COMPUTING SCIENCE

Laws and Semantics for Rely-Guarantee Refinement

Ian J. Hayes, Cliff B. Jones and Robert J. Colvin

TECHNICAL REPORT SERIES

No. CS-TR-1425 July 2014

TECHNICAL REPORT SERIES

No. CS-TR-1425 July, 2014

Laws and Semantics for Rely-Guarantee Refinement

Ian J. Hayes, Cliff B. Jones and Robert 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 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.

Laws and semantics for rely-guarantee refinement
[By] I. J. Hayes, C.B. Jones, and R. J. Colvin
Newcastle upon Tyne: Newcastle University: Computing Science, 2014.

(Newcastle University, Computing Science, Technical Report Series, No. CS-TR-1425)

Added entries

NEWCASTLE UNIVERSITY
Computing Science. Technical Report Series. CS-TR-1425

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 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

Laws and semantics for rely-guarantee refinement[*]

Ian J. Hayes[1] Cliff B. Jones[2]
Robert J. Colvin[1]

[1]School of Information Technology and Electrical Engineering, The University of Queensland, Australia
[2]School of Computing Science, Newcastle University, UK

July 14, 2014

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 two new commands: a guarantee command ($\textbf{guar}\, g \bullet c$) 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 ($\textbf{rely}\, r \bullet c$) 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.

Contents

[*] This report supersedes and replaces CS-TR-1395 (“Reasoning about concurrent programs: Refining rely-guarantee thinking”) by the same authors.

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

1 Introduction

The rely-guarantee rules of Jones [Jon81, Jon83] provide a compositional approach to reasoning about concurrent processes (the most accessible reference is [Jon96]; an exhaustive analysis of various compositional and non-compositional approaches can be found in [dR01]). Based on many other contributions such as [Stø90, CJ00, CJ07] these rules have been absorbed into a more general rely-guarantee "thinking" as exemplified in [JP11].

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. To handle reasoning about concurrent programs, [Jon81, Jon83] extended Hoare triples to quintuples of the form $\{p, r\}\ c\ \{g, q\}$, in which the precondition p is augmented with a rely condition r and the post condition q is augmented with a guarantee condition g. A set of Hoare-style inference rules based on these quintuples allows reasoning about concurrent programs.

In this paper we use an approach based on the refinement calculus rather than Hoare logic. The refinement calculus has separate primitives for precondition assumptions $\{p\}$ and post condition specifications $[q]$. To extend the refinement calculus for concurrent programs, we add two new constructs: one to support a guarantee condition g on a command c, $(\mathbf{guar}\, g \bullet c)$, and the other to support a rely condition r, $(\mathbf{rely}\, r \bullet c)$. The main advantage of our new approach is that one can develop separate simpler refinement laws for each construct in isolation and then combine them to produce more complex laws involving rely and guarantee conditions as well as pre- and post-conditions. Proofs of laws equivalent to the Jones-style inference rules are then straightforward to derive from the more basic laws and it is easier to derive new variants of laws.

The guarantee command. To simplify reasoning about types and invariants in the refinement calculus, Morgan and Vickers [MV90] 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 [CJ00] 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 satisfy the relation g between their before-state and after-state or stutter (i.e. do not change the 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 $[q]$, in which q is a relation (expressed as a two-state predicate) giving its post condition [Mor88]. Any implementation of $[q]$ must terminate in a state such that the initial and final states are related by q. The command $(\mathbf{guar}\, g \bullet [q])$ not only satisfies the post condition q but also only uses atomic program 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 $(\mathbf{guar}\, g \bullet [q])$ 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.

The rely command. 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 $(\mathbf{rely}\, r \bullet c)$, 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 $\langle r \vee \mathsf{id}\rangle^*$ represents any finite number, zero or more, of atomic steps that satisfy the relation r or the identity relation id (i.e. stuttering). A specification $[q]$ is refined by ($\sqsubseteq$) the command $(\mathbf{rely}\, r \bullet [q])$ run in parallel with interference bounded by r, i.e.

$$[q] \;\sqsubseteq\; (\mathbf{rely}\, r \bullet [q]) \parallel \langle r \vee \mathsf{id}\rangle^* .$$

The stronger the rely condition, the more constrained the acceptable environments and hence the easier it is to implement $[q]$. The empty relation is the strongest rely condition: it represents only stuttering interference from the environment. Common rely conditions are those that require that 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.

Parallel. The key step in any concurrent refinement is the introduction of parallel composition, splitting a sequential specification into two processes. The quintessential quintuple rule of Jones allows a postcondition of $q_0 \wedge q_1$ to be achieved by two parallel processes that achieve q_0 and q_1, respectively, provided each branch of the parallel guarantees the rely condition of the other. This can be expressed in the notation used here as follows.

$$[q_0 \wedge q_1] \;\sqsubseteq\; (\mathbf{guar}\, g_0 \bullet (\mathbf{rely}\, g_1 \bullet [q_0])) \parallel (\mathbf{guar}\, g_1 \bullet (\mathbf{rely}\, g_0 \bullet [q_1])) \qquad (1)$$

Using the strict conjunction operator "$\Cap$", the specification $[q_0 \wedge q_1]$ can be written as a conjunction of two specifications: $[q_0] \Cap [q_1]$. This motivates a generalisation of (1) to refine a conjunction of two commands c_0 and c_1 to a parallel composition.

$$c_0 \Cap c_1 \;\sqsubseteq\; (\mathbf{guar}\, g_0 \bullet (\mathbf{rely}\, g_1 \bullet c_0)) \parallel (\mathbf{guar}\, g_1 \bullet (\mathbf{rely}\, g_0 \bullet c_1))$$

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 post condition 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.

In the literature, early justifications of proof obligations for rely/guarantee conditions had to confront the complete set of assumptions and commitments at once. [Pre01, Pre03] limits the task both by disallowing nesting of parallelism (but allowing multi-way parallelism) and by simplifying assumptions about atomicity; she does however provide a complete Isabelle-checked proof. In contrast, [Jon81] and [CJ07] only offer proof outlines, but with minimal atomicity assumptions and using a language with arbitrarily nested parallelism. In this paper the proof of the rule for introducing a parallel composition is comparatively short and elegant because it utilises more basic laws that were not expressible in the earlier approaches.

[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.

Refinement with interference. 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, $(\mathbf{uses}\, X \bullet c)$, is introduced; it restricts c to only access variables in the set X.

Section 8 examines expression evaluation, which in the presence of non-trivial interference is non-deterministic. Section 9 examines local variable blocks. Local variables are not subject to interference from external processes and, within a local variable block, code cannot modify any other variables with the same name as the local variables and hence guarantees they are unchanged.

Reasoning about control structures in the presence of interference requires accepting that interference can affect test evaluation. In the following refinement, the specification $[q]$ is refined to an **if** statement, in a context in which the variable x may be decreased, as represented by the rely relation $x' \leq x$ in which x and x' stand for the initial and final values of the variable x, respectively, within the relation. For the "then" branch, if $x < 0$ is true, decreasing x will not falsify it and hence $x < 0$ can be assumed as a precondition of the "then" branch. For the "else" branch, if $x \geq 0$ is true, decreasing x may falsify $x \geq 0$, and hence only the trivial precondition true can be assumed for the "else" branch. A specification command with a precondition p and postcondition q is written $[p,\ q]$; any behaviour is possible if p does not hold initially.

$$(\mathbf{rely}\, x' \leq x \bullet [q]) \ \sqsubseteq\ \mathbf{if}\, x < 0\ \mathbf{then}\ (\mathbf{rely}\, x' \leq x \bullet [x < 0,\ q])\ \mathbf{else}\ (\mathbf{rely}\, x' \leq x \bullet [\mathsf{true},\ q])$$

To handle tests within control structures, a new test construct $[[b]]$, for boolean expression b, is used to allow for interference during test evaluation. Section 10 examines rules for introducing conditionals and "while" loops. Showing termination of loops requires reasoning about the effect of the interference on a well-founded relation, in addition to the loop guard. A non-trivial example of a parallel search algorithm involving relies, guarantees, parallelism, local variables, conditionals and iteration is developed in full in Section 11.

Theory of atomic steps. In order to define the new constructs, we found it beneficial to develop a theory of programs based on atomic program/environment steps. 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. Using the theory, it is only necessary to justify that about two dozen basic lemmas respect the semantic model: all of the remaining laws are derived from this basic set. 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. The theory of atomic steps is presented, as needed, throughout the paper. Interestingly, it allows program properties to be represented as refinements, for example, termination of a command c from states satisfying the precondition p in an environment in which all the interference steps satisfy the relation r can be represented by c satisfying the refinement:

$$\{p\}\langle\mathsf{true}\rangle^* \ \sqsubseteq_r\ c$$

where the command $\langle\mathsf{true}\rangle^*$ can perform a finite number, zero or more, of any atomic program step.

The semantics of the language is given in Appendix A and proofs of the basic lemmas are contained in Appendix B.

2 Programming language and refinement

2.1 Syntax

The syntax of expressions and commands is given in Figure 1. 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 in the before state but otherwise performs a single atomic program step that satisfies relation q between its before and after states. The environment is free to interleave any environment steps either before or after the program step. Note that p is evaluated

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

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

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.

$$c \;::=\; \langle p, q\rangle \mid [q] \mid \{p\} \mid \bigsqcap C \mid c_1 \Cap c_2 \mid c_1 \,;\, c_2 \mid c_1 \parallel c_2 \mid [[b]] \mid \\ \mathbf{uses}\, X \bullet c \mid \mathbf{state}\; x \mapsto v \bullet c$$

Figure 1: Syntax of expressions and basic commands

in the state immediately prior to the program step. The word "step" always means an atomic step. The specification command $[q]$ may take any finite number (zero or more) of atomic program steps to achieve the relation q between its before and after states but it assumes that the environment will not interfere – only stuttering environment steps are allowed otherwise the specification aborts. Refinement in an environment that only performs stuttering steps in our concurrent theory corresponds to refinement in a sequential program theory.

The precondition command $\{p\}$ terminates immediately if p holds initially, otherwise it aborts, at which point any behaviour is possible. For a preconditioned command, "$\{p\}\,;\,c$" is written with the ";" elided as "$\{p\}c$". The operator "$\bigsqcap$" is (demonic) nondeterministic choice from a set of commands and "$\Cap$" is strict conjunction of commands (a specification rather than implementation construct – see Section 3). Sequential composition ";" and parallel composition "$\parallel$" are standard. 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); only traces for tests that succeed are retained as program traces. Tests are used to define the "if" and "while" commands. 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.

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 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. 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 for zero or more and $c^{\omega+}$ for one or more). Although fixed point operators are not part of the programming language, iterations of commands are defined via greatest (ν) and least (μ) fixed points with respect to the refinement ordering [vW04].

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 $(\forall \sigma : \Sigma \bullet p_0(\sigma) \Rightarrow p_1(\sigma))$, and $q_0 \Rrightarrow q_1$ means $(\forall \sigma, \sigma' \in \Sigma \bullet q_0(\sigma, \sigma') \Rightarrow q_1(\sigma, \sigma'))$. If the implications hold in both directions for all states, the symbol "$\equiv$" is used. The predicate p' stands for the predicate p with all variables replaced by their primed versions, i.e. p holds

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 $\mathsf{id}(X)$ is the identity relation on X. For a variable x, $\bar{x}$ is the set of all variables other than x.

$$\mathbf{skip} \mathrel{\widehat{=}} \{\mathsf{true}\} \tag{2}$$
$$\mathbf{abort} \mathrel{\widehat{=}} \{\mathsf{false}\} \tag{3}$$
$$\langle q \rangle \mathrel{\widehat{=}} \langle \mathsf{true}, q \rangle \tag{4}$$
$$\left[p,\, q\right] \mathrel{\widehat{=}} \{p\}\left[q\right] \tag{5}$$
$$\mathbf{magic} \mathrel{\widehat{=}} \bigsqcap\{\} \tag{6}$$
$$c_0 \sqcap c_1 \mathrel{\widehat{=}} \bigsqcap\{c_0, c_1\} \tag{7}$$
$$c^* \mathrel{\widehat{=}} \nu x \bullet \mathbf{skip} \sqcap c \,;\, x \tag{8}$$
$$c^\infty \mathrel{\widehat{=}} \mu x \bullet c \,;\, x \tag{9}$$
$$c^\omega \mathrel{\widehat{=}} \mu x \bullet \mathbf{skip} \sqcap c \,;\, x \tag{10}$$
$$c^+ \mathrel{\widehat{=}} c^* \,;\, c \tag{11}$$
$$c^{\omega+} \mathrel{\widehat{=}} c^\omega \,;\, c \tag{12}$$
$$\mathbf{if}\, b\, \mathbf{then}\, c_0\, \mathbf{else}\, c_1 \mathrel{\widehat{=}} ([[b]] \,;\, c_0) \sqcap ([[\neg b]] \,;\, c_1) \tag{13}$$
$$\mathbf{while}\, b\, \mathbf{do}\, c \mathrel{\widehat{=}} ([[b]] \,;\, c)^\omega \,;\, [[\neg b]] \tag{14}$$
$$x := e \mathrel{\widehat{=}} \bigsqcap\{v \in \mathit{Val} \bullet [[e = v]] \,;\, \langle x' = v \wedge \mathsf{id}(\bar{x})\rangle\} \tag{15}$$
$$\mathbf{var}\, x \bullet c \mathrel{\widehat{=}} \bigsqcap\{v \in \mathit{Val} \bullet (\mathbf{state}\, x \mapsto v \bullet c)\} \tag{16}$$

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

in the after state. The composition of two relations q_0 and q_1 is defined by

$$q_0 \fatsemi q_1 \mathrel{\widehat{=}} (\exists\, \mathit{Var}'' \bullet q_0[\mathit{Var}''/\mathit{Var}'] \wedge q_1[\mathit{Var}''/\mathit{Var}]) \tag{17}$$

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 [Bac81, Mor87, Mor88, Mor94, MV94a, BvW98] 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 both precondition assumptions $\{p\}$ and postcondition specifications $\left[q\right]$ as commands in the wide-spectrum language. In this paper 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 ($\mathbf{guar}\, g \bullet (\mathbf{rely}\, r \bullet \{p\}\left[q\right])$). The statement that this specification is refined by a command c is written

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

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

[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.

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 [Acz83], which distinguishes program steps and environment steps [dBHdR99, dR01]. A state is a total mapping from program variable names (*Var*) to values (*Val*) or the undefined state "$\bot$".

$$\Sigma \quad \widehat{=} \quad \bot \mid \mathit{Var} \to \mathit{Val} \tag{18}$$

Each atomic step of a program, $\alpha \in \mathcal{L}$, is either a program step, $\pi(\sigma,\sigma')$, an environment step, $\epsilon(\sigma,\sigma')$, or a termination step, $\checkmark(\sigma)$, where σ and σ' represent the pre (before) and post (after) states of a step.

$$\begin{array}{rclrclrcl} \mathit{pre}(\pi(\sigma,\sigma')) & \widehat{=} & \sigma & \mathit{pre}(\epsilon(\sigma,\sigma')) & \widehat{=} & \sigma & \mathit{pre}(\checkmark(\sigma)) & \widehat{=} & \sigma \\ \mathit{post}(\pi(\sigma,\sigma')) & \widehat{=} & \sigma' & \mathit{post}(\epsilon(\sigma,\sigma')) & \widehat{=} & \sigma' & \mathit{post}(\checkmark(\sigma)) & \widehat{=} & \sigma \end{array}$$

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 and $\#t$ stands for the length of t.

$$\begin{array}{rcll} \mathcal{L}_\checkmark & \widehat{=} & \{\sigma : \Sigma \bullet \checkmark(\sigma)\} & (19) \\ \mathit{consistent}(t) & \widehat{=} & \forall\, i \in \mathrm{dom}(t) - \{0\} \bullet \mathit{post}(t(i-1)) = \mathit{pre}(t(i)) & (20) \\ \mathit{Trace} & \widehat{=} & \{t : \mathcal{L}^\omega \mid \; \mathit{consistent}(t) \wedge & (21) \\ & & \qquad (\forall\, i \in \mathrm{dom}(t) \bullet t(i) \in \mathcal{L}_\checkmark \Rightarrow t \in \mathcal{L}^* \wedge i = \#t - 1)\} & \\ \mathit{Complete} & \widehat{=} & \{t : \mathit{Trace} \mid \mathrm{dom}(t) \neq \mathbb{N} \Rightarrow t(\#t-1) \in \mathcal{L}_\checkmark\} & (22) \end{array}$$

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. Once a command aborts any trace of behaviour is possible, i.e. the transition $(\bot,\sigma)$ is possible for any σ. Parallel execution of two commands is given an interleaving semantics that matches a program step $\pi(\sigma,\sigma')$ of one command with an equivalent environment step $\epsilon(\sigma,\sigma')$ of the other to give a step $\pi(\sigma,\sigma')$ of the composition, or matches equal environment steps of both to give an environment step of the composition. Termination of one of the commands via a $\checkmark(\sigma)$ step means the parallel composition reduces to the other command.

In general, the semantics of a command is given for arbitrary environment transitions but often we wish to restrict consideration to a particular environment; given a trace t, its environment relation $\mathit{env}(t)$ may be extracted by collecting the pairs of states in each environment step.

$$\mathit{env}(t) \quad \widehat{=} \quad \{(\sigma,\sigma') \mid \epsilon(\sigma,\sigma') \in \mathrm{ran}(t)\} \tag{23}$$

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

$$[\![c]\!]_r \quad \widehat{=} \quad \{t \in [\![c]\!] \mid \mathit{env}(t) \subseteq r \vee \mathsf{id}\} \tag{24}$$

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 partial order.

Definition 2.1 (refinement-in-context) *For any relation r and commands c and d,*

$$\begin{array}{rcll} c \sqsubseteq_r d & \widehat{=} & [\![d]\!]_r \subseteq [\![c]\!] & (25) \\ c =_r d & \widehat{=} & (c \sqsubseteq_r d) \wedge (d \sqsubseteq_r c) & (26) \end{array}$$

Note that (25) is equivalent to $[\![d]\!]_r \subseteq [\![c]\!]_r$. Sequential refinement corresponds to refinement in a context in which the environment may only stutter: $c \sqsubseteq_{\mathsf{id}} d$. Refinement in any context corresponds to choosing r to be the universal relation true because this allows any environment steps.

Definition 2.2 (refinement) *For any commands c and d,*

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

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

Law 2.3 (refinement-monotonic) *For any commands c and d and relations r_0 and r_1, if $r_0 \Rrightarrow r_1 \lor \mathsf{id}$ and $c \sqsubseteq_{r_1} d$, then $c \sqsubseteq_{r_0} d$.*

Proof. As $r_0 \Rrightarrow r_1 \lor \mathsf{id}$ by (24), $[\![d]\!]_{r_0} \subseteq [\![d]\!]_{r_1}$ and by Definition 2.1 (refinement-in-context) $[\![d]\!]_{r_1} \subseteq [\![c]\!]$ and hence the result follows by transitivity. □

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 semantics of that construct, and the term "law" for 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 precondition command $\{p\}$ aborts (i.e. any behaviour is possible) 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, $\{p\}$ terminates immediately.

Lemma 2.4 (precondition) *For any predicates p, p_0 and p_1, relation r and commands c and d,*

$$\{p_0\}\{p_1\} \quad = \quad \{p_0 \land p_1\} \tag{27}$$

$$(p_0 \Rrightarrow p_1) \quad \Rightarrow \quad (\{p_0\} \sqsubseteq \{p_1\}) \tag{28}$$

$$(\{p\}c \sqsubseteq_r \{p\}d) \quad \Leftrightarrow \quad (\{p\}c \sqsubseteq_r d) \tag{29}$$

Because these properties are so basic, we often make use of them without explicit reference to this lemma.

Lemma 2.5 (parallel-precondition) *For any predicate p, and commands c and d,*

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

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. The only interesting case for refinement of a specification is refinement in an environment of id, because a specification command is defined to abort in any other environment.

Lemma 2.6 (refine-specification) *For any predicate p, relation q, and command c,*

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

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

Lemma 2.7 (make-atomic) *For any predicate p and relation q,* $[p,\ q] \sqsubseteq \langle p, q\rangle$.

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

Lemma 2.8 (consequence) *For any predicates p_0 and p_1, and relations q_0 and q_1, provided $p_0 \Rrightarrow p_1$ and $p_0 \land q_1 \Rrightarrow q_0$,*

$$\langle p_0, q_0\rangle \quad \sqsubseteq \quad \langle p_1, q_1\rangle \tag{30}$$

$$[p_0,\ q_0] \quad \sqsubseteq \quad [p_1,\ q_1] \tag{31}$$

Lemma 2.9 (sequential) *For any predicates p_0 and p_1, and any relations q, q_0 and q_1, such that $p_0 \wedge ((q_0 \wedge p_1') \fatsemi q_1) \Rrightarrow q$,*

$$\lfloor p_0,\ q\rceil \quad \sqsubseteq \quad \lfloor p_0,\ q_0 \wedge p_1'\rceil \ ; \lfloor p_1,\ q_1\rceil \ .$$

Recall that p_1' stands for p_1 holding in the after-state. The traces of a nondeterministic choice over a set of commands consists of the union of their traces.

Lemma 2.10 (nondeterminism-traces) *For a set of commands C,*

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

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

Law 2.11 (nondeterministic-choice) *For any relation r, command c and set of commands C,*

$$(\forall d \in D \bullet (\exists c \in C \bullet c \sqsubseteq_r d)) \quad \Rrightarrow \quad (\textstyle\bigsqcap C) \sqsubseteq_r (\textstyle\bigsqcap D) \tag{32}$$

$$c \in C \quad \Rrightarrow \quad (\textstyle\bigsqcap C) \sqsubseteq_r c \tag{33}$$

$$(\forall d \in D \bullet c \sqsubseteq_r d) \quad \Rrightarrow \quad c \sqsubseteq_r (\textstyle\bigsqcap D) \tag{34}$$

Proof. Property (32) follows from Lemma 2.10 (nondeterminism-traces) and Definition 2.1 (refinement-in-context). The other two parts are special cases of (32) given that $c = \bigsqcap\{c\}$. □

In the sequential refinement calculus the specification $\lfloor def(b),\ b \wedge \mathsf{id}\rceil$, where predicate $def(b)$ holds if and only if b is well 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 $\lfloor def(b),\ b \wedge \mathsf{id}\rceil$ in an environment consisting of only stuttering. If b evaluates to false the test is infeasible.

Lemma 2.12 (introduce-test) *For any boolean expression b,* $\quad \lfloor def(b),\ b \wedge \mathsf{id}\rceil \sqsubseteq [[b]]$.

Lemmas 2.13 to 2.17 are based on standard properties of iterations defined in terms of fixed points [vW04]. They follow from definitions (8), (9) and (10).

Lemma 2.13 (fold/unfold-iteration) *For any command c,*

$$c^* \quad = \quad \mathbf{skip} \sqcap c \ ; c^* \tag{35}$$

$$c^\infty \quad = \quad c \ ; c^\infty \tag{36}$$

$$c^\omega \quad = \quad \mathbf{skip} \sqcap c \ ; c^\omega \tag{37}$$

Lemma 2.14 (iteration-induction) *For any relation r and commands c, d and x,*

$$x \sqsubseteq_r d \sqcap c \ ; x \quad \Rrightarrow \quad x \sqsubseteq_r c^* \ ; d \tag{38}$$

$$c \ ; x \sqsubseteq_r x \quad \Rrightarrow \quad c^\infty \sqsubseteq_r x \tag{39}$$

$$d \sqcap c \ ; x \sqsubseteq_r x \quad \Rrightarrow \quad c^\omega \ ; d \sqsubseteq_r x \tag{40}$$

Lemma 2.15 (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 [vW04].

Lemma 2.16 (isolation) *For any command c,* $\quad c^\omega = c^* \sqcap c^\infty$.

The following lemma allows reasoning about fixed points on a complete lattice [BvW98]. The lattice join operation is the standard one induced by the refinement ordering, i.e. intersection on set of traces.

Lemma 2.17 (fusion) *For monotonic functions F and G on complete lattices and a function H, if $H \circ F = G \circ H$ then*

$$\begin{array}{rl} H(\mu F) = \mu G & \textit{provided } H \textit{ is continuous and} \\ H(\nu F) = \nu G & \textit{provided } H \textit{ is co-continuous.} \end{array}$$

A terminating command can only perform a finite number of atomic steps. As in the sequential refinement calculus, a command may be guaranteed to stop 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.

