A structural proof of the soundness of rely/guarantee rules

Joey W. Coleman
Cliff B. Jones

School of Computing Science
Newcastle University
NE1 7RU, UK
e-mail: {j.w.coleman, cliff.jones}@ncl.ac.uk

Abstract. The challenge of finding compositional ways of (formally) developing concurrent programs is considerable. Various forms of rely/guarantee conditions have been used to record and reason about interference in ways which do indeed provide compositional development methods for such programs. Our underlying concurrent language allows fine-grained interleaving and nested concurrency; it is defined by an operational semantics; the proof that the rely/guarantee rules are consistent with that semantics (including termination) is by a structural induction. This paper presents a new approach to justifying the soundness of rely/guarantee inference rules. A key lemma which relates the states which can arise from the extra interference that results from taking a portion of the program out of context makes it possible to do the proofs without having to perform induction over the computation history. This lemma also offers a way to think about some elusive expressibility issues around auxiliary variables in rely/guarantee conditions.

1 Introduction

Floyd/Hoare rules provide a way of reasoning about non-interfering programs: for such *sequential* programs, inference rules are now well known; their soundness can be proved and one can even obtain a (relatively) complete "axiomatic semantics" for simple languages [Apt81]. Moreover, because the rules are "compositional" (see [Jon03a]), they can be used in a design process rather than just in *post-facto* proofs.

Even for sequential programs, the rules used in the VDM literature differ in two important respects from, say, those used in [Hoa69,Dij76,GS96]: VDM authors have always insisted on using post conditions which are predicates of two states and on recognising the problems which result from undefined expressions. Section 3.2 expands on both of these points because they permeate the subsequent research on concurrency.

Finding compositional proof rules for concurrent programs has proved to be challenging precisely because of the "interference" which is the essence of concurrency. The "Owicki/Gries approach" [Owi75,OG76] is not compositional because a multi-stage development might have to be repeated if Owicki's final *Einmischungsfrei* proof obligation cannot be discharged.

Rely and guarantee conditions (e.g. [Jon81,Jon83b,Jon83a,Jon96]) offer a way of handling interference during the development process. Crucially this way of documenting and reasoning about interference does provide a compositional development method for concurrent programs. John Reynolds characterised rely/guarantee conditions as providing a way of reasoning about "racy" programs (whereas "separation logic" [O'H07] lets one show that race conditions are avoided).

There has been a lot written about various forms of rely/guarantee conditions.[1] We have recently revisited a number of issues as part of a project on "Splitting (software) atoms safely" [Jon07]; one of us was also motivated by our study of new forms of "interference reasoning" in connection with "deriving specifications" [HJJ03,JHJ07]. Having decided to undertake the task of providing a reference point on rely/guarantee conditions, we believe that we have come up with a novel approach to the soundness proof.

The current paper provides an underlying operational semantics for a concurrent language; one particular set of rely/guarantee rules for that language; and a justification of those inference rules with respect to the operational semantics. The approach here follows that of Tom Melham [CM92] and Tobias Nipkow [KNvO$^+$02] which were presumably influenced by [BH88]: the rules of an operational semantics can be taken to provide an inductive definition of a relation ($\xrightarrow{s}$) over "configurations" (i.e. pairs of program texts and states). Results about specific programs could be proved *directly* in terms of this inference system.

[1] An annotated list of publications on rely/guarantee concepts can be found at http://homepages.cs.ncl.ac.uk/cliff.jones/home.formal
A key (encyclopaedic) reference is [dR01].

We view the "Floyd/Hoare-like" rules for reasoning directly about rely/guarantee conditions as extra inference rules which have to be shown to be consistent with the operational semantics (and thus abbreviate longer proofs which might have been written directly in terms of that semantics).

Thus this paper presents one version of a collection of rely/guarantee rules for reasoning about interference (they are collected in Appendix A); a semantic model of a small, fine-grained concurrent, shared-variable language — discussed in Section 2); and shows a novel justification of the formal rules with respect to the language semantics (discussed in Section 4). To aid the reader's understanding, Section 3 introduces the rely/guarantee rules by means of an example of a small concurrent program whose design is justified in terms of the aforementioned rules.

1.1 Introducing rely/guarantee conditions

This section offers a brief introduction to rely/guarantee concepts for the benefit of those unfamiliar with them. Program development using Floyd/Hoare-like pre and post conditions can be visualised as shown in Figure 1a. The horizontal line represents the system states over time; P and Q are –respectively– pre and post condition predicates of a state; they are positioned to show where they are expected to hold and the execution of the program is indicated by the box along the top of the diagram. This model is adequate for isolated, sequential systems, but it assumes atomicity with respect to the program's environment, making it unsuitable for concurrent programs.

