in the proof implicitly use Law 41 (guarantee-monotonic).

$$\begin{array}{ll}
 & x : \mathbf{guar}\, g \bullet (\mathbf{rely}\, \mathrm{id}(X) \bullet \lceil p,\ q \rceil) \\
\sqsubseteq & \text{by Law 101 (rely-uses) as } \lceil p,\ q \rceil \text{ tolerates } \mathrm{id}(X) \\
 & x : \mathbf{guar}\, g \bullet (\mathbf{uses}\, X \bullet \lceil p,\ q \rceil) \\
= & \text{as } \mathbf{uses} \text{ distributes through guarantees and Law 59 (guarantee-frame)} \\
 & \mathbf{uses}\, X \bullet (\mathbf{guar}\, g \bullet x \colon \lceil p,\ q \rceil) \\
\sqsubseteq & \text{by Law 60 (guarantee-assignment) as } p \Rrightarrow \mathit{def}(e) \text{ and } p \wedge x' = e \wedge \mathrm{id}(\bar{x}) \Rrightarrow q \wedge (g \vee \mathrm{id}) \\
 & \mathbf{uses}\, X \bullet x := e \\
\sqsubseteq & \text{as } \mathit{vars}(e) \cup \{x\} \subseteq X \\
 & x := e
\end{array}$$

The restriction on the variables of e and x ensures that every atomic step of $x := e$ does not modify or access variables outside X. □

The VDM rules for rely-guarantee handle this issue via disjoint sets of read and write variables. The union of the variables in VDM read and write sets corresponds to the set of "used" variables, and the variables in the VDM write set correspond to the set of variables in the frame here.

7.1 Local variables

A local variable is immune from interference from its environment. Hence when refining the body of a local variable block, the environment will not change the local variable. Furthermore the values of x in environment steps of a local variable block have no effect on its behaviour.

Lemma 103 (refine-var) *For any variable x, and commands c and d,*

$$c \sqsubseteq_{\mathrm{id}(x)} d \qquad \Rightarrow \qquad (\mathbf{var}\, x \bullet c) \sqsubseteq (\mathbf{var}\, x \bullet d)\ .$$

Law 104 (strengthen-rely-in-context) *For any relations r, rx and z, and command c,*

$$(\mathbf{rely}\, r \bullet c_z) \sqsubseteq_{rx} (\mathbf{rely}\, rx \wedge r \bullet c_z)\ .$$

Proof. By Definition 69 (rely) the law holds provided

$$\begin{array}{ll}
 & \bigsqcap\{d_1 \mid (\{\mathit{stps}(c,z)\}c \sqsubseteq_z d_1 \parallel \langle r \vee \mathrm{id}\rangle^*) \wedge (\mathit{stps}(c,z) \Rrightarrow \mathit{stps}(d_1, z \vee r))\} \\
\sqsubseteq_{rx} & \bigsqcap\{d_0 \mid (\{\mathit{stps}(c,z)\}c \sqsubseteq_z d_0 \parallel \langle (rx \wedge r) \vee \mathrm{id}\rangle^*) \wedge (\mathit{stps}(c,z) \Rrightarrow \mathit{stps}(d_0, z \vee (rx \wedge r)))\} \\
\Leftarrow & \text{by Law 14 (nondeterministic-choice)} \\
 & \forall d_0 \bullet (\{\mathit{stps}(c,z)\}c \sqsubseteq_z d_0 \parallel \langle (rx \wedge r) \vee \mathrm{id}\rangle^*) \wedge (\mathit{stps}(c,z) \Rrightarrow \mathit{stps}(d_0, z \vee (rx \wedge r))) \Rightarrow \\
 & \quad \exists d_1 \bullet (\{\mathit{stps}(c,z)\}c \sqsubseteq_z d_1 \parallel \langle r \vee \mathrm{id}\rangle^*) \wedge (\mathit{stps}(c,z) \Rrightarrow \mathit{stps}(d_1, z \vee r)) \wedge d_1 \sqsubseteq_{rx} d_0
\end{array}$$

As the witness for the existential quantifier choose d_1 to be the same as d_0 except that d_1 only allows environment steps satisfying $rx \vee \mathrm{id}$, that is, $[\![d_1]\!] = [\![d_0]\!]_{rx}$. By Definition 1 (refinement-in-context) one can deduce $d_0 \sqsubseteq d_1 \sqsubseteq_{rx} d_0$. Because the environment steps of d_1 only allow interference satisfying $rx \vee \mathrm{id}$, it follows that

$$\{\mathit{stps}(c,z)\}c \ \sqsubseteq_z\ d_0 \parallel \langle (rx \wedge r) \vee \mathrm{id}\rangle^* \ \sqsubseteq\ d_1 \parallel \langle (rx \wedge r) \vee \mathrm{id}\rangle^* \ =\ d_1 \parallel \langle r \vee \mathrm{id}\rangle^*\ .$$

In addition, $[\![d_1]\!]_{z \vee r} = [\![d_0]\!]_{rx \wedge (z \vee r)} \subseteq [\![d_0]\!]_{z \vee (rx \wedge r)}$. Hence $\mathit{stps}(c,z) \Rrightarrow \mathit{stps}(d_0, z \vee (rx \wedge r)) \Rrightarrow \mathit{stps}(d_1, z \vee r)$. □

Law 105 (variable-rely) *For any command c, relations z and r, variable x, and set of variables y, where x is not in y,*

$$(\mathbf{var}\, x \bullet x, y : (\mathbf{rely}\, r \bullet c_z)) \quad = \quad (\mathbf{var}\, x \bullet x, y : (\mathbf{rely}\, \mathrm{id}(x) \wedge r \bullet c_z)) \ .$$

Proof. The refinement from right to left follows by Law 77 (weaken-rely). The refinement from left to right holds as follows.

$$
\begin{array}{ll}
 & (\mathbf{var}\, x \bullet x, y : (\mathbf{rely}\, r \bullet c_z)) \sqsubseteq (\mathbf{var}\, x \bullet x, y : (\mathbf{rely}\, \mathrm{id}(x) \wedge r \bullet c_z)) \\
\Lleftarrow & \text{by Lemma 103 (refine-var)} \\
 & x, y : (\mathbf{rely}\, r \bullet c_z) \sqsubseteq_{\mathrm{id}(x)} x, y : (\mathbf{rely}\, \mathrm{id}(x) \wedge r \bullet c_z) \\
\Lleftarrow & \text{by Law 41 (guarantee-monotonic) applied to the frames} \\
 & (\mathbf{rely}\, r \bullet c_z) \sqsubseteq_{\mathrm{id}(x)} (\mathbf{rely}\, \mathrm{id}(x) \wedge r \bullet c_z)
\end{array}
$$

The latter follows by Law 104 (strengthen-rely-in-context). □

In the following law the guarantee on the left applies to any global occurrence of x, which cannot be modified by any implementation of c on the right because all its references to x are to the new local x[9], while the rely on the right refers to the local variable x, which because it is local to the process cannot be subject to external interference.

Law 106 (variable-rely-guarantee) *For a command c, relations z, g and r, a set of variables y, and a variable x that is not in y and that does not occur free in any of g, r, z and c,*

$$(\mathbf{guar}\, g \wedge \mathrm{id}(x) \bullet y : (\mathbf{rely}\, r \bullet c_z)) \sqsubseteq (\mathbf{var}\, x \bullet x, y : (\mathbf{guar}\, g \bullet (\mathbf{rely}\, \mathrm{id}(x) \wedge r \bullet c_z)))$$

Proof.

$$
\begin{array}{ll}
 & \mathbf{guar}\, g \wedge \mathrm{id}(x) \bullet y : (\mathbf{rely}\, r \bullet c_z) \\
\sqsubseteq & \text{by Lemma 61 (introduce-variable) as } x \text{ not in } y \text{ and not used by } c \text{ or } r \\
 & \mathbf{guar}\, g \wedge \mathrm{id}(x) \bullet (\mathbf{var}\, x \bullet x, y : (\mathbf{rely}\, r \bullet c_z)) \\
= & \text{by Law 46 (nested-guarantees) and Law 62 (guarantee-variable)} \\
 & \mathbf{guar}\, g \bullet (\mathbf{var}\, x \bullet x, y : (\mathbf{rely}\, r \bullet c_z)) \\
= & \text{by Law 105 (variable-rely) as variable declaration ensures rely } \mathrm{id}(x) \text{ locally} \\
 & \mathbf{guar}\, g \bullet (\mathbf{var}\, x \bullet x, y : (\mathbf{rely}\, \mathrm{id}(x) \wedge r \bullet c_z)) \\
= & \text{by Law 47 (distribute-guarantee) over variable declaration (57) as } g \text{ is independent of } x \\
 & \mathbf{var}\, x \bullet x, y : (\mathbf{guar}\, g \bullet (\mathbf{rely}\, \mathrm{id}(x) \wedge r \bullet c_z))
\end{array}
$$

The last step also uses Law 59 (guarantee-frame). □

8 Control structures and rely commands

Evaluation of an expression becomes nondeterministic if, during its evaluation, a concurrent process can modify variables used within the expression. Properties of nondeterministic expression evaluation have been investigated elsewhere (Coleman and Jones 2007; Coleman 2008; Wickerson, Dodds, and Parkinson 2010; Hayes, Burns, Dongol, and Jones 2013); here we use the results of those investigations. Our treatment of nondeterministic expressions considers a common special case where there is at most a single reference within an expression e to a single variable y that may be modified by the environment and all variables in e other than y are stable (unchanged by the environment) during the evaluation of e. For example, the test $x \leq y$ satisfies the single reference property provided x is a local variable (and hence may not be modified by the environment) and y is a global variable (which may be modified by the environment). For expressions that do not satisfy this property, such as $x \leq y$ when both x and y are global variables, the development typically requires the use of rely conditions that are specific to the application and the refinement needs to use more primitive rules such as Lemma 54 (tests).

[9]This can be violated in a language that allows another variable to be an alias for the global variable x.

If an expression e satisfies the *single reference property*, the value of e is its value in the state in which y is sampled. If the environment respects a rely condition r (which preserves all variables in e other than y), the evaluation state is related to the initial state by r^*. This property is encapsulated by the following law, which is similar in structure to Law 100 (uses-atomic-effective).

Law 107 (test-single-reference) *Given any predicate p, relations q and r, and a boolean expression b, if there is at most one variable y such that $r \Rrightarrow \mathrm{id}(vars(b) - \{y\})$, and if there is at most a single reference to y within b,*

$$\left[p,\ q\right] \sqsubseteq [[b]] \parallel \langle r \vee \mathrm{id}\rangle^* \qquad \Leftrightarrow \qquad \left[p,\ q\right] \sqsubseteq \langle r \vee \mathrm{id}\rangle^* \mathbin{;} [[b]] \mathbin{;} \langle r \vee \mathrm{id}\rangle^* \ .$$

Proof. The implication from left to right is straightforward. For the reverse implication, any trace of $[[b]]$ consists of a single atomic step that references y together with other steps that are either stuttering (calculation) steps or reference variables that are stable under r. For any trace of $[[b]] \parallel \langle r \vee \mathrm{id}\rangle^*$, the stable steps of $[[b]]$ occurring before (or after) the reference to y may all be shifted right or left over the interference steps so that all the steps of $[[b]]$ are together and the overall end-to-end relation between the initial and final states is unchanged. Hence the whole trace of $[[b]]$ is preceded and followed by the interference steps and hence is a trace of $\langle r \vee \mathrm{id}\rangle^* \mathbin{;} [[b]] \mathbin{;} \langle r \vee \mathrm{id}\rangle^*$. □

In practice, it is not enough to simply evaluate a test, but rather one would like to be able to use the successful evaluation as an assumption within the body of a conditional or loop. In the presence of interference this requires some care. For instance, if x is a shared variable that may be modified arbitrarily by the environment and the test $[[x \leq 0]]$ succeeds, that does not allow one to assume that $x \leq 0$ continues to hold after its evaluation because the environment may increase x. However, the property is preserved if the environment respects the rely condition $x' \leq x$. More generally, if $[[b]]$ is a test then one may subsequently assume a weaker condition b_0, provided b_0 is preserved by r, i.e., $r \Rrightarrow (b_0 \Rightarrow b_0')$. This weakening of a test may be useful for various purposes; in Section 9 it is used to allow an early exit from a loop. We now consider this special case for introducing a test, before using it to prove the laws for introducing conditionals and loops.