Definition 2.18 (stops) *For any command c and relation r, $stops(c,r)$ is the weakest predicate such that from states satisfying $stops(c,r)$, c is guaranteed to stop in environment r.*

$$\{stops(c,r)\}\langle \mathsf{true}\rangle^* \quad \sqsubseteq_r \quad c \tag{41}$$

$$\forall p \bullet (\{p\}\langle \mathsf{true}\rangle^* \sqsubseteq_r c) \quad \Leftrightarrow \quad (p \Rrightarrow stops(c,r)) \tag{42}$$

Law 2.19 (term-monotonic) *For any commands c and d and relations r, r_0 and r_1,*

$$(r_0 \Rrightarrow r_1 \vee \mathsf{id}) \quad \Rightarrow \quad (stops(c,r_1) \Rrightarrow stops(c,r_0)) \tag{43}$$

$$(c \sqsubseteq_r d) \quad \Rightarrow \quad (stops(c,r) \Rrightarrow stops(d,r)) \tag{44}$$

Proof. For (43) by Definition 2.18 (stops) part (41),

$$\begin{array}{ll} & \{stops(c,r_1)\}\langle \mathsf{true}\rangle^* \sqsubseteq_{r_1} c \\ \Rightarrow & \text{by Law 2.3 (refinement-monotonic) as } r_0 \Rrightarrow r_1 \vee \mathsf{id} \\ & \{stops(c,r_1)\}\langle \mathsf{true}\rangle^* \sqsubseteq_{r_0} c \end{array}$$

and hence by Definition 2.18 (stops) part (42), $stops(c,r_1) \Rrightarrow stops(c,r_0)$. For (44) from Definition 2.18 (stops) part (41)

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

and hence by Definition 2.18 (stops) part (42), $stops(c,r) \Rrightarrow stops(d,r)$. □

Lemma 2.20 (precondition-term) *For any predicate p, relation r and command c,*

$$stops(\{p\}c,r) \quad \equiv \quad p \wedge stops(c,r) \ .$$

Lemma 2.21 (specification-term) *For any relations q and r,*

$$\begin{array}{rcl} stops(\left[q\right],r) & \equiv & \mathsf{true}, \quad \textit{if } r \Rrightarrow \mathsf{id} \\ stops(\left[q\right],r) & \equiv & \mathsf{false}, \quad \textit{if } r \not\Rrightarrow \mathsf{id} \end{array}$$

Lemma 2.22 (sequential-term) *For any relation r and commands c_0 and c_1,*

$$stops(c_0 \, ; c_1, r) \quad = \quad stops(c_0\{stops(c_1,r)\}, r) \ .$$

Lemma 2.23 (atomic-term) *For any predicate p, and relations q and r, such that $r \Rrightarrow (p \Rightarrow p')$,*

$$stops(\langle p,q\rangle, r) \quad \equiv \quad p \ .$$

For an atomic step $\langle p,q\rangle$ not to abort, its precondition p must hold in the state in which the atomic program step takes place, which may be after a sequence of environment steps. If r preserves p and p holds initially, p holds after any number of environment steps that respect r, and hence $p \Rrightarrow stops(\langle p,q\rangle, r)$.

3 The guarantee command

The guarantee command $(\mathbf{guar}\, g \bullet c)$ constrains the possible implementations of the command c such that each program step must either satisfy the relation g or stutter. Intuition for its semantics is 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 11).

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.

$$\begin{array}{ll} & \mathbf{guar}\, x < x' \bullet \left[x' = x + 1\right] \\ \sqsubseteq & \text{by Law 3.38 (guarantee-assignment) as } x' = x + 1 \Rrightarrow x < x' \\ & x := x + 1 \end{array}$$

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

$$\begin{array}{ll} & \mathbf{guar}\, x < x' \bullet \left[x' = x + 2\right] \\ \sqsubseteq & \text{by Law 3.21 (guarantee-monotonic) as } \left[x' = x + 2\right] \sqsubseteq \left[x' = x + 1\right] ; \left[x' = x + 1\right] \\ & \mathbf{guar}\, x < x' \bullet (\left[x' = x + 1\right] ; \left[x' = x + 1\right]) \\ = & \text{by Law 3.27 (distribute-guarantee) over sequential (61)} \\ & (\mathbf{guar}\, x < x' \bullet \left[x' = x + 1\right]) ; (\mathbf{guar}\, x < x' \bullet \left[x' = x + 1\right]) \\ \sqsubseteq & \text{by example (1) above (twice)} \\ & x := x + 1 ; x := x + 1 \end{array}$$

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

$$\begin{array}{ll} & \mathbf{guar}\, x < x' \bullet \left[x' = x + 1 \lor x' = x - 1\right] \\ = & \text{by Law 3.31 (trading-post-guarantee) and as } (x < x')^* = (x \le x') \\ & \mathbf{guar}\, x < x' \bullet \left[(x' = x + 1 \lor x' = x - 1) \land x \le x'\right] \\ = & \mathbf{guar}\, x < x' \bullet \left[x' = x + 1\right] \end{array}$$

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

$$\begin{array}{ll} & \mathbf{guar}\, x < x' \bullet \left[x' = x - 1\right] \\ = & \text{by Law 3.31 (trading-post-guarantee) and as } (x < x')^* = (x \le x') \\ & \mathbf{guar}\, x < x' \bullet \left[x' = x - 1 \land (x \le x')\right] \\ = & \mathbf{guar}\, x < x' \bullet \left[\mathsf{false}\right] \\ = & \left[\mathsf{false}\right] \end{array}$$

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

Definition 3.1 (guarantee) *For any relation g and command c,* $\mathbf{guar}\, g \bullet c \;\widehat{=}\; \langle g \vee \mathsf{id} \rangle^{\omega} \Cap c$.

3.3 Properties of atomic steps and iterations

Law 3.2 (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 2.8 (consequence) for atomic steps (30) with Lemma 2.15 (iteration-monotonic). □

Law 3.3 (refine-iterated-relation) *For any relation g,* $\left[g^*\right] \sqsubseteq \langle g \rangle^*$.

Proof. The proof follows using Lemma 2.14 (iteration-induction) for finite iteration (38) provided the refinement, $\left[g^*\right] \sqsubseteq \mathbf{skip} \sqcap \langle g \rangle \mathbin{;} \left[g^*\right]$, holds, which is shown as follows.

$$
\begin{array}{ll}
 & \left[g^*\right] \\
= & \text{as } g^* = \mathsf{id} \vee (g \mathbin{⨾} g^*) \\
 & \left[\mathsf{id} \vee (g \mathbin{⨾} g^*)\right] \\
\sqsubseteq & \text{by Law 2.11 (nondeterministic-choice) and Lemma 2.9 (sequential)} \\
 & \left[\mathsf{id}\right] \sqcap \left[g\right] \mathbin{;} \left[g^*\right] \\
\sqsubseteq & \text{by Lemma 2.7 (make-atomic)} \\
 & \mathbf{skip} \sqcap \langle g \rangle \mathbin{;} \left[g^*\right] \ .
\end{array}
$$

□

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

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

Lemma 3.5 (conjunction-strict) *For any predicate p and commands c and d,*

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

Lemma 3.6 (conjunction-atomic) *For any predicates p_0 and p_1, relations q_0 and q_1, and commands c_0 and c_1,*

$$\mathbf{skip} \Cap \mathbf{skip} \quad = \quad \mathbf{skip} \tag{45}$$

$$\langle p_0, q_0 \rangle \Cap \langle p_1, q_1 \rangle \quad = \quad \langle p_0 \wedge p_1, q_0 \wedge q_1 \rangle \tag{46}$$

$$(\langle p_0, q_0 \rangle \mathbin{;} c_0) \Cap (\langle p_1, q_1 \rangle \mathbin{;} c_1) \quad = \quad (\langle p_0, q_0 \rangle \Cap \langle p_1, q_1 \rangle) \mathbin{;} (c_0 \Cap c_1) \tag{47}$$

$$(\langle p_0, q_0 \rangle \mathbin{;} c_0) \Cap \mathbf{skip} \quad = \quad \{p_0\}\mathbf{magic} \tag{48}$$

Strict conjunction does not distribute through parallel or sequential composition but the following interchange properties hold.

Lemma 3.7 (interchange-conjunction) *For any commands c_0, c_1, d_0 and d_1, the following hold.*

$$(c_0 \parallel c_1) \Cap (d_0 \parallel d_1) \quad \sqsubseteq \quad (c_0 \Cap d_0) \parallel (c_1 \Cap d_1) \tag{49}$$
$$(c_0 \mathbin{;} c_1) \Cap (d_0 \mathbin{;} d_1) \quad \sqsubseteq \quad (c_0 \Cap d_0) \mathbin{;} (c_1 \Cap d_1) \tag{50}$$

Note that (49) is a refinement rather than an equality because on the left behaviour of c_0 may synchronise with behaviour of either d_0 or d_1, whereas on the right it can only synchronise with behaviour of d_0; (50) is similar.

Lemma 3.8 (refine-conjunction) *For 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 3.9 (simplify-conjunction) *For commands c and d, if $c \sqsubseteq_r d$, $\quad c \Cap d \sqsubseteq_r d$.*

Proof. The proof follows from Lemma 3.8 (refine-conjunction) as $c \sqsubseteq_r d$ and $d \sqsubseteq_r d$. □

Law 3.10 (conjunction-term) *For any relation r and commands c_0 and c_1,*

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

Proof. Let $p \equiv stops(c_0, r) \wedge stops(c_1, r)$, by Definition 2.18 (stops) part (42) both $\{p\}\langle \mathsf{true} \rangle^* \sqsubseteq_r c_0$ and $\{p\}\langle \mathsf{true} \rangle^* \sqsubseteq_r c_1$ and hence by Lemma 3.4 (conjunction-monotonic)

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

and because "$\Cap$" is idempotent the left side reduces to $\{p\}\langle \mathsf{true} \rangle^*$, and hence by Definition 2.18 (stops) part (42), $p \Rrightarrow stops((c_0 \Cap c_1), r)$. □

For a local variable block $(\mathbf{var}\, x \bullet c)$, any program step of c that modifies only x becomes a stuttering step of the block but any steps that modify variables other than x are preserved.

Lemma 3.11 (no-change-local) *For any command c, and variable x,*

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

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

For a relation g that does not depend on x, if every program step of c satisfies g, then every program step of $(\mathbf{var}\, x \bullet c)$ also satisfies g. A relation g depends only on a set of variables X, if the effect of g in a state is independent of all variables other than X. Recall that $\mathsf{id}(X)$ is the identity relation on X; it allows chaos for variables other than X.

Definition 3.12 (depends-only) *For any relation g and set of variables X,*

$$depends\_only(g, X) \quad \widehat{=} \quad (\mathsf{id}(X) \mathbin{⨾} g \mathbin{⨾} \mathsf{id}(X)) \equiv g .$$

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

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 3.13 (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. depends_only$(g_1, \bar{x})$,*

$$c \Cap (d_0 \Cap d_1) \;=\; (c \Cap d_0) \Cap (c \Cap d_1) \tag{51}$$
$$c \Cap (\textstyle\bigsqcap D) \;=\; \textstyle\bigsqcap \{d \in D \bullet (c \Cap d)\} \tag{52}$$
$$\langle g\rangle^{\omega} \Cap (c \,;\, d) \;=\; (\langle g\rangle^{\omega} \Cap c) \,;\, (\langle g\rangle^{\omega} \Cap d) \tag{53}$$
$$\langle g\rangle^{\omega} \Cap (c \parallel d) \;=\; (\langle g\rangle^{\omega} \Cap c) \parallel (\langle g\rangle^{\omega} \Cap d) \tag{54}$$
$$\langle g_1\rangle^{\omega} \Cap (\mathbf{var}\, x \bullet c) \;=\; \mathbf{var}\, x \bullet (\langle g_1\rangle^{\omega} \Cap c) \tag{55}$$
$$\langle g\rangle^{\omega} \Cap (c^{\omega}) \;=\; (\langle g\rangle^{\omega} \Cap c)^{\omega} \tag{56}$$

Law 3.14 (conjunction-with-atomic) *For any relations g and q,* $\langle g\rangle^{\omega} \Cap \langle p, q\rangle \;=\; \langle p, g \wedge q\rangle$.

Proof.

$\langle g\rangle^{\omega} \Cap \langle p, q\rangle$
$=$ by Lemma 2.13 (fold/unfold-iteration)
$(\mathbf{skip} \sqcap \langle g\rangle \,;\, \langle g\rangle^{\omega}) \Cap \langle p, q\rangle$
$=$ by Lemma 3.13 (distribute-conjunction) over choice (52) and $\langle p, q\rangle = \langle p, q\rangle \,;\, \mathbf{skip}$
$(\mathbf{skip} \Cap \langle p, q\rangle) \sqcap (\langle g\rangle \,;\, \langle g\rangle^{\omega} \Cap \langle p, q\rangle \,;\, \mathbf{skip})$
$=$ by (48), (47) and (46) of Lemma 3.6 (conjunction-atomic); Lemma 2.13 (fold/unfold-iteration)
$(\{p\}\mathbf{magic}) \sqcap (\langle p, g \wedge q\rangle \,;\, ((\mathbf{skip} \sqcap \langle g\rangle \,;\, \langle g\rangle^{\omega}) \Cap \mathbf{skip}))$
$=$ by Lemma 3.13 (distribute-conjunction) over choice (52); (45)
$(\{p\}\mathbf{magic}) \sqcap (\langle p, g \wedge q\rangle \,;\, (\mathbf{skip} \sqcap (\langle g\rangle \,;\, \langle g\rangle^{\omega} \Cap \mathbf{skip}))$
$=$ by Lemma 3.6 (conjunction-atomic) part (48)
$(\{p\}\mathbf{magic}) \sqcap (\langle p, g \wedge q\rangle \,;\, (\mathbf{skip} \sqcap \mathbf{magic})$
$=$ as $\mathbf{skip} \sqcap \mathbf{magic} = \mathbf{skip}$ and $\langle p, g \wedge q\rangle \sqsubseteq \{p\}\mathbf{magic}$
$\langle p, g \wedge q\rangle$

□

Law 3.15 (terminating-iteration) *For any relation q,* $\langle \mathsf{true}\rangle^{*} \Cap \langle q\rangle^{\omega} \;=\; \langle q\rangle^{*}$.

Proof. Lemma 2.17 (fusion) can be applied by choosing $F = (\lambda x \bullet \mathbf{skip} \sqcap \langle \mathsf{true}\rangle \,;\, x)$ and hence $\nu F = \langle \mathsf{true}\rangle^{*}$, and choosing $G = (\lambda x \bullet \mathbf{skip} \sqcap \langle q\rangle \,;\! x)$, and hence $\nu G = \langle q\rangle^{*}$, and choosing $H = (\lambda x \bullet x \Cap \langle q\rangle^{\omega})$, and hence $H(\nu F) = \langle \mathsf{true}\rangle^{*} \Cap \langle q\rangle^{\omega}$. Both F and G are monotone because non-deterministic choice and sequential composition are monotone operators. By Lemma 2.17 (fusion) $\langle \mathsf{true}\rangle^{*} \Cap \langle q\rangle^{\omega} = \langle q\rangle^{*}$ if,

$$(\mathbf{skip} \sqcap \langle \mathsf{true}\rangle \,;\, x) \Cap \langle q\rangle^{\omega} \quad = \quad \mathbf{skip} \sqcap \langle q\rangle \,;\, (x \Cap \langle q\rangle^{\omega})$$

which we show as follows starting from the left side.

$(\mathbf{skip} \sqcap \langle \mathsf{true}\rangle \,;\, x) \Cap \langle q\rangle^{\omega}$
$=$ by Lemma 2.13 (fold/unfold-iteration)
$(\mathbf{skip} \sqcap \langle \mathsf{true}\rangle \,;\, x) \Cap (\mathbf{skip} \sqcap \langle q\rangle \,;\, \langle q\rangle^{\omega})$
$=$ by Lemma 3.13 (distribute-conjunction) over choice (52)
$(\mathbf{skip} \Cap \mathbf{skip}) \sqcap ((\langle \mathsf{true}\rangle \,;\, x) \Cap \mathbf{skip}) \sqcap (\mathbf{skip} \Cap (\langle q\rangle \,;\, \langle q\rangle^{\omega})) \sqcap ((\langle \mathsf{true}\rangle \,;\, x) \Cap (\langle q\rangle \,;\, \langle q\rangle^{\omega}))$
$=$ by Lemma 3.6 (conjunction-atomic) parts (45), (48) twice, (47) and (46)
$\mathbf{skip} \sqcap \mathbf{magic} \sqcap \mathbf{magic} \sqcap (\langle q\rangle \,;\, (x \Cap \langle q\rangle^{\omega}))$
$=$ as $\mathbf{skip} \sqcap \mathbf{magic} = \mathbf{skip}$
$\mathbf{skip} \sqcap (\langle q\rangle \,;\, (x \Cap \langle q\rangle^{\omega}))$

Function H is co-continuous by Lemma 3.13 (distribute-conjunction) over non-deterministic choice (52) and as $\mathbf{magic} \Cap \langle q\rangle^{\omega} = \mathbf{magic}$. □

Law 3.16 (conjunction-atomic-iterated) *For any relations g_0 and g_1 both the following hold.*

$$\langle g_0\rangle^{\omega} \Cap \langle g_1\rangle^{\omega} \;=\; \langle g_0 \wedge g_1\rangle^{\omega} \tag{57}$$
$$\langle g_0\rangle^{*} \Cap \langle g_1\rangle^{*} \;=\; \langle g_0 \wedge g_1\rangle^{*} \tag{58}$$

Proof. By Law 3.2 (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 Lemma 3.8 (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 (57) holds by Lemma 2.14 (iteration-induction) provided

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

which we show by expanding the right side as follows.

$$\begin{array}{ll}
& \langle g_0\rangle^\omega \Cap \langle g_1\rangle^\omega \\
= & \text{by Lemma 2.13 (fold/unfold-iteration) unfolding} \\
& (\mathbf{skip} \sqcap \langle g_0\rangle \mathbin{;} \langle g_0\rangle^\omega) \Cap (\mathbf{skip} \sqcap \langle g_1\rangle \mathbin{;} \langle g_1\rangle^\omega) \\
= & \text{by Lemma 3.13 (distribute-conjunction) over choice (52)} \\
& (\mathbf{skip} \Cap \mathbf{skip}) \sqcap (\mathbf{skip} \Cap \langle g_1\rangle \mathbin{;} \langle g_1\rangle^\omega) \sqcap ((\langle g_0\rangle \mathbin{;} \langle g_0\rangle^\omega \Cap \mathbf{skip}) \sqcap ((\langle g_0\rangle \mathbin{;} \langle g_0\rangle^\omega) \Cap (\langle g_1\rangle \mathbin{;} \langle g_1\rangle^\omega) \\
= & \text{by Lemma 3.6 (conjunction-atomic) parts (45), (48) twice, (47) and (46)} \\
& \mathbf{skip} \sqcap \mathbf{magic} \sqcap \mathbf{magic} \sqcap \langle g_0 \wedge g_1\rangle \mathbin{;} (\langle g_0\rangle^\omega \Cap \langle g_1\rangle^\omega) \\
= & \text{as } \mathbf{skip} \sqcap \mathbf{magic} = \mathbf{skip} \\
& \mathbf{skip} \sqcap \langle g_0 \wedge g_1\rangle \mathbin{;} (\langle g_0\rangle^\omega \Cap \langle g_1\rangle^\omega)
\end{array}$$

To prove (58) we start by using (57).

$$\begin{array}{ll}
& \langle g_0\rangle^\omega \Cap \langle g_1\rangle^\omega = \langle g_0 \wedge g_1\rangle^\omega \\
\Rightarrow & \text{by Lemma 3.4 (conjunction-monotonic)} \\
& \langle \mathsf{true}\rangle^* \Cap \langle g_0\rangle^\omega \Cap \langle g_1\rangle^\omega = \langle \mathsf{true}\rangle^* \Cap \langle g_0 \wedge g_1\rangle^\omega \\
\Rightarrow & \text{by Law 3.15 (terminating-iteration) as conjunction is idempotent and associative} \\
& \langle g_0\rangle^* \Cap \langle g_1\rangle^* = \langle g_0 \wedge g_1\rangle^*
\end{array}$$

□

Law 3.17 (conjunction-with-terminating) *For any predicate p, relations r and g, and command c, such that $p \Rrightarrow stops(c, r)$,*

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

Proof. Definition 2.18 (stops) part (42) gives $\{p\}\langle \mathsf{true}\rangle^* \sqsubseteq_r \{p\}c$, which is used below.

$$\begin{array}{ll}
& \{p\}\langle g\rangle^* \\
= & \text{by Law 3.15 (terminating-iteration) and Lemma 3.5 (conjunction-strict)} \\
& \langle g\rangle^\omega \Cap (\{p\}\langle \mathsf{true}\rangle^*) \\
\sqsubseteq_r & \text{by Lemma 3.4 (conjunction-monotonic) as } \{p\}\langle \mathsf{true}\rangle^* \sqsubseteq_r \{p\}c \\
& \langle g\rangle^\omega \Cap (\{p\}c)
\end{array}$$

□

Lemma 3.18 (conjoined-specifications) *For any relations q_0 and q_1,* $\quad [q_0 \wedge q_1] \; = \; [q_0] \Cap [q_1]$.

Proof. In any environment other than id both sides may abort and hence by Lemma 2.6 (refine-specification) one only needs to consider equivalence in environment id. A trace of $[q_0] \Cap [q_1]$ must be a trace of both $[q_0]$ and a trace of $[q_1]$, 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 $[q_0 \wedge q_1]$. By Lemma 2.8 (consequence) both $[q_0] \sqsubseteq [q_0 \wedge q_1]$ and $[q_1] \sqsubseteq [q_0 \wedge q_1]$ and hence by Lemma 3.8 (refine-conjunction) $[q_0] \Cap [q_1] \sqsubseteq [q_0 \wedge q_1]$. □

Law 3.19 (specification-finite) *For any relation q,* $\quad \langle \mathsf{true}\rangle^* \Cap [q] \;\sqsubseteq\; [q]$

Proof. By Lemma 2.21 (specification-term) $\langle \mathsf{true}\rangle^* \sqsubseteq_{\mathsf{id}} [q]$ and hence by Law 3.9 (simplify-conjunction) $\langle \mathsf{true}\rangle^* \Cap [q] \sqsubseteq_{\mathsf{id}} [q]$. In a non-$\mathsf{id}$ environment both sides abort and the refinement holds. □

3.5 Laws for refining guarantee commands

Law 3.20 (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 "$\Cap$".
$(\mathbf{guar}\,\mathsf{true} \bullet c) = \langle\mathsf{true} \vee \mathsf{id}\rangle^\omega \Cap c = \langle\mathsf{true}\rangle^\omega \Cap c = c$ □

Law 3.21 (guarantee-monotonic) *For any commands c and d, and relations g and r,*

$$c \sqsubseteq_r d \quad\Rightarrow\quad (\mathbf{guar}\,g \bullet c) \sqsubseteq_r (\mathbf{guar}\,g \bullet d)\ .$$

Proof. The proof relies on Lemma 3.4 (conjunction-monotonic). If $c \sqsubseteq_r d$,
$(\mathbf{guar}\,g \bullet c) = \langle g \vee \mathsf{id}\rangle^\omega \Cap c \sqsubseteq_r \langle g \vee \mathsf{id}\rangle^\omega \Cap d = (\mathbf{guar}\,g \bullet d)$. □

Law 3.22 (strengthen-guarantee) *For any command c and relations g_0 and g_1,*

$$(g_0 \Rrightarrow g_1 \vee \mathsf{id}) \quad\Rightarrow\quad (\mathbf{guar}\,g_1 \bullet c) \sqsubseteq (\mathbf{guar}\,g_0 \bullet c)\ .$$

Proof. The proof relies on Lemma 3.4 (conjunction-monotonic) and Law 3.2 (strengthen-iterated-atomic). Assume $g_0 \Rrightarrow g_1 \vee \mathsf{id}$,
$(\mathbf{guar}\,g_1 \bullet c) = \langle g_1 \vee \mathsf{id}\rangle^\omega \Cap c \sqsubseteq \langle g_0 \vee \mathsf{id}\rangle^\omega \Cap c = (\mathbf{guar}\,g_0 \bullet c)$. □

Law 3.23 (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 3.5 a strict conjunction aborts if either of its operands aborts,
$(\mathbf{guar}\,g \bullet \{p\}c) = \langle g \vee \mathsf{id}\rangle^\omega \Cap \{p\}c = \{p\}(\langle g \vee \mathsf{id}\rangle^\omega \Cap c) = \{p\}(\mathbf{guar}\,g \bullet c)$. □

Law 3.24 (introduce-guarantee) *For any command c and relation g,* $c \sqsubseteq (\mathbf{guar}\,g \bullet c)$.

Proof. The proof can be shown using Law 3.20 (guarantee-true) and Law 3.22 (strengthen-guarantee) as follows: $c = (\mathbf{guar}\,\mathsf{true} \bullet c) \sqsubseteq (\mathbf{guar}\,g \bullet c)$ □

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

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

Proof. The law holds by Definition 2.18 (stops) part (42) if $\{stops(c,r)\}\langle\mathsf{true}\rangle^* \sqsubseteq_r (\mathbf{guar}\,g \bullet c)$, which holds using Definition 2.18 (stops) part (41) and Law 3.24 (introduce-guarantee) as follows.
$\{stops(c,r)\}\langle\mathsf{true}\rangle^* \sqsubseteq_r c \sqsubseteq (\mathbf{guar}\,g \bullet c)$. □

Law 3.26 (nested-guarantees) *For a command c and relations g_0 and g_1,*

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

Proof. The proof relies on the property of relations: $(g_0 \vee \mathsf{id}) \wedge (g_1 \vee \mathsf{id}) = (g_0 \wedge g_1) \vee \mathsf{id}$.