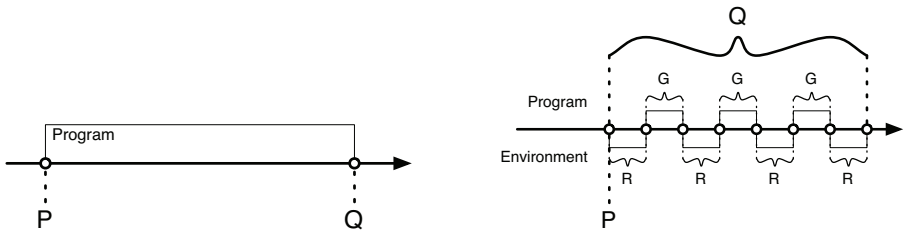

Fig. 1. (a) Pre/Post conditions and (b) Rely/Guarantee conditions

Rely/guarantee conditions can be visualised as in Figure 1b. As with Figure 1a, the horizontal line represents the system state over time and P represents the program's pre condition. Unlike Figure 1a, however, Q is a relation — it is required to hold over the initial and final states. The execution of the program is displayed as boxes above the state line and actions taken by the environment are represented below it. Every change to the state made by the program must conform to the restrictions given by the guarantee condition, G. The program specification indicates that all actions taken by the environment can be assumed to conform to the rely condition, R.

It is important to note the asymmetry in the use of R and G: whereas the latter is a restriction on the program to be created, the former does not constrain anything. In fact, a rely condition is an invitation to the *developer* to make assumptions about the environment in which the software that is to be created will be deployed.[2] It is the responsibility of the user of the program to ensure that it is run in an environment that conforms to R. (Of course, one way to do this is to construct a program which can be proven to provide such a context.) However, for the purposes of reasoning about interference in proofs, both R and G are required in proofs and, when dealing with concurrency, we find that guarantee of one thread becomes (part of) the rely condition of another thread.

Typical rely conditions might record that a flag is only set (say **false** to **true**) by one process or that a variable is unchanged when a certain flag is set. Another class of rely conditions that are useful in practice is that the value of a variable changes monotonically. Without locks, this latter class of conditions presents intriguing implementation problems and the example in Section 3 shows how careful choice of data representations can play a key part in their solution.

With Figure 1b in mind, then, the thrust of a rely/guarantee development lies in formalising the behaviour of both the program and its intended environment. Once the assumptions about the environment have been characterised in the rely condition, that condition can be used both in the proofs regarding the program and also by a

[2] Much the same can actually be said about pre conditions: there is a sense in which the original Hoare triple notation $\{P\}S\{Q\}$ rather hides the distinction between design-time assumptions and obligations on the created code.

potential user of the program to determine its suitability to the actual environment at hand. The guarantee condition serves not only to indicate the potential behaviour of the program, but it also becomes critical when reasoning about different branches of a program or about the behavioural interaction of the two separately developed parallel programs.

It is worth noting that there are decisions to be made in Floyd/Hoare-like rules for sequential programs (e.g. whether to have separate "weakening" rules or to incorporate the relaxation of pre/post conditions into the other rules); such decisions are more numerous when there are four clauses to a specification. There are in fact several forms of rely/guarantee conditions even where the idea is applied to shared-variable concurrency. There are further decisions to be made about whether the predicates written in $If/While$ statements are stable under interference — this point is discussed below. Further discussion of the trade-offs in designing rely/guarantee rules can be found in [CJ00] which includes the useful idea of a "dynamic invariant" that is not discussed further in the current paper.

As in all research on "formal methods", the expectation is that such work will inspire guidelines even for less formal approaches. This expectation appears to be fulfilled for rely/guarantee conditions which have proved to be a useful way of thinking about a whole range of issues.

2 The language

In order to keep the proofs to a reasonable length, the language has only five statement constructs and **nil** (to represent a completed statement), as well as a subsidiary expression construct. Its formal description follows the "VDM tradition" of basing the semantics on an *abstract* syntax and restricting the class of programs by using "context conditions". The abstract syntax is in Figure 2. (An example program is given in Figure 4 but the code is obtained by steps of design which employ the rely/guarantee rules.) We omit here the context conditions which are routine and make the obvious type checks (they are contained in [CJ07]).

$$Stmt = Par \mid While \mid If \mid Seq \mid Assign \mid \textbf{nil}$$

$$\begin{array}{rcl} Par & :: & sl : Stmt \\ & & sr : Stmt \end{array}$$

$$\begin{array}{rcl} While & :: & b : Expr \\ & & body : Stmt \end{array}$$

$$\begin{array}{rcl} If & :: & b : Expr \\ & & body : Stmt \end{array}$$

$$\begin{array}{rcl} Seq & :: & sl : Stmt \\ & & sr : Stmt \end{array}$$

$$\begin{array}{rcl} Assign & :: & id : Id \\ & & e : Expr \end{array}$$

$$Expr = \mathbb{B} \mid \mathbb{Z} \mid Id \mid Dyad$$

$$\begin{array}{rcl} Dyad & :: & op : + \mid - \mid < \mid = \mid > \mid \wedge \mid \vee \\ & & a : Expr \\ & & b : Expr \end{array}$$

Fig. 2. Abstract syntax of the language to be defined

In VDM [Jon90] a record such as $While$ is constructed by a make function

$$mk\text{-}While\colon Expr \times Stmt \to While$$

Elements of $While$ are disjoint from any other record type so that unions like $Stmt$ can be formed safely. The use of the constructors in pattern matching positions such as the SOS rules was common in VDM from the 1970's but should be familiar from functional languages.