Law 108 (rely-test) *For any relation r, predicate p that is preserved by r, boolean expression b that has the single reference property and predicate b_0 such that $p \wedge b \Rrightarrow b_0$ and $p \wedge r \Rrightarrow (b_0 \Rightarrow b_0')$ and $p \Rrightarrow def(b)$,*

$$(\mathbf{rely}\, r \bullet \left[p,\ r^* \wedge b_0'\right]) \sqsubseteq [[b]] \ .$$

Proof. Note that due to the assumption that p is preserved by r and the assumption on b_0,

$$p \wedge r^* \Rrightarrow (b_0 \Rightarrow b_0') \ .$$

Hence the specification $\left[p,\ r^* \wedge b_0'\right]$ tolerates interference r. The termination condition, $p \Rrightarrow stps([[b]], r)$, holds because $p \Rrightarrow def(b)$ and p is preserved by r. Hence by Law 74 (rely-specification) the theorem holds if,

$$\left[p,\ r^* \wedge b_0'\right] \sqsubseteq [[b]] \parallel \langle r \vee \mathrm{id}\rangle^*$$
$\equiv$ by Law 107 (test-single-reference)
$$\left[p,\ r^* \wedge b_0'\right] \sqsubseteq \langle r \vee \mathrm{id}\rangle^* \mathbin{;} [[b]] \mathbin{;} \langle r \vee \mathrm{id}\rangle^*$$
$\Leftarrow$ by Law 97 (tolerate-interference)
$$\left[p,\ r^* \wedge b_0'\right] \sqsubseteq [[b]]$$
$\Leftarrow$ by Lemma 11 (consequence) as $p \Rrightarrow def(b)$ and $p \wedge b \Rrightarrow b_0$ and $\mathrm{id} \Rrightarrow r^*$
$$\left[def(b),\ b \wedge \mathrm{id}\right] \sqsubseteq [[b]]$$

which holds by Lemma 15 (introduce-test). □ As discussed above, a boolean expression b satisfying the single reference property may not be invariant under r but it may imply a weaker predicate b_0, that is invariant under r. Similarly, $\neg\, b$ may imply a weaker predicate b_1 that is invariant under r. Hence in a conditional command, if b evaluates to true, b may no longer hold at the start of the "then" part, but b_0 will hold. Similarly, if b evaluates to false, $\neg b$ may no longer hold at the start of the "else" part, but b_1 will hold. Note that $p \wedge (b \vee \neg b) \Rrightarrow b_0 \vee b_1$ that is, $p \Rrightarrow b_0 \vee b_1$, and that b_0 and b_1 may overlap to give nondeterminism in the choice of branch.

Law 109 (rely-conditional) *For any predicate p, relations r and q, such that $\lceil p,\ q\rceil$ tolerates interference r, and boolean expressions b, b_0 and b_1 such that b satisfies the single-reference property and $p \wedge b \Rrightarrow b_0$ and $p \wedge r \Rrightarrow (b_0 \Rightarrow b_0')$, and $p \wedge \neg b \Rrightarrow b_1$ and $p \wedge r \Rrightarrow (b_1 \Rightarrow b_1')$ and $p \Rrightarrow def(b)$,*

$$(\textbf{rely}\, r \bullet \lceil p,\ q\rceil) \sqsubseteq \textbf{if}\, b\ \textbf{then}\ (\textbf{rely}\, r \bullet \lceil p \wedge b_0,\ q\rceil\ \textbf{else}\ (\textbf{rely}\, r \bullet \lceil p \wedge b_1,\ q\rceil)$$

Proof.

$$\begin{array}{ll}
 & \textbf{rely}\, r \bullet \lceil p,\ q\rceil \\
= & \text{as nondeterministic choice is idempotent} \\
 & (\textbf{rely}\, r \bullet \lceil p,\ q\rceil) \sqcap (\textbf{rely}\, r \bullet \lceil p,\ q\rceil) \\
\sqsubseteq & \text{by Law 98 (rely-sequential) twice as } \lceil p,\ q\rceil \text{ tolerates interference } r\text{; hence } p \wedge r^* \operatorname{⨾} q \Rrightarrow q \\
 & ((\textbf{rely}\, r \bullet \lceil p,\ r^* \wedge b_0'\rceil); (\textbf{rely}\, r \bullet \lceil p \wedge b_0,\ q\rceil)) \sqcap ((\textbf{rely}\, r \bullet \lceil p,\ r^* \wedge b_1'\rceil); (\textbf{rely}\, r \bullet \lceil p \wedge b_1,\ q\rceil)) \\
\sqsubseteq & \text{by Law 108 (rely-test) twice using the assumptions on } b_0 \text{ and } b_1 \\
 & ([[b]]\,;(\textbf{rely}\, r \bullet \lceil p \wedge b_0,\ q\rceil)) \sqcap ([[\neg b]]\,;(\textbf{rely}\, r \bullet \lceil p \wedge b_1,\ q\rceil)) \\
= & \text{by the definition of a conditional (7)} \\
 & \textbf{if}\, b\ \textbf{then}\ (\textbf{rely}\, r \bullet \lceil p \wedge b_0,\ q\rceil))\ \textbf{else}\ (\textbf{rely}\, r \bullet \lceil p \wedge b_1,\ q\rceil)
\end{array}$$

□ If the rely condition implies that all the variables used in the boolean expression b are stable, then b_0 and b_1 can be chosen to be b and $\neg b$, respectively. Note that this law may be combined with Law 56 (guarantee-conditional) to handle refinement of a specification to a conditional in both guarantee and rely contexts.

The proof of Law 112 (rely-loop) below depends on properties of an iteration of a specification with a postcondition satisfying a well-founded relation. Iteration of a specification that establishes a well-founded relation terminates.

Law 110 (well-founded-termination) *For any predicate p and relation w, such that w is well founded on p,*

$$\lceil p,\ p' \wedge w\rceil^{\omega+} = \lceil p,\ p' \wedge w\rceil^{+}\ .$$

Proof.

$$\begin{array}{ll}
 & \lceil p,\ p' \wedge w\rceil^{\omega+} \\
= & \text{by Law 20 (isolation)} \\
 & \lceil p,\ p' \wedge w\rceil^{+} \sqcap \lceil p,\ p' \wedge w\rceil^{\infty} \\
= & \text{as } w \text{ is well founded on } p,\ \lceil p,\ p' \wedge w\rceil^{\infty} = \lceil p,\ \mathsf{false}\rceil \\
 & \lceil p,\ p' \wedge w\rceil^{+}
\end{array}$$

□ A relation ω^+ may be established by finitely iterating a specification that establishes ω^+.

Law 111 (refine-to-iteration) *For any precondition p and relation w,*

$$\lceil p,\ p' \wedge w^+\rceil \sqsubseteq \lceil p,\ p' \wedge w^+\rceil^{+}$$

Proof. By Law 18 (iteration-induction) for Kleene-iteration (35)

$$d \sqsubseteq c \sqcap (c\,;d) \quad \Rightarrow \quad d \sqsubseteq c^+ \tag{90}$$

Applying this to the theorem requires one to show,

$$[p,\, p' \wedge w^+] \;\sqsubseteq\; [p,\, p' \wedge w^+] \sqcap [p,\, p' \wedge w^+]\,;[p,\, p' \wedge w^+]$$

which holds as $w^+ \mathbin{⨾} w^+ \Rrightarrow w^+$. □

The law for a "while" loop treats the test in a similar manner to the test in a conditional. To guarantee termination of a loop a well-founded relation w is required. The body of the loop must satisfy w and in addition any interference step satisfying r must either satisfy the transitive closure w^+ or not change any variables in some set X, where w depends only on the variables in X and hence satisfies $(\mathrm{id}(X) \mathbin{⨾} w) \equiv w \equiv (w \mathbin{⨾} \mathrm{id}(X))$. The reflexive transitive closure w^* would be too strong here because it would require that r implies no variables at all are changed if r did not satisfy w^+. We define $w^*_X \mathrel{\widehat{=}} w^+ \vee \mathrm{id}(X)$ and note that if w is well founded then so is w^+.

Law 112 (rely-loop) *For any predicate p, relations r, w and q, and set of variables X, such that $r \Rrightarrow (p \Rightarrow p')$ and w is well-founded on p and depends_only(w,X) and $p \wedge r \Rrightarrow w^*_X$ and boolean expressions b, b_0 and b_1 such that b satisfies the single-reference property and $p \wedge b \Rrightarrow b_0$ and $p \wedge r \Rrightarrow (b_0 \Rightarrow b_0')$, and $p \wedge \neg\, b \Rrightarrow b_1$ and $p \wedge r \Rrightarrow (b_1 \Rightarrow b_1')$ and $p \Rrightarrow def(b)$,*

$$(\mathbf{rely}\, r \bullet [p,\, p' \wedge b_1' \wedge w^*_X]) \;\sqsubseteq\; \mathbf{while}\, b\, \mathbf{do} (\mathbf{rely}\, r \bullet [p \wedge b_0,\, p' \wedge w])$$

Proof. Note that $w^+ \mathbin{⨾} w^*_X \Rrightarrow w^*_X$ because $w \mathbin{⨾} \mathrm{id}(X) \equiv w$.

$\mathbf{rely}\, r \bullet [p,\, p' \wedge b_1' \wedge w^*_X]$

$\sqsubseteq$ as nondeterministic choice is idempotent and Lemma 12 (sequential) as $w^+ \mathbin{⨾} w^*_X \Rrightarrow w^*_X$

$\mathbf{rely}\, r \bullet ([p,\, p' \wedge w^+]\,;[p,\, p' \wedge b_1' \wedge w^*_X] \sqcap [p,\, p' \wedge b_1' \wedge w^*_X])$

$\sqsubseteq$ by Law 81 (distribute-rely-choice) and Law 82 (distribute-rely-sequential)

$((\mathbf{rely}\, r \bullet [p,\, p' \wedge w^+])\,; (\mathbf{rely}\, r \bullet [p,\, p' \wedge b_1' \wedge w^*_X])) \sqcap (\mathbf{rely}\, r \bullet [p,\, p' \wedge b_1' \wedge w^*_X])$

$=$ distribute sequential over nondeterministic choice

$((\mathbf{rely}\, r \bullet [p,\, p' \wedge w^+]) \sqcap \mathbf{nil})\,; (\mathbf{rely}\, r \bullet [p,\, p' \wedge b_1' \wedge w^*_X])$

$\sqsubseteq$ by Lemma 11 (consequence) as $p \wedge r^* \Rrightarrow p' \wedge w^*_X$

$((\mathbf{rely}\, r \bullet [p,\, p' \wedge w^+]) \sqcap \mathbf{nil})\,; (\mathbf{rely}\, r \bullet [p,\, r^* \wedge b_1'])$

$\sqsubseteq$ by Law 108 (rely-test) as $p \wedge \neg\, b \Rrightarrow b_1$ and $p \wedge r \Rrightarrow (b_1 \Rightarrow b_1')$

$((\mathbf{rely}\, r \bullet [p,\, p' \wedge w^+]) \sqcap \mathbf{nil})\,; [[\neg\, b]]$

$\sqsubseteq$ by Law 111 (refine-to-iteration)

$((\mathbf{rely}\, r \bullet [p,\, p' \wedge w^+]^+) \sqcap \mathbf{nil})\,; [[\neg\, b]]$

$=$ as w^+ is well founded using Law 110 (well-founded-termination) but with w^+ rather than w

$((\mathbf{rely}\, r \bullet [p,\, p' \wedge w^+]^{\omega +}) \sqcap \mathbf{nil})\,; [[\neg\, b]]$

$\sqsubseteq$ by Law 85 (distribute-rely-iteration)

$((\mathbf{rely}\, r \bullet [p,\, p' \wedge w^+])^{\omega +} \sqcap \mathbf{nil})\,; [[\neg\, b]]$

$\sqsubseteq$ by Law 17 (fold/unfold-iteration) and $c^{\omega +} = c^{\omega}\,; c$

$(\mathbf{rely}\, r \bullet [p,\, p' \wedge w^+])^{\omega}\,; [[\neg\, b]]$

$\sqsubseteq$ by Law 98 (rely-sequential) as $w^*_X \mathbin{⨾} w \equiv w^+$ because $\mathrm{id}(X) \mathbin{⨾} w \equiv w$

$((\mathbf{rely}\, r \bullet [p,\, p' \wedge b_0' \wedge w^*_X])\,; (\mathbf{rely}\, r \bullet [p \wedge b_0,\, p' \wedge w]))^{\omega}\,; [[\neg\, b]]$

$\sqsubseteq$ by Lemma 11 (consequence) as $p \wedge r^* \Rrightarrow p' \wedge w^*_X$

$((\mathbf{rely}\, r \bullet [p,\, r^* \wedge b_0'])\,; (\mathbf{rely}\, r \bullet [p \wedge b_0,\, p' \wedge w]))^{\omega}\,; [[\neg\, b]]$

$\sqsubseteq$ by Law 108 (rely-test) as $p \wedge b \Rrightarrow b_0$ and $p \wedge r \Rrightarrow (b_0 \Rightarrow b_0')$

$([[b]]\,; (\mathbf{rely}\, r \bullet [p \wedge b_0,\, p' \wedge w]))^{\omega}\,; [[\neg\, b]]$

$=$ by the definition of a loop (8)

$\mathbf{while}\, b\, \mathbf{do} (\mathbf{rely}\, r \bullet [p \wedge b_0,\, p' \wedge w])$

□ If the rely condition implies that all the variables used in the boolean expression b are stable, then b_0 and b_1 can be chosen to be b and $\neg b$, respectively. Also note that if $r \Rrightarrow \mathrm{id}(X)$ then $r \Rrightarrow w_X^*$. This law may be combined with Law 57 (guarantee-loop) to handle refinement of a specification to a loop in both guarantee and rely contexts.