$(\mathbf{guar}\,g_0 \bullet (\mathbf{guar}\,g_1 \bullet c))$
= by Definition 3.1 (guarantee) twice, "$\Cap$" is associative
$\langle g_0 \vee \mathsf{id}\rangle^\omega \Cap \langle g_1 \vee \mathsf{id}\rangle^\omega \Cap c$
= by Law 3.16 (conjunction-atomic-iterated) part (57) using the above property of relations
$\langle (g_0 \wedge g_1) \vee \mathsf{id}\rangle^\omega \Cap c$
= $(\mathbf{guar}\,g_0 \wedge g_1 \bullet c)$

□

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

Law 3.27 (distribute-guarantee) *For any relations g and g_1, commands c and d, set of commands C, and variable x such that g_1 does not depend on x, i.e. $depend\_only(g_1, \bar{x})$, the following hold.*

$$\mathbf{guar}\, g \bullet (c \Cap d) = (\mathbf{guar}\, g \bullet c) \Cap (\mathbf{guar}\, g \bullet d) \tag{59}$$

$$\mathbf{guar}\, g \bullet (\textstyle\bigsqcap C) = \textstyle\bigsqcap \{c \in C \bullet (\mathbf{guar}\, g \bullet c)\} \tag{60}$$

$$\mathbf{guar}\, g \bullet (c \,;\, d) = (\mathbf{guar}\, g \bullet c) \,;\, (\mathbf{guar}\, g \bullet d) \tag{61}$$

$$\mathbf{guar}\, g \bullet (c \parallel d) = (\mathbf{guar}\, g \bullet c) \parallel (\mathbf{guar}\, g \bullet d) \tag{62}$$

$$\mathbf{guar}\, g_1 \bullet (\mathbf{var}\, x \bullet c) = \mathbf{var}\, x \bullet (\mathbf{guar}\, g_1 \bullet c) \tag{63}$$

$$\mathbf{guar}\, g \bullet (c^\omega) = (\mathbf{guar}\, g \bullet c)^\omega \tag{64}$$

Proof. The proofs of (59)-(64) follow directly from Lemma 3.13 (distribute-conjunction) properties (51)-(56) respectively. □

A guarantee g on an atomic step that satisfies q must satisfy both q and the guarantee.

Law 3.28 (guarantee-atomic) *For any predicate p and relations g and q,*

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

Proof. The proof follows using Law 3.14 (conjunction-with-atomic).

$(\mathbf{guar}\, g \bullet \langle p, q \rangle) = \langle g \vee \mathsf{id} \rangle^\omega \Cap \langle p, q \rangle = \langle p, (g \vee \mathsf{id}) \wedge q \rangle$ □

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

Lemma 3.29 (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 3.30 (refine-in-context) *For any relations r_0 and r_1 and commands c_0 and c_1, if $c_0 \sqsubseteq_{r_0 \vee r_1} c_1$,*

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

Proof. Using Lemma 3.29 (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.

$c_0 \parallel (\mathbf{guar}\, r_0 \bullet \langle \mathsf{true} \rangle^*) \sqsubseteq_{r_0 \vee r_1} c_1 \parallel (\mathbf{guar}\, r_0 \bullet \langle \mathsf{true} \rangle^*)$

$\Leftrightarrow$ by Definition 3.1 (guarantee)

$c_0 \parallel (\langle r_0 \vee \mathsf{id} \rangle^\omega \Cap \langle \mathsf{true} \rangle^*) \sqsubseteq_{r_0 \vee r_1} c_1 \parallel (\langle r_0 \vee \mathsf{id} \rangle^\omega \Cap \langle \mathsf{true} \rangle^*)$

$\Leftrightarrow$ by Law 3.15 (terminating-iteration) twice

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

□

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 post condition ensuring g^* to be traded for a guarantee of g on every atomic step.

Law 3.31 (trading-post-guarantee) *For any relations g and q,*

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

Proof. By Lemma 2.8 (consequence) $[q] \sqsubseteq [g^* \wedge q]$ and hence by Law 3.21 (guarantee-monotonic) the refinement holds from right to left. The proof of refinement from left to right follows.

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

$=$ by Definition 3.1 (guarantee), Lemma 3.18 (conjoined-specifications)

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

$\sqsubseteq$ by Law 3.3 (refine-iterated-relation) and $(g \vee \mathsf{id})^* = g^*$
$\langle g \vee \mathsf{id}\rangle^\omega \Cap \langle g \vee \mathsf{id}\rangle^* \Cap [q]$
$=$ by Law 3.15 (terminating-iteration)
$\langle g \vee \mathsf{id}\rangle^\omega \Cap \langle g \vee \mathsf{id}\rangle^\omega \Cap \langle \mathsf{true}\rangle^* \Cap [q]$
$\sqsubseteq$ as "$\Cap$" is idempotent and by Law 3.19 (specification-finite)
$\langle g \vee \mathsf{id}\rangle^\omega \Cap [q]$
$=$ Definition 3.1 (guarantee)
$\mathbf{guar}\, g \bullet [q]$

□

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 infeasible and that trace of behaviour is eliminated. 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**)[3] and "conditional or" (**cor**), that evaluate their second operand depending on the evaluation of the first. Tests satisfy the following laws.

Lemma 3.32 (tests) *For any boolean expressions a and b the following hold.*

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

For arithmetic expressions e_0 and e_1,

$$[[e_0 < e_1]] \quad = \quad \bigsqcap \{v \in \mathit{Val}, w \in \mathit{Val} \mid v < w \bullet ([[e_0 = v]] \parallel [[e_1 = w]])\} \;.$$

Other relational operators are treated similarly. Assuming atomic access to a variable x, for any value v

$$[[x = v]] \quad = \quad \langle \mathsf{id}\rangle^* \,;\, \langle x = v \wedge \mathsf{id}\rangle \,;\, \langle \mathsf{id}\rangle^* \;.$$

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 3.33 (test-guarantee) *For any relation g and test predicate b,* $(\mathbf{guar}\, g \bullet [[b]]) = [[b]]$.

Proof. Refinement from right to left holds by Law 3.24 (introduce-guarantee). Because a test does not modify any variables $(\mathbf{guar}\,\mathsf{id} \bullet [[b]]) = [[b]]$, which is used in the following refinement.

$(\mathbf{guar}\, g \bullet [[b]])$
$\sqsubseteq$ by Law 3.22 (strengthen-guarantee) as $\mathsf{id} \Rrightarrow g \vee \mathsf{id}$
$(\mathbf{guar}\,\mathsf{id} \bullet [[b]])$
$= [[b]]$

□

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

Law 3.34 (guarantee-conditional) *For any relation g, boolean expression b, and commands c and d,*

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

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

Proof. The proof is mechanical: it expands the left side conditional using its definition (13), applies Law 3.27 (distribute-guarantee) to distribute the guarantee over the nondeterministic choice and sequential compositions, then applies Law 3.33 (test-guarantee) twice to eliminate the guarantee around the tests, and finally rewrites the result as a conditional using definition (13). □

Law 3.35 (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 (14), then applies Law 3.27 (distribute-guarantee) to distribute the guarantee over the sequential composition and iteration, then applies Law 3.33 (test-guarantee) to eliminate the guarantees around the tests, and finally rewrites the result as a while loop using definition (14). □

3.7 Frames and assignment

The refinement calculus as described by [Mor88] 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 $\mathsf{id}(\bar{x})$ to stand for the identity relation on all variables other than x, in Morgan's sequential refinement calculus $x\colon [q]$ is defined as $[q \wedge \mathsf{id}(\bar{x})]$, 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,

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

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 $\mathsf{id}(\bar{x})$ a guarantee, i.e. $(\mathbf{guar}\, \mathsf{id}(\bar{x}) \bullet [x' = y + 1])$, the above refinement is no longer valid. Hence, rather than Morgan's definition above, a frame on a specification satisfies the following.

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

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

Definition 3.36 (frame) *For any set of variables x and command c,* $\quad x : c \ \widehat{=}\ (\mathbf{guar}\, \mathsf{id}(\bar{x}) \bullet c)$.

The importance of framing in reasoning about concurrency is highlighted by framing being 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 3.26 (nested-guarantees)

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

Law 3.37 (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 3.36 (frame) and Law 3.26 (nested-guarantees).

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

□

An assignment is defined in terms of a test $[[e = v]]$ representing the expression evaluation, followed by an atomic update of x (15).

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

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 $x' = e \wedge \mathsf{id}(\bar{x})$ and hence this relation must imply $g \vee \mathsf{id}$ in order for the assignment to implement the guarantee g.

Law 3.38 (guarantee-assignment) *For any precondition p, relation g, variable x and expression e, such that $p \Rrightarrow def(e)$ and such that $p \wedge x' = e \wedge \mathsf{id}(\bar{x}) \Rrightarrow q \wedge (g \vee \mathsf{id})$,*

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

Proof.

$$\begin{array}{ll}
 & \mathbf{guar}\, g \bullet [p,\ q] \\
\sqsubseteq & \text{by Lemma 2.8 (consequence) as } p \wedge x' = e \wedge \mathsf{id}(\bar{x}) \Rrightarrow q \\
 & \mathbf{guar}\, g \bullet [p,\ x' = e \wedge \mathsf{id}(\bar{x})] \\
\sqsubseteq & \text{by Law 2.11 (nondeterministic-choice) and Lemma 2.9 (sequential)} \\
 & \mathbf{guar}\, g \bullet \bigsqcap \{v \in Val \bullet [p,\ p \wedge v = e \wedge \mathsf{id}]\ ;\ [p \wedge v = e,\ x' = v \wedge \mathsf{id}(\bar{x})]\} \\
\sqsubseteq & \text{by Law 3.27 (distribute-guarantee) into choice (60) and sequential (61)} \\
 & \bigsqcap \{v \in Val \bullet (\mathbf{guar}\, g \bullet [p,\ p \wedge v = e \wedge \mathsf{id}])\ ;\ (\mathbf{guar}\, g \bullet [p \wedge v = e,\ x' = v \wedge \mathsf{id}(\bar{x})])\} \\
\sqsubseteq & \text{by Lemma 2.12 (introduce-test) as } p \Rrightarrow def(e)\text{; Lemma 2.7 (make-atomic)} \\
 & \bigsqcap \{v \in Val \bullet (\mathbf{guar}\, g \bullet [[e = v]])\ ;\ (\mathbf{guar}\, g \bullet \langle p \wedge v = e, x' = v \wedge \mathsf{id}(\bar{x})\rangle)\} \\
\sqsubseteq & \text{by Law 3.33 (test-guarantee); Law 3.28 (guarantee-atomic) as } p \wedge x' = e \wedge \mathsf{id}(\bar{x}) \Rrightarrow (g \vee \mathsf{id}) \\
 & \bigsqcap \{v \in Val \bullet [[e = v]]\ ;\ \langle x' = v \wedge \mathsf{id}(\bar{x})\rangle\} \\
= & \text{by the definition of an assignment (15)} \\
 & x := e
\end{array}$$

□

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

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

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 3.39 (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.[4]

Law 3.40 (guarantee-variable) *For any command c and variable x,*

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

[4]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.

Proof. Because $(\mathsf{id}(x) \vee \mathsf{id}) = \mathsf{id}(x)$, from Definition 3.1 (guarantee),

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

and the proof follows by Lemma 3.11 (no-change-local). □

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 3.41 (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

$$(p \Rightarrow p')^* \equiv (p \Rightarrow p') \tag{65}$$

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 3.31 (trading-post-guarantee) to give the following law.

Law 3.42 (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.

$$\begin{array}{ll}
 & [p,\ p' \wedge q] \\
\sqsubseteq & \text{by Lemma 2.8 (consequence) using (65); Law 3.24 (introduce-guarantee)} \\
 & \mathbf{guar}(p \Rightarrow p') \bullet [p,\ (p \Rightarrow p')^* \wedge q] \\
= & \text{by Law 3.31 (trading-post-guarantee); Definition 3.41 (guarantee-invariant)} \\
 & \mathbf{guar\text{-}inv}\, p \bullet [p,\ q]
\end{array}$$

□

3.10 Extended example (sequential version)

Sect. 11 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,[5] or if p does not hold for any element of v, to set t to $len(v) + 1$. As shown in Sect. 11, the unconventional approach using guarantee invariants is useful when deriving a concurrent implementation from the same specification. To avoid repetition, a refinement with no explicit left side applies to the most recent command marked by "◁" and hence the specification of the *findp* program is marked with "◁" as the starting point of the refinement.

$$\mathit{findp} \mathrel{\widehat{=}} t\colon \left[(t' = len(v) + 1 \vee satp(v, t')) \wedge notp(v, \mathrm{dom}(v), t')\right] \qquad \lhd$$

[5]For brevity, it is assumed here that $p(x)$ is always defined (undefinedness is considered by [CJ07] but it has little bearing on the actual design).

where

$$\begin{array}{rcl} satp(v,t) & \mathrel{\widehat{=}} & t \in \mathrm{dom}(v) \wedge p(v(t)) \\ notp(v,s,t) & \mathrel{\widehat{=}} & (\forall i \in s \bullet i < t \Rightarrow \neg p(v(i))) \end{array}$$

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 ($t = len(v)+1 \vee satp(v,t)$) and an initialisation of t that establishes the invariant, using Lemma 2.9 (sequential) and Law 3.38 (guarantee-assignment) to handle the frame.

$$\begin{array}{l} \sqsubseteq t := len(v) + 1; \\ \quad t\colon \big[t = len(v) + 1 \vee satp(v,t),\ (t' = len(v) + 1 \vee satp(v,t')) \wedge notp(v, \mathrm{dom}(v), t')\big] \end{array} \qquad \lhd$$

The second specification can be refined using Law 3.42 (trade-guarantee-invariant).

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

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 3.39 (introduce-variable) for c and initialise it using Lemma 2.9 (sequential), and Law 3.38 (guarantee-assignment).

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

This can be refined using Law 3.42 (trade-guarantee-invariant).

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

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 $t - c$. We use the sequential refinement calculus rule for introducing a "while" loop.[6]

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

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 $p(v(c))$. We use the sequential refinement calculus rule for introducing a conditional.[6] Hence the body of the loop is refined by

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

[6] The rules for introducing "while" loops and conditionals are not given here; they are special cases of Law 10.6 (rely-loop) and Law 10.2 (rely-conditional) given later.