In order to show that the inference rules used for (concurrent) program constructs are sound, an independent semantics is needed. The semantics used here is a structural operational semantics [Plo81].[3] [4] We view the rules of the semantics as (inductively) defining a relation over configurations of program texts and states.

The language given contains no means to create fresh variables nor to restrict access to any variable. A program in this language has all of its variables contained within a single global scope: the state object, σ. All of the variables that the program requires must be present and initialized at the start of execution. The state object in the language maps all variables to integer values.

[3] Republished as [Plo04b] — see also [Plo04a,Jon03b].

[4] We do not enter here into a debate about the merits of operational versus denotational semantics (but see [Plo04a,Jon03b]); we do, of course, avoid the Baroque excesses caused by using what McCarthy called a "Grand State".

$\Sigma = Id \xrightarrow{m} Value$

In our SOS below, the main semantic relation of the language is:
$\xrightarrow{s} : \mathcal{P}\left((Stmt \times \Sigma) \times (Stmt \times \Sigma)\right)$
This symmetry between the type of the domain and range of the relation allows us to form transitive closures giving us a convenient mechanism to talk about two configurations related by many steps of the semantics (the transitive closure is $\xrightarrow{s}*$).

The *raison d'être* of the language is its Par construct which interleaves parallel execution of two statements. The parallel execution of more than two statements can be achieved by nesting Par constructs.

$$\boxed{\text{Par-L}}\ \frac{(sl, \sigma) \xrightarrow{s} (sl', \sigma')}{(mk\text{-}Par(sl, sr), \sigma) \xrightarrow{s} (mk\text{-}Par(sl', sr), \sigma')}$$

$$\boxed{\text{Par-R}}\ \frac{(sr, \sigma) \xrightarrow{s} (sr', \sigma')}{(mk\text{-}Par(sl, sr), \sigma) \xrightarrow{s} (mk\text{-}Par(sl, sr'), \sigma')}$$

$$\boxed{\text{Par-E}}\ \frac{}{(mk\text{-}Par(\textbf{nil}, \textbf{nil}), \sigma) \xrightarrow{s} (\textbf{nil}, \sigma')}$$

The SOS rules have no inherent notion of fairness: the choice of which branch to follow is unspecified.

Conditional execution is provided by the If construct and is a pure conditional rather than a choice between two statements (which is not required by the example in Section 3) so the "else" branch has been omitted from the conditional.

$$\boxed{\text{If-Eval}}\ \frac{(b, \sigma) \xrightarrow{e} b'}{(mk\text{-}If(b, body), \sigma) \xrightarrow{s} (mk\text{-}If(b', body), \sigma)}$$

$$\boxed{\text{If-T-E}}\ \frac{}{(mk\text{-}If(\textbf{true}, body), \sigma) \xrightarrow{s} (body, \sigma)}$$

$$\boxed{\text{If-F-E}}\ \frac{}{(mk\text{-}If(\textbf{false}, body), \sigma) \xrightarrow{s} (\textbf{nil}, \sigma)}$$

Repetition is achieved with the $While$ construct — our description gives the behaviour for this construct indirectly: the SOS rule that specifically deals with $While$ rewrites the program text in terms of an If that contains a sequence with the loop body and the original $While$.

$$\boxed{\text{While}}\ \frac{}{(mk\text{-}While(b, body), \sigma) \xrightarrow{s} (mk\text{-}If(b, mk\text{-}Seq(body, mk\text{-}While(b, body))), \sigma)}$$

The Seq construct provides sequential execution.

$$\boxed{\text{Seq-Step}}\ \frac{(sl, \sigma) \xrightarrow{s} (sl', \sigma')}{(mk\text{-}Seq(sl, sr), \sigma) \xrightarrow{s} (mk\text{-}Seq(sl', sr), \sigma')}$$

$$\boxed{\text{Seq-E}}\ \frac{}{(mk\text{-}Seq(\textbf{nil}, sr), \sigma) \xrightarrow{s} (sr, \sigma)}$$

Assignment (to a scalar variable) is represented by the $Assign$ construct and is the only means to alter the state. The only step of the Assignment which is atomic is the actual mutation of the state object.