9 Extended example (concurrent version)

The task here is as described in Section 3.10; here, however, the focus is on refining to a concurrent algorithm. This choice of example facilitates comparison with other publications: it is taken from Susan Owicki's thesis (1975) (and is also used by Jones (1981) and de Roever (2001) to contrast the compositional rely/guarantee approach with the "Owicki/Gries" method).

9.1 Specification

The specification requires that *findp* sets the final value of variable t to the lowest index of array v such that $p(v(t))$ for some predicate p; it assumes that neither v nor t is changed by the environment; the frame prefix guarantees no changes to variables other than t are made.

$$\mathit{findp} \mathrel{\widehat{=}} t : \mathbf{rely}\, \mathrm{id}(\{v,t\}) \bullet \left[(t' = \mathit{len}(v) + 1 \lor \mathit{satp}(v,t')) \land \mathit{notp}(v, \mathit{dom}(v), t')\right] \qquad \lhd$$

For a parallel implementation the essential change from Sect. 3.10 is the addition of a rely condition which records the assumption that the key variables v and t experience no interference.

The majority of the laws developed above do not explicitly handle frames, however, a frame of x corresponds to a guarantee of $\mathrm{id}(\bar{x})$, and hence the laws using guarantees can be used wherever needed.

9.2 Representing t using two variables

The implementation developed in Section 3.10 uses one process with an index c and loops from one upwards until either $p(v(c))$ is satisfied or the index goes beyond $\mathit{len}(v)$. Utilising concurrency, one can split the task into two[10] processes and have each process consider a subset of the domain of v.

One danger here is to end up with a design that needs to "lock" the result variable t during updates and there is a better approach followed here. One can avoid the difficulties of sharing t by having a separate index for each of the concurrent processes and representing t by $\mathit{min}(ot, et)$.[11] A poor solution would then use disjoint parallelism where the two processes ignore each other's progress — our aim is an algorithm where the processes "interfere" to achieve better performance. Two local variables ot and et are introduced with the intention that on termination the minimum of ot and et will be the least index satisfying p.

$\sqsubseteq$ by Law 106 (variable-rely-guarantee) for ot and et and Lemma 11 (consequence)

$$\mathbf{var}\; ot, et \bullet$$
$$ot, et, t : \mathbf{rely}\, \mathrm{id}(\{v,t,ot,et\}) \bullet \begin{bmatrix}(\mathit{min}(ot',et') = \mathit{len}(v) + 1 \lor \mathit{satp}(v, \mathit{min}(ot',et'))) \land \\ \mathit{notp}(v, \mathit{dom}(v), \mathit{min}(ot',et')) \land t' = \mathit{min}(ot',et')\end{bmatrix} \qquad \lhd$$

Note that reducing a frame corresponds to strengthening the corresponding guarantee.

$\sqsubseteq$ by Law 98 (rely-sequential), Law 47 (distribute-guarantee) and Law 77 (weaken-rely)

$$ot, et : \mathbf{rely}\, \mathrm{id}(\{v,ot,et\}) \bullet \begin{bmatrix}(\mathit{min}(ot',et') = \mathit{len}(v) + 1 \lor \mathit{satp}(v, \mathit{min}(ot',et'))) \land \\ \mathit{notp}(v, \mathit{dom}(v), \mathit{min}(ot',et'))\end{bmatrix}; \qquad \lhd$$
$$t : \mathbf{rely}\, \mathrm{id}(\{t,ot,et\}) \bullet \left[t' = \mathit{min}(ot,et)\right]$$

[10]Generalising to an arbitrary number of threads presents no conceptual difficulties; in common with the earlier papers using this example, a two way split of the index values into even and odd is considered because this keeps formulae short.

[11]Jones (2007) observes that achieving rely and/or guarantee conditions is often linked with data reification, for instance, viewing $\mathit{min}(ot,et)$ as a representation of the abstract variable t; this point is not pursued here.

The specification $t : \textbf{rely}\, \mathrm{id}(\{t, ot, et\}) \bullet \left[t' = min(ot, et)\right]$ can be refined to $t := min(ot, et)$ using Law 102 (assignment-rely-guarantee) because all the variables involved are stable in the rely condition. A guarantee invariant can be employed in a manner similar to the use in Sect. 3.10; the invariant is established by setting both *ot* and *et* to $len(v) + 1$ using Law 102 (assignment-rely-guarantee).

$\sqsubseteq$ by Law 95 (trade-rely-guarantee-invariant); Law 98 (rely-sequential)
$ot := len(v) + 1\,;\, et := len(v) + 1;$
$\textbf{guar-inv}\, min(ot, et) = len(v) + 1 \vee satp(v, min(ot, et)) \bullet$
$\quad ot, et : \textbf{rely}\, \mathrm{id}(\{v, ot, et\}) \bullet \left[notp(v, \mathrm{dom}(v), min(ot', et'))\right]$ $\lhd$

9.3 Concurrency

The motivation for the parallel algorithm comes from the observation that the set of indices to be searched, $\mathrm{dom}(v)$, can be partitioned into the odd and even indices of v, namely $evens(v)$ and $odds(v)$, respectively, which can be searched in parallel.

$$notp(v, odds(v), min(ot', et')) \wedge notp(v, evens(v), min(ot', et')) \Rrightarrow notp(v, \mathrm{dom}(v), min(ot', et'))$$

The next step is the epitome of rely-guarantee refinement: splitting the specification command.

$\sqsubseteq$ by Law 94 (introduce-parallel-spec-weakened-relies)
$ot, et : \textbf{guar}\, ot' \leq ot \wedge et' = et \bullet \textbf{rely}\, et' \leq et \wedge \mathrm{id}(\{ot, v\}) \bullet$
$\quad \left[notp(v, odds(v), min(ot', et'))\right]$ $\lhd$
$\|$
$ot, et : \textbf{guar}\, et' \leq et \wedge ot' = ot \bullet \textbf{rely}\, ot' \leq ot \wedge \mathrm{id}(\{et, v\}) \bullet$
$\quad \left[notp(v, evens(v), min(ot', et'))\right]$

The above is all in the context of the guarantee invariant given above. As with Sect. 3.10, the guarantee invariant will eventually need to be discharged for each atomic step but it is only possible to do that when the final code has been developed. However, during the development one needs to be aware of this requirement to avoid making design decisions that result in code that is inconsistent with the guarantee invariant.

9.4 Refining the branches to code

For the first branch of the parallel, the guarantee $et' = et$ is by Definition 58 (frame) equivalent to removing *et* from the frame of the branch.

$= \textbf{guar}\, ot' \leq ot \bullet$
$\quad ot : \textbf{rely}\, et' \leq et \wedge \mathrm{id}(\{ot, v\}) \bullet \left[notp(v, odds(v), min(ot', et'))\right]$ $\lhd$

The body of this can be refined to sequential code in a manner similar to that used in Sect. 3.10, however, because the specification refers to et' it is subject to interference from the parallel (evens) process which may update *et*. That interference is however bounded by the rely condition which assumes the parallel process only ever decreases *et*.

$\sqsubseteq$ by Law 106 (variable-rely-guarantee) for *oc*
$\textbf{var}\, oc \bullet$
$\quad oc, ot : \textbf{rely}\, et' \leq et \wedge \mathrm{id}(\{oc, ot, v\}) \bullet \left[notp(v, odds(v), min(ot', et'))\right]$ $\lhd$

As in Sect. 3.10, a loop invariant is introduced as a (stronger) guarantee invariant

$notp(v, odds(v), oc) \wedge bnd(oc, v)$

where the bounding conditions on oc are not quite the same as earlier because oc only takes on odd values.

$$bnd(oc, v) \mathrel{\widehat{=}} 1 \leq oc \leq len(v) + 2$$

This invariant is established by setting oc to one. The guarantee invariant combined with the post-condition $oc' \geq min(ot', et')$ implies the postcondition of the above specification. The postcondition $oc' \geq min(ot', et')$ uses "$\geq$" rather than "$=$" because the parallel process may decrease et. The above can be refined using Law 98 (rely-sequential), Law 95 (trade-rely-guarantee-invariant) and Law 102 (assignment-rely-guarantee) as follows.

```
⊑ $oc := 1;$
  $\textbf{guar-inv}\, notp(v, odds(v), oc) \wedge bnd(oc, v) \bullet$
      $oc, ot : \textbf{rely}\, et' \leq et \wedge \mathrm{id}(\{oc, ot, v\}) \bullet \lceil oc' \geq min(ot', et') \rceil$        $\lhd$
```

A while loop is introduced using Law 112 (rely-loop). Although the loop guard is $oc < ot \wedge oc < et$, only the first conjunct of this is preserved by the rely condition because et may be decreased. Hence the boolean expression b_0 for this application of the law is $oc < ot$. However, the loop termination condition $oc \geq ot \vee oc \geq et$ is preserved by the rely condition as decreasing et will not falsify it. Hence b_1 is $oc \geq ot \vee oc \geq et$, which ensures $oc \geq min(ot, et)$ as required. For loop termination a well founded relation reducing $ot - oc$ is used.

```
⊑   by Law 112 (rely-loop)
  $\textbf{while}\, oc < ot \wedge oc < et\, \textbf{do}$
      $oc, ot : \textbf{rely}\, et' \leq et \wedge \mathrm{id}(\{oc, ot, v\}) \bullet \lceil oc < ot,\ 0 \leq ot' - oc' < ot - oc \rceil$        $\lhd$
```

The specification of the loop body only involves variables which are stable under interference.

```
⊑   by Law 77 (weaken-rely)
  $oc, ot : \textbf{rely}\, \mathrm{id}(\{oc, ot, v\}) \bullet \lceil oc < ot,\ 0 \leq ot' - oc' < ot - oc \rceil$        $\lhd$
```

At this stage one could use Law 101 (rely-uses) to introduce a "uses" clause and allow a sequential refinement to be used; we follow an alternative path in order to illustrate other laws. The refinement is now similar to that used in Sect. 3.10 but uses Law 109 (rely-conditional).

```
⊑ $\textbf{if}\, p(v(oc))\ \textbf{then}\ oc, ot : \textbf{rely}\, \mathrm{id}(\{oc, ot, v\}) \bullet \lceil p(v(oc)) \wedge oc < ot,\ 0 \leq ot' - oc' < ot - oc \rceil$
                  $\textbf{else}\ oc, ot : \textbf{rely}\, \mathrm{id}(\{oc, ot, v\}) \bullet \lceil \neg p(v(oc)) \wedge oc < ot,\ 0 \leq ot' - oc' < ot - oc \rceil$
```

Finally, Law 102 (assignment-rely-guarantee) can be applied to each of the branches. Each assignment ensures the guarantee invariant $(min(ot, et) = len(v) + 1 \vee satp(v, min(ot, et))) \wedge notp(v, odds(v), oc) \wedge bnd(oc, v)$ is maintained.

```
⊑ $\textbf{if}\, p(v(oc))\ \textbf{then}\ ot := oc\ \textbf{else}\ oc := oc + 2$
```

9.5 Collected code

The development of the "evens" branch of the parallel composition follows the same pattern as that of the "odds" branch given above. The collected code follows.

```
$\textbf{var}\ ot, et \bullet$
$ot := len(v) + 1\ ;$
$et := len(v) + 1\ ;$
(  $\textbf{var}\ oc \bullet$                                  ||  $\textbf{var}\ ec \bullet$                                  )
   $oc := 1\ ;$                                       ||  $ec := 2\ ;$
   $\textbf{while}\, oc < ot \wedge oc < et\, \textbf{do}$                ||  $\textbf{while}\, ec < ot \wedge ec < et\, \textbf{do}$                ) ;
       $\textbf{if}\, p(v(oc))\ \textbf{then}\ ot := oc$                 ||      $\textbf{if}\, p(v(ec))\ \textbf{then}\ et := ec$
                        $\textbf{else}\ oc := oc + 2$     ||                       $\textbf{else}\ ec := ec + 2$   )
$t := min(ot, et)$
```

The two branches (*odds*/*evens*) step through their respective subsets of the indices of v looking for the first element that satisfies p. The efficiency gain over a sequential implementation comes from allowing one of the processes to exit its loop early if the other has found an index i such that $p(v(i))$ that is lower than the remaining indexes that the first process has yet to consider. The extra complications for reasoning about this interprocess communication manifests itself particularly in the steps that introduce concurrency and the while loop because the interference affects variables mentioned in the test of the loop.