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

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

The branches of the conditional can be refined using Law 3.38 (guarantee-assignment) to give the following final program.

```
$t := len(v) + 1;$
(var $c \bullet$  $c := 1;$
          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. [Jon81, Jon83] 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 $(\mathbf{rely}\, r \bullet c)$ is introduced; it is the most general command that 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 $\langle r \vee \mathsf{id}\rangle^*$, 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. $p \Rrightarrow stops(c, \mathsf{id})$) the rely command satisfies the following property.

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

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 weakest environment of [CH81] and [Cha82], and the weakest pre-specification of [HH86], although the latter deals 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. $(\mathbf{rely}\, x < x' \bullet \big[x + 1 \le x'\big])$ 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. $(\mathbf{rely}\, x < x' \bullet \big[x' = x\big])$ 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, **skip**, 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. $(\mathbf{rely}\, x = x' \bullet \big[x' = x + 1\big])$ 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. $(\mathbf{rely}\, x = x' \wedge y = y' \bullet \lceil x' = x + 1 \rceil)$ 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 example 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 4.1 (parallel-interference) *For any relations r_0 and r_1,* $\langle r_0 \rangle^* \parallel \langle r_1 \rangle^* \ =\ \langle r_0 \vee r_1 \rangle^*$.

Repeated interference is equivalent to interference.

Lemma 4.2 (repeated-interference) *For any relation r,* $\langle r \rangle^* \,;\, \langle r \rangle^* = \langle r \rangle^*$.

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

Lemma 4.3 (interference-atomic) *For any predicate p and relations q and r,*

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

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, one can only distribute interference (rather than an arbitrary command).

Lemma 4.4 (distribute-parallel) *For any commands c_0, c_1 and d, set of commands C, and relation r,*

$$(\textstyle\bigsqcap C) \parallel d \quad = \quad \bigsqcap\{c \in C \bullet (c \parallel d)\} \tag{66}$$

$$(c_0 \,;\, c_1) \parallel \langle r \rangle^* \quad = \quad (c_0 \parallel \langle r \rangle^*) \,;\, (c_1 \parallel \langle r \rangle^*) \tag{67}$$

$$(c_0 \parallel c_1) \parallel \langle r \rangle^* \quad = \quad (c_0 \parallel \langle r \rangle^*) \parallel (c_1 \parallel \langle r \rangle^*) \tag{68}$$

Law 4.5 (term-in-context) *For any relations r_0 and r_1, and command c,*

$$stops(c, r_0 \vee r_1) \quad \equiv \quad stops(c \parallel \langle r_1 \vee \mathsf{id} \rangle^*, r_0 \vee r_1) .$$

Proof. The implication from right to left holds by Law 2.19 (term-monotonic) because $\langle r \vee \mathsf{id} \rangle^* \sqsubseteq \mathbf{skip}$. For the implication from left to right, by Definition 2.18 (stops) part (42) it is sufficient to show

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

which holds as follows.

$\quad \{stops(c, r_0 \vee r_1)\}\langle \mathsf{true} \rangle^*$
$\sqsubseteq$ by Lemma 4.1 (parallel-interference) and Lemma 2.5 (parallel-precondition)
$\quad \{stops(c, r_0 \vee r_1)\}\langle \mathsf{true} \rangle^* \parallel \langle r_1 \vee \mathsf{id} \rangle^*$
$\sqsubseteq_{r_0 \vee r_1}$ by Law 3.30 (refine-in-context) as by (41) $\{stops(c, r_0 \vee r_1)\}\langle \mathsf{true} \rangle^* \sqsubseteq_{r_0 \vee r_1} c$
$\quad c \parallel \langle r_1 \vee \mathsf{id} \rangle^*$

$\square$

4.3 Fundamental properties of rely

Our theory makes use of a generalisation of the rely command discussed above. The more general command $(\textbf{rely}\, r \bullet c_z)$ 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, $(\textbf{rely}\, r \bullet c_z)$ is the most general command that implements c in an environment bounded by z if started in a state from which c terminates in environment z, i.e.

$$\{stops(c, z)\}c \quad \sqsubseteq_z \quad (\textbf{rely}\, r \bullet c_z) \parallel \langle r \vee \mathsf{id}\rangle^* \ . \tag{69}$$

The additional parameter z to the rely command is necessary to determine the environment in which the refinement (69) is valid and the states from which the rely command is required to terminate, i.e. $stops(c, z)$. The default value for the relation z is id, i.e.

$$(\textbf{rely}\, r \bullet c) \quad \widehat{=} \quad (\textbf{rely}\, r \bullet c_{\mathsf{id}}) \tag{70}$$

Condition (69) does not cover the case of $(\textbf{rely}\, r \bullet c_z)$ failing to terminate in the presence of infinite interference from its environment. For example, the left loop in

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

is guaranteed to terminate in the presence of finite interference satisfying $(i' = i + 1 \vee \mathsf{id})$ but not in the presence of potentially infinite interference as represented by the right loop. Hence condition (69) 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 $(\textbf{rely}\, r \bullet [p,\ q])$ must be guaranteed to terminate from states satisfying p in an environment that satisfies r. Hence the termination condition that the rely command should satisfy is captured by the following.

$$stops((\textbf{rely}\, r \bullet [p,\ q]), r) \equiv stops([p,\ q]\, , \mathsf{id}) \equiv p \tag{71}$$

More generally, if the specification is included in nested relies, the following should hold.

$$\begin{array}{rcl} p & \equiv & stops([p,\ q]\, , \mathsf{id}) \\ & \equiv & stops((\textbf{rely}\, r_1 \bullet [p,\ q]), r_1) \\ & \equiv & stops((\textbf{rely}\, r_0 \bullet (\textbf{rely}\, r_1 \bullet [p,\ q])_{r_1}), r_0 \vee r_1) \end{array}$$

The definition of the rely command below is designed so that these termination conditions hold (they follow from Law 4.7 (rely-stops) below).

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

Definition 4.6 (rely) *For any command c and relations r and z,*

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

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 4.6 (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 4.7 (rely-stops) *For any relations z and r and command c,*

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

Proof.

$$\begin{array}{ll}
& stops((\mathbf{rely}\, r \bullet c_z), z \vee r) \\
\equiv & \text{by Definition 4.6 (rely)} \\
& stops((\bigsqcap\{d \mid (\{stops(c,z)\}c \sqsubseteq_z d \parallel \langle r \vee \mathsf{id}\rangle^*) \wedge (stops(c,z) \Rrightarrow stops(d, z \vee r))\}), z \vee r) \\
\equiv & \text{termination of non-deterministic choice} \\
& \forall d \bullet (\{stops(c,z)\}c \sqsubseteq_z d \parallel \langle r \vee \mathsf{id}\rangle^*) \wedge (stops(c,z) \Rrightarrow stops(d, z \vee r)) \Rightarrow stops(d, z \vee r) \\
\equiv & stops(c,z)
\end{array}$$

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

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 $\langle r \vee \mathsf{id}\rangle^*$, 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 4.8 (rely-refinement) *For any relations z and r, and commands c and d,*

$$((\mathbf{rely}\, r \bullet c_z) \sqsubseteq d) \quad \Leftrightarrow \quad (\{stops(c,z)\}c \sqsubseteq_z d \parallel \langle r \vee \mathsf{id}\rangle^*) \wedge (stops(c,z) \Rrightarrow stops(d, z \vee r))\ .$$

Proof. The reverse implication holds as follows.

$$\begin{array}{ll}
& (\mathbf{rely}\, r \bullet c_z) \sqsubseteq d \\
\Leftrightarrow & \text{by Definition 4.6 (rely)} \\
& \bigsqcap\{d_0 \mid (\{stops(c,z)\}c \sqsubseteq_z d_0 \parallel \langle r \vee \mathsf{id}\rangle^*) \wedge (stops(c,z) \Rrightarrow stops(d_0, z \vee r))\} \sqsubseteq d \\
\Leftarrow & \text{by Law 2.11 (nondeterministic-choice) part (33)} \\
& \exists d_0 \bullet (\{stops(c,z)\}c \sqsubseteq_z d_0 \parallel \langle r \vee \mathsf{id}\rangle^*) \wedge (stops(c,z) \Rrightarrow stops(d_0, z \vee r)) \wedge (d_0 \sqsubseteq d) \\
\Leftarrow & \text{a witness for the existential quantifier is } d \\
& (\{stops(c,z)\}c \sqsubseteq_z d \parallel \langle r \vee \mathsf{id}\rangle^*) \wedge (stops(c,z) \Rrightarrow stops(d, z \vee r))
\end{array}$$

For the forward implication by Law 2.3 (refinement-monotonic)

$$((\mathbf{rely}\, r \bullet c_z) \sqsubseteq d) \quad \Rightarrow \quad ((\mathbf{rely}\, r \bullet c_z) \sqsubseteq_{z \vee r} d) \tag{72}$$

and hence it is sufficient to show the following.

$$((\mathbf{rely}\, r \bullet c_z) \sqsubseteq_{z \vee r} d) \Rightarrow (\{stops(c,z)\}c \sqsubseteq_z d \parallel \langle r \vee \mathsf{id}\rangle^*) \wedge (stops(c,z) \Rrightarrow stops(d, z \vee r)) \tag{73}$$

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

$$\begin{array}{ll}
& \{p\}c \sqsubseteq_z d \parallel \langle r \vee \mathsf{id}\rangle^* \\
\Leftarrow & \text{by Law 3.30 (refine-in-context) as } (\mathbf{rely}\, r \bullet c_z) \sqsubseteq_{z \vee r} d \\
& \{p\}c \sqsubseteq_z (\mathbf{rely}\, r \bullet c_z) \parallel \langle r \vee \mathsf{id}\rangle^* \\
\Leftrightarrow & \text{by Definition 4.6 (rely)} \\
& \{p\}c \sqsubseteq_z \bigsqcap\{d_1 \mid (\{p\}c \sqsubseteq_z d_1 \parallel \langle r \vee \mathsf{id}\rangle^*) \wedge (p \Rrightarrow stops(d_1, z \vee r))\} \parallel \langle r \vee \mathsf{id}\rangle^* \\
\Leftrightarrow & \text{by Lemma 4.4 (distribute-parallel) part (66)} \\
& \{p\}c \sqsubseteq_z \bigsqcap\{d_1 \mid (\{p\}c \sqsubseteq_z d_1 \parallel \langle r \vee \mathsf{id}\rangle^*) \wedge (p \Rrightarrow stops(d_1, z \vee r)) \bullet (d_1 \parallel \langle r \vee \mathsf{id}\rangle^*)\}^{7} \\
\Leftarrow & \text{by Law 2.11 (nondeterministic-choice)} \\
& \forall d_1 \bullet (\{p\}c \sqsubseteq_z d_1 \parallel \langle r \vee \mathsf{id}\rangle^*) \wedge (p \Rrightarrow stops(d_1, z \vee r)) \Rightarrow (\{p\}c \sqsubseteq_z d_1 \parallel \langle r \vee \mathsf{id}\rangle^*)
\end{array}$$

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

$$\begin{array}{ll}
& stops(d, z \vee r) \\
\Lleftarrow & \text{by Law 2.19 (term-monotonic) as } (\mathbf{rely}\, r \bullet c_z) \sqsubseteq_{z \vee r} d \\
& stops((\mathbf{rely}\, r \bullet c_z), z \vee r) \\
\equiv & \text{by Law 4.7 (rely-stops)} \\
& stops(c,z)
\end{array}$$

[7] Recall that the notation $\{d \mid p \bullet e\}$ stands for the set of values of the expression e for d ranging over values that satisfy the predicate p.

□

Law 4.9 (rely-environment) *For any relations z and r, and commands c and d,*

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

Proof. The forward implication holds by Law 2.3 (refinement-monotonic). For the reverse implication by Law 4.8 (rely-refinement) it is sufficient to show

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

which was shown as property (73) above. □

Law 4.9 (rely-environment) allows one to use the stronger "$\sqsubseteq$" instead of "$\sqsubseteq_{z\vee r}$" when dealing with rely commands.

Law 4.10 (rely-refinement-precondition) *For any predicate p, relations z and r, and commands c and d, such that $p \Rrightarrow stops(c,z) \wedge stops(d, z \vee r)$,*

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

Proof. Because $stops(\{p\}c, z) \equiv p \wedge stops(c,z) \equiv p \Rrightarrow stops(d, z \vee r)$, the law follows by Law 4.8 (rely-refinement) . □

Law 4.11 (rely-specification) *For any predicate p, relations q and r, and command d, such that $p \Rrightarrow stops(d,r)$,*

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

Proof. The law follows from Law 4.10 (rely-refinement-precondition) and Lemma 2.6 (refine-specification) because $stops(\{p\}\ [q]\ , \mathsf{id}) \equiv p \wedge stops([q]\ , \mathsf{id}) \equiv p$ by Lemma 2.20 (precondition-term) and Lemma 2.21 (specification-term). □

Law 4.12 (rely) *For any predicate p, relations z and r, and command c, such that $p \Rrightarrow stops(c,z)$,*

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

Proof. Because by Law 4.7 (rely-stops), $stops((\mathbf{rely}\, r \bullet \{p\}c_z), z \vee r) \equiv stops(\{p\}c, z) \equiv p$, the law follows from Law 4.10 (rely-refinement-precondition) by taking d to be $(\mathbf{rely}\, r \bullet \{p\}c_z)$. □

Law 4.13 (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 4.6 (rely) the law holds provided

$$\begin{array}{l}
\bigsqcap\{d_1 \mid (\{stops(c,z)\}c \sqsubseteq_z d_1 \parallel \langle r \vee \mathsf{id}\rangle^*) \wedge (stops(c,z) \Rrightarrow stops(d_1, z \vee r))\} \sqsubseteq_{rx} \\
\quad \bigsqcap\{d_0 \mid (\{stops(c,z)\}c \sqsubseteq_z d_0 \parallel \langle (rx \wedge r) \vee \mathsf{id}\rangle^*) \wedge (stops(c,z) \Rrightarrow stops(d_0, z \vee (rx \wedge r)))\} \\
\Leftarrow \quad \text{by Law 2.11 (nondeterministic-choice)} \\
\quad \forall d_0 \bullet (\{stops(c,z)\}c \sqsubseteq_z d_0 \parallel \langle (rx \wedge r) \vee \mathsf{id}\rangle^*) \wedge (stops(c,z) \Rrightarrow stops(d_0, z \vee (rx \wedge r))) \Rightarrow \\
\qquad \exists d_1 \bullet (\{stops(c,z)\}c \sqsubseteq_z d_1 \parallel \langle r \vee \mathsf{id}\rangle^*) \wedge (stops(c,z) \Rrightarrow stops(d_1, z \vee r)) \wedge d_1 \sqsubseteq_{rx} d_0
\end{array}$$

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 $rx \vee \mathsf{id}$, that is, $[\![d_1]\!] = [\![d_0]\!]_{rx}$. By Definition 2.1 (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 $rx \vee \mathsf{id}$, it follows that

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

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

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 [CJ07]) 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 4.8 (rely-refinement) requires one to show $(\mathbf{guar}\, g \bullet [p,\ q]) \sqsubseteq d \parallel \langle r \vee \mathsf{id}\rangle^*$, 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 [CJ00].

Law 4.14 (guarantee-plus-rely) *For any relations g, z and r, and commands c and d,*

$$(\mathbf{guar}\, g \bullet (\mathbf{rely}\, r \bullet c_z)) \quad \sqsubseteq \quad d, \tag{74}$$

if both the following hold

$$(\mathbf{rely}\, r \bullet c_z) \quad \sqsubseteq \quad d \tag{75}$$

$$(\mathbf{guar}\, g \bullet d) \quad \sqsubseteq \quad d\,. \tag{76}$$

Proof. Applying Law 3.21 (guarantee-monotonic) to (75) gives the following.

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

When combined with (76) this implies (74). □

4.4 Laws for refining rely commands

Law 4.15 (weaken-rely) *For any command c, and any relations z, r_0 and r_1, such that $r_0 \Rrightarrow r_1 \vee \mathsf{id}$,*

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

Proof. By Law 4.8 (rely-refinement) the theorem holds provided

$$(\{stops(c,z)\}c \sqsubseteq_z (\mathbf{rely}\, r_1 \bullet c_z) \parallel \langle r_0 \vee \mathsf{id}\rangle^*) \wedge (stops(c,z) \Rrightarrow stops((\mathbf{rely}\, r_1 \bullet c_z), z \vee r_0))$$

$\Lleftarrow$ Law 3.2 (strengthen-iterated-atomic) and Law 2.19 (term-monotonic) as $r_0 \Rrightarrow r_1 \vee \mathsf{id}$

$$(\{stops(c,z)\}c \sqsubseteq_z (\mathbf{rely}\, r_1 \bullet c_z) \parallel \langle r_1 \vee \mathsf{id}\rangle^*) \wedge (stops(c,z) \Rrightarrow stops((\mathbf{rely}\, r_1 \bullet c_z), z \vee r_1)$$

which follow by Law 4.12 (rely) and Law 4.7 (rely-stops). □

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

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

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 4.16 (rely-monotonic) *For relations z and r, and commands c and d, such that $\{stops(c,z)\}c \sqsubseteq_z d$,*

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

Proof. The theorem holds by Law 4.8 (rely-refinement) provided the following holds.

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

$\Lleftarrow$ by Law 4.12 (rely), $\{stops(d,z)\}d \sqsubseteq_z (\mathbf{rely}\, r \bullet d_z) \parallel \langle r \vee \mathsf{id}\rangle^*$ and Law 4.7 (rely-stops)

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

By assumption $\{stops(c,z)\}c \sqsubseteq_z d$ and hence by Law 2.19 (term-monotonic) $stops(c,z) \Rrightarrow stops(d,z)$. □

Note that refining the body of a rely command does not necessarily preserve feasibility. For example, $\left[x = 0,\ x < x'\right] \sqsubseteq \left[x' = 1\right]$ is a valid refinement but while $(\mathbf{rely}\, x \leq x' \bullet \left[x = 0,\ x < x'\right])$ is feasible, $(\mathbf{rely}\, x \leq x' \bullet \left[x' = 1\right])$ is not feasible because the interference may increase x beyond one.

Law 4.17 (rely-precondition) *For any predicate p, relations z and r, and command c, if* $p \Rrightarrow stops(c,z)$*,*

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

Proof. Refinement from right to left is simply removing the precondition. Refinement from left to right holds by Law 4.10 (rely-refinement-precondition) provided both the following hold.

$$\{p\}c \quad \sqsubseteq_z \quad (\{p\}(\mathbf{rely}\, r \bullet \{p\}c_z)) \parallel \langle r \vee \mathsf{id}\rangle^* \tag{77}$$

$$p \quad \Rrightarrow \quad stops(\{p\}(\mathbf{rely}\, r \bullet \{p\}c_z), z \vee r) \tag{78}$$

Because by Lemma 2.4 (precondition) part (27), $\{p\} = \{p\}\{p\}$, (77) can be shown as follows.

$$\begin{array}{ll}
 & \{p\}\{p\}c \\
\sqsubseteq_z & \text{by Law 4.12 (rely) using assumption } p \Rrightarrow stops(c,z) \\
 & \{p\}((\mathbf{rely}\, r \bullet \{p\}c_z) \parallel \langle r \vee \mathsf{id}\rangle^*) \\
\sqsubseteq & \text{by Lemma 2.5 (parallel-precondition)} \\
 & (\{p\}(\mathbf{rely}\, r \bullet \{p\}c_z)) \parallel \langle r \vee \mathsf{id}\rangle^*
\end{array}$$

One can use Law 4.7 (rely-stops) to show (78) as follows.

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

□

Law 4.18 (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 $stops(d, z \vee r_0 \vee r_1) \Rrightarrow stops(d \parallel \langle r_0 \vee \mathsf{id}\rangle^*, z \vee r_1)$ by Law 4.5 (term-in-context).

$$\begin{array}{ll}
 & (\mathbf{rely}\, r_0 \bullet (\mathbf{rely}\, r_1 \bullet c_z)_{z \vee r_1}) \\
= & \text{by Definition 4.6 (rely)} \\
 & \bigsqcap\{d \mid \ (\{stops((\mathbf{rely}\, r_1 \bullet c_z), z \vee r_1)\}(\mathbf{rely}\, r_1 \bullet c_z) \sqsubseteq_{z \vee r_1} d \parallel \langle r_0 \vee \mathsf{id}\rangle^*) \wedge \\
 & \qquad (stops((\mathbf{rely}\, r_1 \bullet c_z), z \vee r_1) \Rrightarrow stops(d, z \vee r_0 \vee r_1))\} \\
= & \text{by Law 4.7 (rely-stops)} \\
 & \bigsqcap\{d \mid (\{stops(c,z)\}(\mathbf{rely}\, r_1 \bullet c_z) \sqsubseteq_{z \vee r_1} d \parallel \langle r_0 \vee \mathsf{id}\rangle^*) \wedge (stops(c,z) \Rrightarrow stops(d, z \vee r_0 \vee r_1))\} \\
= & \text{by Law 4.17 (rely-precondition) and Law 4.8 (rely-refinement)} \\
 & \bigsqcap\{d \mid \ (\{stops(c,z)\}c \sqsubseteq_z d \parallel \langle r_0 \vee \mathsf{id}\rangle^* \parallel \langle r_1 \vee \mathsf{id}\rangle^*) \wedge \\
 & \qquad (stops(c,z) \Rrightarrow stops(d \parallel \langle r_0 \vee \mathsf{id}\rangle^*, z \vee r_1)) \wedge (stops(c,z) \Rrightarrow stops(d, z \vee r_0 \vee r_1))\} \\
= & \text{by Lemma 4.1 (parallel-interference) and Law 4.5 (term-in-context) – see above} \\
 & \bigsqcap\{d \mid \ (\{stops(c,z)\}c \sqsubseteq_z d \parallel \langle r_0 \vee r_1 \vee \mathsf{id}\rangle^*) \wedge (stops(c,z) \Rrightarrow stops(d, z \vee r_0 \vee r_1))\} \\
= & \text{by Definition 4.6 (rely)} \\
 & (\mathbf{rely}\, r_0 \vee r_1 \bullet c_z)
\end{array}$$

□

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

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

Proof. By Law 4.8 (rely-refinement) we must show both the following.

$$\{stops(\textstyle\bigsqcap C, z)\}(\textstyle\bigsqcap C) \quad \sqsubseteq_z \quad (\textstyle\bigsqcap\{c \in C \bullet (\mathbf{rely}\, r \bullet c_z)\}) \parallel \langle r \vee \mathsf{id}\rangle^* \tag{79}$$
$$stops(\textstyle\bigsqcap C, z) \quad \Rrightarrow \quad stops(\textstyle\bigsqcap\{c \in C \bullet (\mathbf{rely}\, r \bullet c_z)\}, z \vee r) \tag{80}$$

The proof of (79) follows starting from the right side.

$(\bigsqcap\{c \in C \bullet (\mathbf{rely}\, r \bullet c_z)\}) \parallel \langle r \vee \mathsf{id}\rangle^*$
$=$ by Lemma 4.4 (distribute-parallel) over nondeterministic choice (66)
$\bigsqcap\{c \in C \bullet (\mathbf{rely}\, r \bullet c_z) \parallel \langle r \vee \mathsf{id}\rangle^*\}$
$\sqsupseteq_z$ by Law 2.11 (nondeterministic-choice) part (32) and Law 4.12 (rely)
$\bigsqcap\{c \in C \bullet \{stops(c, z)\}c\}$
$= \{stops(\bigsqcap C, z)\}(\bigsqcap C)$

The proof of (80) follows.

$stops(\bigsqcap\{c \in C \bullet (\mathbf{rely}\, r \bullet c_z)\}, z \vee r)$
$\equiv$ termination of a nondeterministic choice
$(\forall c \in C \bullet stops((\mathbf{rely}\, r \bullet c_z), z \vee r))$
$\equiv$ by Law 4.7 (rely-stops)
$(\forall c \in C \bullet stops(c, z))$
$\equiv$ termination of nondeterministic choice
$stops(\bigsqcap C, z)$

□

Law 4.20 (rely-post-assertion)

$$(\mathbf{rely}\, r \bullet (c\{p\})_z) \quad \sqsubseteq \quad (\mathbf{rely}\, r \bullet c_z)\{p\}$$

Proof. By Law 4.8 (rely-refinement) one must show both the following.

$$\{stops(c\{p\}, z)\}c\{p\} \quad \sqsubseteq_z \quad (\mathbf{rely}\, r \bullet c_z)\{p\} \parallel \langle r \vee \mathsf{id}\rangle^* \tag{81}$$
$$stops(c\{p\}, z) \quad \Rrightarrow \quad stops((\mathbf{rely}\, r \bullet c_z)\{p\}, z \vee r) \tag{82}$$

For (81)

$\{stops(c\{p\}, z)\}c\{p\} \sqsubseteq_z ((\mathbf{rely}\, r \bullet c_z)\{p\}) \parallel \langle r \vee \mathsf{id}\rangle^*$
$\Leftarrow$ moving assertion out of parallel
$\{stops(c\{p\}, z)\}c\{p\} \sqsubseteq_z ((\mathbf{rely}\, r \bullet c_z) \parallel \langle r \vee \mathsf{id}\rangle^*)\{p\}$
$\Leftarrow$ as $stops(c\{p\}, z) \Rrightarrow stops(c, z)$ by Law 2.19 (term-monotonic) as $c\{p\} \sqsubseteq c$
$\{stops(c, z)\}c \sqsubseteq_z (\mathbf{rely}\, r \bullet c_z) \parallel \langle r \vee \mathsf{id}\rangle^*$

which holds by Law 4.12 (rely). □

Law 4.21 (distribute-rely-sequential) *For any predicate p, relations z and r, commands c_0 and c_1, such that $(p \Rrightarrow stops((c_0 \,;\, c_1), z)$*

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

Proof. By Law 4.10 (rely-refinement-precondition) we must show both the following.

$$\{p\}c_0 \mathbin{;} c_1 \quad \sqsubseteq_z \quad ((\mathbf{rely}\, r \bullet (\{p\}c_0)_z) \mathbin{;} (\mathbf{rely}\, r \bullet (c_1)_z)) \parallel \langle r \vee \mathsf{id}\rangle^* \tag{83}$$

$$p \quad \Rrightarrow \quad stops(((\mathbf{rely}\, r \bullet (\{p\}c_0)_z) \mathbin{;} (\mathbf{rely}\, r \bullet (c_1)_z)), z \vee r) \tag{84}$$

Note that $p \Rrightarrow stops((c_0 \mathbin{;} c_1), z) \Rrightarrow stops(c_0, z)$. The proof of (83) follows starting from its right side.

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

$=$ by Lemma 4.4 (distribute-parallel) over sequential (67)

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

$\sqsupseteq_z$ by Law 4.12 (rely) twice as $p \Rrightarrow stops(c_0, z)$

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

$=_z$ as $p \Rrightarrow stops((c_0 \mathbin{;} c_1), z)$

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

The proof of (84) follows.

$stops((\mathbf{rely}\, r \bullet (c_0)_z) \mathbin{;} (\mathbf{rely}\, r \bullet (c_1)_z), z \vee r)$

$\equiv$ by Lemma 2.22 (sequential-term)

$stops((\mathbf{rely}\, r \bullet (c_0)_z)\{stops((\mathbf{rely}\, r \bullet (c_1)_z), z \vee r)\}, z \vee r)$

$\equiv$ by Law 4.7 (rely-stops)

$stops((\mathbf{rely}\, r \bullet (c_0)_z)\{stops(c_1, z)\}, z \vee r)$

$\equiv$ by Law 4.20 (rely-post-assertion) and Law 2.19 (term-monotonic)

$stops((\mathbf{rely}\, r \bullet (c_0\{stops(c_1, z)\})_z), z \vee r)$

$\equiv$ by Law 4.7 (rely-stops)

$stops(c_0\{stops(c_1, z)\}, z)$

$\equiv$ by Lemma 2.22 (sequential-term)

$stops(c_0 \mathbin{;} c_1, z)$

$\Lleftarrow$ by assumption

p

□

Law 4.22 (distribute-rely-conjunction) *For any predicate p, relations z and r, and commands c_0 and c_1, such that $p \Rrightarrow stops(c_0, z) \wedge stops(c_1, z)$,*

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

Proof. By Law 4.10 (rely-refinement-precondition) we must show both the following.

$$\{p\}(c_0 \Cap c_1) \quad \sqsubseteq_z \quad ((\mathbf{rely}\, r \bullet (\{p\}c_0)_z) \Cap (\mathbf{rely}\, r \bullet (\{p\}c_1)_z)) \parallel \langle r \vee \mathsf{id}\rangle^* \tag{85}$$

$$p \quad \Rrightarrow \quad stops(((\mathbf{rely}\, r \bullet (\{p\}c_0)_z) \Cap (\mathbf{rely}\, r \bullet (\{p\}c_1)_z)), z \vee r)) \tag{86}$$

The proof of (85) follows starting from the right side using the fact that conjunction is idempotent.

$((\mathbf{rely}\, r \bullet (\{p\}c_0)_z) \Cap (\mathbf{rely}\, r \bullet (\{p\}c_1)_z)) \parallel (\langle r \vee \mathsf{id}\rangle^* \Cap \langle r \vee \mathsf{id}\rangle^*)$

$\sqsupseteq$ by Lemma 3.7 (interchange-conjunction) with parallel (49)

$((\mathbf{rely}\, r \bullet (\{p\}c_0)_z) \parallel \langle r \vee \mathsf{id}\rangle^*) \Cap ((\mathbf{rely}\, r \bullet (\{p\}c_1)_z) \parallel \langle r \vee \mathsf{id}\rangle^*$

$\sqsupseteq_z$ by Law 4.12 (rely) twice using termination assumptions

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

$=$ by Lemma 3.5 (conjunction-strict)

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

For the proof of termination property (86), using Law 4.7 (rely-stops) one can deduce the property $p \Rrightarrow stops((\mathbf{rely}\, r \bullet (\{p\}c_0)_z), z \vee r) \wedge stops((\mathbf{rely}\, r \bullet (\{p\}c_1)_z), z \vee r)$ and hence that (86) holds by Law 3.10 (conjunction-term). □

Law 4.23 (distribute-rely-iteration) *For any predicate p, relations z and r, and command c, such that* $p \Rrightarrow stops((\{p\}c)^{\omega+}, z)$*,*

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

Proof. Because $p \Rrightarrow stops((\{p\}c)^{\omega+}, z)$, we have $p \Rrightarrow stops(\{p\}c\ ;\ (\{p\}c)^{\omega}, z) \Rrightarrow stops(c, z)$. Using Law 4.10 (rely-refinement-precondition) one must show both the following.

$$(\{p\}c)^{\omega+} \quad \sqsubseteq_z \quad (\mathbf{rely}\ r \bullet \{p\}c_z)^{\omega+} \parallel \langle r \vee \mathsf{id}\rangle^* \tag{87}$$

$$p \quad \Rrightarrow \quad stops((\mathbf{rely}\ r \bullet \{p\}c_z)^{\omega+}, z \vee r) \tag{88}$$

For $c^{\omega+}$ iteration we use Lemma 2.14 (iteration-induction) for ω-iteration (40). In general,

$$c \sqcap (c\ ;\ d) \sqsubseteq_{r_x} d \qquad \Rightarrow \qquad c^{\omega+} \sqsubseteq_{r_x} d \tag{89}$$

and hence (87) holds if

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

which we show as follows.

$\{p\}c \sqcap (\{p\}c\ ;\ ((\mathbf{rely}\ r \bullet \{p\}c_z)^{\omega+} \parallel \langle r \vee \mathsf{id}\rangle^*))$

$\sqsubseteq_z$ by Law 4.12 (rely) twice as $p \Rrightarrow stops(c, z)$

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

$(((\mathbf{rely}\ r \bullet \{p\}c_z) \parallel \langle r \vee \mathsf{id}\rangle^*);((\mathbf{rely}\ r \bullet \{p\}c_z)^{\omega+} \parallel \langle r \vee \mathsf{id}\rangle^*))$

$=$ by Lemma 4.4 (distribute-parallel) over choice (66) and sequential (67)

$((\mathbf{rely}\ r \bullet \{p\}c_z) \sqcap ((\mathbf{rely}\ r \bullet \{p\}c_z)\ ;\ (\mathbf{rely}\ r \bullet \{p\}c_z)^{\omega+})) \parallel \langle r \vee \mathsf{id}\rangle^*$

$=$ by Lemma 2.13 (fold/unfold-iteration)

$(\mathbf{rely}\ r \bullet \{p\}c_z)^{\omega+} \parallel \langle r \vee \mathsf{id}\rangle^*$

For termination condition (88) by Law 4.12 (rely)

$\{p\}c \sqsubseteq_z (\mathbf{rely}\ r \bullet \{p\}c_z) \parallel \langle r \vee \mathsf{id}\rangle^*$

$\Rrightarrow$ by Lemma 2.15 (iteration-monotonic)

$(\{p\}c)^{\omega+} \sqsubseteq_z ((\mathbf{rely}\ r \bullet \{p\}c_z) \parallel \langle r \vee \mathsf{id}\rangle^*)^{\omega+}$

$\Rrightarrow$ by Law 2.19 (term-monotonic) and assumption

$p \Rrightarrow stops((\{p\}c)^{\omega+}, z) \Rrightarrow stops(((\mathbf{rely}\ r \bullet \{p\}c_z) \parallel \langle r \vee \mathsf{id}\rangle^*)^{\omega+}, z)$

$\Rrightarrow$ by Lemma 4.24 (term-iteration) as $p \Rrightarrow stops((\mathbf{rely}\ r \bullet \{p\}c_z), z \vee r)$ by Law 4.7 (rely-stops)

$p \Rrightarrow stops((\mathbf{rely}\ r \bullet \{p\}c_z)^{\omega+}, z \vee r)$

□

Lemma 4.24 (term-iteration) *For any predicate p, relations r and z, and command c, such that* $p \Rrightarrow stops(c, z \vee r)$*,*

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

Law 4.23 (distribute-rely-iteration) applies to $c^{\omega+}$ but not c^{ω} because $(\mathbf{rely}\ r \bullet \mathbf{skip})$ is not refined by **skip** (because by Law 4.8 (rely-refinement) this would require $\mathbf{skip} \sqsubseteq \mathbf{skip} \parallel \langle r \vee \mathsf{id}\rangle^*$).

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

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

However, the 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, which after all is their *raison d'être*.

A conjunction of commands may be implemented by a parallel combination of commands provided each respects the rely of the other; Law 5.1 captures the core of the rely-guarantee approach to developing a parallel program. This law generalises the Jones-quintuple rule because it applies to a conjunction of commands, rather than a conjunction of postconditions. Law 5.2 extends that to handle a conjunction of commands within a rely context, and Law 5.3 specialises that to refining a specification (with a conjunction of post conditions) by a parallel composition.

Law 5.1 (introduce-parallel) *For any predicate p, relations z, g_0 and g_1, and commands c_0 and c_1, such that $p \Rrightarrow stops(c_0, z) \wedge stops(c_1, z)$,*

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

Proof. By Law 4.12 (rely) both the following hold.

$$\{p\}c_0 \quad \sqsubseteq_z \quad (\mathbf{rely}\, g_1 \bullet (\{p\}c_0)_z) \parallel \langle g_1 \vee \mathsf{id}\rangle^* \qquad (90)$$
$$\{p\}c_1 \quad \sqsubseteq_z \quad (\mathbf{rely}\, g_0 \bullet (\{p\}c_1)_z) \parallel \langle g_0 \vee \mathsf{id}\rangle^* \qquad (91)$$

Hence the law holds as follows.

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

$=$ by Lemma 3.5 (conjunction-strict)

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

$\sqsubseteq_z$ by Lemma 3.4 (conjunction-monotonic) using (90) and (91)

$((\mathbf{rely}\, g_1 \bullet (\{p\}c_0)_z) \parallel \langle g_1 \vee \mathsf{id}\rangle^*) \Cap (\langle g_0 \vee \mathsf{id}\rangle^* \parallel (\mathbf{rely}\, g_0 \bullet (\{p\}c_1)_z))$

$\sqsubseteq$ by Lemma 3.7 (interchange-conjunction) with parallel

$((\mathbf{rely}\, g_1 \bullet (\{p\}c_0)_z) \Cap \langle g_0 \vee \mathsf{id}\rangle^*) \parallel (\langle g_1 \vee \mathsf{id}\rangle^* \Cap (\mathbf{rely}\, g_0 \bullet (\{p\}c_1)_z))$

$=$ by Law 3.15 (terminating-iteration)

$(\langle g_0 \vee \mathsf{id}\rangle^\omega \Cap \langle \mathsf{true}\rangle^* \Cap (\mathbf{rely}\, g_1 \bullet (\{p\}c_0)_z)) \parallel (\langle g_1 \vee \mathsf{id}\rangle^\omega \Cap \langle \mathsf{true}\rangle^* \Cap (\mathbf{rely}\, g_0 \bullet (\{p\}c_1)_z))$

$=$ by Definition 3.1 (guarantee)

$(\mathbf{guar}\, g_0 \bullet (\langle \mathsf{true}\rangle^* \Cap (\mathbf{rely}\, g_1 \bullet (\{p\}c_0)_z))) \parallel (\mathbf{guar}\, g_1 \bullet (\langle \mathsf{true}\rangle^* \Cap (\mathbf{rely}\, g_0 \bullet (\{p\}c_1)_z)))$

$\sqsubseteq_z$ by Lemma 3.29 (refine-in-guarantee-context) – see below

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

For the proof of the last step above, the body of left branch of the parallel is refined as follows.

$\langle \mathsf{true}\rangle^* \Cap (\mathbf{rely}\, g_1 \bullet (\{p\}c_0)_z)$

$=$ by Law 4.17 (rely-precondition) and Lemma 3.5 (conjunction-strict)

$\{p\}\langle \mathsf{true}\rangle^* \Cap (\mathbf{rely}\, g_1 \bullet (\{p\}c_0)_z)$

$\sqsubseteq_{z \vee g_1}$ by Law 3.9 (simplify-conjunction) using (92) below

$(\mathbf{rely}\, g_1 \bullet (\{p\}c_0)_z)$

The right branch is refined similarly. By Law 4.7 (rely-stops), $p \Rrightarrow stops((\mathbf{rely}\, g_1 \bullet (\{p\}c_0)_z), z \vee g_1)$ and hence by Definition 2.18 (stops) part (42),

$$\{p\}\langle \mathsf{true}\rangle^* \quad \sqsubseteq_{z \vee g_1} \quad (\mathbf{rely}\, g_1 \bullet (\{p\}c_0)_z) . \qquad (92)$$

which is used in the proof of the last step above. □

Law 5.2 (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 $p_0 \Rrightarrow stops(c_0, z)$ and $p_1 \Rrightarrow stops(c_1, z)$,*

$$\begin{array}{ll} & (\textbf{rely}\, r \bullet (\{p\}(c_0 \Cap c_1))_z) \\ \sqsubseteq & (\textbf{guar}\, g_0 \bullet (\textbf{rely}\, g_1 \vee r \bullet (\{p_0\}c_0)_z)) \parallel (\textbf{guar}\, g_1 \bullet (\textbf{rely}\, g_0 \vee r \bullet (\{p_1\}c_1)_z))\ . \end{array} \tag{93}$$

Proof. From the assumptions one can deduce both $p_0 \Rrightarrow stops((\textbf{rely}\, r \bullet (\{p_0\}c_0)_z), z \vee r)$ and $p_1 \Rrightarrow stops((\textbf{rely}\, r \bullet (\{p_1\}c_1)_z), z \vee r)$. These are used in the application of Law 5.1 (introduce-parallel) below. By Law 4.9 (rely-environment) it is sufficient to show (93) in context $z \vee r$.

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

$\sqsubseteq$ by Lemma 3.5 (conjunction-strict) and as $p \Rrightarrow p_0$ and $p \Rrightarrow p_1$

$(\textbf{rely}\, r \bullet (\{p_0\}c_0 \Cap \{p_1\}c_1)_z)$

$\sqsubseteq$ Law 4.22 (distribute-rely-conjunction) using termination conditions

$(\textbf{rely}\, r \bullet (\{p_0\}c_0)_z) \Cap (\textbf{rely}\, r \bullet (\{p_1\}c_1)_z)$

$\sqsubseteq$ by Law 4.17 (rely-precondition) twice using termination conditions

$(\{p_0\}(\textbf{rely}\, r \bullet (\{p_0\}c_0)_z)) \Cap (\{p_1\}(\textbf{rely}\, r \bullet (\{p_1\}c_1)_z))$

$=$ by Lemma 3.5 (conjunction-strict)

$\{p_0 \wedge p_1\}((\textbf{rely}\, r \bullet (\{p_0\}c_0)_z) \Cap (\textbf{rely}\, r \bullet (\{p_1\}c_1)_z)$