$$\boxed{\text{Assign-Eval}}\ \frac{(e, \sigma) \xrightarrow{e} e'}{(mk\text{-}Assign(id, e), \sigma) \xrightarrow{s} (mk\text{-}Assign(id, e'), \sigma)}$$

$$\boxed{\text{Assign-E}}\ \frac{n \in \mathbb{Z}}{(mk\text{-}Assign(id, n), \sigma) \xrightarrow{s} (\textbf{nil}, \sigma \dagger \{id \mapsto n\})}$$

Expression evaluation is *non-atomic*; interference makes it possible that $(x + x) \neq 2x$ (or indeed $x + x$ can be an odd number). This decision allows parallel statements to interfere with each other at a very fine level of granularity. Not surprisingly, such fine grained interference complicates reasoning but it is a realistic decision that permits efficient implementation: it would be ridiculous to introduce locks which forced statement level atomicity and extravagant to require a compiler to detect where they were (not) required.

The **nil** statement acts as a *skip* or empty statement, indicating that there is no computation to perform. At first glance it may seem that a simple self-assignment such as $x \leftarrow x$ might be equivalent to a statement which does nothing, but that overlooks the nature of the interference which this language allows. The addition of **nil** to the $Stmt$ type has the useful side effect of simplifying the SOS rules for nearly all of the constructs which can contain a statement: without the **nil** statement we would be required to distinguish those transitions that terminate the contained statement. However, it should be noted programs containing many **nil** statements can be normalised to a form without most of those **nil** statements and still have the same behaviour.[5] Finally, a completed program is always a configuration of the form $(\mathbf{nil}, \sigma)$, which contains a valid program in its own right.

It is important to understand how the fine-grained interleaving of steps is achieved in the SOS. Essentially, the whole of the unexecuted program is available (as an abstract syntax tree). To perform one (small) step of the $\xrightarrow{s}$ transition can require making non-deterministic choices all the way down to a leaf statement (even to a leaf operand of an expression). Each step of the transition relation results in a new configuration containing the remainder of the program to be executed; the next step is then chosen in this new configuration. This permits a thread to be interrupted by other threads which might be considered to be its uncles or nephews in the program text. This complicates the proof of soundness of the rules of Appendix A but it is a useful freedom for writing realistic concurrent programs.

The subsidiary type $Expr$ is used by the rules for the $Assign$, If and $While$ constructs. It has its own semantic relation that models the process of expression evaluation.

$\xrightarrow{e} : \mathcal{P}\left((Expr \times \Sigma) \times Expr\right)$

$$\boxed{\text{Id-E}}\ \frac{}{(id, \sigma) \xrightarrow{e} \sigma(id)}$$

$$\boxed{\text{Dyad-L}}\ \frac{(a, \sigma) \xrightarrow{e} a'}{(mk\text{-}Dyad(op, a, b), \sigma) \xrightarrow{e} mk\text{-}Dyad(op, a', b)}$$

$$\boxed{\text{Dyad-R}}\ \frac{(b, \sigma) \xrightarrow{e} b'}{(mk\text{-}Dyad(op, a, b), \sigma) \xrightarrow{e} mk\text{-}Dyad(op, a, b')}$$

Predicates like P and Q are written as assertions about states (or pairs thereof); they are obviously pieces of text and in the proofs that follow we need to apply the corresponding semantic object to states. Thus

$[\![P]\!] \triangleq \lambda\sigma.P$
$[\![Q]\!] \triangleq \lambda\overleftarrow{\sigma}, \sigma.Q$

$$\boxed{\text{Dyad-E}}\ \frac{a \in \mathbb{Z} \wedge b \in \mathbb{Z}}{(mk\text{-}Dyad(op, a, b), \sigma) \xrightarrow{e} [\![op]\!](a, b)}$$

Unlike $\xrightarrow{s}$, this semantic relation is not symmetric, as the type of the range is an $Expr$, as compared to the type of the domain which is a pair of an $Expr$ and a state object. Expressions in our language cannot cause side effects, that is, they are unable to mutate state; this allowed the simplification of the $\xrightarrow{e}$ relation and simplifies the proofs of the development rules needed. The language has no notion of function or procedure calls and this lack is part of what keeps expressions side effect free.

3 A set of rely/guarantee rules

As indicated in the introductory section, there is no single set of rely/guarantee rules which fit all situations (cf. Section 5); a useful set is given in Appendix A. We illustrate these rules with an example.

3.1 An example development

We use as an example here a problem –FINDP– which originated in [Owi75] and was tackled by rely/guarantee reasoning in [Jon81]. The FINDP example is presented as it is a minimal example in terms of which we can demonstrate the ideas needed in this paper.

The task is to find the least index i (to a vector v) such that for some predicate $pred$, $pred(v(i))$ holds. In order not to have to describe function calls in the semantics in Section 2, we note that the predicate could be as simple

[5] Because of termination issues, a $While$ with just **nil** as the body cannot, in general be eliminated.

as $\lambda x \cdot x > 0$. In order to justify parallelism, we wish to view it as expensive to implement. It is straightforward to add semantics for function calls as we would not allow any side-effects from their evaluation.