This implementation is guaranteed to satisfy the original specification due to its use at every step of the refinement laws. In many ways, this mirrors the development in (Coleman and Jones 2007). In particular, the use of Law 94 (introduce-parallel-spec-weakened-relies) in Section 9.3 mirrors the main thrust of "traditional" rely/guarantee thinking. What is novel in the new development is both the use of a guarantee invariant and the fact that there are rules for every construct used. Moreover, because all of the results are derived from a small number of basic lemmas, it is possible to add new styles of development without needing to go back to the semantics.

10 Conclusions

Summary

The idea of adding state assertions to imperative programs is crucial to the ability to decompose proofs about such program texts. The most influential source of this idea is due to Floyd (1967) although it is interesting to note that pioneers such as Turing and von Neumann recognised that something of the sort would aid reasoning (Jones 2003a). The key contribution of Hoare (1969) was to change the viewpoint away from program annotations to a system of judgements about "Hoare triples". One huge advantage of this viewpoint was that it offered a notion of compositional development. This is not the place to attempt a complete history but it is important to note that the refinement calculus (Back 1981; Morris 1987; Morgan 1988; Morgan 1994; Back and von Wright 1998) brings together the strands of development for sequential programs into an elegant calculus in which algebraic properties are clear. Other key developments include the idea of data refinement (or reification) and the importance of data type invariants.

There are several extra challenges when one moves from confronting sequential programs to their concurrent counterparts. Here again, this is not the place for a complete catalogue of the issues — and certainly not a history. The aspect of parallelism that goes under the heading of data races can be divided into avoiding such races and reasoning about those that are unavoidable. Conventional wisdom indicates that concurrent separation logic (O'Hearn, Yang, and Reynolds 2009) is useful for the former way of reasoning and that rely-guarantee thinking is useful for the latter.

The current paper shows how the sort of explicit reasoning about interference that underlies rely-guarantee thinking can be recast into a refinement calculus mould. It transpires that there is a very good fit of basic objectives. This includes the simple observation that the refinement calculus also embraced relations (rather than single state predicates) as the cornerstone of specifications; more important is the shared acknowledgement that compositional reasoning is a necessity if a method is to scale up to large problems.

Rather than treat a specification of a program as a four-tuple of pre, rely, guarantee and postcondition, the approach taken here has been to consider initially guarantees and relies separately and, rather than just apply them to a pre-post specification, allow them to be applied more generally to commands. The guarantee command ($\mathbf{guar}\, g \bullet c$) constrains the behaviour of c so that only atomic program steps that respect g are permitted. It generalises nicely to being applied to an arbitrary command. Refining a guarantee command can be decomposed into refining the body of the command, distributing the guarantee into the components of the refinement and then checking that each atomic step maintains the guarantee. Alternatively, in order to ensure that the guarantee is not broken, one can interleave refinement steps with checking that the guarantee is preserved. The choice of strategies is up to the developer.

The guarantee command also provides a neat way to define a frame for a command in a manner suitable to handle concurrency. Motivation for the guarantee construct was drawn from the invariant construct of Morgan and Vickers (1994) and its behaviour in restricting the possible atomic steps mirrors the restrictions introduced by the invariant command on possible states. The guarantee construct is also related to a form of enforced property used in action system refinement in which every action of a system is constrained to satisfy a relation (Dongol and Hayes 2010). The guarantee construct can be thought of as providing a context in which its body is refined. Nickson and Hayes (1997) have investigated tool support for contexts such as preconditions and the Morgan-Vickers invariant, and it is clear that the guarantee context could be treated similarly in tool support.

Invariants play an important role in reasoning about "while" loops because if a single iteration of a loop maintains an invariant, any finite number of iterations of the loop will also maintain the invariant. In a similar vein, if every atomic step of a computation maintains an invariant, the whole computation will also maintain the invariant. This motivates the introduction of the guarantee invariant construct (Sect. 3.9). As illustrated in the developments of *findp* as both sequential (Sect. 3.10) and concurrent (Sect. 9) programs, one can make use of guarantee invariants to ensure that every atomic step of a computation maintains the invariant, and hence the whole computation (including any loops within it) also maintains the invariant. The negative is that one must ensure every atomic step maintains the invariant, although in many cases this is trivial if no variables within the invariant are modified by the step. Guarantee invariants also play a role in the context of concurrency because if every atomic step of a process c maintains an invariant, then a concurrent process d can rely on the invariant being maintained by any interference generated by c.

The thesis by Jürgen Dingel (2000) and the more accessible (Dingel 2002) offer an approach towards a "refinement calculus" view of rely/guarantee reasoning. There are however clear differences about what constitutes such a view in Dingel's writings and in the current paper. Dingel (2000, 2002) has also produced a refinement calculus that supports rely-guarantee style reasoning. One difference between his approach and that presented here is that he used a four-tuple of pre, rely, guarantee and post conditions rather than the separate rely and guarantee constructs used here; in fact, his writings do not follow Jones' original relational view because he, for example, has post conditions that are sets of predicates of single states — on this point he follows Stirling (1986). To the current authors, the key reason for a refinement calculus view is to get away from any fixed packaging of the assertions that comprise a specification and to view guarantee/rely commands in a way that opens up a relational view of the constructs. Given a basic set of commands and their basic laws, it is possible to derive more specific laws that often show nice algebraic properties. To give just one specific example, Law 92 (trade-rely-guarantee) presents an intuitive rule for what often appear to be arbitrary manipulations in earlier papers.

The rely command $(\textbf{rely}\, r \bullet [p,\ q])$ guarantees to implement $[p,\ q]$ under interference bounded by the relation r. The generalisation of a rely command to allow a body containing any arbitrary command c is complicated by the need to make the rely context z of c explicit. The explicit context is required because the termination set of $(\textbf{rely}\, r \bullet c_z)$ in context $z \vee r$ is $stps(c, z)$, which depends on the context z.

In order for $(\textbf{rely}\, r \bullet c)$ to be feasible c must allow inference bounded by r, and this usually requires c to be in the form of a pre-post specification (or a composition of such specifications) rather than more basic commands like assignments because the latter are too restrictive to be feasible when included in a rely.

The advantage of treating the guarantee and rely constructs separately is that we have been able to develop sets of laws specific to each. This has the advantage of providing a better understanding of the role of guarantees and relies. In particular the main defining property of the rely construct

$$\{stps(c,z)\}c \sqsubseteq (\textbf{rely}\, r \bullet c_z) \parallel \langle r \vee \mathrm{id} \rangle^*$$

given in Section 4.3 brings out the essence of the role of the rely condition.

Of course, the main point of introducing rely and guarantee constructs is to allow them to be used to express the bounds on interference in parallel compositions. To this end, in Section 5, we have developed refinement laws for parallel composition that have been proved using the more fundamental laws for guarantee and rely constructs along with basic laws about conjoined specifications/commands.

Our theory of rely and guarantee commands is built on a more basic theory of atomic steps. The guarantee command is defined in terms of a strict conjunction with an iteration of atomic steps, each of which satisfies the guarantee. The rely command again makes use of atomic steps but this time to represent that the interference is bounded by the rely condition. This shows the basic relationship required for the parallel composition law in which the atomic steps of one process must guarantee the assumed interference of all other processes. All the refinement laws have been proven in terms of these more basic theories, and from our experience it is clear that it is much easier to develop new refinement laws using the theory than proving new laws directly from the semantics.

Further work

In the sequential refinement calculus because one is only concerned with the end-to-end behaviour from the initial state to the final state, any program can be reduced to an equivalent specification statement. However for concurrent programs the intermediate behaviour of a process can be as important as its overall effect. The rely-guarantee approach augments pre-post specifications with rely and guarantee relations which allow a pre-rely-guarantee-post specification to express both the assumption it makes of steps by its environment and the guarantee about the steps its takes. As rely (guarantee) conditions abstract all interference (program) steps, pre-rely-guarantee-post specifications are not rich enough to be able to express precisely the behaviour of all processes.[12] One challenge is to increase the expressive power of rely and guarantee conditions to allow a more precise specification of a greater range of processes, while retaining the elegance of the rely-guarantee approach.

As should be clear from the preceding material, the reformulation of rely/guarantee thinking in a refinement calculus mould has suggested new notation and laws. A nice example is the shift from the heavy VDM-like keyword framing in which read/write access is declared to the compact prefix listing of write variables. There is however much scope for further research and progress. In the same area as framing, it would be useful to have a direct notation that showed that a "variable" essentially became a constant throughout a portion of code; in the same vein, Jones and Pierce (2011) use **owns**. Furthermore, the hope for deep links to Separation Logic could be advanced by laws that directly indicate separate access (cf. ot, et in Sect. 9.3).

There is also a clear case for reconsidering data reification in the new framework. The concept of "possible values" was introduced by Jones and Pierce (2011) and linked to non-deterministic states in (Hayes, Burns, Dongol, and Jones 2013) — re-examining the idea in the new framework might also be enlightening.

Acknowledgements

The authors would like to thank our colleagues Joey Coleman, Larissa Meinicke, Jeff Sanders and (anonymous) referees for feedback on our earlier work.

This research was supported by Australian Research Council Grants DP130102901 and LP0989643 and the EPSRC (UK) TrAmS-2 Platform grant and Taming Concurrency responsive mode grant.

[12]Technically this can be overcome by adding some form of program counter to each process and labels to each step of the program but such an approach destroys the elegance of the rely/guarantee abstractions.

APPENDIX

A A rely-able semantics

A.1 Basic definitions

Memory is represented by a *state* that maps variables to their values, type $\Sigma \mathrel{\widehat{=}} \mathit{Var} \rightarrow \mathit{Val}$. We use $\sigma, \sigma', \sigma_i$ for elements of Σ. A sequence t of type T, $t \in T^\omega$, may be either finite ($t \in T^*$) or infinite ($t \in T^\infty$). The domain of a finite sequence of length n is the set of natural numbers $0..n-1$, and the domain of an infinite sequence t is the entire set of natural numbers ($\mathrm{dom}(t) = \mathbb{N}$).

A.2 Interpretation of programs

A semantics for sequential programs is classically given in terms of a binary relation between the pre-state and post-state (perhaps augmented by a termination set as in VDM (Jones 1987)). However to handle concurrency and the possibility of interference, we need to divide a program's behaviour – its *trace*– into its atomic steps. Moreover, to conveniently represent the behaviour of a program c in the presence of interference from the environment, we include the steps of the environment within the traces of c, distinguishing program steps, $\pi(\sigma, \sigma')$, from environment steps, $\epsilon(\sigma, \sigma')$, where each step has an associated pair of states, (σ, σ'), representing the pre- and post-states of the step (de Boer, Hannemann, and de Roever 1999).[13] A step $\alpha \in \mathcal{L}$ may label a transition arrow, and its syntax follows.

$$\alpha ::= \pi(\sigma, \sigma') \mid \epsilon(\sigma, \sigma') \tag{A.91}$$

Recall a *trace* is a *consistent* (12) sequence of steps. An execution of a command c is of the form

$$c \xrightarrow{\pi(\sigma_0,\sigma_1)} c' \xrightarrow{\epsilon(\sigma_1,\sigma_2)} c'' \xrightarrow{\pi(\sigma_2,\sigma_3)} c''' \xrightarrow{\pi(\sigma_3,\sigma_4)} \ldots$$

where $c, c', \ldots$ is the successive evolution of c as it is executed, and each transition represents an atomic step from state σ_i to state σ_{i+1}. The trace generated by the above execution is the sequence $\pi(\sigma_0, \sigma_1), \epsilon(\sigma_1, \sigma_2), \pi(\sigma_2, \sigma_3), \pi(\sigma_3, \sigma_4), \ldots$.

An operational semantics defining the above transition relation via a set of inference rules is used below to define the behaviour of the basic commands in the language; the transition relation is the smallest relation induced by those rules. The style of placing pairs of states on the transition arrows follows that of Modular Structural Operational Semantics (MSOS) (Mosses 2004a; Mosses 2004b), which has some advantages over the seminal Plotkin-style operational semantics (Plotkin 2004) in which the states form part of the configuration rather than the label: $\langle c_0, \sigma_0 \rangle \rightarrow \langle c_1, \sigma_1 \rangle$. Of particular importance for the semantics presented here is that by collecting the actions in the trace we are able to more succinctly describe properties of the traces and hence interference.