$\sqsubseteq_{z \vee r}$ by Law 5.1 (introduce-parallel) using termination conditions

$(\textbf{guar}\, g_0 \bullet (\textbf{rely}\, g1 \bullet (\textbf{rely}\, r \bullet (\{p_0\}c_0)_z)_{z \vee r})) \parallel (\textbf{guar}\, g_1 \bullet (\textbf{rely}\, g_0 \bullet (\textbf{rely}\, r \bullet (\{p_1\}c_1)_z)_{z \vee r}))$

$=$ by Law 4.18 (nested-rely) twice

$(\textbf{guar}\, g_0 \bullet (\textbf{rely}\, g1 \vee r \bullet (\{p_0\}c_0)_z)) \parallel (\textbf{guar}\, g_1 \bullet (\textbf{rely}\, g_0 \vee r \bullet (\{p_1\}c_1)_z))$

□

The following law applies Law 5.2 to specifications. This corresponds to the parallel introduction law of [Jon83].

Law 5.3 (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 $p \wedge q_0 \wedge q_1 \Rrightarrow q$,*

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

Proof. The termination assumptions for the application of Law 5.2 (introduce-parallel-with-rely) below are $p \Rrightarrow stops([q_0], \mathsf{id})$ and $p \Rrightarrow stops([q_1], \mathsf{id})$, which are trivial.

$(\textbf{rely}\, r \bullet [p,\ q])$

$\sqsubseteq$ by Lemma 2.8 (consequence) as $p \wedge q_0 \wedge q_1 \Rrightarrow q$ and Law 4.16 (rely-monotonic)

$(\textbf{rely}\, r \bullet \{p\}\ [q_0 \wedge q_1])$

$=$ by Lemma 3.18 (conjoined-specifications)

$(\textbf{rely}\, r \bullet \{p\}([q_0] \Cap [q_1]))$

$\sqsubseteq$ by Law 5.2 (introduce-parallel-with-rely)

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

□

6 Trading post conditions with rely and guarantee

Any command in a guarantee context of g and a rely context of r only executes atomic program steps satisfying $(g \vee \mathsf{id})$ and assumes the environment only executes atomic steps satisfying $(r \vee \mathsf{id})$ and hence any single step (whether program or environment) satisfies $(g \vee r \vee \mathsf{id})$ 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 6.1 (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 post condition. The refinement from left to right can be shown using Law 4.14 (guarantee-plus-rely). The guarantee proviso (76) of Law 4.14 is trivial; to prove the other proviso (75) one can use Law 4.11 (rely-specification) which requires one to show both the following.

$$[p,\ q \wedge (g \vee r)^*] \quad \sqsubseteq \quad (\mathbf{guar}\, g \bullet (\mathbf{rely}\, r \bullet [p,\ q])) \parallel \langle r \vee \mathsf{id}\rangle^* \tag{94}$$

$$p \quad \Rrightarrow \quad stops((\mathbf{guar}\, g \bullet (\mathbf{rely}\, r \bullet [p,\ q])), r) \tag{95}$$

The proof for (95) follows by Lemma 2.20 (precondition-term), Lemma 2.21 (specification-term), Law 4.7 (rely-stops) and Law 3.25 (guarantee-term).

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

The proof of the refinement (94) follows.

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

$\sqsubseteq$ by Law 3.24 (introduce-guarantee) of $g \vee r$ and Law 3.31 (trading-post-guarantee)

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

$\sqsubseteq$ by Law 4.12 (rely)

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

$=$ by Law 3.27 (distribute-guarantee) over parallel (62)

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

$\sqsubseteq$ by Law 3.22 (strengthen-guarantee) twice

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

$=$ by Law 3.16 (conjunction-atomic-iterated) part (58) and Law 3.15 (terminating-iteration)

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

Because conjunction is idempotent the last step holds as follows.

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

□

Law 6.2 (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 $p \wedge q_0 \wedge q_1 \Rrightarrow q$,*

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

Proof. We use Law 6.1 (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 3.24 (introduce-guarantee) of $g_0 \vee g_1$ and Law 6.1 (trade-rely-guarantee)

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

$\sqsubseteq$ by Law 5.3 (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 3.27 (distribute-guarantee) and Law 3.26 (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 6.3 (introduce-parallel-spec-weaken-rely) *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 $g_0 \vee r \Rrightarrow r_1 \vee \mathsf{id}$ and $g_1 \vee r \Rrightarrow r_0 \vee \mathsf{id}$,*

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

Proof. The law follows from Law 6.2 (introduce-parallel-spec-with-trading) and two applications of Law 4.15 (weaken-rely). □

Trading can also be applied to a guarantee invariant.

Law 6.4 (trade-rely-guarantee-invariant) *For predicate p and relations r and q, such that $r \Rrightarrow (p \Rightarrow p')$,*

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

Proof.

$$\begin{array}{ll} & \mathbf{rely}\, r \bullet [p,\ p' \wedge q] \\ \sqsubseteq & \text{by Lemma 2.8 (consequence) as } r \Rrightarrow (p \Rightarrow p') \\ & \mathbf{rely}\, r \bullet [p,\ ((p \Rightarrow p') \vee r)^* \wedge q] \\ \sqsubseteq & \text{by Law 3.24 (introduce-guarantee)} \\ & \mathbf{guar}(p \Rightarrow p') \bullet \mathbf{rely}\, r \bullet ([p,\ ((p \Rightarrow p') \vee r)^* \wedge q]) \\ = & \text{by Law 6.1 (trade-rely-guarantee) and Definition 3.41 (guarantee-invariant)} \\ & \mathbf{guar}\text{-}\mathbf{inv}\, p \bullet (\mathbf{rely}\, r \bullet [p,\ q]) \end{array}$$

□

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. If the precondition p holds initially, it must hold after any interference steps satisfying r – this leads to condition (96) below. If the specification with post condition q is preceded by an interference step satisfying r, then a step satisfying r followed by the step satisfying q should also satisfy q – this leads to condition (97), which also assumes p holds initially. Condition (98) is similarly required to handle an interference step following the specification.

Definition 7.1 (tolerate-interference) *A specification $[p,\ q]$* tolerates interference *r provided*

$$r \Rrightarrow (p \Rightarrow p') \tag{96}$$
$$p \wedge (r \mathbin{⨟} q) \Rrightarrow q \tag{97}$$
$$p \wedge (q \mathbin{⨟} r) \Rrightarrow q \tag{98}$$

Properties (96), (97) and (98) imply both the following.

$$r^* \Rrightarrow (p \Rightarrow p') \tag{99}$$
$$p \wedge (r^* \mathbin{⨟} q \mathbin{⨟} r^*) \Rrightarrow q \tag{100}$$

Law 7.2 (tolerate-interference) *If a specification $[p,\ q]$ tolerates interference r then*

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

Proof.

$$\begin{array}{ll} & [p,\ q] \\ \sqsubseteq & \text{by Lemma 2.8 (consequence) as } p \wedge (r^* \fatsemi q \fatsemi r^*) \Rrightarrow q \\ & [p,\ r^* \fatsemi q \fatsemi r^*] \\ \sqsubseteq & \text{by Lemma 2.9 (sequential) twice as } r^* \Rrightarrow (p \Rightarrow p') \\ & [r^*]\ ;\ [p,\ q]\ ;\ [r^*] \\ \sqsubseteq & \text{by Law 3.3 (refine-iterated-relation) twice as } (r \vee \mathsf{id})^* = r^* \\ & \langle r \vee \mathsf{id}\rangle^*\ ;\ [p,\ q]\ ;\ \langle r \vee \mathsf{id}\rangle^* \end{array}$$

□

Conditions (96), (97) and (98) are slight generalisations of conditions **PR-ident**, **RQ-ident**, and **QR-ident** used by Coleman and Jones [CJ07, 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 [WDP10], although that paper limits post conditions to single state predicates rather than relations.

If $[p,\ q]$ 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 $[p,\ q]$ can be implemented because the conditions do not address interference while $[p,\ q]$ is executing, only before and after. Interference during $[p,\ q]$ is handled by distributing the rely. We may have that $[p,\ q]$ tolerates interference r and

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

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

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

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

The following law corresponds to Jones-style sequential introduction [Jon83]; note that by Law 3.21 (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 3.27 (distribute-guarantee) over sequential (61).

Law 7.3 (rely-sequential) *For any preconditions p_0 and p_1, and any relations r, q_0 and q_1, such that $p_0 \wedge ((q_0 \wedge p_1') \fatsemi q_1) \Rrightarrow q$,*

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

Proof.

$$\begin{array}{ll} & (\textbf{rely}\, r \bullet [p_0,\ q]) \\ \sqsubseteq & \text{by Lemma 2.9 (sequential) and Law 4.16 (rely-monotonic)} \\ & (\textbf{rely}\, r \bullet [p_0,\ q_0 \wedge p_1']\ ;\ [p_1,\ q_1]) \\ \sqsubseteq & \text{by Law 4.21 (distribute-rely-sequential)} \\ & (\textbf{rely}\, r \bullet [p_0,\ q_0 \wedge p_1'])\ ;\ (\textbf{rely}\, r \bullet [p_1,\ q_1]) \end{array}$$

□

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 7.4 (rely-to-atomic) *For predicate p, and relations r and q, such that $[p,\ q]$ tolerates interference r,*

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

Note that the precondition p in the specification $[p,\ q]$ 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.
Proof.

$$
\begin{array}{ll}
 & (\mathbf{rely}\, r \bullet [p,\ q]) \sqsubseteq \langle p,q\rangle \\
\equiv & \text{by Law 4.11 (rely-specification) as } p \Rrightarrow \mathit{stops}(\langle p,q\rangle, r) \text{ by Lemma 2.23 (atomic-term)} \\
 & [p,\ q] \sqsubseteq \langle p,q\rangle \parallel \langle r \vee \mathsf{id}\rangle^* \\
\equiv & \text{by Lemma 4.3 (interference-atomic)} \\
 & [p,\ q] \sqsubseteq \langle r \vee \mathsf{id}\rangle^* \mathbin{;} \langle p,q\rangle \mathbin{;} \langle r \vee \mathsf{id}\rangle^* \\
\Leftarrow & \text{by Lemma 2.7 (make-atomic)} \\
 & [p,\ q] \sqsubseteq \langle r \vee \mathsf{id}\rangle^* \mathbin{;} [p,\ q] \mathbin{;} \langle r \vee \mathsf{id}\rangle^*
\end{array}
$$

The last refinement holds by Law 7.2 (tolerate-interference) as $[p,\ q]$ tolerates r. □

If $[p,\ q]$ tolerates interference $\mathsf{id}(X)$, where X contains the free variables of p and q, it is tempting to use a non-atomic specification on the right in Law 7.4. 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 $\mathsf{id}(X)$. 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 the usual care is required with local variable declarations. The rules are straightforward and hence they are not spelled out here.[8]

Lemma 7.5 (uses-atomic-effective) *For any predicate p, relation q, command c, and set of variables X,*

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

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. Hence in a trace of $(\mathbf{uses}\, X \bullet c) \parallel \langle \mathsf{id}(X)\rangle^*$, all steps of $\langle \mathsf{id}(X)\rangle^*$ may 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 $\langle \mathsf{id}(X)\rangle^*$ on either side. □

Law 7.6 (rely-uses) *For any predicate p, relation q, and set of variables X, such that* $[p,\ q]$ *tolerates interference* $\mathsf{id}(X)$*,*

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

Proof. Noting that $\mathsf{id}(X) \vee \mathsf{id} = \mathsf{id}(X)$ and $p \equiv \mathit{stops}((\mathbf{uses}\, X \bullet [p,\ q]), \mathsf{id}(X))$, by Law 4.11 (rely-specification) the theorem holds if,

$$
\begin{array}{ll}
 & [p,\ q] \sqsubseteq (\mathbf{uses}\, X \bullet [p,\ q]) \parallel \langle \mathsf{id}(X)\rangle^* \\
\Leftarrow & \text{by Lemma 7.5 (uses-atomic-effective)} \\
 & [p,\ q] \sqsubseteq \langle \mathsf{id}(X)\rangle^* \mathbin{;} (\mathbf{uses}\, X \bullet [p,\ q]) \mathbin{;} \langle \mathsf{id}(X)\rangle^* \\
\Leftarrow & \text{by Law 7.2 (tolerate-interference) as } [p,\ q] \text{ tolerates } \mathsf{id}(X) \\
 & [p,\ q] \sqsubseteq (\mathbf{uses}\, X \bullet [p,\ q])
\end{array}
$$

[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.

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.

Law 7.7 (assignment-rely-guarantee) *For any variable x, expression e, set of variables X, predicate p and relations g and q, such that $[p,\ q]$ tolerates interference* $\mathsf{id}(X)$, *$p \Rrightarrow \mathit{def}(e)$ and $p \wedge x' = e \wedge \mathsf{id}(\bar{x}) \Rrightarrow q \wedge (g \vee \mathsf{id})$ and $\mathit{vars}(e) \cup \{x\} \subseteq X$*

$$x : (\mathbf{guar}\, g \bullet (\mathbf{rely}\, \mathsf{id}(X) \bullet [p,\ q])) \quad \sqsubseteq \quad x := e\ .$$

Proof. Many of the steps in the proof implicitly use Law 3.21 (guarantee-monotonic).

$x : \mathbf{guar}\, g \bullet (\mathbf{rely}\, \mathsf{id}(X) \bullet [p,\ q])$

$\sqsubseteq$ by Law 7.6 (rely-uses) as $[p,\ q]$ tolerates $\mathsf{id}(X)$

$x : \mathbf{guar}\, g \bullet (\mathbf{uses}\, X \bullet [p,\ q])$

$=$ as **uses** distributes through guarantees and Law 3.37 (guarantee-frame)

$\mathbf{uses}\, X \bullet (\mathbf{guar}\, g \bullet x \colon [p,\ q])$

$\sqsubseteq$ by Law 3.38 (guarantee-assignment) as $p \Rrightarrow \mathit{def}(e)$ and $p \wedge x' = e \wedge \mathsf{id}(\bar{x}) \Rrightarrow q \wedge (g \vee \mathsf{id})$

$\mathbf{uses}\, X \bullet x := e$

$\sqsubseteq$ as $\mathit{vars}(e) \cup \{x\} \subseteq X$

$x := e$

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.

8 Expressions and tests

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 [CJ07, Col08, WDP10, HBDJ13]; 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 3.32 (tests).

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 Lemma 7.5 (uses-atomic-effective).

Lemma 8.1 (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 $r \Rrightarrow \mathsf{id}(\mathit{vars}(b) - \{y\})$, and if there is at most a single reference to y within b,*

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

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 $[[b]] \parallel \langle r \vee \mathsf{id}\rangle^*$, 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 $\langle r \vee \mathsf{id}\rangle^* ; [[b]] ; \langle r \vee \mathsf{id}\rangle^*$. □

For Law 7.7 (assignment-rely-guarantee) there is an assumption of no interference on e and x. Below we develop a law that handles interference in the form of the single reference property.

Law 8.2 (assignment-single-reference) *For any variable x, expression e, predicate p, and relations r and q, such that* $\left[p,\ q\right]$ *tolerates interference r,* $p \Rrightarrow def(e)$*, e satisfies the single-reference property and is preserved by r, i.e.* $r \Rrightarrow (e = e')$*, and* $p \wedge x' = e \wedge \mathsf{id}(\bar{x}) \Rrightarrow q$*,*

$$(\textbf{rely}\, r \bullet \left[p,\ q\right]) \quad \sqsubseteq \quad x := e\ .$$

Proof. From the definition of assignment (15) one must show the following refinement in which the bound variable v does not occur free in either e or x.

$$\begin{array}{ll}
& (\textbf{rely}\, r \bullet \left[p,\ q\right]) \ \sqsubseteq\ \bigsqcap\{v \in \mathit{Val} \bullet [[e = v]] ; \langle x' = v \wedge \mathsf{id}(\bar{x})\rangle\} \\
\Lleftarrow & \text{by Law 2.11 (nondeterministic-choice) part (34) if for all } v \in \mathit{Val} \\
& (\textbf{rely}\, r \bullet \left[p,\ q\right]) \ \sqsubseteq\ [[e = v]] ; \langle x' = v \wedge \mathsf{id}(\bar{x})\rangle \\
\equiv & \text{by Law 4.8 (rely-refinement)} \\
& \left[p,\ q\right] \ \sqsubseteq\ ([[e = v]] ; \langle x' = v \wedge \mathsf{id}(\bar{x})\rangle) \parallel \langle r \vee \mathsf{id}\rangle^* \wedge \\
& (\mathit{stops}(\left[p,\ q\right], \mathsf{id}) \Rrightarrow \mathit{stops}([[e = v]] ; \langle x' = v \wedge \mathsf{id}(\bar{x})\rangle, r))
\end{array}$$

The termination condition holds because $\mathit{stops}(\left[p,\ q\right], \mathsf{id}) \equiv p$ holds by Lemma 2.20 (precondition-term) and Lemma 2.21 (specification-term), and $p \Rrightarrow \mathit{def}(e)$ and p is preserved by r and hence the test terminates; an atomic step with no (i.e. true) precondition terminates. The refinement holds by Lemma 4.4 (distribute-parallel) over sequential (67) if,

$$\begin{array}{ll}
& \left[p,\ q\right] \ \sqsubseteq\ ([[e = v]] \parallel \langle r \vee \mathsf{id}\rangle^*) ; (\langle x' = v \wedge \mathsf{id}(\bar{x})\rangle \parallel \langle r \vee \mathsf{id}\rangle^*) \\
\equiv & \text{by Lemma 8.1 (test-single-reference) and Lemma 4.3 (interference-atomic)} \\
& \left[p,\ q\right] \ \sqsubseteq\ \langle r \vee \mathsf{id}\rangle^* ; [[e = v]] ; \langle r \vee \mathsf{id}\rangle^* ; \langle r \vee \mathsf{id}\rangle^* ; \langle x' = v \wedge \mathsf{id}(\bar{x})\rangle ; \langle r \vee \mathsf{id}\rangle^* \\
\Lleftarrow & \text{by Law 7.2 (tolerate-interference) and Lemma 4.2 (repeated-interference)} \\
& \left[p,\ q\right] \ \sqsubseteq\ [[e = v]] ; \langle r \vee \mathsf{id}\rangle^* ; \langle x' = v \wedge \mathsf{id}(\bar{x})\rangle \\
\Lleftarrow & \text{by Law 3.3 (refine-iterated-relation) and Lemma 2.7 (make-atomic)} \\
& \left[p,\ q\right] \ \sqsubseteq\ [[e = v]] ; \left[r^*\right] ; \left[x' = v \wedge \mathsf{id}(\bar{x})\right] \\
\Lleftarrow & \text{by Lemma 2.12 (introduce-test)} \\
& \left[p,\ q\right] \ \sqsubseteq\ \left[\mathit{def}(e),\ e = v \wedge \mathsf{id}\right] ; \left[r^*\right] ; \left[x' = v \wedge \mathsf{id}(\bar{x})\right] \\
\Lleftarrow & \text{as } p \Rrightarrow \mathit{def}(e) \text{ and Lemma 2.9 (sequential) twice} \\
& p \wedge (e = v \wedge \mathsf{id}) \fatsemi r^* \fatsemi (x' = v \wedge \mathsf{id}(\bar{x})) \Rrightarrow q \\
\Lleftarrow & \text{as } r \text{ preserves } p \text{ and } e \\
& p \wedge r^* \fatsemi (p \wedge x' = e \wedge \mathsf{id}(\bar{x})) \Rrightarrow q \\
\Lleftarrow & \text{as } p \wedge x' = e \wedge \mathsf{id}(\bar{x}) \Rrightarrow q \\
& p \wedge r^* \fatsemi q \Rrightarrow q
\end{array}$$

which holds as $\left[p,\ q\right]$ tolerates interference r. □

9 Local variables

A local variable is immune from interference from its environment. Hence when refining the body of a local variable block, the environment can 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 9.1 (refine-var) *For any variable x, and commands c and d,*

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

Law 9.2 (variable-rely) *For any command c, relations z and r, variable x, and set of variables Y, where x is not in Y,*

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

Proof. The refinement from right to left follows by Law 4.15 (weaken-rely). The refinement from left to right holds as follows.

$(\mathbf{var}\, x \bullet x, Y : (\mathbf{rely}\, r \bullet c_z)) \sqsubseteq (\mathbf{var}\, x \bullet x, Y : (\mathbf{rely}\, \mathsf{id}(x) \wedge r \bullet c_z))$
$\Lleftarrow$ by Lemma 9.1 (refine-var)
$x, Y : (\mathbf{rely}\, r \bullet c_z) \sqsubseteq_{\mathsf{id}(x)} x, Y : (\mathbf{rely}\, \mathsf{id}(x) \wedge r \bullet c_z)$
$\Lleftarrow$ by Law 3.21 (guarantee-monotonic) applied to the frames
$(\mathbf{rely}\, r \bullet c_z) \sqsubseteq_{\mathsf{id}(x)} (\mathbf{rely}\, \mathsf{id}(x) \wedge r \bullet c_z)$

The latter follows by Law 4.13 (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 9.3 (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,*

$$(\mathbf{guar}\, g \wedge \mathsf{id}(x) \bullet Y : (\mathbf{rely}\, r \bullet c_z)) \quad \sqsubseteq \quad (\mathbf{var}\, x \bullet x, Y : (\mathbf{guar}\, g \bullet (\mathbf{rely}\, \mathsf{id}(x) \wedge r \bullet c_z))) \ .$$

Proof.

$\mathbf{guar}\, g \wedge \mathsf{id}(x) \bullet Y : (\mathbf{rely}\, r \bullet c_z)$
$\sqsubseteq$ by Lemma 3.39 (introduce-variable) as x not in Y and not used by c or r
$\mathbf{guar}\, g \wedge \mathsf{id}(x) \bullet (\mathbf{var}\, x \bullet x, Y : (\mathbf{rely}\, r \bullet c_z))$
$=$ by Law 3.26 (nested-guarantees) and Law 3.40 (guarantee-variable)
$\mathbf{guar}\, g \bullet (\mathbf{var}\, x \bullet x, Y : (\mathbf{rely}\, r \bullet c_z))$
$=$ by Law 9.2 (variable-rely) as variable declaration ensures rely $\mathsf{id}(x)$ locally
$\mathbf{guar}\, g \bullet (\mathbf{var}\, x \bullet x, Y : (\mathbf{rely}\, \mathsf{id}(x) \wedge r \bullet c_z))$
$=$ by Law 3.27 (distribute-guarantee) over variable declaration (63) as g is independent of x
$\mathbf{var}\, x \bullet x, Y : (\mathbf{guar}\, g \bullet (\mathbf{rely}\, \mathsf{id}(x) \wedge r \bullet c_z))$

The last step also uses Law 3.37 (guarantee-frame). □

10 Control structures and rely commands

For control structures it is not enough to simply evaluate a test, but rather one would like to be able to use its 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 11 it is used to allow an early exit from a loop. We now consider this special case for introducing a test that satisfies the single reference property, before using it to prove the laws for introducing conditionals and loops.

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

Law 10.1 (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 $p \Rrightarrow def(b)$,*

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

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

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

Hence $[p,\ r^* \wedge b_0']$ tolerates interference r. The termination condition, $p \Rrightarrow stops([[b]], r)$, holds because $p \Rrightarrow def(b)$ and p is preserved by r. Hence by Law 4.11 (rely-specification) the theorem holds if,

$[p,\ r^* \wedge b_0'] \sqsubseteq [[b]] \parallel \langle r \vee \mathsf{id}\rangle^*$

$\equiv$ by Lemma 8.1 (test-single-reference)

$[p,\ r^* \wedge b_0'] \sqsubseteq \langle r \vee \mathsf{id}\rangle^* \,;\, [[b]] \,;\, \langle r \vee \mathsf{id}\rangle^*$

$\Lleftarrow$ by Law 7.2 (tolerate-interference)

$[p,\ r^* \wedge b_0'] \sqsubseteq [[b]]$

$\Lleftarrow$ by Lemma 2.8 (consequence) as $p \Rrightarrow def(b)$ and $p \wedge b \Rrightarrow b_0$ and $\mathsf{id} \Rrightarrow r^*$

$[def(b),\ b \wedge \mathsf{id}] \sqsubseteq [[b]]$

which holds by Lemma 2.12 (introduce-test). □

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. 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 sets of states satisfying b_0 and b_1 may overlap.

Law 10.2 (rely-conditional) *For any predicate p, relations r and q, such that $[p,\ q]$ 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 $p \Rrightarrow def(b)$,*

$$(\mathbf{rely}\, r \bullet [p,\ q]) \quad \sqsubseteq \quad \mathbf{if}\, b\, \mathbf{then}\, (\mathbf{rely}\, r \bullet [p \wedge b_0,\ q]\, \mathbf{else}\, (\mathbf{rely}\, r \bullet [p \wedge b_1,\ q]) .$$

Proof.

$\mathbf{rely}\, r \bullet [p,\ q]$

$=$ as nondeterministic choice is idempotent

$(\mathbf{rely}\, r \bullet [p,\ q]) \sqcap (\mathbf{rely}\, r \bullet [p,\ q])$

$\sqsubseteq$ by Law 7.3 (rely-sequential) twice as $[p,\ q]$ tolerates interference r; hence $p \wedge r^* \fatsemi q \Rrightarrow q$

$((\mathbf{rely}\, r \bullet [p,\ r^* \wedge b_0']);(\mathbf{rely}\, r \bullet [p \wedge b_0,\ q])) \sqcap ((\mathbf{rely}\, r \bullet [p,\ r^* \wedge b_1']);(\mathbf{rely}\, r \bullet [p \wedge b_1,\ q]))$

$\sqsubseteq$ by Law 10.1 (rely-test) twice using the assumptions on b_0 and b_1

$([[b]] \,;\, (\mathbf{rely}\, r \bullet [p \wedge b_0,\ q])) \sqcap ([[\neg\, b]] \,;\, (\mathbf{rely}\, r \bullet [p \wedge b_1,\ q]))$

$=$ by the definition of a conditional (13)

$\mathbf{if}\, b\, \mathbf{then}\, (\mathbf{rely}\, r \bullet [p \wedge b_0,\ q])) \,\mathbf{else}\, (\mathbf{rely}\, r \bullet [p \wedge b_1,\ q])$

□

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 3.34 (guarantee-conditional) to handle refinement of a specification to a conditional in both guarantee and rely contexts.

The proof of Law 10.6 (rely-loop) below depends on properties of an iteration of a specification with a post condition satisfying a well-founded relation.

Definition 10.3 (well-founded) *For a predicate p, a relation w is* well-founded on p *if for any state σ_0 satisfying p there does not exist an infinite sequence of states starting with σ_0 in which every adjacent pair of states in the sequence is related by w.*

Iteration of a specification that establishes a well-founded relation terminates.

Law 10.4 (well-founded-termination) *For any predicate p and relation w, such that w is well founded on p,*

$$[p,\ p' \wedge w]^{\omega+} \quad = \quad [p,\ p' \wedge w]^{+}\ .$$

Proof.

$$
\begin{array}{ll}
& [p,\ p' \wedge w]^{\omega+} \\
= & \text{by Lemma 2.16 (isolation)} \\
& [p,\ p' \wedge w]^{+} \sqcap [p,\ p' \wedge w]^{\infty} \\
= & \text{as } w \text{ is well founded on } p,\ [p,\ p' \wedge w]^{\infty} = [p,\ \mathsf{false}] \\
& [p,\ p' \wedge w]^{+}
\end{array}
$$

□

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

Law 10.5 (refine-to-iteration) *For any precondition p and relation w,*

$$[p,\ p' \wedge w^+] \quad \sqsubseteq \quad [p,\ p' \wedge w^+]^{+}\ .$$

Proof. By Lemma 2.14 (iteration-induction) for finite iteration (38)

$$d \sqsubseteq c \sqcap (c\ ;\ d) \quad \Rrightarrow \quad d \sqsubseteq c^+\ . \tag{101}$$

Applying this to the theorem requires one to show,

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

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 $(\mathsf{id}(X) \mathbin{⨾} w) \equiv w \equiv (w \mathbin{⨾} \mathsf{id}(X))$. 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 $w^*_X \mathrel{\widehat=} w^+ \vee \mathsf{id}(X)$ and note that if w is well founded then so is w^+.

Law 10.6 (rely-loop) *For 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 $p \Rrightarrow def(b)$,*

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

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

$$
\begin{array}{ll}
& \mathbf{rely}\ r \bullet [p,\ p' \wedge b_1' \wedge w^*_X] \\
\sqsubseteq & \text{as nondeterministic choice is idempotent and Lemma 2.9 (sequential) as } w^+ \mathbin{⨾} w^*_X \Rrightarrow w^*_X \\
& \mathbf{rely}\ r \bullet ([p,\ p' \wedge w^+]\ ;\ [p,\ p' \wedge b_1' \wedge w^*_X] \sqcap [p,\ p' \wedge b_1' \wedge w^*_X]) \\
\sqsubseteq & \text{by Law 4.19 (distribute-rely-choice) and Law 4.21 (distribute-rely-sequential)} \\
& ((\mathbf{rely}\ r \bullet [p,\ p' \wedge w^+])\ ;\ (\mathbf{rely}\ r \bullet [p,\ p' \wedge b_1' \wedge w^*_X])) \sqcap (\mathbf{rely}\ r \bullet [p,\ p' \wedge b_1' \wedge w^*_X]) \\
= & \text{distribute sequential over nondeterministic choice}
\end{array}
$$

$((\textbf{rely}\, r \bullet \lfloor p,\ p' \wedge w^+ \rfloor) \sqcap \textbf{skip})\ ;\ (\textbf{rely}\, r \bullet \lfloor p,\ p' \wedge b_1' \wedge w_X^* \rfloor)$
$\sqsubseteq$ by Lemma 2.8 (consequence) as $p \wedge r^* \Rrightarrow p' \wedge w_X^*$
$((\textbf{rely}\, r \bullet \lfloor p,\ p' \wedge w^+ \rfloor) \sqcap \textbf{skip})\ ;\ (\textbf{rely}\, r \bullet \lfloor p,\ r^* \wedge b_1' \rfloor)$
$\sqsubseteq$ by Law 10.1 (rely-test) as $p \wedge \neg\, b \Rrightarrow b_1$ and $p \wedge r \Rrightarrow (b_1 \Rightarrow b_1')$ and $p \Rrightarrow def(b)$
$((\textbf{rely}\, r \bullet \lfloor p,\ p' \wedge w^+ \rfloor) \sqcap \textbf{skip})\ ;\ [[\neg\, b]]$
$\sqsubseteq$ by Law 10.5 (refine-to-iteration)
$((\textbf{rely}\, r \bullet \lfloor p,\ p' \wedge w^+ \rfloor^+) \sqcap \textbf{skip})\ ;\ [[\neg\, b]]$
$=$ as w^+ is well founded using Law 10.4 (well-founded-termination) but with w^+ rather than w
$((\textbf{rely}\, r \bullet \lfloor p,\ p' \wedge w^+ \rfloor^{\omega+}) \sqcap \textbf{skip})\ ;\ [[\neg\, b]]$
$\sqsubseteq$ by Law 4.23 (distribute-rely-iteration)
$((\textbf{rely}\, r \bullet \lfloor p,\ p' \wedge w^+ \rfloor)^{\omega+} \sqcap \textbf{skip})\ ;\ [[\neg\, b]]$
$\sqsubseteq$ by Lemma 2.13 (fold/unfold-iteration) and $c^{\omega+} = c^{\omega}\ ;\ c$
$(\textbf{rely}\, r \bullet \lfloor p,\ p' \wedge w^+ \rfloor)^{\omega}\ ;\ [[\neg\, b]]$
$\sqsubseteq$ by Law 7.3 (rely-sequential) as $w_X^* \fatsemi w \equiv w^+$ because $\mathsf{id}(X) \fatsemi w \equiv w$
$((\textbf{rely}\, r \bullet \lfloor p,\ p' \wedge b_0' \wedge w_X^* \rfloor)\ ;\ (\textbf{rely}\, r \bullet \lfloor p \wedge b_0,\ p' \wedge w \rfloor))^{\omega}\ ;\ [[\neg\, b]]$
$\sqsubseteq$ by Lemma 2.8 (consequence) as $p \wedge r^* \Rrightarrow p' \wedge w_X^*$
$((\textbf{rely}\, r \bullet \lfloor p,\ r^* \wedge b_0' \rfloor)\ ;\ (\textbf{rely}\, r \bullet \lfloor p \wedge b_0,\ p' \wedge w \rfloor))^{\omega}\ ;\ [[\neg\, b]]$
$\sqsubseteq$ by Law 10.1 (rely-test) as $p \wedge b \Rrightarrow b_0$ and $p \wedge r \Rrightarrow (b_0 \Rightarrow b_0')$
$([[b]]\ ;\ (\textbf{rely}\, r \bullet \lfloor p \wedge b_0,\ p' \wedge w \rfloor))^{\omega}\ ;\ [[\neg\, b]]$
$=$ by the definition of a loop (14)
$\textbf{while}\, b\, \textbf{do}(\textbf{rely}\, r \bullet \lfloor p \wedge b_0,\ p' \wedge w \rfloor)$

□

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 $r \Rrightarrow \mathsf{id}(X)$ then $r \Rrightarrow w_X^*$. This law may be combined with Law 3.35 (guarantee-loop) to handle refinement of a specification to a loop in both guarantee and rely contexts.

11 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 [Owi75] (and is also used by [Jon81] and [dR01] to contrast the compositional rely/guarantee approach with the "Owicki/Gries" method).

11.1 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.

$$findp \mathrel{\widehat{=}} t : \textbf{rely}\, \mathsf{id}(\{v, t\}) \bullet \lfloor (t' = len(v) + 1 \vee satp(v, t')) \wedge notp(v, dom(v), t') \rfloor \qquad \lhd$$

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 $\mathsf{id}(\bar{x})$, and hence the laws using guarantees can be used wherever needed.

11.2 Representing the result 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 $len(v)$. 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 $min(ot, et)$.[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 9.3 (variable-rely-guarantee) for ot and et and Lemma 2.8 (consequence)

$$\begin{array}{l}\mathbf{var}\, ot, et \bullet \\ \quad ot, et, t : \mathbf{rely}\, \mathsf{id}(\{v, t, ot, et\}) \bullet \begin{bmatrix}(min(ot', et') = len(v) + 1 \vee satp(v, min(ot', et'))) \wedge \\ notp(v, dom(v), min(ot', et')) \wedge t' = min(ot', et')\end{bmatrix}\end{array} \quad \lhd$$

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

$\sqsubseteq$ by Law 7.3 (rely-sequential), Law 3.27 (distribute-guarantee) and Law 4.15 (weaken-rely)

$$\begin{array}{l} ot, et : \mathbf{rely}\, \mathsf{id}(\{v, ot, et\}) \bullet \begin{bmatrix}(min(ot', et') = len(v) + 1 \vee satp(v, min(ot', et'))) \wedge \\ notp(v, dom(v), min(ot', et'))\end{bmatrix}; \\ t : \mathbf{rely}\, \mathsf{id}(\{t, ot, et\}) \bullet \left[t' = min(ot, et)\right]\end{array} \quad \lhd$$

The specification $t : \mathbf{rely}\, \mathsf{id}(\{t, ot, et\}) \bullet \left[t' = min(ot, et)\right]$ can be refined to $t := min(ot, et)$ using Law 7.7 (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 7.7 (assignment-rely-guarantee).

$\sqsubseteq$ by Law 6.4 (trade-rely-guarantee-invariant); Law 7.3 (rely-sequential)

$$\begin{array}{l} ot := len(v) + 1\,; et := len(v) + 1; \\ \mathbf{guar\text{-}inv}\, min(ot, et) = len(v) + 1 \vee satp(v, min(ot, et)) \bullet \\ \quad ot, et : \mathbf{rely}\, \mathsf{id}(\{v, ot, et\}) \bullet \left[notp(v, \mathrm{dom}(v), min(ot', et'))\right]\end{array} \quad \lhd$$

11.3 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 6.3 (introduce-parallel-spec-weaken-rely)

$$\begin{array}{l} ot, et : \mathbf{guar}\, ot' \le ot \wedge et' = et \bullet \mathbf{rely}\, et' \le et \wedge \mathsf{id}(\{ot, v\}) \bullet \\ \quad \left[notp(v, odds(v), min(ot', et'))\right] \\ \| \\ ot, et : \mathbf{guar}\, et' \le et \wedge ot' = ot \bullet \mathbf{rely}\, ot' \le ot \wedge \mathsf{id}(\{et, v\}) \bullet \\ \quad \left[notp(v, evens(v), min(ot', et'))\right]\end{array} \quad \lhd$$

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.

[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] [Jon07] observes that achieving rely and/or guarantee conditions is often linked with data reification, for instance, viewing $min(ot, et)$ as a representation of the abstract variable t; this point is not pursued here.

11.4 Refining the branches to code

For the first branch of the parallel, the guarantee $et' = et$ is by Definition 3.36 (frame) equivalent to removing et from the frame of the branch.

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

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 9.3 (variable-rely-guarantee) for oc

$$\textbf{var}\, oc \bullet$$
$$oc, ot : \textbf{rely}\, et' \leq et \wedge \mathsf{id}(\{oc, ot, v\}) \bullet \left[notp(v, odds(v), min(ot', et'))\right] \qquad \lhd$$

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 postcondition $oc' \geq min(ot', et')$ 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 7.3 (rely-sequential), Law 6.4 (trade-rely-guarantee-invariant) and Law 7.7 (assignment-rely-guarantee) as follows.

$$\sqsubseteq oc := 1;$$
$$\textbf{guar-inv}\, notp(v, odds(v), oc) \wedge bnd(oc, v) \bullet$$
$$oc, ot : \textbf{rely}\, et' \leq et \wedge \mathsf{id}(\{oc, ot, v\}) \bullet \left[oc' \geq min(ot', et')\right] \qquad \lhd$$

A while loop is introduced using Law 10.6 (rely-loop). Only the first conjunct of the loop guard $oc < ot \wedge oc < et$ 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.

$\sqsubseteq$ by Law 10.6 (rely-loop)

$$\textbf{while}\, oc < ot \wedge oc < et\, \textbf{do}$$
$$oc, ot : \textbf{rely}\, et' \leq et \wedge \mathsf{id}(\{oc, ot, v\}) \bullet \left[oc < ot,\ -1 \leq ot' - oc' < ot - oc\right] \qquad \lhd$$

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

$\sqsubseteq$ by Law 4.15 (weaken-rely)

$$oc, ot : \textbf{rely}\, \mathsf{id}(\{oc, ot, v\}) \bullet \left[oc < ot,\ -1 \leq ot' - oc' < ot - oc\right] \qquad \lhd$$

At this stage one could use Law 7.6 (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 10.2 (rely-conditional).

$$\sqsubseteq \textbf{if}\, p(v(oc))\ \textbf{then}\ oc, ot : \textbf{rely}\, \mathsf{id}(\{oc, ot, v\}) \bullet \left[p(v(oc)) \wedge oc < ot,\ -1 \leq ot' - oc' < ot - oc\right]$$
$$\textbf{else}\ oc, ot : \textbf{rely}\, \mathsf{id}(\{oc, ot, v\}) \bullet \left[\neg p(v(oc)) \wedge oc < ot,\ -1 \leq ot' - oc' < ot - oc\right]$$

Finally, Law 7.7 (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.

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

11.5 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.