A specification is presented in Figure 3 using normal VDM pre/post conditions plus rely/guarantee conditions. All except the pre-condition are predicates over pairs of states; the distinction between components of the first and second states is made by "hooking" the former. Thus $v = \overleftarrow{v}$ in the rely condition indicates that the value of v is unchanged by interference.

At this level of abstraction, the required program has access to two variables: v and r. The former will only be read by $FINDP$ whereas the program can both read and write the latter. The pre condition allows for any starting state providing that $pred(v(i))$ is defined for all indices (this includes the requirement that the indices are within the vector; $\delta(e)$ requires that e is defined — see discussion of δ in Section 3.2). The guarantee condition places no constraint on the internal behaviour of the program.

The rely condition requires that the environment does not change v or r during execution of $FINDP$ (without this assumption, it would be impossible to devise an implementation). Finally, the post condition asserts that, if r is a valid index into v, then $pred$ holds on $v(i)$; alternatively, if there are no values in v for which $pred$ holds, then r will be precisely one greater than the length of v.[6] The final conjunct requires that the least such r is found. Notice the type of r: $\mathbb{N}_1$: this is an implied restriction that its minimum value is 1; the step to some form of dependent type –restricting the highest value (with respect to v)– has not been taken here.

$FINDP$
rd v: X^*
wr r: $\mathbb{N}_1$
pre $\forall i \in \{1..\textbf{len}\ v\} \cdot \delta(pred(v(i)))$
rely $v = \overleftarrow{v} \wedge r = \overleftarrow{r}$
guar true
post $(r = \textbf{len}\ v + 1 \vee 1 \leq r \leq \textbf{len}\ v \wedge pred(v(r))) \wedge$
$\quad \forall i \in \{1..r-1\} \cdot \neg\, pred(v(i))$

Fig. 3. Specification of $FINDP$

3.2 Sequential aside

Apart from the rely/guarantee extension, there are aspects of VDM's design rules –even as applied to sequential programs– that do not fit with "mainstream" verification work (although in some cases, others have moved toward the VDM position). Of particular relevance here is that VDM uses post conditions of two states (relations) which means that standard Floyd/Hoare rules for reasoning about programming constructs are not applicable. This decision actually goes back to work that predates the christening of VDM (e.g. [Jon72]) and was made widely visible in [Jon80]. The presentation of the proof rules used here is as in the first (1986) edition of [Jon90]. These are essentially the same as the rules proposed by Peter Aczel in [Acz82] (which contains the generous characterisation of the first attempt to give rules in [Jon80] as "unmemorable").

The other unconventional feature of VDM is that it takes seriously the problem of expressions which might be "undefined". Both of these points can be illustrated in a short development of a sequential implementation of the specification in Figure 3. Here, S **sim-sat** (P, Q) is written instead of the "Hoare triple" $\{P\}S\{Q\}$.

The rule used for **while** in VDM ensures termination by requiring that the relation over the body of the loop is well-founded; this fits with the relational view of post conditions and is in many ways more natural than Dijkstra's "variant function" [DS90].[7] (We mark the rules for sequential constructs with a prefix **sim-** — the parallel rules below have no prefix.) This might suggest a rule like:

$$\frac{S \textbf{ sim-sat } (P \wedge b, P \wedge W)}{mk\text{-}While(b, S) \textbf{ sim-sat } (P, P \wedge \neg\, b \wedge W^*)}$$

[6] As an alternative one could insert a value at the end of v for which $pred$ is definitely true; the changes to what follows are inconsequential.

[7] At the Schloß Dagstuhl Seminar in July 2006, Josh Berdine of Microsoft observed that their experience with the "Terminator" tool (which attempts automatic verification of termination) supported the use of well-founded relations rather than "variant functions".

Where W^* is the reflexive closure of W (which is already transitive.)

The issue of undefinedness can be seen if one considers the following putative implementation

$r \leftarrow 1;$
while $r \leq$ **len** $v \wedge \neg\, pred(v(r))$ **do** $r \leftarrow r + 1$ **od**

Remembering that the pre condition in Figure 3 only guarantees the $v(i)$ (and thus $pred(v(i))$) is defined for $i \in \{1..$**len** $v\}$, the definedness of the test in this while turns on how expressions are evaluated. If the evaluation of the conjunction short circuits the second conjunct when the first is **false**, all is well; but, if both conjuncts are evaluated, then the second can be undefined when $v = r+1$. Proofs in VDM employ a "logic of partial functions" (LPF) [BCJ84] which ensures that logical expressions are defined whenever possible: the semantics of the logical operators are the weakest extensions over the ordering $\perp_{\mathbb{B}} \leq$ **true**, $\perp_{\mathbb{B}} \leq$ **false** compatible with standard first-order predicate calculus. The "law of the excluded middle" does not in general hold in LPF which means that there are special natural deduction rules for negated disjunction introduction etc.