The single-step transition relation induces a (multi-step) trace relation in the obvious way. For a finite trace of steps, t, the relation is written $c \overset{t}{\Rightarrow} c'$ and if t is infinite, it is written $c \overset{t}{\Rightarrow} \infty$. The meaning of a command c, $[\![c]\!]$, is the collection of all (finite and infinite) complete traces it may generate that either terminate or are non-terminating. A command c that can terminate immediately is indicated by $(c)_\checkmark$. The predicate $(c)_\checkmark$ is defined in Appendix A.6.

$$[\![c]\!] \quad \widehat{=} \quad \{t \in \mathit{Trace} \mid (\exists c' \bullet c \overset{t}{\Rightarrow} c' \land (c')_\checkmark) \lor c \overset{t}{\Rightarrow} \infty\}$$

We start with the semantics of expressions (Appendix A.3) and commands that can be described by a small-step semantics (Appendix A.4). We then move on to commands that require a big-step semantics (A.5). We prefer the operational semantics style to direct denotational semantics as the former is generally held to be more readable (Jones 2003b), although the latter is certainly possible (as demonstrated by Brookes (2007)).

[13]The use of explicit environment steps goes back to Peter Aczel's use of direct (program) and interference (environment) steps in traces to give a semantics for rely-guarantee inference rules (Aczel 1983).

A.3 Operational semantics of expression evaluation

The evaluation of a variable x to a value v generates a single program step (label) which is allowed only when x has the value v in state σ; the step does not change the value of any variables.

$$\frac{\sigma(x) = v \quad v \in Val}{x \xrightarrow{\pi(\sigma,\sigma)} v}$$

To evaluate a binary expression $e_1 \oplus e_2$ the operands may be evaluated to values in any order.[14] The final value is found by applying the underlying mathematical operator to the values, which we write $eval(\oplus, (v_1, v_2))$. The function $eval$ may return the undefined value $\bot$ if the operator is not defined for those values (e.g., division by zero). Unary operators are evaluated similarly.

$$\frac{e_1 \xrightarrow{\alpha} e_1'}{e_1 \oplus e_2 \xrightarrow{\alpha} e_1' \oplus e_2} \qquad \frac{e_2 \xrightarrow{\alpha} e_2'}{e_1 \oplus e_2 \xrightarrow{\alpha} e_1 \oplus e_2'} \qquad \frac{v_1, v_2 \in Val}{v_1 \oplus v_2 \xrightarrow{\pi(\sigma,\sigma)} eval(\oplus, (v_1, v_2))}$$

$$\frac{e \xrightarrow{\alpha} e'}{\ominus e \xrightarrow{\alpha} \ominus e'} \qquad \frac{v \in Val}{\ominus v \xrightarrow{\pi(\sigma,\sigma)} eval(\ominus, v)}$$

In addition, the evaluation of an expression may be interrupted by an environment step at any time, as given by the following rule.

$$e \xrightarrow{\epsilon(\sigma,\sigma')} e \tag{A.92}$$

The interference step does not change the expression, but may change the state, possibly affecting subsequent evaluations of variables. Note that this rule allows any finite or infinite interference in the trace generated by the evaluation.

A.4 Operational semantics of primitive code commands

The most basic terminated command is **nil**. A terminated command may allow further environment steps, including an infinite number to handle the case that it is in parallel with a non-terminating loop.

$$(\mathbf{nil})_{\checkmark} \qquad \frac{(c)_{\checkmark}}{c \xrightarrow{\epsilon(\sigma,\sigma')} \mathbf{nil}} \tag{A.93}$$

The interference may be infinite, that is, unfair. In our model, unfairness results in a trace that ends with an infinite number of environment steps; the definition of refinement, $c \sqsubseteq d$, requires that d preserves behaviours of c under unfair interference, but this does not affect the set of commands related by $\sqsubseteq$, because the behaviours of d that occur without interruption must also be behaviours of c. Note the distinction between **nil** and **magic**: the former allows traces containing environment steps only, while the latter allows no traces whatsoever.

A command c that has aborted may partake in any further behaviour.

$$\mathbf{abort} \xrightarrow{\alpha} c' \tag{A.94}$$

Note that command c' on the right may be any command, including **nil** or **abort**.

[14]Operators like "conditional and" (which only evaluates its second operand if its first operand evaluates to true) are not covered by these rules and would need separate specific rules.

For a single state predicate p and relation q, the atomic step $\langle p, q\rangle$ can do a q program step if p holds or can abort if p does not hold. The execution of the step may be preceded by any number of environment steps.

$$\frac{\sigma \in p \wedge (\sigma, \sigma') \in q}{\langle p, q\rangle \xrightarrow{\pi(\sigma,\sigma')} \mathbf{nil}} \qquad \frac{\sigma \notin p}{\langle p, q\rangle \xrightarrow{\pi(\sigma,\sigma')} \mathbf{abort}} \qquad \langle p, q\rangle \xrightarrow{\epsilon(\sigma,\sigma')} \langle p, q\rangle \tag{A.95}$$

Note that if the precondition p does not hold, there is no constraint on the final state, and that $\langle \mathsf{true}, \mathsf{false}\rangle$ has no program transitions: all its traces are infinite in length and contain only environment steps.

A preconditioned command $\{p\}c$ requires the precondition p to hold on the first step of c, whether that be a program or environment step.

$$\frac{c \xrightarrow{\alpha} c' \quad pre(\alpha) \in p}{\{p\}c \xrightarrow{\alpha} c'} \qquad \frac{pre(\alpha) \notin p}{\{p\}c \xrightarrow{\alpha} \mathbf{abort}} \tag{A.96}$$

Nondeterministic choice between a set of commands C, $(\bigsqcap C)$, can behave as any command within C.

$$\frac{c \in C \quad c \xrightarrow{\alpha} c'}{(\bigsqcap C) \xrightarrow{\alpha} c'} \tag{A.97}$$

A sequential composition, $(c_1 \ ; \ c_2)$, is defined to execute c_1 until it terminates, after which c_2 may begin. If c_1 aborts, the sequential composition aborts.

$$\frac{c_1 \xrightarrow{\alpha} c_1'}{c_1 \ ; c_2 \xrightarrow{\alpha} c_1' \ ; c_2} \qquad \frac{(c_1)\checkmark \quad c_2 \xrightarrow{\alpha} c_2'}{c_1 \ ; c_2 \xrightarrow{\alpha} c_2'} \qquad \frac{c_1 \xrightarrow{\alpha} \mathbf{abort}}{c_1 \ ; c_2 \xrightarrow{\alpha} \mathbf{abort}} \tag{A.98}$$

Note that sequential composition implicitly fails to terminate if c_1 fails to terminate, i.e., c_2 is never executed.

A strict conjunction of two commands $c \Cap d$ behaves in a manner consistent with both c and d. If either can abort, so can their conjunction.

$$\frac{c_1 \xrightarrow{\alpha} c_1' \quad c_2 \xrightarrow{\alpha} c_2'}{c_1 \Cap c_2 \xrightarrow{\alpha} c_1' \Cap c_2'} \qquad \frac{c_1 \xrightarrow{\alpha} \mathbf{abort}}{c_1 \Cap c_2 \xrightarrow{\alpha} \mathbf{abort}} \qquad \frac{c_2 \xrightarrow{\alpha} \mathbf{abort}}{c_1 \Cap c_2 \xrightarrow{\alpha} \mathbf{abort}} \tag{A.99}$$

A parallel composition $c \parallel d$ matches a program step of c with an environment step of d (or vice versa) to give a program step of the composition. It can also match environment steps of both commands to give an environment step of their composition. If either command can abort on a step matched by the other, the parallel composition can abort. To define parallel composition we define a relation match between a pair of steps (representing the transitions of the two components) and a single step (representing the the transition of the parallel composition) as follows.

$$(\pi(\sigma, \sigma'), \epsilon(\sigma, \sigma')) \ \ \mathsf{match} \ \ \pi(\sigma, \sigma') \tag{A.100}$$

$$(\epsilon(\sigma, \sigma'), \pi(\sigma, \sigma')) \ \ \mathsf{match} \ \ \pi(\sigma, \sigma') \tag{A.101}$$

$$(\epsilon(\sigma, \sigma'), \epsilon(\sigma, \sigma')) \ \ \mathsf{match} \ \ \epsilon(\sigma, \sigma') \tag{A.102}$$

The rule for the normal case allows any combinations that match while if either of the commands can abort on matching transitions, the whole can.

$$\frac{c_1 \xrightarrow{\alpha_1} c_1' \qquad c_2 \xrightarrow{\alpha_2} c_2' \qquad (\alpha_1, \alpha_2) \ \mathsf{match} \ \alpha}{(c_1 \parallel c_2) \xrightarrow{\alpha} (c_1' \parallel c_2')} \tag{A.103}$$

$$\frac{c_1 \xrightarrow{\alpha_1} \textbf{abort} \qquad c_2 \xrightarrow{\alpha_2} c_2' \qquad (\alpha_1, \alpha_2) \text{ match } \alpha}{(c_1 \parallel c_2) \xrightarrow{\alpha} \textbf{abort}} \tag{A.104}$$

$$\frac{c_1 \xrightarrow{\alpha_1} c_1' \qquad c_2 \xrightarrow{\alpha_2} \textbf{abort} \qquad (\alpha_1, \alpha_2) \text{ match } \alpha}{(c_1 \parallel c_2) \xrightarrow{\alpha} \textbf{abort}} \tag{A.105}$$

A local state command ($\textbf{state}\ y \mapsto v \bullet c$) limits the scope of y. Modifications to y are kept locally (and have no effect on any (global) declarations of y) and the environment is explicitly prevented from modifying the local variable y (but may modify other non-local variables called y). The state $\sigma[y \mapsto v]$ is the state σ with the value at y updated to v.

$$\frac{c \xrightarrow{\pi(\sigma[y \mapsto v], \sigma'[y \mapsto v'])} c' \quad \sigma'(y) = \sigma(y)}{(\textbf{state}\ y \mapsto v \bullet c) \xrightarrow{\pi(\sigma, \sigma')} (\textbf{state}\ y \mapsto v' \bullet c')} \tag{A.106}$$

$$\frac{c \xrightarrow{\epsilon(\sigma[y \mapsto v], \sigma'[y \mapsto v])} c'}{(\textbf{state}\ y \mapsto v \bullet c) \xrightarrow{\epsilon(\sigma, \sigma')} (\textbf{state}\ y \mapsto v \bullet c')} \tag{A.107}$$

$$\frac{c \xrightarrow{\pi(\sigma[y \mapsto v], \sigma'[y \mapsto v'])} \textbf{abort}}{(\textbf{state}\ y \mapsto v \bullet c) \xrightarrow{\pi(\sigma, \sigma')} \textbf{abort}} \tag{A.108}$$

$$\frac{c \xrightarrow{\epsilon(\sigma[y \mapsto v], \sigma'[y \mapsto v])} \textbf{abort}}{(\textbf{state}\ y \mapsto v \bullet c) \xrightarrow{\epsilon(\sigma, \sigma')} \textbf{abort}} \tag{A.109}$$

Rule (A.106) states that if c transitions with a program step in which the global pre-post values for y are overwritten by the local values, then the new post-state local value for y becomes v', but to an external observer the global value of y is unchanged. The latter is enforced by the premise of the rule.

Rule (A.107) states that environment steps *within* the scope of the declaration of y may not modify y, however environment steps *outside* the scope of the local y may modify some global y. Thus, the local declaration of y protects it from interference.

Promoted program steps of the body of a local state construct cannot modify any non-local variable y. To remove this restriction in the case in which the body of the command can abort in environment $\mathrm{id}(y)$, Rule (A.108) and Rule (A.109) promote a local state with a body that aborts in $\mathrm{id}(y)$ to **abort**.

A program step $\pi(\sigma, \sigma')$ of ($\textbf{uses}\, X \bullet c$) is permitted provided c may take essentially the same program step for every pair of states that is equal to (σ, σ') in X. Let $\sigma \overset{X}{=} \sigma'$ abbreviate $(\sigma, \sigma') \in \mathrm{id}(X)$, and similarly lifted to pairs of states.

$$\frac{c \xrightarrow{\pi(\sigma, \sigma')} c' \quad \sigma \overset{\overline{X}}{=} \sigma' \quad \forall \sigma_1, \sigma_1' \bullet (\sigma_1, \sigma_1') \overset{X}{=} (\sigma, \sigma') \wedge \sigma_1 \overset{\overline{X}}{=} \sigma_1' \Rightarrow c \xrightarrow{\pi(\sigma_1, \sigma_1')} c'}{(\textbf{uses}\, X \bullet c) \xrightarrow{\pi(\sigma, \sigma')} (\textbf{uses}\, X \bullet c')} \tag{A.110}$$

$$\frac{c \xrightarrow{\epsilon(\sigma, \sigma')} c'}{(\textbf{uses}\, X \bullet c) \xrightarrow{\epsilon(\sigma, \sigma')} (\textbf{uses}\, X \bullet c')} \tag{A.111}$$

$$\frac{c \xrightarrow{\alpha} \textbf{abort}}{(\textbf{uses}\, X \bullet c) \xrightarrow{\alpha} \textbf{abort}} \tag{A.112}$$

Environment steps for the body of a uses command are simply promoted and if the body of a "uses" command can abort, the "uses" command aborts.