```
var $ot, et$ •
$ot := len(v) + 1$ ;
$et := len(v) + 1$ ;
( var $oc$ •                                  ||  var $ec$ •                                  ) ;
( $oc := 1$ ;                                 ||  $ec := 2$ ;                                 )
( while $oc < ot \wedge oc < et$ do           ||  while $ec < ot \wedge ec < et$ do           )
(     if $p(v(oc))$ then $ot := oc$           ||      if $p(v(ec))$ then $et := ec$           )
(                   else $oc := oc + 2$       ||                    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 [CJ07]. In particular, the use of Law 6.3 (introduce-parallel-spec-weaken-rely) in Section 11.3 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.

12 Conclusions

12.1 Summary

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 post condition, 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 (**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 [MV94b] 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 [DH10]. The guarantee construct can be thought of as providing a context in which its body is refined. [NH97] 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. 11) 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 rely command $(\mathbf{rely}\, r \bullet [p,\ q])$ 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 $(\mathbf{rely}\, r \bullet c_z)$ in context $z \vee r$ is $stops(c, z)$, which depends on the context z.

In order for $(\mathbf{rely}\, r \bullet c)$ 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

$$\{stops(c,z)\}c \sqsubseteq_z (\mathbf{rely}\, r \bullet c_z) \parallel \langle r \vee \mathsf{id}\rangle^*$$

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.

12.2 Related work

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 [Flo67] although it is interesting to note that pioneers such as Turing and von Neumann recognised that something of the sort would aid reasoning [Jon03a]. The key contribution of [Hoa69] was to change the viewpoint away from program annotations to a system of judgements about "Hoare triples". One crucial 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 [Bac81, Mor87, Mor88, Mor94, BvW98] 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.

The refinement calculus presented in this paper provides a method for deriving correct implementations from specifications via a sequence of intermediate steps, as opposed to verifying an implementation against a specification. The latter style is exemplified by Jones' quintuples [Jon81, Jon83], as discussed in Sect. 2.2. The derivational style provides a formal way of structuring large proofs into intuitive chunks and incrementally introducing the implementation control structure. The inference-rule style is more suited to monolithic verifications, although it is often the case that applications of the method to non-trivial examples are presented in an incremental style (e.g., [CJ07]).

The thesis by [Din00] and the more accessible [Din02] offer an approach towards a refinement calculus view of rely/guarantee reasoning. The main difference is that Dingel does not treat rely and guarantees as separate commands, and instead each refinement step includes a four-tuple of pre, rely, guarantee and post conditions. Our command-based approach is more flexible in both writing specifications and structuring proofs. Furthermore, Dingel's writings do not follow Jones' original relational view because, for example, he uses post conditions that are sets of predicates of single states — on this point he follows [Sti86]. 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. Dingel's semantics, like that of [Bro07], defines semantic equivalence modulo finite stuttering (program steps that don't change the state) and mumbling (two program steps may be merged into a single step with the same overall effect). Because the set of traces of a specification command $\lceil q \rceil$ is closed under finite stuttering and mumbling[12] 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.

Rely-guarantee thinking is suited to dealing with concurrency where interference (or co-operation, as in the example in Sect. 11) between processes is unavoidable. Separation Logic [Rey02, IO01, OYR09] is especially suited to cases where for the most part interference is avoided, for instance where two programs (sequential or concurrent) operate on separate parts of a shared heap. The combination of the two areas of research is a current topic; for instance, [VP07] provide inference rules for rely-guarantee style quintuples where assertions from Separation Logic may be used inside the relations. SAGL [FFS07] makes a similar attempt to bring together separation logic with rely/guarantee methods. [TW11] also make use of separation logic combined with ideas from the rely-guarantee approach to reason about concurrent objects. Operations on local state are not affected by interference and can be considered atomic; that has similarities with the **uses** construct here, which allows sequential refinement laws to be used. Current attempts to find a deeper way of combining the approaches include ideas by Matt Parkinson (private communication) and [Jon12, JHC14]. It is not appropriate to explore further this distinct avenue of research in the current paper.

A large body of work has been developed for verifying concurrent programs based on using linearizability [HW90] as the correctness criteria: in inference rule-style, see for instance [DSW11], and in derivation (refinement calculus) style see [GC09]. Specifications of operations in the context of linearizability are typically a single atomic operation, which is successively decomposed into an implementation which is not atomic but appears so to an observer. For instance, a linearisable "push" of an element onto a stack must take place at some point between the invocation of the operation and its completion, although by the time of completion that element may already have been "popped" by some other process. This type of specification is in contrast to our own, in which a relation between the pre (invocation-time) and post (completion-time) states must hold, regardless of interference from other processes. As such our specifications consist of the set of all possible traces, formed from single atomic steps or otherwise, that satisfy the relation when taken as a whole. Hence we do not "split" [JP08] atomicity during refinement, but rather successively direct the development towards an implementation that generates correct traces.

12.3 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 reactive 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

[12]A finite trace t_0 of $\lceil q \rceil$ satisfies q (end-to-end) and hence any trace t_1 that is equivalent to t_0 modulo finite stuttering also satisfies q and is therefore a trace of $\lceil q \rceil$. If a trace t_0 satisfying q contains two consecutive program steps $\pi(\sigma_1, \sigma_2)$ and $\pi(\sigma_2, \sigma_3)$ then the trace t_0 with those two steps replaced with a single step $\pi(\sigma_1, \sigma_3)$ also satisfies q, and hence is a trace of $\lceil q \rceil$.

all interference (program) steps, pre-rely-guarantee-post specifications are not rich enough to be able to express precisely the behaviour of all processes.[13] 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. Nice examples are the representation of the frame of a command by a guarantee, and the novel test command used both for guards in control structures and to define the assignment command.

There is also a clear case for reconsidering data reification in the new framework. The concept of "possible values" was introduced by [JP11] and linked to non-deterministic states by [HBDJ13] — 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, Gerhard Schellhorn and (anonymous) referees for feedback on our earlier work.