The use of LPF in proofs does *not* presuppose that a programming language will implement such generous operators. The proof rule for while therefore contains a hypothesis $P \;\Rightarrow\; \delta_l(b)$ which requires that b is defined in the implementation language.[8] We have chosen not to add arrays to the semantics because in this restrictive case they behave like simple (unchanging) sequences whose semantics is obvious. (The full semantics is given in [CJ07].)

The rule for sequential while statements is:

$$\boxed{\textbf{sim-While-I}}\ \frac{\begin{array}{l} S\ \textbf{sim-sat}\ (P \wedge b, P \wedge W) \\ P \;\Rightarrow\; \delta_l(b) \end{array}}{mk\text{-}While(b, S)\ \textbf{sim-sat}\ (P, P \wedge \neg\, b \wedge W^*)}$$

Which would lead us to a sequential implementation like:

$r \leftarrow 1;$
while $r \leq$ **len** v **do**
  if $\neg\, pred(v(r))$ **then** $r \leftarrow r + 1$
od

Notice that this design is cautious about undefinedness by nesting the statements in a way that avoids evaluating $pred(v(r))$ outside the domain of v.

The justification of this design step would use **sim-While-I** with $W(\overleftarrow{\sigma}, \sigma)$ that shows σ is closer to termination than $\overleftarrow{\sigma}$:

$\overleftarrow{r} < r \leq$ **len** v

and P:

$r \in \{1..$**len** $v + 1\} \wedge \forall i \in \{1..r-1\} \cdot \neg\, pred(v(i))$

Notice the W is well-founded over P because of the upper limit on r.

Further sequential rules are:

$$\boxed{\textbf{sim-If-I}}\ \frac{\begin{array}{l} S_t\ \textbf{sim-sat}\ (P \wedge b, Q) \\ S_e\ \textbf{sim-sat}\ (P \wedge \neg\, b, Q) \\ P \;\Rightarrow\; \delta_l(b) \end{array}}{mk\text{-}If(b, S_t, S_e)\ \textbf{sim-sat}\ (P, Q)}$$

In fact, we use a simple conditional with no else clause throughout this paper; the obvious simplification for the identity of the false branch is:

$$\boxed{\textbf{sim-If-I}}\ \frac{\begin{array}{l} body\ \textbf{sim-sat}\ (P \wedge b, Q) \\ \overleftarrow{P} \wedge \overleftarrow{\neg\, b} \;\Rightarrow\; Q \\ P \;\Rightarrow\; \delta_l(b) \end{array}}{mk\text{-}If(b, body)\ \textbf{sim-sat}\ (P, Q)}$$

[8] In the language definition in Section 2, there are no operators (e.g. division) which would result in undefined expressions. Only the possibility of indexing outside the array exists in this example to illustrate the need for δ.

One more piece of notation is needed to define the rule for reasoning about the use of sequential statement composition: $R_1 \mid R_2$ is the predicate which expresses the composition of the two relations. Thus:

$$\textbf{sim-Seq-I}\ \frac{\begin{array}{l} l\ \textbf{sim-sat}\ (P, Q_l \wedge P_r) \\ r\ \textbf{sim-sat}\ (P_r, Q_r) \\ Q_l \mid Q_r\ \Rightarrow\ Q \end{array}}{mk\text{-}Seq(l, r)\ \textbf{sim-sat}\ (P, Q)}$$

An important advantage of "hooking the pre state" (rather than "priming the post state") is visible here. Writing $Q_l \wedge P_r)$ as the post condition of l has exactly the right effect: P_r defines the interface between l and r.

Our interest is *development*; we have argued in several publications that the steps of design provide the outline proof of correctness. The formal rules offer the ability to generate verification conditions and to use a theorem prover if required. Proof rules for assignment statements are therefore of less interest than those for the combinators which let the designer decompose a large task into sub-programs. In fact, in the absence of complicated concepts like location sharing or interference, assignments are unlikely to be wrong. Be that as it may, a rule can be given:

$$\textbf{sim-Assn-I}\ \frac{\textbf{true}}{mk\text{-}Assign(v, e)\ \textbf{sim-sat}\ (\delta(e), v = \overleftarrow{e} \wedge I_{comp(v)})}$$

where $I_{comp(v)}$ indicates the identity relation on all variables except v. In general we denote the identity relation as I. If there is a subscript, i.e. I_{vars}, then the subscript indicates the set of variables to which the identity relation is restricted; so, $I_{\{x\}}$ is the relation where the variable x does not change. Furthermore, the function $comp(v)$ gives the complement of variables relative to its given parameter; so $I_{comp(v)}$ is the relation where all variables except v do not change.