A.5 Semantics of tests and specifications

We now turn our attention to commands that cannot be adequately described using a small-step semantics. The test command $[[b]]$ evaluates its boolean expression b. Only terminating traces that evaluate to true will survive; evaluations to false are eliminated. If the evaluation of b results in undefined, the test aborts, and a non-terminating expression evaluation results in non-terminated test. Variations in the evaluation strategy used in an implementation may lead to evaluation traces that differ only in stuttering steps. For example, stuttering steps allow equivalences like $[[a \wedge b]] = [[a]] \parallel [[b]]$ and refinements like $[[b \wedge b]] \sqsubseteq [[b]]$. The semantics of the test command allows traces that are equivalent to the trace defined by the expression evaluation modulo finite stuttering. The notation $t_0 \overset{st}{=} t_1$ states that t_0 and t_1 are identical modulo finite stuttering of program steps.

$$\frac{b \xRightarrow{t_0} \mathsf{true} \quad t_0 \overset{st}{=} t_1}{[[b]] \xRightarrow{t_1} \mathbf{nil}} \qquad \frac{b \xRightarrow{t_0} \bot \quad t_0 \overset{st}{=} t_1}{[[b]] \xRightarrow{t_1} \mathbf{abort}} \qquad \frac{b \xRightarrow{t_0}\infty \quad t_0 \overset{st}{=} t_1}{[[b]] \xRightarrow{t_1}\infty} \tag{A.113}$$

If $b \overset{t}{\Rightarrow} \mathsf{false}$, that trace is not promoted to a trace of $[[b]]$. The exclusion of evaluations to false is required for the definition of the conditional command, in which there is a nondeterministic choice between a branch with test $[[b]]$ and another with $[[\neg b]]$. Whichever branch evaluates to false must "lose", which is modelled by a lack of traces, or magic.

A specification command $\lceil q \rceil$ can perform any finite sequence of program steps that end-to-end satisfies q, provided all environment steps are stuttering steps. However, if the environment changes the state, then $\lceil q \rceil$ aborts. Recall from (14) that $env(t) \subseteq \mathrm{id}$ holds if every environment step in t satisfies id and that $pre(t)$ abbreviates $pre(t(0))$ and $post(t)$ abbreviates $post(t(\#t-1))$.

$$\begin{gathered}\frac{env(t) \subseteq \mathrm{id} \quad (pre(t), post(t)) \in q \quad t \in \mathcal{L}^*}{\lceil q \rceil \xRightarrow{t} \mathbf{nil}} \\ \frac{env(t) \not\subseteq \mathrm{id} \quad t \in \mathcal{L}^*}{\lceil q \rceil \xRightarrow{t} \mathbf{abort}} \\ \frac{env(t) \subseteq \mathrm{id} \quad interrupted(t) \quad t \in \mathcal{L}^\infty}{\lceil q \rceil \overset{t}{\Rightarrow}\infty}\end{gathered} \tag{A.114}$$

The premises of the first rule require the first and final states in t to satisfy q. Any behaviour may occur in between as long as q is established on termination. The final two rules state that a specification may fail to meet its postcondition if the environment changes any variable or if the environment unfairly interrupts execution. The latter case uses the predicate $interrupted(t)$, which holds if t ends with an infinite sequence of environment steps.

$$interrupted(t) \Leftrightarrow (t \in \mathcal{L}^\infty \wedge finite\_pgm\_steps(t)) \tag{A.115}$$

The command $(\mathbf{env}\, r \bullet c)$ behaves as c in an environment that respects r but aborts otherwise. It is defined directly in terms of its set of traces.

$$[\![\mathbf{env}\, r \bullet c]\!] \;\widehat{=}\; \{t \in Trace \mid env(t) \subseteq r \vee \mathrm{id} \Rightarrow t \in [\![c]\!]\} \tag{A.116}$$

All other commands in the language are built out of these basic operators.

A.6 Propagating terminated behaviour

One method of reducing to terminated commands is to use explicit id steps, however we wish to reduce the amount of stuttering induced by the semantics so as to relate as many commands as possible by the refinement relation. Hence we inductively define the set of terminated commands, for use primarily in the sequential composition rule.

$$(\mathbf{nil})_{\checkmark} \quad \frac{(c)_{\checkmark}}{(\{p\}c)_{\checkmark}} \quad \frac{\forall c \in C \bullet (c)_{\checkmark}}{(\bigsqcap C)_{\checkmark}} \quad \frac{(c_0)_{\checkmark} \quad (c_1)_{\checkmark}}{(c_0 \Cap c_1)_{\checkmark}} \quad \frac{(c_0)_{\checkmark} \quad (c_1)_{\checkmark}}{(c_0 \;;\; c_1)_{\checkmark}}$$

$$\frac{(c_0)_{\checkmark} \quad (c_1)_{\checkmark}}{(c_0 \parallel c_1)_{\checkmark}} \quad \frac{(c)_{\checkmark}}{(\mathbf{uses}\, X \bullet c)_{\checkmark}} \quad \frac{(c)_{\checkmark}}{(\mathbf{state}\; y \mapsto v \bullet c)_{\checkmark}} \tag{A.117}$$

B Proofs of lemmas

In this section we prove soundness of some of the lemmas in the body of the paper.

Proof technique

Refinement is defined as reverse trace inclusion (Definition 2 (refinement)). As such it has elements of both sequential program refinement and notions such as (bi)simulation from the process algebra literature (Milner 1989). For the majority of the proofs of laws of the form $c \sqsubseteq d$ we enumerate all possible transitions $d \xrightarrow{\alpha} d'$, check that there exists a corresponding transition $c \xrightarrow{\alpha} c'$, requiring furthermore that d' is a refinement of c'.

Theorem 113 *The refinement $c \sqsubseteq_r d$ holds if, for all α and d' such that $d \xrightarrow{\alpha} d'$ and if α is of the form $\epsilon(\sigma, \sigma')$ then $(\sigma, \sigma') \in r \vee \mathrm{id}$, there exists a c' such that both*

$$c \xrightarrow{\alpha} c' \tag{B.118}$$
$$c' \sqsubseteq_r d' \tag{B.119}$$

Proof. It is straightforward by induction that any complete trace of d generated by the small-step operational semantics must be a trace of c under the above conditions. This is similar to the definition of *bisimulation* given by Milner (1989), but in only one direction (d is a *simulation* of c). □

Proofs using Theorem 113 proceed by first discharging B.118 by case-analysis on the possible transitions of d, of which there are typically two, and sometimes three: a program step, an environment step, and a step that ends in **abort**. Condition B.119 is usually trivial because the basic laws are defined so that c and d (and hence c' and d') are structurally similar.

Theorem 113 is applied when c and d are commands defined using the small-step operational semantics rules; for the remaining commands, defined using big-step operational semantics rules, in particular the specification command $\left[q\right]$, we instead justify refinements against complete traces.

For convenience, when proving a refinement of the form $c \sqsubseteq d$, we refer to c as the *source* and d as the *target*, and use the same terms when proving an equality $c = d$.

Lemma 5 (precondition-traces)

For any predicate p and command c,

$$[\![\{p\}c]\!] = \{t \in \mathit{Trace} \mid \mathit{pre}(t) \in p \Rightarrow t \in [\![c]\!]\} \tag{B.120}$$
$$\{p\}c \sqsubseteq_r d \Leftrightarrow (\forall t \in [\![d]\!]_r \bullet \mathit{pre}(t) \in p \Rightarrow t \in [\![c]\!]) \tag{B.121}$$

Proof. By Rule (A.96), if $\mathit{pre}(t) \notin p$ then $t \in [\![\{p\}c]\!]$, otherwise $t \in [\![\{p\}c]\!]$ if and only if $t \in [\![c]\!]$. Hence (B.120) holds. Property (B.121) follows directly from (B.120) and Definition 1 (refinement-in-context). □

Lemma 13 (nondeterminism-traces)

$$[\![\bigsqcap C]\!] \quad = \quad \bigcup_{c \in C} [\![c]\!]$$

Proof. By Rule (A.97) any trace of $\bigsqcap C$ is a trace of a command $c \in C$ and hence the set of traces of $\bigsqcap C$ is the union of the traces of all $c \in C$. If the set C is empty no behaviour is possible, i.e., as expected, the empty set of choices is equivalent to **magic**. □

Lemma 7 (parallel-precondition)

$$\{p\}(c \parallel d) = (\{p\}c) \parallel (\{p\}d)$$

Proof. Assume $c \parallel d \xrightarrow{\alpha} c' \parallel d'$, where α may be a program step of either c or d, or an environment step of both (by Rule (A.103)). For α to be a non-aborting step of the source, then $pre(\alpha) \in p$. This is step is matched exactly by the target, since α is a step of either or both of c and d. In the aborting case of Rule (A.96), the steps on each side are straightforwardly matched. □

Lemma 8 (refine-specification)

$$(\big[p,\ q\big] \sqsubseteq c) \quad \Leftrightarrow \quad (\big[p,\ q\big] \sqsubseteq_{\mathrm{id}} c)$$

Proof. By Lemma 5 (precondition-traces) $\{p\}\ \big[q\big] \sqsubseteq c$ if and only if

$$\forall t \in [\![c]\!] \bullet pre(t) \in p \Rightarrow t \in [\![\big[q\big]]\!]$$

If $t \in [\![c]\!]$ and $env(t) \not\subseteq \mathrm{id}$ then by Rule (A.114), $t \in [\![\big[q\big]]\!]$. The remaining case is when $env(t) \subseteq \mathrm{id}$ and requires

$$\forall t \in [\![c]\!]_{\mathrm{id}} \bullet pre(t) \in p \Rightarrow t \in [\![\big[q\big]]\!]$$

which by Lemma 5 (precondition-traces) is equivalent to $\{p\}\ \big[q\big] \sqsubseteq_{\mathrm{id}} c$. □

Lemma 11 (consequence)

Assuming $p_0 \Rrightarrow p_1$, and $p_0 \wedge q_1 \Rrightarrow q_0$,

$$\begin{array}{lcl} \langle p_0, q_0 \rangle & \sqsubseteq & \langle p_1, q_1 \rangle \\ \big[p_0,\ q_0\big] & \sqsubseteq & \big[p_1,\ q_1\big] \end{array}$$

Proof. From Theorem 113 and Rule (A.95), consider three cases.

- Case $\langle p_1, q_1 \rangle \xrightarrow{\epsilon(\sigma,\sigma')} \langle p_1, q_1 \rangle$. This is directly matched by a step of the source.
- Case $\langle p_1, q_1 \rangle \xrightarrow{\pi(\sigma,\sigma')}$ **nil**. It must be the case the $\sigma \in p_1$ and $(\sigma, \sigma') \in q_1$. In the case where $\sigma \in p_0$ also, because $p_0 \wedge q_1 \Rrightarrow q_0$, then $\langle p_0, q_0 \rangle \xrightarrow{\pi(\sigma,\sigma')}$ **nil** by the first case of Rule (A.95). If $\sigma \notin p_0$, then $\langle p_0, q_0 \rangle \xrightarrow{\pi(\sigma,\sigma')}$ **abort** . Since **abort** $\sqsubseteq$ **nil**, condition (B.119) holds.
- Case $\langle p_1, q_1 \rangle \xrightarrow{\pi(\sigma,\sigma')}$ **abort**. We may assume $\sigma \notin p_1$, and hence $\sigma \notin p_0$, thus the source also transitions to **abort**.