APPENDIX

A A rely-able semantics

A.1 Basic definitions

Memory is represented by a *state* that is either undefined ($\bot$) or maps variables to their values, which is represented formally by the type $\Sigma \mathrel{\widehat{=}} \bot \mid (\mathit{Var} \to \mathit{Val})$. 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 [Jon87]). 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 [dBHdR99].[14] In addition, we distinguish termination of a command in state σ by the label "$\checkmark(\sigma)$". A step $\alpha \in \mathcal{L}$ may adorn a transition arrow, and its syntax follows.

$$\alpha ::= \pi(\sigma, \sigma') \mid \epsilon(\sigma, \sigma') \mid \checkmark(\sigma) \tag{A.102}$$

Recall a *trace* is a *consistent* (20) sequence of steps. For the semantics the additional command **nil** is used to indicate a terminated process; **nil** has no transitions. An example execution of a command c is of the form

$$c \xrightarrow{\epsilon(\sigma_0,\sigma_1)} c_1 \xrightarrow{\pi(\sigma_1,\sigma_2)} c_2 \xrightarrow{\epsilon(\sigma_2,\sigma_3)} c_3 \xrightarrow{\epsilon(\sigma_3,\sigma_4)} c_4 \xrightarrow{\pi(\sigma_4,\sigma_5)} c_5 \xrightarrow{\epsilon(\sigma_5,\sigma_6)} c_6 \xrightarrow{\checkmark(\sigma_6)} \mathbf{nil}$$

where $c, c_1, \ldots$ 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

$$\epsilon(\sigma_0, \sigma_1), \pi(\sigma_1, \sigma_2), \epsilon(\sigma_2, \sigma_3), \epsilon(\sigma_3, \sigma_4), \pi(\sigma_4, \sigma_5), \epsilon(\sigma_5, \sigma_6), \checkmark(\sigma_6)\ .$$

[13]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.

[14] 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 [Acz83].

A trace representing a terminated behaviour always ends with a label of the form $\surd(\sigma)$, where σ matches the final state of the preceding trace (if it is non-empty). Infinite traces do not contain a $\surd(\sigma)$ step.

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) [Mos04a, Mos04b], which has some advantages over the seminal Plotkin-style operational semantics [Plo04] in which the states form part of the configuration rather than the label: $\langle c_0, \sigma_0 \rangle \to \langle c_1, \sigma_1 \rangle$. Of particular importance for the semantics presented here is that by collecting the steps in a trace we are able to more succinctly describe properties of the traces, including 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.

$$[\![c]\!] \;\widehat{=}\; \{t \in \mathit{Trace} \mid c \overset{t}{\Rightarrow} \mathbf{nil} \vee c \overset{t}{\Rightarrow} \infty\}$$

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 [Jon03b], although the latter is certainly possible (as demonstrated by [Bro07]).

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.

$$\frac{\sigma(x) = v \quad v \in \mathit{Val}}{x \xrightarrow{\pi(\sigma,\sigma)} v}$$

To evaluate a binary expression $e_1 \oplus e_2$ the operands may be evaluated to values in any order.[15] The final value is found by applying the underlying mathematical operator to the values, which we write $\mathit{eval}(\oplus, (v_1, v_2))$. The function eval may return the undefined value $\perp$ if the operator is not defined for those values (e.g., division by zero). Unary operators are evaluated similarly.

$$\frac{e_1 \xrightarrow{\alpha} e_1'}{e_1 \oplus e_2 \xrightarrow{\alpha} e_1' \oplus e_2} \qquad \frac{e_2 \xrightarrow{\alpha} e_2'}{e_1 \oplus e_2 \xrightarrow{\alpha} e_1 \oplus e_2'} \qquad \frac{v_1, v_2 \in \mathit{Val}}{v_1 \oplus v_2 \xrightarrow{\pi(\sigma,\sigma)} \mathit{eval}(\oplus, (v_1, v_2))}$$

$$\frac{e \xrightarrow{\alpha} e'}{\ominus e \xrightarrow{\alpha} \ominus e'} \qquad \frac{v \in \mathit{Val}}{\ominus v \xrightarrow{\pi(\sigma,\sigma)} \mathit{eval}(\ominus, v)}$$

In addition, the evaluation of an expression may be interrupted by an environment step at any time, as given by the following rule.

$$e \xrightarrow{\epsilon(\sigma,\sigma')} e \qquad \text{(A.103)}$$

The interference step does not change the expression e, 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.

[15] 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.

A.4 Operational semantics of primitive code commands

Termination of a command in state σ is indicated by the transition $c \xrightarrow{\surd(\sigma)} \mathbf{nil}$, where **nil** represents the terminated command that can perform no steps whatsoever. That a command c aborts in state σ is indicated by the predicate $c_{\sigma\times}$. A precondition $\{p\}$ may terminate immediately in any state σ in which p holds, and abort in any state in which it does not.

$$\frac{\sigma \in p}{\{p\} \xrightarrow{\surd(\sigma)} \mathbf{nil}} \qquad \frac{\sigma \notin p}{\{p\}_{\sigma\times}} \tag{A.104}$$

The "worst" command is **abort**, defined as $\{\mathsf{false}\}$, and the command that terminates immediately, **skip**, is defined as $\{\mathsf{true}\}$. A command that has aborted in state σ may take any further finite or infinite behaviour, or terminate immediately.

$$\frac{c_{\sigma\times}}{c \xrightarrow{\pi(\sigma,\sigma')} \mathbf{abort}} \qquad \frac{c_{\sigma\times}}{c \xrightarrow{\epsilon(\sigma,\sigma')} \mathbf{abort}} \qquad \frac{c_{\sigma\times}}{c \xrightarrow{\surd(\sigma)} \mathbf{nil}} \tag{A.105}$$

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.

$$\frac{\sigma \in p \quad (\sigma,\sigma') \in q}{\langle p, q\rangle \xrightarrow{\pi(\sigma,\sigma')} \mathbf{skip}} \qquad \langle p, q\rangle \xrightarrow{\epsilon(\sigma,\sigma')} \langle p, q\rangle \qquad \frac{\sigma \notin p}{\langle p, q\rangle_{\sigma\times}} \tag{A.106}$$

Note that $\langle \mathsf{true}, \mathsf{false}\rangle$ has no program transitions: all its traces are infinite in length and contain only environment steps.

Nondeterministic choice between a set of commands C, $(\bigsqcap C)$, can behave as any command within C. If any $c \in C$ can terminate, i.e. c' is **nil**, the choice can terminate and if any $c \in C$ can abort, the choice can abort.

$$\frac{c \in C \quad c \xrightarrow{\alpha} c'}{(\bigsqcap C) \xrightarrow{\alpha} c'} \tag{A.107}$$

A sequential composition, $(c_1 \mathbin{;} c_2)$, is defined to execute c_1 until it terminates, after which c_2 may begin, provided c_2 begins execution in the state in which c_1 terminated.

$$\frac{c_1 \xrightarrow{\alpha} c_1' \quad (\forall \sigma \bullet \alpha \neq \surd(\sigma))}{c_1 \mathbin{;} c_2 \xrightarrow{\alpha} c_1' \mathbin{;} c_2} \qquad \frac{(c_1)_{\sigma\times}}{(c_1 \mathbin{;} c_2)_{\sigma\times}} \tag{A.108}$$

$$\frac{c_1 \xrightarrow{\surd(\sigma)} \mathbf{nil} \quad c_2 \xrightarrow{\alpha} c_2' \quad pre(\alpha) = \sigma}{c_1 \mathbin{;} c_2 \xrightarrow{\alpha} c_2'} \tag{A.109}$$

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, terminating when both may terminate. If either can abort, so can their conjunction.

$$\frac{c_1 \xrightarrow{\alpha} c_1' \quad c_2 \xrightarrow{\alpha} c_2' \quad (\forall \sigma \bullet \alpha \neq \surd(\sigma))}{c_1 \Cap c_2 \xrightarrow{\alpha} c_1' \Cap c_2'} \qquad \frac{c_1 \xrightarrow{\surd(\sigma)} \mathbf{nil} \quad c_2 \xrightarrow{\surd(\sigma)} \mathbf{nil}}{c_1 \Cap c_2 \xrightarrow{\surd(\sigma)} \mathbf{nil}} \tag{A.110}$$

$$\frac{(c_1)_{\sigma\times}}{(c_1 \Cap c_2)_{\sigma\times}} \qquad \frac{(c_2)_{\sigma\times}}{(c_1 \Cap c_2)_{\sigma\times}} \tag{A.111}$$

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 transition of the parallel composition) as follows.

$$(\pi(\sigma,\sigma'),\epsilon(\sigma,\sigma')) \;\; \mathsf{match} \;\; \pi(\sigma,\sigma') \tag{A.112}$$
$$(\epsilon(\sigma,\sigma'),\pi(\sigma,\sigma')) \;\; \mathsf{match} \;\; \pi(\sigma,\sigma') \tag{A.113}$$
$$(\epsilon(\sigma,\sigma'),\epsilon(\sigma,\sigma')) \;\; \mathsf{match} \;\; \epsilon(\sigma,\sigma') \tag{A.114}$$

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. When either command terminates, it is removed from the composition (note that Rule (A.116) also allows synchronised termination).

$$\frac{c_1 \xrightarrow{\alpha_1} c_1' \qquad c_2 \xrightarrow{\alpha_2} c_2' \qquad (\alpha_1,\alpha_2)\ \mathsf{match}\ \alpha}{(c_1 \parallel c_2) \xrightarrow{\alpha} (c_1' \parallel c_2')} \tag{A.115}$$

$$\frac{c_1 \xrightarrow{\checkmark(\sigma)} \mathbf{nil} \quad c_2 \xrightarrow{\alpha} c_2' \quad pre(\alpha)=\sigma}{(c_1 \parallel c_2) \xrightarrow{\alpha} c_2'} \qquad \frac{c_2 \xrightarrow{\checkmark(\sigma)} \mathbf{nil} \quad c_1 \xrightarrow{\alpha} c_1' \quad pre(\alpha)=\sigma}{(c_1 \parallel c_2) \xrightarrow{\alpha} c_1'} \tag{A.116}$$

$$\frac{(c_1)_{\sigma\times}}{(c_1 \parallel c_2)_{\sigma\times}} \qquad \frac{(c_2)_{\sigma\times}}{(c_1 \parallel c_2)_{\sigma\times}} \tag{A.117}$$

A local state command $(\mathbf{state}\ y \mapsto v \bullet c)$ 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.

$$\frac{c \xrightarrow{\pi(\sigma[y\mapsto v],\sigma'[y\mapsto v'])} c' \quad \sigma'(y)=\sigma(y)}{(\mathbf{state}\ y \mapsto v \bullet c) \xrightarrow{\pi(\sigma,\sigma')} (\mathbf{state}\ y \mapsto v' \bullet c')} \tag{A.118}$$

$$\frac{c \xrightarrow{\epsilon(\sigma[y\mapsto v],\sigma'[y\mapsto v])} c'}{(\mathbf{state}\ y \mapsto v \bullet c) \xrightarrow{\epsilon(\sigma,\sigma')} (\mathbf{state}\ y \mapsto v \bullet c')} \tag{A.119}$$

$$\frac{c \xrightarrow{\checkmark(\sigma[y\mapsto v])} \mathbf{nil}}{(\mathbf{state}\ y \mapsto v \bullet c) \xrightarrow{\checkmark(\sigma)} \mathbf{nil}} \qquad \frac{c_{\sigma[y\mapsto v]\times}}{(\mathbf{state}\ y \mapsto v \bullet c)_{\sigma\times}} \tag{A.120}$$

Rule (A.118) 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.119) 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. Rule (A.120) states that a local state may terminate in any state that is consistent with the local value. For instance, $(\mathbf{state}\ y \mapsto 4 \bullet [y'=4])$: this can terminate immediately, but this must not be allowed if the specification is $y'=5$, for instance.

A program step $\pi(\sigma,\sigma')$ of $(\mathbf{uses}\,X \bullet c)$ is permitted if and only if 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 $(\sigma,\sigma') \in \mathsf{id}(X)$, and similarly lifted to pairs of states.

$$\frac{c \xrightarrow{\pi(\sigma,\sigma')} c' \quad \sigma \overset{\overline{X}}{=} \sigma' \quad \forall \sigma_1,\sigma_1' \bullet (\sigma_1,\sigma_1') \overset{X}{=} (\sigma,\sigma') \wedge \sigma_1 \overset{\overline{X}}{=} \sigma_1' \Rightarrow c \xrightarrow{\pi(\sigma_1,\sigma_1')} c'}{(\mathbf{uses}\,X \bullet c) \xrightarrow{\pi(\sigma,\sigma')} (\mathbf{uses}\,X \bullet c')} \tag{A.121}$$

$$\frac{c \xrightarrow{\epsilon(\sigma,\sigma')} c'}{(\textbf{uses}\, X \bullet c) \xrightarrow{\epsilon(\sigma,\sigma')} (\textbf{uses}\, X \bullet c')} \tag{A.122}$$

$$\frac{c \xrightarrow{\surd(\sigma)} \textbf{nil}}{(\textbf{uses}\, X \bullet c) \xrightarrow{\surd(\sigma)} \textbf{nil}} \qquad \frac{c_{\sigma\times}}{(\textbf{uses}\, X \bullet c)_{\sigma\times}} \tag{A.123}$$

Environment steps for the body of a uses command are simply promoted, and if the body of a "uses" command can terminate or abort, the "uses" command terminates or aborts, respectively.

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.

$$\frac{b \overset{t_0}{\Longrightarrow} \mathsf{true} \quad t_0 \overset{st}{=} t_1}{[[b]] \overset{t_1}{\Longrightarrow} \textbf{skip}} \qquad \frac{b \overset{t_0}{\Longrightarrow} \bot \quad t_0 \overset{st}{=} t_1}{[[b]] \overset{t_1}{\Longrightarrow} \textbf{abort}} \qquad \frac{b \overset{t_0}{\Longrightarrow} \infty \quad t_0 \overset{st}{=} t_1}{[[b]] \overset{t_1}{\Longrightarrow} \infty} \tag{A.124}$$

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 a trace.

A specification command $\left[q\right]$ 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 $\left[q\right]$ aborts. Recall from (23) that $env(t) \subseteq \mathsf{id}$ holds if every environment step in t satisfies id and that $pre(t)$ abbreviates $pre(t(0))$ and $post(t)$ abbreviates $post(t(\#t-1))$.

$$\frac{env(t) \subseteq \mathsf{id} \quad (pre(t), post(t)) \in q \quad t \in \mathcal{L}^* \cap Complete}{\left[q\right] \overset{t}{\Longrightarrow} \textbf{nil}} \tag{A.125}$$

$$\frac{env(t) \nsubseteq \mathsf{id} \quad t \in \mathcal{L}^* - Complete}{\left[q\right] \overset{t}{\Longrightarrow} \textbf{abort}} \tag{A.126}$$

$$\frac{env(t) \subseteq \mathsf{id} \quad interrupted(t) \quad t \in \mathcal{L}^\infty \cap Trace}{\left[q\right] \overset{t}{\Rightarrow} \infty} \tag{A.127}$$

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. This includes the possibility of immediate termination – the trace $\surd(\sigma)$, if $(\sigma, \sigma) \in q$. 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 a sequence of environment steps.

$$interrupted(t) \quad \widehat{=} \quad (\exists\, i \in \mathrm{dom}(t) \bullet (\forall j \in \mathrm{dom}(t) \bullet i < j \Rightarrow (\exists\, \sigma, \sigma' \bullet t(j) = \epsilon(\sigma, \sigma'))))$$

Because the rule for a specification command only gives the full traces for the command we introduce the command (c **after** t) to stand for the remaining execution of command c after it has executed trace t. A trace with a single step α is equivalent to a transition on α.

$$\frac{c_1 \xRightarrow{t_1 \frown t_2} c_2}{c_1 \xRightarrow{t_1} (c_1 \textbf{ after } t_1) \qquad (c_1 \textbf{ after } t_1) \xRightarrow{t_2} c_2} \qquad \frac{c_1 \xRightarrow{[\alpha]} c_2}{c_1 \xrightarrow{\alpha} c_2} \tag{A.128}$$

A rule similar to the first is required for infinite traces but with c_2 replaced by ∞.

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 2.2). It has elements of both sequential program refinement and notions such as (bi)simulation from the process algebra literature [Mil89]. 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 B.1 *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 $(\sigma, \sigma') \in r \vee \mathsf{id}$, there exists a c' such that both*

$$c \xrightarrow{\alpha} c' \tag{B.129}$$
$$c' \sqsubseteq_r d' \tag{B.130}$$

Proof. By induction, under the above conditions, any complete trace of d generated by the small-step operational semantics must be a trace of c. This is similar to the definition of *simulation* given by [Mil89]. □

Proofs using Theorem B.1 proceed by first discharging B.129 by case-analysis on the possible transitions of d. Condition B.130 is usually trivial because the basic laws are defined so that c and d (and hence c' and d') are structurally similar. Theorem B.1 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 $[q]$, 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 B.2 (precondition-traces) *For any predicate p and command c,*

$$[\![\{p\}c]\!] = \{t \in \mathit{Trace} \mid \mathit{pre}(t) \in p \Rightarrow t \in [\![c]\!]\} \tag{B.131}$$
$$\{p\}c \sqsubseteq_r d \Leftrightarrow (\forall t \in [\![d]\!]_r \bullet \mathit{pre}(t) \in p \Rightarrow t \in [\![c]\!]) \tag{B.132}$$

Proof. By Rules (A.104) and (A.105), if $\mathit{pre}(t) \notin p$ then $\{p\}$ may take any behaviour in state $\mathit{pre}(t)$, hence $t \in [\![\{p\}c]\!]$. Otherwise, if $\mathit{pre}(t) \in p$, then $t \in [\![\{p\}c]\!]$ if and only if $t \in [\![c]\!]$. Hence (B.131) holds. Property (B.132) follows directly from (B.131) and Definition 2.1 (refinement-in-context). □

Lemma 2.5 (parallel-precondition) $\{p\}(c \parallel d) = (\{p\}c) \parallel (\{p\}d)$.
Proof. The proof is based on Lemma B.2 (precondition-traces). Consider a trace t of $\{p\}(c \parallel d)$. If $\mathit{pre}(t) \in p$ then t is a trace of $c \parallel d$, and hence there exist traces t_c and t_d of c and d respectively, which match in a parallel combination to give t and furthermore both $\mathit{pre}(t_c) \in p$ and $\mathit{pre}(t_d) \in p$. Hence t_c is a trace of $\{p\}c$ that matches t_d which is a trace of $\{p\}d$ and it follows that t is a trace of $(\{p\}c) \parallel (\{p\}d)$. If $\mathit{pre}(t) \notin p$, t is a trace of both $\{p\}(c \parallel d)$ and $\{p\}c$. Let t_1 be the trace t with all program steps replaced with the equivalent environment steps. Because $\mathit{pre}(t_1) \notin p$, t_1 is a trace of $\{p\}d$ and hence t is a trace of the right side. □

Lemma 2.6 (refine-specification) $([p,\ q] \sqsubseteq c) \Leftrightarrow ([p,\ q] \sqsubseteq_{\mathsf{id}} c)$.

Proof. By Lemma B.2 (precondition-traces) $\{p\}\ \left[q\right] \sqsubseteq c$ if and only if

$$\forall\, t \in [\![c]\!] \bullet pre(t) \in p \Rightarrow t \in [\![\left[q\right]]\!]$$

If $t \in [\![c]\!]$ and $env(t) \not\subseteq \mathsf{id}$ then by Rule (A.125), $t \in [\![\left[q\right]]\!]$. The remaining case is when $env(t) \subseteq \mathsf{id}$ and requires

$$\forall\, t \in [\![c]\!]_{\mathsf{id}} \bullet pre(t) \in p \Rightarrow t \in [\![\left[q\right]]\!]$$

which by Lemma B.2 (precondition-traces) is equivalent to $\{p\}\ \left[q\right] \sqsubseteq_{\mathsf{id}} c$. □

Lemma 2.7 (make-atomic) $\left[p,\ q\right] \sqsubseteq \langle p, q\rangle$.
Proof. By Lemma 2.6 (refine-specification) it is sufficient to show $\{p\}\ \left[q\right] \sqsubseteq_{\mathsf{id}} \langle p, q\rangle$ and hence by Lemma B.2 (precondition-traces) that

$$\forall\, t \in [\![\langle p, q\rangle]\!]_{\mathsf{id}} \bullet pre(t) \in p \Rightarrow t \in [\![\left[q\right]]\!]$$

By Rule (A.106) any complete trace $t \in [\![\langle p, q\rangle]\!]_{\mathsf{id}}$ has one of the following forms: a) a finite number of id environment steps containing a single program step (satisfying p and q), followed by termination; b) a finite number of id environment steps, followed by a single program step that does not satisfy p, followed by any trace; or c) 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), because every initial environment step satisfies id, if $pre(t) \in p$ holds then p must also hold for the atomic step, and hence no traces t in case b) satisfy $pre(t) \in p$.

Case c) covers *interrupted* traces which are in $[\![\left[q\right]]\!]$. □

Lemma 2.8 (consequence) Assuming $p_0 \Rrightarrow p_1$, and $p_0 \wedge q_1 \Rrightarrow q_0$,

$$\begin{array}{rcl}\langle p_0, q_0\rangle & \sqsubseteq & \langle p_1, q_1\rangle \\ \left[p_0,\ q_0\right] & \sqsubseteq & \left[p_1,\ q_1\right]\end{array}$$

Proof. Consider a trace $t \in [\![\langle p_1, q_1\rangle]\!]$. By Rule (A.106), t may start with any sequence of environment steps followed by either a program step satisfying q_1 if p_1 holds or it may abort if p_1 does not hold. Similarly, a trace of $\langle p_0, q_0\rangle$ may do any sequence of environment steps. If after the environment steps p_0 holds, by assumption so so p_1 and as $p_0 \wedge q_1 \Rrightarrow q_0$ any step satisfying q_1 also satisfies q_0 and hence the step can be taken by $\langle p_0, q_0\rangle$. If p_0 does not hold, $\langle p_0, q_0\rangle$ aborts allowing any behaviour for $\langle p_1, q_1\rangle$.

For specifications we need to show $\left[p_0,\ q_0\right] \sqsubseteq \left[p_1,\ q_1\right]$. By Lemma 2.6 (refine-specification) and Lemma 2.4 (precondition) part (29) as $p_0 \Rrightarrow p_1$ it is sufficient to show $\{p_0\}\ \left[q_0\right] \sqsubseteq_{\mathsf{id}} \left[q_1\right]$ and hence by Lemma B.2 (precondition-traces) to show that

$$\forall\, t \in [\![\left[q_1\right]]\!]_{\mathsf{id}} \bullet pre(t) \in p_0 \Rightarrow t \in [\![\left[q_0\right]]\!] \ . \qquad \text{(B.133)}$$

By Rule (A.125), $t \in [\![\left[q_1\right]]\!]_{\mathsf{id}}$ 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.125) 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 2.9 (sequential) Assume $p_0 \wedge ((q_0 \wedge p_1') \mathbin{⨾} q_1) \Rrightarrow q$,

$$\left[p_0,\ q\right] \ \sqsubseteq\ \left[p_0,\ q_0 \wedge p_1'\right] \,;\, \left[p_1,\ q_1\right] \ .$$

Proof. By Lemma 2.4 (precondition) part (29) and Lemma 2.6 (refine-specification) it is sufficient to show $\{p_0\}\ \left[q\right] \sqsubseteq_{\mathsf{id}} \left[q_0 \wedge p_1'\right] \,;\, \left[p_1,\ q_1\right]$ and hence by Lemma B.2 (precondition-traces),

$$\forall\, t \in [\![\left[q_0 \wedge p_1'\right] \,;\, \left[p_1,\ q_1\right]]\!]_{\mathsf{id}} \bullet pre(t) \in p_0 \Rightarrow t \in [\![\left[q\right]]\!] \ .$$

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 $(pre(t_0), post(t_0)) \in (q_0 \wedge p_1')$ and hence $post(t_0) \in p_1$. Because t is consistent (20), $pre(t_1) \in p_1$ and hence as t_1 is a trace of $\left[q_1\right]$, either $interrupted(t_1)$ or t_1 is finite and $(pre(t_1), post(t_1)) \in q_1$. 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 $pre(t) \in p_0$ and $(pre(t), post(t)) \in ((q_0 \wedge p_1') \fatsemi q_1)$ and because $p_0 \wedge ((q_0 \wedge p_1') \fatsemi q_1) \Rrightarrow q$, $(pre(t), post(t)) \in q$ and hence t is a trace of $\left[q\right]$. □

Lemma 2.10 (nondeterminism-traces) $[\![\bigsqcap C]\!] = \bigcup_{c \in C} [\![c]\!]$.
Proof. By Rule (A.107) 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 2.12 (introduce-test) $\left[def(b),\ b \wedge \mathsf{id}\right] \sqsubseteq [[b]]$.
Proof. By Lemma 2.6 (refine-specification) and Lemma B.2 (precondition-traces) it is sufficient to show

$$\forall t \in [\![[[b]]]\!]_{\mathsf{id}} \bullet pre(t) \in def(b) \Rightarrow t \in [\![\left[b \wedge \mathsf{id}\right]]\!]$$

By Rule (A.124) 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 $def(b)$ 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 $\left[b \wedge \mathsf{id}\right]$. Finally, any interrupted trace of $[[b]]$ is also a possible behaviour of the $\left[b \wedge \mathsf{id}\right]$ (by Rule (A.125)). □

Lemma 2.14 (iteration-induction) For any relation r and commands c, d and x,

$$x \sqsubseteq_r d \sqcap c \mathbin{;} x \quad \Rightarrow \quad x \sqsubseteq_r c^* \mathbin{;} d \tag{B.134}$$

$$c \mathbin{;} x \sqsubseteq_r x \quad \Rightarrow \quad c^\infty \sqsubseteq_r x \tag{B.135}$$

$$d \sqcap c \mathbin{;} x \sqsubseteq_r x \quad \Rightarrow \quad c^\omega \mathbin{;} d \sqsubseteq_r x \tag{B.136}$$

To aid in the proof of this lemma we introduce the command $(\mathbf{env}\, r \bullet c)$ that behaves as c if all the environment steps satisfy r but otherwise aborts; it is used to simplify some proofs in the theory. Its traces are given by $[\![\mathbf{env}\, r \bullet c]\!] = \{t \in Trace \mid env(t) \subseteq r \vee \mathsf{id} \Rightarrow t \in [\![c]\!]\}$.
Proof. Standard fixed point theory gives

$$y \sqsubseteq d \sqcap c \mathbin{;} y \quad \Rightarrow \quad y \sqsubseteq c^* \mathbin{;} d\ . \tag{B.137}$$

To show (B.134), note that using (B.137)

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

As a witness for y choose $(\mathbf{env}\, r \bullet x)$. This gives $x \sqsubseteq_r (\mathbf{env}\, r \bullet x)$ by Definition 2.1 (refinement-in-context). The refinement $y \sqsubseteq d \sqcap c \mathbin{;} y$ also holds as follows.