3.3 Introducing parallelism

Rather than the sequential algorithm of Section 3.2, we actually have in mind a development (from the original specification in Figure 3) which uses parallel tasks to check subsets of v. It would be possible to have these processes work entirely independently on a partition of the indices; after their completion, they would choose the lowest index where $pred$ was found to be satisfied by $v(i)$. However, even with separate processors for each thread, this would actually present the risk of it taking longer than the sequential solution (consider a split over two processes, if there is only one index where $pred(v(i))$ holds, no result can be confirmed until at least half of the indices are examined). So the interest is in having the processes communicate in a way that permits any process to avoid searching higher indices than one where a value which satisfies $pred$ has already been located.

The particular set of rely/guarantee rules that we use here assumes specific properties about the relations used in a specification. These assumptions reflect choices that we have made and are not required for all possible rely/guarantee rule sets. (See the discussion on completeness in Section 1.) A major assumption that we make is that the rely and guarantee conditions must be both reflexive and transitive. Reflexivity allows for atomic steps that do not change the state. Transitivity implies that consecutive actions of the same kind must satisfy their condition when taken as a whole. For example, if a program does several steps that, individually, satisfy that program's guarantee condition, and between these steps the environment does nothing that might mutate the state, then that whole sequence of actions will also satisfy the program's guarantee condition.

We are also assuming that the components of a rely/guarantee specification satisfy certain constraints with respect to each other. Although these constraints must be shown to be valid for each specification, they are stated here as axioms since that is how they are used in the proofs.

The first constraint regards interaction between the pre condition and interference from the environment

$$\textbf{PR-ident}\ \frac{}{\overleftarrow{P} \wedge R\ \Rightarrow\ P}$$

This indicates that, if the environment acts on a state that satisfied the pre condition, then the following state still satisfies the pre condition. It is important to note that this property alone is *not* sufficient to establish that the test of a conditional construct such as *If* or *While* still holds during the execution of that construct's body; this point is dealt with in more detail in Section 3.4.

There is also a pair of constraints that describe how interference influences the states that satisfy the post condition

RQ-ident $\dfrac{}{\overleftarrow{P} \wedge (R \,|\, Q) \;\Rightarrow\; Q}$

QR-ident $\dfrac{}{Q \,|\, R \;\Rightarrow\; Q}$

Of these, the first essentially allows a post condition to be "stretched" to an earlier state in the program's execution history. This is useful when proving that the post condition of the body of an *If* construct also applies to the whole construct. The second allows us to assert that interference from the environment does not make a previously true post condition false.

Returning to the design of FINDP, the overall structure of the program sets a temporary variable t beyond the top of v; then executes parallel threads whose overall effect is specified here as $SEARCHES$; and finally copies the value of t into r.[9]

$t \leftarrow$ **len** $v + 1$;
$SEARCHES$;
$r \leftarrow t$

$SEARCHES$
rd $v\colon X^*$
wr $t\colon \mathbb{N}_1$
pre $\forall i \in \{1..t-1\} \cdot \delta(pred(v(i)))$
rely $v = \overleftarrow{v} \wedge t = \overleftarrow{t}$
guar true
post $t \leq \overleftarrow{t} \wedge (t = \overleftarrow{t} \vee pred(v(t))) \wedge$
$\quad \forall i \in \{1..t-1\} \cdot \neg\, pred(v(i))$

This can be justified by **Assign-I**, **Seq-I** and **weaken** rules of Appendix A. The introduction of a new variable, t, requires some care in this step as we are assuming it to be local even though our target implementation language only has global variables. However, as t is not already referenced, we are taking it as unused and assuming that there is an implicit $t = \overleftarrow{t}$ in the rely condition of $FINDP$'s specification.

The statement that a particular implementation –call it $searches$– satisfies the specification of $SEARCHES$, would be written as

$$\{pre\text{-}SEARCHES, rely\text{-}SEARCHES\} \vdash searches \textbf{ sat } (guar\text{-}SEARCHES, post\text{-}SEARCHES)$$

which has the general form $\{P, R\} \vdash S$ **sat** (G, Q). This form of writing satisfaction nicely separates the developer's assumptions from the constraints on the execution of the program. Furthermore, this statement is a deduction within the rely/guarantee logic that, given a state which satisfies the pre condition, P, and a run-time environment in which all changes made by the environment to the state satisfy the rely condition, R, asserts that execution of the program S will

1. only change the state such that each individual change satisfies the guarantee condition, G;
2. always terminate; and
3. when finished, produce a state that, when taken with the initial pre condition-satisfying state, satisfy the post condition, Q.

This is the semantic notion which has to be proved of the introduction rules in Appendix A.