For specifications we need to show $\left[p_0,\ q_0\right] \sqsubseteq \left[p_1,\ q_1\right]$. By Lemma 8 (refine-specification) and Law 6 (precondition) as $p_0 \Rrightarrow p_1$ it is sufficient to show $\{p_0\}\left[q_0\right] \sqsubseteq_{\mathrm{id}} \left[q_1\right]$ and hence by Lemma 5 (precondition-traces) to show that

$$\forall t \in [\![\left[q_1\right]]\!]_{\mathrm{id}} \bullet pre(t) \in p_0 \Rightarrow t \in [\![\left[q_0\right]]\!] . \tag{B.122}$$

By Rule (A.114), $t \in [\![\left[q_1\right]]\!]_{\mathrm{id}}$ if and only if $interrupted(t)$ or t is finite and $(pre(t), post(t)) \in q_1$. If $interrupted(t)$ then by Rule (A.114) we also have $t \in [\![\left[q_0\right]]\!]$, otherwise if $pre(t) \in p_0$ then $t \in [\![\left[q_0\right]]\!]$ because $p_0 \wedge q_1 \Rrightarrow q_0$. □

Lemma 9 (specification-term)

$$stps(\left[p,\ q\right], r) \equiv \quad p, \qquad \text{if } r \Rrightarrow \mathrm{id} \tag{B.123}$$
$$stps(\left[p,\ q\right], r) \equiv \quad \mathsf{false}, \qquad \text{if } r \not\Rrightarrow \mathrm{id} \tag{B.124}$$

Proof. By Rule (A.114), every trace t of $\left[q\right]$ in environment id is finite (or interrupted) and hence $stps(\left[q\right], \mathrm{id}) \equiv \mathsf{true}$. If $r \not\Rrightarrow \mathrm{id}$, then in an environment satisfying r, by Rule (A.114), $\left[q\right]$ can abort and hence has a nonterminating trace and hence $stps(\left[q\right], r) \equiv \mathsf{false}$. For property (B.123), if $r \Rrightarrow \mathrm{id}$, by Law 6 (precondition) and as $r \vee \mathrm{id} \equiv \mathrm{id}$, we have

$$stps(\{p\}\left[q\right], r) \equiv p \wedge stps(\left[q\right], \mathrm{id}) \equiv p \wedge \mathsf{true} \equiv p$$

and for property (B.124), if $r \not\Rrightarrow \mathrm{id}$,

$$stps(\{p\}\left[q\right], r) \equiv p \wedge stps(\left[q\right], r) \equiv p \wedge \mathsf{false} \equiv \mathsf{false} .$$

□

Lemma 10 (make-atomic)

$$\left[p,\ q\right] \quad \sqsubseteq \quad \langle p, q\rangle$$

Proof. By Lemma 8 (refine-specification) is is sufficient to show $\{p\}\left[q\right] \sqsubseteq_{\mathrm{id}} \langle p, q\rangle$ and hence by Lemma 5 (precondition-traces) that

$$\forall t \in [\![\langle p, q\rangle]\!]_{\mathrm{id}} \bullet pre(t) \in p \Rightarrow t \in [\![\left[q\right]]\!]$$

By Rule (A.95) and Rule (A.93), any complete trace $t \in [\![\langle p, q\rangle]\!]_{\mathrm{id}}$ has one of the following forms: a) a finite number of id environment steps, followed by a single program step (satisfying p and q), followed by a finite number of further id environment steps; b) a finite number of id environment steps, followed by a single program step (satisfying p and q), followed by an infinite number of further id environment steps; c) a finite number of id environment steps, followed by a single program step that does not satisfy p, followed by any trace; or d) an infinite number of id environment steps.

In case a), because all environment steps of t satisfy id, $(pre(t), post(t)) \in q$ holds (due to the single program step) and hence t is a finite trace of $\left[q\right]$.

In case b), $interrupted(t)$ holds, and hence t is an interrupted trace of $\left[q\right]$.

In case c), because every environment step satisfies id, if $pre(t) \in p$ holds then p holds for the program step and hence there are no traces for which case c) applies.

Case d) is similar to case b).

□

Lemma 12 (sequential)

Assume $p_0 \wedge ((q_0 \wedge p_1') \fatsemi q_1) \Rrightarrow q$,

$$\left[p_0,\ q\right] \quad \sqsubseteq \quad \left[p_0,\ q_0 \wedge p_1'\right]\ ;\ \left[p_1,\ q_1\right]$$

Proof. By Law 6 (precondition) and Lemma 8 (refine-specification) it is sufficient to show $\{p_0\}\ \left[q\right] \sqsubseteq_{\mathrm{id}} \left[q_0 \wedge p_1'\right]\ ;\ \left[p_1,\ q_1\right]$ and hence by Lemma 5 (precondition-traces) to show that

$$\forall\, t \in \llbracket \left[q_0 \wedge p_1'\right]\ ;\ \left[p_1,\ q_1\right] \rrbracket_{\mathrm{id}} \bullet \mathit{pre}(t) \in p_0 \Rightarrow t \in \llbracket \left[q\right] \rrbracket\ .$$

A trace t of $\left[q_0 \wedge p_1'\right]\ ;\ \left[p_1,\ q_1\right]$ in environment id is either an interrupted trace of $\left[q_0 \wedge p_1'\right]$ or a finite trace t_0 of $\left[q_0 \wedge p_1'\right]$ followed by a trace t_1 of $\left[p_1,\ q_1\right]$. If *interrupted*(t) then t is also a trace of $\left[q\right]$, otherwise $t = t_0 \frown t_1$. Because t_0 is a finite trace of $\left[q_0 \wedge p_1'\right]$, it follows that $(\mathit{pre}(t_0), \mathit{post}(t_0)) \in (q_0 \wedge p_1')$ and hence $\mathit{post}(t_0) \in p_1$. Because t is consistent (12), $\mathit{pre}(t_1) \in p_1$ and hence as t_1 is a trace of $\left[q_1\right]$, either *interrupted*(t_1) or t_1 is finite and $(\mathit{pre}(t_1), \mathit{post}(t_1)) \in q_1$. If *interrupted*$(t_1)$, then *interrupted*$(t)$ and hence t is a trace of $\left[q\right]$, otherwise as $t = t_0 \frown t_1$, both $\mathit{pre}(t) \in p_0$ and $(\mathit{pre}(t), \mathit{post}(t)) \in ((q_0 \wedge p_1') \fatsemi q_1)$ and because $p_0 \wedge ((q_0 \wedge p_1') \fatsemi q_1) \Rrightarrow q$, $(\mathit{pre}(t), \mathit{post}(t)) \in q$ and hence t is a trace of $\left[q\right]$. □

Lemma 15 (introduce-test)

$$\left[\mathit{def}(b),\ b \wedge \mathrm{id}\right] \sqsubseteq [[b]]$$

Proof. By Lemma 8 (refine-specification) and Lemma 5 (precondition-traces) it is sufficient to show

$$\forall\, t \in \llbracket [[b]] \rrbracket_{\mathrm{id}} \bullet \mathit{pre}(t) \in \mathit{def}(b) \Rightarrow t \in \llbracket \left[b \wedge \mathrm{id}\right] \rrbracket$$

By Rule (A.113) a trace of the target ($[[b]]$) is any evaluation of b to true using the expression evaluation rules (Appendix A.3), or an evaluation ending in **abort** if b is not defined. Because the evaluation of b does not change the state and neither do the environment steps, if $\mathit{def}(b)$ holds initially then it holds for all states in the trace, and hence the evaluation to $\perp$ is not possible. Moreover, because the state does not change in any finite trace of $[[b]]$, b must be true for all states in the trace, including in the final state. Therefore, all finite traces of $[[b]]$ are traces of $\left[b \wedge \mathrm{id}\right]$. Finally, any interrupted trace of $[[b]]$ is also a possible behaviour of the $\left[b \wedge \mathrm{id}\right]$ (by Rule (A.114)). □

Lemma 30 (conjunction-strict)

We focus on $\{p\}(c \Cap d) = (\{p\}c) \Cap d$, the other cases follow similarly.
Proof. By Rule (A.99) there are two cases of $c \Cap d$ to consider: 1) when both c and d may take the same step α, and 2) when either may take a step to **abort**. The precondition p requires a further two subcases by Rule (A.96): a) when p is satisfied by the α, and b) when it is not.

- Case 1a. In this case α is trivially a step of each side of the equality.
- Case 1b. If α does not satisfy p then the source may take a step ending in **abort**. The subcommand $\{p\}c$ may similarly take a step ending in **abort**, and hence so may the conjunction.
- Case 2a. The aborting step of the conjunction is promoted to an aborting step of the source. If the aborting step is due to c, this is promoted to an aborting step of $\{p\}c$, which is promoted to an aborting step of the target. If the aborting step is due to d, both source and target promote this label.
- Case 2b. As with case 2a above, the aborting step is either promoted by the conjunction, or permits an aborting step of the precondition.

□

Lemma 50 (refine-in-guarantee-context)

For any relations g and r and commands c_0, c_1 and d, such that $c_0 \sqsubseteq_{g \vee r} c_1$,

$$c_0 \parallel (\mathbf{guar}\, g \bullet d) \ \ \sqsubseteq_r \ \ c_1 \parallel (\mathbf{guar}\, g \bullet d) \ .$$

Proof. We use Theorem 113, recalling that $(\mathbf{guar}\, g \ \bullet \ d) \ \widehat{=} \ d \Cap \langle g \ \vee \ \mathrm{id}\rangle^{\omega}$. We first show (B.118). Assume $d \xrightarrow{\alpha} d'$. In the case where d' is **abort** then both source and target may abort (Rule (A.99) and Rule (A.103)). In the case where d' is not abort, then we may further assume that a program step α satsifies $g \vee \mathrm{id}$.

Because the context is r, by Definition 1 (refinement-in-context) we need only consider traces of the target where the environment steps satsify r. Consider the case where the target takes an environment step,

$$c_1 \parallel (\mathbf{guar}\, g \bullet d) \xrightarrow{\epsilon(\sigma,\sigma')} c_1' \parallel (\mathbf{guar}\, g \bullet d')$$

assuming $(\sigma, \sigma') \in r \vee \mathrm{id}$. By Rule (A.103) both subprocesses must have also taken this step, and hence $c_1 \xrightarrow{\epsilon(\sigma,\sigma')} c_1'$. By assumption $(\sigma, \sigma') \in r \vee \mathrm{id}$, which immediately implies $(\sigma, \sigma') \in g \vee r \vee \mathrm{id}$. Hence from the assumption $c_0 \sqsubseteq_{g\vee r} c_1$, we have that $\epsilon(\sigma, \sigma')$ is a step of c_0, and therefore by extension is also a step of the source.

Now consider a program step of the target.

$$c_1 \parallel (\mathbf{guar}\, g \bullet d) \xrightarrow{\pi(\sigma,\sigma')} c_1' \parallel (\mathbf{guar}\, g \bullet d')$$

This is a program step of either operand. If $\pi(\sigma, \sigma')$ is a program step of c_1 (matched by an environment step of $(\mathbf{guar}\, g \ \bullet \ d)$), then it is also a step of the source, by assumption. The interesting case of the proof is when $\pi(\sigma, \sigma')$ is a program step of $(\mathbf{guar}\, g \ \bullet \ d)$. Such a program step must be matched by a corresponding environment step of c_1. By the reasoning above, any program step of $(\mathbf{guar}\, g \ \bullet \ d)$ satisfies $g \vee \mathrm{id}$, and hence also satisfies $g \vee r \vee \mathrm{id}$. The corresponding environment step of c_1 therefore also satisfies $g \vee r \vee \mathrm{id}$, and by assumption this is a valid step of c_0 in the context $g \vee r \vee \mathrm{id}$. This completes the proof of (B.118). To prove (B.119) is straightforward as both source and target evolve similarly. □

Lemma 61 (introduce-variable)

Assuming x nfi c and $x \notin y$,

$$y : c \sqsubseteq (\mathbf{var}\, x \bullet x, y : c)$$

Proof. From the definition of a local variable block (10), the statement in the law is equivalent to showing that for all $v \in \mathit{Val}$,

$$y : c \sqsubseteq (\mathbf{state}\, x \mapsto v \bullet x, y : c) \ .$$

Expanding the definition of the frame and hence guarantee gives:

$$c \Cap \langle \mathrm{id}(\bar{y})\rangle^{\omega} \sqsubseteq (\mathbf{state}\, x \mapsto v \bullet c \Cap \langle \mathrm{id}(\overline{x, y})\rangle^{\omega})$$

For clarity we use the following equivalent version of Rule (A.106).

$$\frac{c \xrightarrow{\pi(\sigma,\sigma')} c' \quad v = \sigma(y) \quad v' = \sigma'(y) \quad w \in \mathit{Val}}{(\mathbf{state}\, y \mapsto v \bullet c) \xrightarrow{\pi(\sigma[y\mapsto w],\sigma'[y\mapsto w])} (\mathbf{state}\, y \mapsto v' \bullet c')}$$

Consider an arbitrary environment step $\epsilon(\sigma, \sigma')$ of c. There are no restrictions on the allowed environment step of the source. However by Rule (A.107) environment steps $\epsilon(\sigma, \sigma')$ of c in the target will be disallowed unless $\sigma(x) = \sigma'(x)$. By Rule (A.107) the promoted step overrides the values of x with any arbitrary values; and by the argument above, these are allowed also by the source. This reasoning shows how the semantics of the local variable command prevent the environment from modifying the local declaration, but allow it to modify other declarations at a higher level of scope.