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

Hence (B.134) holds. The proofs of (B.135) and (B.136) are similar, except the witness for y is a process with traces $[\![x]\!]_r$. □

Lemma 2.21 (specification-term)

$$stops(\left[q\right], r) \equiv \mathsf{true}, \qquad \text{if } r \Rrightarrow \mathsf{id} \tag{B.138}$$

$$stops(\left[q\right], r) \equiv \mathsf{false}, \qquad \text{if } r \not\Rrightarrow \mathsf{id} \tag{B.139}$$

Proof. By Rule (A.125), every trace t of $\left[q\right]$ in an environment satisfying id is finite (or interrupted) and hence $stops(\left[q\right], \mathsf{id}) \equiv \mathsf{true}$. If $r \not\Rrightarrow \mathsf{id}$, then in an environment satisfying r, by Rule (A.126), $\left[q\right]$ can abort and hence has a nonterminating trace and hence $stops(\left[q\right], r) \equiv \mathsf{false}$. □

Lemma 3.5 (conjunction-strict) We focus on $\{p\}(c \Cap d) = (\{p\}c) \Cap d$, the other cases follow similarly.

Proof. By Rule (A.110) 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 both may terminate. The precondition p requires a further two sub-cases by Rule (A.104): a) when p is satisfied by α, and b) when it is not. It is clear from Rule (A.111) that both sides may abort in any state σ in which p does not hold, hence we need consider the case where $pre(\alpha) \in p$ only.

- Case 1. In this case α is trivially a step of each side of the equality.
- Case 2. Since all three commands are terminating (recalling $pre(\alpha) \in p$), both sides trivially terminate.

□

Lemma 3.13 (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. $depends\_only(g_1, \bar{x})$,

$$\begin{aligned} c \Cap (d_0 \Cap d_1) &= (c \Cap d_0) \Cap (c \Cap d_1) && \text{(B.140)}\\ c \Cap (\textstyle\bigsqcap D) &= \textstyle\bigsqcap \{d \in D \bullet (c \Cap d)\} && \text{(B.141)}\\ \langle g\rangle^{\omega} \Cap (c \,;\, d) &= (\langle g\rangle^{\omega} \Cap c) \,;\, (\langle g\rangle^{\omega} \Cap d) && \text{(B.142)}\\ \langle g\rangle^{\omega} \Cap (c \parallel d) &= (\langle g\rangle^{\omega} \Cap c) \parallel (\langle g\rangle^{\omega} \Cap d) && \text{(B.143)}\\ \langle g_1\rangle^{\omega} \Cap (\mathbf{var}\, x \bullet c) &= \mathbf{var}\, x \bullet (\langle g_1\rangle^{\omega} \Cap c) && \text{(B.144)}\\ \langle g\rangle^{\omega} \Cap (c^{\omega}) &= (\langle g\rangle^{\omega} \Cap c)^{\omega} && \text{(B.145)} \end{aligned}$$

Proof. Property (B.140) holds since strict conjunction is associative, commutative and idempotent.

For (B.141), 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 (B.142) 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 (B.143) 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 (B.144) $(\langle g_1\rangle^{\omega} \Cap c)$ 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 $\pi(\sigma[x \mapsto v], \sigma'[x \mapsto v])$ for any v. If c can do a step $\pi(\sigma, \sigma')$ then $(\mathbf{var}\, x \bullet c)$ can do a step $\pi(\sigma[x \mapsto v], \sigma'[x \mapsto v])$ for any v and as g_1 is independent of x, $(\sigma[x \mapsto v], \sigma'[x \mapsto v]) \in g_1 \Leftrightarrow (\sigma, \sigma') \in g_1$. Hence the traces of the left and right sides are the same.

Lemma 2.17 (fusion) is applied to prove (B.145) by choosing $F = (\lambda x \bullet \mathbf{skip} \sqcap c \,;\, x)$ and hence $\mu F = c^{\omega}$, choosing $G = (\lambda x \bullet \mathbf{skip} \sqcap (\langle g\rangle^{\omega} \Cap c) \,;\, x)$, and hence $\mu G = (\langle g\rangle^{\omega} \Cap c)^{\omega}$, 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 Lemma 2.17 (fusion), (B.145) holds if,

$$\langle g\rangle^{\omega} \Cap (\mathbf{skip} \sqcap c \,;\, x) \quad = \quad \mathbf{skip} \sqcap (\langle g\rangle^{\omega} \Cap c) \,;\, (\langle g\rangle^{\omega} \Cap x)$$

which holds by a combination of parts (B.141) and (B.142), Lemma 2.13 (fold/unfold-iteration) and Lemma 3.6 (conjunction-atomic). The function H is continuous because strict conjunction is continuous.
□

Lemma 3.29 (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 B.1, recalling that $(\mathbf{guar}\, g \bullet d) \mathrel{\widehat{=}} d \Cap \langle g \vee \mathsf{id}\rangle^{\omega}$. We first show (B.129). Assume $d \xrightarrow{\alpha} d'$. In the case where d' is **abort** then both source and target may abort (Rule (A.110) and Rule (A.115)). In the case where d' is not abort, then we may further assume that a program step α satsifies $g \vee \mathsf{id}$.

Because the context is r, by Definition 2.1 (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 $(\sigma, \sigma') \in r \vee \mathsf{id}$. By Rule (A.115) both subprocesses must have also taken this step, and hence $c_1 \xrightarrow{\epsilon(\sigma,\sigma')} c_1'$. By assumption $(\sigma, \sigma') \in r \vee \mathsf{id}$, which implies $(\sigma, \sigma') \in g \vee r \vee \mathsf{id}$. 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 $g \vee \mathsf{id}$, and hence also satisfies $g \vee r \vee \mathsf{id}$. The corresponding environment step of c_1 therefore also satisfies $g \vee r \vee \mathsf{id}$, and by assumption this is a valid step of c_0 in the context $g \vee r \vee \mathsf{id}$. This completes the proof of (B.129). To prove (B.130) is straightforward as both source and target evolve similarly. □

Lemma 3.39 (introduce-variable) Assuming x does not occur free in c and $x \notin Y$,

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

Proof. From the definition of a local variable block (16), the statement in the law is equivalent to showing that for all $v \in \mathit{Val}$,

$$Y : c \quad \sqsubseteq \quad (\mathbf{state}\, x \mapsto v \bullet x, Y : c)\ .$$

Expanding using the the definition of the frame and hence guarantee gives:

$$c \Cap \langle \mathsf{id}(\overline{Y})\rangle^{\omega} \quad \sqsubseteq \quad (\mathbf{state}\, x \mapsto v \bullet c \Cap \langle \mathsf{id}(\overline{x, Y})\rangle^{\omega})$$

The proof uses Theorem B.1. For property (B.129) for program steps

$$\begin{array}{ll}
& (\mathbf{state}\, x \mapsto v \bullet c \Cap \langle \mathsf{id}(\overline{x, Y})\rangle^{\omega}) \xrightarrow{\pi(\sigma,\sigma')} (\mathbf{state}\, x \mapsto v' \bullet c' \Cap \langle \mathsf{id}(\overline{x, Y})\rangle^{\omega}) \\
\Leftrightarrow & \text{by Rule (A.118)} \\
& (c \Cap \langle \mathsf{id}(\overline{x, Y})\rangle^{\omega} \xrightarrow{\pi(\sigma[x\mapsto v], \sigma'[x\mapsto v'])} c' \Cap \langle \mathsf{id}(\overline{x, Y})\rangle^{\omega}) \wedge \sigma'(x) = \sigma(x) \\
\Rightarrow & \text{by Rule (A.106) the program step must satisfy } \mathsf{id}(\overline{x, Y}) \\
& (c \xrightarrow{\pi(\sigma[x\mapsto v], \sigma'[x\mapsto v'])} c') \wedge (\sigma, \sigma') \in \mathsf{id}(\overline{Y}) \\
\Leftrightarrow & \text{as } x \text{ does not occur free in } c \\
& (c \xrightarrow{\pi(\sigma,\sigma')} c') \wedge (\sigma, \sigma') \in \mathsf{id}(\overline{Y}) \\
\Leftrightarrow & \text{as the step } \pi(\sigma, \sigma') \text{ satisfies } \mathsf{id}(\overline{Y}) \\
& c \Cap \langle \mathsf{id}(\overline{Y})\rangle^{\omega} \xrightarrow{\pi(\sigma,\sigma')} c' \Cap \langle \mathsf{id}(\overline{Y})\rangle^{\omega}
\end{array}$$

and for environment steps

$$\begin{array}{ll}
& (\mathbf{state}\ x \mapsto v \bullet c \Cap \langle \mathsf{id}(\overline{x,Y}\rangle^{\omega}) \xrightarrow{\epsilon(\sigma,\sigma')} (\mathbf{state}\ x \mapsto v \bullet c' \Cap \langle \mathsf{id}(\overline{x,Y}\rangle^{\omega}) \\
\Leftrightarrow & \text{by Rule (A.119)} \\
& c \Cap \langle \mathsf{id}(\overline{x,Y}\rangle^{\omega} \xrightarrow{\epsilon(\sigma[x\mapsto v],\sigma'[x\mapsto v])} c' \Cap \langle \mathsf{id}(\overline{x,Y}\rangle^{\omega} \\
\Leftrightarrow & \text{by Rule (A.106) an atomic step allows any environment step} \\
& c \xrightarrow{\epsilon(\sigma[x\mapsto v],\sigma'[x\mapsto v])} c' \\
\Rightarrow & \text{as } x \text{ does not occur free in } c \\
& c \xrightarrow{\epsilon(\sigma,\sigma')} c' \\
\Leftrightarrow & \text{by Rule (A.106) as an atomic step allows any environment step} \\
& c \Cap \langle \mathsf{id}(\overline{Y}\rangle^{\omega} \xrightarrow{\epsilon(\sigma,\sigma')} c' \Cap \langle \mathsf{id}(\overline{Y}\rangle^{\omega}
\end{array}$$

□

Lemma 4.1 (parallel-interference) For any relations r_0 and r_1, $\langle r_0\rangle^* \parallel \langle r_1\rangle^* = \langle r_0 \vee r_1\rangle^*$.
Proof. Follows straightforwardly from Rule (A.106) and Rule (A.115). □

Lemma 4.3 (interference-atomic) For any predicate p and relations q and r,

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

Proof. Follows straightforwardly from Rule (A.106), Rule (A.115), and Rule (A.109). □

Lemma 9.1 (refine-var) $c \sqsubseteq_{\mathsf{id}(x)} d \quad \Rightarrow \quad (\mathbf{var}\, x \bullet c) \sqsubseteq (\mathbf{var}\, x \bullet d)$.
Proof. Using the definition of a local variable block (16) and Law 2.11 (nondeterministic-choice) part (32), it is sufficient to show

$$c \sqsubseteq_{\mathsf{id}(x)} d \qquad \Rightarrow \qquad \forall v \bullet (\mathbf{state}\ x \mapsto v \bullet c) \sqsubseteq (\mathbf{state}\ x \mapsto v \bullet d) .$$

We assume the property on the left of the implication and show the property on the right holds for all v using Theorem B.1. For property (B.129) for program steps

$$\begin{array}{ll}
& (\mathbf{state}\ x \mapsto v \bullet d) \xrightarrow{\pi(\sigma,\sigma')} (\mathbf{state}\ x \mapsto v' \bullet d') \\
\equiv & \text{by Rule (A.118)} \\
& (d \xrightarrow{\pi(\sigma[x\mapsto v],\sigma'[x\mapsto v'])} d') \wedge \sigma'(x) = \sigma(x) \\
\Rrightarrow & \text{as } c \sqsubseteq_{\mathsf{id}(x)} d \\
& (c \xrightarrow{\pi(\sigma[x\mapsto v],\sigma'[x\mapsto v'])} c') \wedge \sigma'(x) = \sigma(x) \\
\equiv & \text{by Rule (A.118)} \\
& (\mathbf{state}\ x \mapsto v \bullet c) \xrightarrow{\pi(\sigma,\sigma')} (\mathbf{state}\ x \mapsto v \bullet c')
\end{array}$$

and for environment steps

$$\begin{array}{ll}
& (\mathbf{state}\ x \mapsto v \bullet d) \xrightarrow{\epsilon(\sigma,\sigma')} (\mathbf{state}\ x \mapsto v \bullet d') \\
\equiv & \text{by Rule (A.119)} \\
& d \xrightarrow{\epsilon(\sigma[x\mapsto v],\sigma'[x\mapsto v])} d' \\
\Rrightarrow & \text{as } c \sqsubseteq_{\mathsf{id}(x)} d \text{ and } (\sigma[x \mapsto v], \sigma'[x \mapsto v]) \in \mathsf{id}(x) \\
& c \xrightarrow{\epsilon(\sigma[x\mapsto v],\sigma'[x\mapsto v])} c' \\
\equiv & \text{by Rule (A.119)} \\
& (\mathbf{state}\ x \mapsto v \bullet c) \xrightarrow{\epsilon(\sigma,\sigma')} (\mathbf{state}\ x \mapsto v \bullet c')
\end{array}$$

Property (B.130) is straightforward as both source and target evolve similarly. □

Index

References

[Acz83] P. H. G. Aczel. On an inference rule for parallel composition, 1983. Private communication to Cliff Jones.

[Bac81] R.-J. R. Back. On correct refinement of programs. *Journal of Computer and System Sciences*, 23(1):49–68, February 1981.

[Bro07] S. Brookes. A semantics for concurrent separation logic. *Theoretical Computer Science*, 375(13):227–270, 2007.

[BvW98] R.-J. R. Back and J. von Wright. *Refinement Calculus: A Systematic Introduction*. Springer, New York, 1998.

[CH81] Zhou Chaochen and C. A. R. Hoare. Partial correctness of communication protocols. In *Technical Monograph PRG-20, Partial Correctness of Communicating Processes and Protocols*, pages 13–23. Oxford University Computing Laboratory, May 1981.

[Cha82] Zhou Chaochen. Weakest environment of communicating processes. In *Proc. of the June 7-10, 1982, National Computer Conf.*, AFIPS '82, pages 679–690, New York, NY, USA, 1982. ACM.

[CJ00] Pierre Collette and Cliff B. Jones. Enhancing the tractability of rely/guarantee specifications in the development of interfering operations. In Gordon Plotkin, Colin Stirling, and Mads Tofte, editors, *Proof, Language and Interaction*, chapter 10, pages 277–307. MIT Press, 2000.

[CJ07] J. W. Coleman and C. B. Jones. A structural proof of the soundness of rely/guarantee rules. *Journal of Logic and Computation*, 17(4):807–841, 2007.

[Col08] Joseph William Coleman. *Constructing a Tractable Reasoning Framework upon a Fine-Grained Structural Operational Semantics*. PhD thesis, Newcastle University, January 2008.

[dBHdR99] F. de Boer, U. Hannemann, and W. de Roever. Formal justification of the rely-guarantee paradigm for shared-variable concurrency: a semantic approach. In Jeannette Wing, Jim Woodcock, and Jim Davies, editors, *FM99 Formal Methods*, volume 1709 of *Lecture Notes in Computer Science*, pages 714–714. Springer Berlin / Heidelberg, 1999.

[DH10] Brijesh Dongol and Ian J. Hayes. Compositional action system derivation using enforced properties. In C. Bolduc, J. Desharnais, and B. Ktari, editors, *Mathematics of Program Construction (MPC)*, volume 6120 of *LNCS*, pages 119–139. Springer Verlag, 2010.

[Din00] Jürgen Dingel. *Systematic Parallel Programming*. PhD thesis, Carnegie Mellon University, 2000. CMU-CS-99-172.

[Din02] J. Dingel. A refinement calculus for shared-variable parallel and distributed programming. *Formal Aspects of Computing*, 14(2):123–197, 2002.

[dR01] W.-P. de Roever. *Concurrency Verification: Introduction to Compositional and Noncompositional Methods*. Cambridge University Press, 2001.

[DSW11] John Derrick, Gerhard Schellhorn, and Heike Wehrheim. Mechanically verified proof obligations for linearizability. *ACM Trans. Program. Lang. Syst.*, 33(1):4, 2011.

[FFS07] Xinyu Feng, Rodrigo Ferreira, and Zhong Shao. On the relationship between concurrent separation logic and assume-guarantee reasoning. In *ESOP: Programming Languages and Systems*, pages 173–188. Springer, 2007.

[Flo67] R. W. Floyd. Assigning meaning to programs. *Math. Aspects of Comput. Sci.*, 19:19–32, 1967.

[GC09] Lindsay Groves and Robert Colvin. Trace-based derivation of a scalable lock-free stack algorithm. *Formal Aspects of Computing*, 21(1–2):187–223, 2009.

[HBDJ13] I. J. Hayes, A. Burns, B. Dongol, and C. B. Jones. Comparing degrees of non-deterministic in expression evaluation. *The Computer Journal*, 56(6):741–755, 2013.

[HH86] C.A.R. Hoare and Jifeng He. The weakest prespecification. *Fundamenta Informaticae*, IX:51–84, 1986.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 583, October 1969.

[HW90] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463 – 492, 1990.

[IO01] S. S. Ishtiaq and P. W. O'Hearn. BI as an assertion language for mutable data structures. In *Proc. of the ACM SIGPLAN-SIGACT Symp. on Princ. of Prog. Languages (POPL)*, pages 14–26. ACM Press, 2001.

[JHC14] Cliff B. Jones, Ian J. Hayes, and Robert J. Colvin. Balancing expressiveness in formal approaches to concurrency. *Formal Aspects of Computing*, (in press), 2014.

[Jon81] C. B. Jones. *Development Methods for Computer Programs including a Notion of Interference*. PhD thesis, Oxford University, June 1981. Printed as: Programming Research Group, Technical Monograph 25.

[Jon83] C. B. Jones. Tentative steps toward a development method for interfering programs. *Transactions on Programming Languages and System*, 5(4):596–619, 1983.

[Jon87] C. B. Jones. Program specification and verification in VDM. In M. Broy, editor, *Logic of Programming and Calculi of Discrete Design*, volume 36 of *NATO ASI Series F: Computer and Systems Sciences*, pages 149–184. Springer-Verlag, 1987.

[Jon96] C. B. Jones. Accommodating interference in the formal design of concurrent object-based programs. *Formal Methods in System Design*, 8(2):105–122, March 1996.

[Jon03a] Cliff B. Jones. The early search for tractable ways of reasonning about programs. *IEEE Annals of the History of Computing*, 25(2):26–49, 2003.

[Jon03b] Cliff B. Jones. Operational semantics: Concepts and their expression. *Inf. Process. Lett.*, 88(1-2):27–32, 2003.

[Jon07] C. B. Jones. Splitting atoms safely. *Theoretical Computer Science*, 375(1–3):109–119, 2007.

[Jon12] Cliff B. Jones. Abstraction as a unifying link for formal approaches to concurrency. In G. Eleftherakis, M. Hinchey, and M. Holcombe, editors, *Software Engineering and Formal Methods*, volume 7504 of *Lecture Notes in Computer Science*, pages 1–15, October 2012.

[JP08] Cliff Jones and Ken Pierce. Splitting atoms with rely/guarantee conditions coupled with data reification. In Egon Börger, Michael Butler, Jonathan Bowen, and Paul Boca, editors, *Abstract State Machines, B and Z*, volume 5238 of *Lecture Notes in Computer Science*, pages 360–377. Springer Berlin / Heidelberg, 2008.

[JP11] Cliff B. Jones and Ken G. Pierce. Elucidating concurrent algorithms via layers of abstraction and reification. *Formal Aspects of Computing*, 23(3):289–306, 2011.

[Mil89] R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.

[Mor87] J. M. Morris. A theoretical basis for stepwise refinement and the programming calculus. *Science of Computer Programming*, 9(3):287–306, 1987.

[Mor88] C. C. Morgan. The specification statement. *ACM Trans. Prog. Lang. and Sys.*, 10(3):403–419, July 1988.

[Mor94] C. C. Morgan. *Programming from Specifications*. Prentice Hall, second edition, 1994.

[Mos04a] Peter D. Mosses. Exploiting labels in structural operational semantics. *Fundam. Inform.*, 60(1-4):17–31, 2004.

[Mos04b] Peter D. Mosses. Modular structural operational semantics. *J. Log. Algebr. Program.*, 60-61:195–228, 2004.

[MV90] C. C. Morgan and T. N. Vickers. Types and invariants in the refinement calculus. *Science of Computer Programming*, 14:281–304, 1990.

[MV94a] C. C. Morgan and T. N. Vickers, editors. *On the Refinement Calculus*. Springer-Verlag, 1994.

[MV94b] C. C. Morgan and T. N. Vickers. Types and invariants in the refinement calculus. In C. C. Morgan and T. N. Vickers [MV94a], pages 127–154. Originally published as [MV90].

[NH97] R. Nickson and I. J. Hayes. Supporting contexts in program refinement. *Science of Computer Programming*, 29(3):279–302, 1997.

[Owi75] S. Owicki. *Axiomatic Proof Techniques for Parallel Programs*. PhD thesis, Department of Computer Science, Cornell University, 1975.

[OYR09] P. W. O'Hearn, H. Yang, and J. C. Reynolds. Separation and information hiding. *ACM TOPLAS*, 31(3):11:1–11:50, April 2009. Preliminary version appeared in 31st POPL, pp268-280, 2004.

[Plo04] Gordon D. Plotkin. A structural approach to operational semantics. *Journal of Logic and Algebraic Programming*, 60–61:17–139, July–December 2004.

[Pre01] Leonor Prensa Nieto. *Verification of Parallel Programs with the Owicki-Gries and Rely-Guarantee Methods in Isabelle/HOL*. PhD thesis, Institut für Informatic der Technischen Universitaet München, 2001.

[Pre03] Leonor Prensa Nieto. The rely-guarantee method in Isabelle/HOL. In *Proceedings of ESOP 2003*, volume 2618 of *LNCS*. Springer-Verlag, 2003.

[Rey02] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *IEEE Symposium on Logic in Computer Science (LICS)*, pages 55–74. IEEE Computer Society, 2002.

[Sti86] C. Stirling. A compositional reformulation of Owicki-Gries' partial correctness logic for a concurrent while language. In *ICALP'86*, volume 226 of *LNCS*, pages 407–415. Springer-Verlag, 1986.

[Stø90] K. Stølen. *Development of Parallel Programs on Shared Data-Structures*. PhD thesis, Manchester University, 1990. Available as UMCS-91-1-1.

[TW11] Aaron Joseph Turon and Mitchell Wand. A separation logic for refining concurrent objects. In Thomas Ball and Mooly Sagiv, editors, *POPL*, pages 247–258. ACM, 2011.

[VP07] V. Vafeiadis and M. Parkinson. A marriage of rely/guarantee and separation logic. In L. Caires and V. Vasconcelos, editors, *CONCUR 2007*, volume 4703 of *LNCS*, pages 256–271. Springer, 2007.

[vW04] J. von Wright. Towards a refinement algebra. *Sci. of Comp. Prog.*, 51:23–45, 2004.

[WDP10] John Wickerson, Mike Dodds, and Matthew Parkinson. Explicit stabilisation for modular rely-guarantee reasoning. Technical Report 774, Computer Laboratory, University of Cambridge, March 2010.