The more interesting cases are for arbitrary program steps $\pi(\sigma, \sigma')$ of the target. If this is an aborting step then by Rule (A.99) and Rule (A.108) this is an aborting step of both source and target. If $\pi(\sigma, \sigma')$ is not an aborting step, then it must be a step of both c and $\langle \mathrm{id}(\overline{x, y}) \rangle$, that is, $(\sigma, \sigma') \in \mathrm{id}(\overline{x, y})$. By the version of Rule (A.106) above, the value $\sigma'(x)$ becomes the new local value for x, and in addition the promoted label sets the value of x in both pre- and post-states to be arbitrary and equal. Hence, because $(\sigma, \sigma') \in \mathrm{id}(\overline{x, y})$, then $(\sigma[x \mapsto w], \sigma'[x \mapsto w]) \in \mathrm{id}(\bar{y})$, that is, the promoted label is stronger in the sense that x does not change. This is now a step of the source. □

Lemma 65 (parallel-interference)

Proof. Follows straightforwardly from Rule (A.95) and Rule (A.103). □

Lemma 66 (interference-atomic)

Proof. Follows straightforwardly from Rule (A.95), Rule (A.103), and Rule (A.98). □

Lemma 103 (refine-var)

$$c \sqsubseteq_{\mathrm{id}(x)} d \qquad \Rightarrow \quad (\mathbf{var}\, x \bullet c) \sqsubseteq (\mathbf{var}\, x \bullet d)$$

Proof. Using the definition of a local variable block (10) and Law 14 (nondeterministic-choice) part (26), it is sufficient to show

$$c \sqsubseteq_{\mathrm{id}(x)} d \qquad \Rightarrow \qquad \forall v \bullet (\mathbf{state}\, x \mapsto v \bullet c) \sqsubseteq (\mathbf{state}\, x \mapsto v \bullet d)\ .$$

We assume the property on the left of the implication and show the property on the right holds for all v using Theorem 113. For property (B.118) for program steps

$$\begin{array}{ll}
& (\mathbf{state}\, x \mapsto v \bullet d) \xrightarrow{\pi(\sigma,\sigma')} (\mathbf{state}\, x \mapsto v' \bullet d') \\
\equiv & \text{by Rule (A.106)} \\
& (d \xrightarrow{\pi(\sigma[x\mapsto v],\sigma'[x\mapsto v'])} d') \wedge \sigma'(x) = \sigma(x) \\
\Rrightarrow & \text{as } c \sqsubseteq_{\mathrm{id}(x)} d \\
& (c \xrightarrow{\pi(\sigma[x\mapsto v],\sigma'[x\mapsto v'])} c') \wedge \sigma'(x) = \sigma(x) \\
\equiv & \text{by Rule (A.106)} \\
& (\mathbf{state}\, x \mapsto v \bullet c) \xrightarrow{\pi(\sigma,\sigma')} (\mathbf{state}\, x \mapsto v \bullet c')
\end{array}$$

and for environment steps

$$\begin{array}{ll}
& (\mathbf{state}\, x \mapsto v \bullet d) \xrightarrow{\epsilon(\sigma,\sigma')} (\mathbf{state}\, x \mapsto v \bullet d') \\
\equiv & \text{by Rule (A.107)} \\
& d \xrightarrow{\epsilon(\sigma[x\mapsto v],\sigma'[x\mapsto v])} d' \\
\Rrightarrow & \text{as } c \sqsubseteq_{\mathrm{id}(x)} d \text{ and } (\sigma[x \mapsto v], \sigma'[x \mapsto v]) \in \mathrm{id}(x) \\
& c \xrightarrow{\epsilon(\sigma[x\mapsto v],\sigma'[x\mapsto v])} c' \\
\equiv & \text{by Rule (A.107)} \\
& (\mathbf{state}\, x \mapsto v \bullet c) \xrightarrow{\epsilon(\sigma,\sigma')} (\mathbf{state}\, x \mapsto v \bullet c')
\end{array}$$

Property (B.119) is straightforward as both source and target evolve similarly. The proofs for the aborting cases are similar using Rule (A.108) and Rule (A.109). □

Index

References

P. H. G. Aczel. 1983. On An Inference Rule for Parallel Composition. (1983). private communication.

R.-J. R. Back. 1981. On Correct Refinement of Programs. *J. Comput. System Sci.* 23, 1 (Feb. 1981), 49–68.

R.-J. R. Back and J. von Wright. 1998. *Refinement Calculus: A Systematic Introduction*. Springer.

S. Brookes. 2007. A semantics for concurrent separation logic. *Theoretical Computer Science* 375, 13 (2007), 227 – 270.

Ernie Cohen. 2000. Separation and Reduction. In *Mathematics of Program Construction, 5th International Conference, Portugal, July 2000*. Springer-Verlag, 45–59.

Joey W. Coleman. 2008. Expression Decomposition in a Rely/Guarantee Context. In *VSTTE (Lecture Notes in Computer Science)*, Natarajan Shankar and Jim Woodcock (Eds.), Vol. 5295. Springer, 146–160.

J. W. Coleman and C. B. Jones. 2007. A Structural Proof of the Soundness of Rely/guarantee Rules. *Journal of Logic and Computation* 17, 4 (2007), 807–841. DOI:http://dx.doi.org/10.1093/logcom/exm030

Pierre Collette and Cliff B. Jones. 2000. Enhancing the Tractability of Rely/Guarantee Specifications in the Development of Interfering Operations. In *Proof, Language and Interaction*, Gordon Plotkin, Colin Stirling, and Mads Tofte (Eds.). MIT Press, Chapter 10, 277–307.

F. de Boer, U. Hannemann, and W. de Roever. 1999. Formal justification of the rely-guarantee paradigm for shared-variable concurrency: a semantic approach. In *FM99 Formal Methods*, Jeannette Wing, Jim Woodcock, and Jim Davies (Eds.). Lecture Notes in Computer Science, Vol. 1709. Springer Berlin / Heidelberg, 714–714.

W. P. de Roever. 2001. *Concurrency Verification: Introduction to Compositional and Noncompositional Methods*. Cambridge University Press.

Jürgen Dingel. 2000. *Systematic Parallel Programming*. Ph.D. Dissertation. Carnegie Mellon University. CMU-CS-99-172.

J. Dingel. 2002. A Refinement Calculus for Shared-Variable Parallel and Distributed Programming. *Formal Aspects of Computing* 14, 2 (2002), 123–197. DOI:http://dx.doi.org/10.1007/s001650200032

Brijesh Dongol and Ian J. Hayes. 2010. Compositional Action System Derivation Using Enforced Properties. In *Mathematics of Program Construction (MPC)* (Berlin) *(LNCS)*, C. Bolduc, J. Desharnais, and B. Ktari (Eds.), Vol. 6120. Springer Verlag, 119–139. DOI:http://dx.doi.org/10.1007/978-3-642-13321-3_9

R. W. Floyd. 1967. Assigning Meanings to Programs. In *Proc. Symp. in Applied Mathematics, Vol.19: Mathematical Aspects of Computer Science*. American Mathematical Society, 19–32.

I. J. Hayes, A. Burns, B. Dongol, and C. B. Jones. 2013. Comparing Degrees of Non-Deterministic in Expression Evaluation. *Comput. J.* 56, 6 (2013), 741–755. doi: 10.1093/comjnl/bxt005.

C.A.R. Hoare and Jifeng He. 1986. The Weakest Prespecification. *Fundamenta Informaticae* IX (1986), 51–84.

C. A. R. Hoare. 1969. An Axiomatic Basis for Computer Programming. *Commun. ACM* 12, 10 (October 1969), 576–580, 583.

C. B. Jones. 1981. *Development Methods for Computer Programs including a Notion of Interference*. Ph.D. Dissertation. Oxford University. Printed as: Programming Research Group, Technical Monograph 25.

C. B. Jones. 1983. Tentative Steps Toward a Development Method for Interfering Programs. *Transactions on Programming Languages and System* 5, 4 (1983), 596–619. `DOI:http://dx.doi.org/10.1145/69575.69577`

C. B. Jones. 1987. Program Specification and Verification in VDM. In *Logic of Programming and Calculi of Discrete Design*, M. Broy (Ed.). NATO ASI Series F: Computer and Systems Sciences, Vol. 36. Springer-Verlag, 149–184.

C. B. Jones. 1996. Accommodating Interference in the Formal Design of Concurrent Object-Based Programs. *Formal Methods in System Design* 8, 2 (March 1996), 105–122. `DOI:http://dx.doi.org/10.1007/BF00122417`

Cliff B. Jones. 2003a. The Early Search for Tractable Ways of Reasonning about Programs. *IEEE, Annals of the History of Computing* 25, 2 (2003), 26–49. `DOI:http://dx.doi.org/10.1109/MAHC.2003.1203057`

Cliff B. Jones. 2003b. Operational semantics: Concepts and their expression. *Inf. Process. Lett.* 88, 1-2 (2003), 27–32.

C. B. Jones. 2007. Splitting Atoms Safely. *Theoretical Computer Science* 375, 1–3 (2007), 109–119. `DOI:http://dx.doi.org/10.1016/j.tcs.2006.12.029`

Cliff B. Jones and Ken G. Pierce. 2011. Elucidating concurrent algorithms via layers of abstraction and reification. *Formal Aspects of Computing* 23, 3 (2011), 289–306. `DOI:http://dx.doi.org/10.1007/s00165-010-0156-1`

R. Milner. 1989. *Communication and Concurrency*. Prentice Hall.

C. C. Morgan. 1988. The Specification Statement. *ACM Trans. on Prog. Lang. and Sys.* 10, 3 (July 1988).

C. C. Morgan. 1994. *Programming from Specifications* (second ed.). Prentice Hall.

C. C. Morgan and T. N. Vickers. 1990. Types and Invariants in the Refinement Calculus. *Science of Computer Programming* 14 (1990), 281–304.

C. C. Morgan and T. N. Vickers (Eds.). 1994. *On the Refinement Calculus*. Springer-Verlag.

C. C. Morgan and T. N. Vickers. 1994. Types and Invariants in the Refinement Calculus. See Morgan and Vickers (1994), 127–154. Originally published as (Morgan and Vickers 1990).

J. M. Morris. 1987. A Theoretical Basis for Stepwise Refinement and the Programming Calculus. *Science of Computer Programming* 9, 3 (1987), 287–306.

Peter D. Mosses. 2004a. Exploiting Labels in Structural Operational Semantics. *Fundam. Inform.* 60, 1-4 (2004), 17–31.

Peter D. Mosses. 2004b. Modular structural operational semantics. *J. Log. Algebr. Program.* 60-61 (2004), 195–228.

R. Nickson and I. J. Hayes. 1997. Supporting Contexts in Program Refinement. *Science of Computer Programming* 29, 3 (1997), 279–302.

P. W. O'Hearn, H. Yang, and J. C. Reynolds. 2009. Separation and Information Hiding. *ACM TOPLAS* 31(3) (April 2009). Preliminary version appeared in 31st POPL, pp268-280, 2004.

S. Owicki. 1975. *Axiomatic Proof Techniques for Parallel Programs*. Ph.D. Dissertation. Department of Computer Science, Cornell University.

Gordon D. Plotkin. 2004. A structural approach to operational semantics. *Journal of Logic and Algebraic Programming* 60–61 (July–December 2004), 17–139. `DOI:http://dx.doi.org/doi:10.1016/j.jlap.2004.03.002`

Leonor Prensa Nieto. 2001. *Verification of Parallel Programs with the Owicki-Gries and Rely-Guarantee Methods in Isabelle/HOL*. Ph.D. Dissertation. Institut für Informatic der Technischen Universitaet München.

Leonor Prensa Nieto. 2003. The Rely-Guarantee Method in Isabelle/HOL. In *Proceedings of ESOP 2003 (LNCS)*, Vol. 2618. Springer-Verlag.

C. Stirling. 1986. A Compositional Reformulation of Owicki-Gries' Partial Correctness Logic for a Concurrent While Language. In *ICALP'86*. Springer-Verlag. LNCS 226.

K. Stølen. 1990. *Development of Parallel Programs on Shared Data-Structures*. Ph.D. Dissertation. Manchester University. Available as UMCS-91-1-1.

J. von Wright. 2004. Towards a refinement algebra. *Sci. of Comp. Prog.* 51 (2004), 23–45.

John Wickerson, Mike Dodds, and Matthew Parkinson. 2010. *Explicit stabilisation for modular rely-guarantee reasoning*. Technical Report 774. Computer Laboratory, University of Cambridge.