The most interesting of the rules is the one for showing how rely and guarantee conditions are combined for the parallel construct. For simplicity, we choose here to use exactly two processes (this is in fact what Owicki did in her thesis [Owi75]; [Jon81] generalised to an arbitrary partition of the indices over n processes). So $SEARCHES$ could be implemented by

$$SEARCH(\{i \in \{1..\textbf{len } v\} \mid is\text{-}odd(i)\}) \parallel SEARCH(\{i \in \{1..\textbf{len } v\} \mid \neg\, is\text{-}odd(i)\})$$

[9] At this stage of development, we might be tempted to use r directly but t will actually be reified into an expression in Section 3.4. The authors have no difficulty in "confessing" that this might cause a designer to backtrack one step of design to arrive at the need for a separate variable t.

where $SEARCH$ can be specified[10] as

$SEARCH(ms\colon \mathbb{N}\text{-}\mathbf{set})$
rd $v\colon X^*$
wr $t\colon \mathbb{N}_1$
pre $\forall i \in ms \cdot \delta(pred(v(i)))$
rely $v = \overleftarrow{v} \wedge t \leq \overleftarrow{t}$
guar $t = \overleftarrow{t} \vee t < \overleftarrow{t} \wedge pred(v(t))$
post $\forall i \in ms \cdot i < t \Rightarrow \neg\, pred(v(i))$

The intuition so far is that the two concurrent processes search their index range in ascending order in much the same way as the sequential program in Section 3.2 but that they communicate the fact that they find i such that $pred(v(i))$ by setting the shared variable t; providing they mutually respect the protocol of never increasing the value of t, a process can quit once it reaches the index where another process has found $pred$ to be satisfied.

We use the key **Par-I** rule to show that the parallel statement satisfies the specification in $SEARCHES$. Notice how the parts of the combination work. Each thread has to tolerate the interference coming either from $rely\text{-}SEARCHES$ or from the other thread; but this disjunction still leaves $t \leq \overleftarrow{t}$ safe. Ensuring $guar\text{-}SEARCHES$ is trivial because it is identically **true**. The combination of the post conditions of the two threads only achieves $\forall i \in \{1..t-1\} \cdot \neg\, pred(v(i))$; so we really do need the (transitive closure of) the two guarantee conditions in order to achieve $post\text{-}SEARCHES$.[11] We should echo here the point about mergings including with uncle and nephew threads: the proof rule specifically copes with interference from such threads.

3.4 Decomposing $SEARCH$ and reifying t

There is however a problem hidden in the preceding section: updating the variable t which is shared between the two instances of $SEARCH$ could lead to a race condition. An assignment such as $\langle t \leftarrow min(t, \cdots)\rangle$ would need to be made "atomic" since the language of Section 2 permits interference during expression evaluation.[12] A key idea in [Jon07] is that a useful strategy to avoid such problems is by the choice of suitable reifications of abstract variables. We choose to implement t as $min(ot, et)$. The specification of the process responsible for the odd indices becomes:

$SEARCH\text{-}Odd$
rd $v\colon X^*$
rd $et\colon \mathbb{N}_1$
wr $ot, oc\colon \mathbb{N}_1$
pre $\forall i \in odds(\mathbf{len}\ v) \cdot \delta(pred(v(i)))$
rely $v = \overleftarrow{v} \wedge et \leq \overleftarrow{et} \wedge ot \leq \overleftarrow{ot}$
guar $ot = \overleftarrow{ot} \vee ot < \overleftarrow{ot} \wedge pred(v(ot))$
post $\forall i \in odds(\mathbf{len}\ v) \cdot i < ot \Rightarrow \neg\, pred(v(i))$

This alone does not fully resolve the problem but the issue left open makes it possible, with a minimum of artifice, to illustrate two different ways of coping with interference in the if/while rules. Given the level of interference allowed in the language defined in Section 2, we can either add a proof requirement that evaluation of the b test is stable under R or we have to prove facts about the body of the while statement (respectively, the embedded statement of the if statement) without being able to take the b as an extra pre condition (cf. **sim-If-I**, **sim-While-I** in Section 3.2).

To illustrate these two possibilities, we chose the latter course for the while rule (i.e. **While-I** in Appendix A only has P as a pre condition for proofs about $body$) whereas **If-I** has the requirement that b **indep** R so that b can be used as a pre condition for reasoning about its $body$. The b **indep** R requirement in **If-I** is actually stronger than we require since it means that the test expression b is completely unaffected by interference bounded by R. This is much stronger than the **PR-ident** assumption on the overall rule set that we make earlier: that only requires that any evaluation in a single state is stable. However, **PR-ident** does not require any stability when an expression

[10] Strictly, each $SEARCH$ process need only rely on v being unchanged over its ms but stating this formally requires yet one more VDM operator ($\lhd$) and adds nothing to the understanding of the rules.

[11] This can be compared with the rule in [Pre03] which needs a stronger (and less isolating) rely condition.

[12] The (obvious) functions min and $odds$ are used in the explanation but not the final code: they are not part of the language defined in Section 2; the use of the **if** construct avoids the need for min.

is evaluated across a series of interfered-with states; we therefore need a stronger condition for expressions in general.

The strength of the requirements on the rules is a choice that is partly motivated by the development of the FINDP example: the implementation in Figure 4 is written to take advantage of the rules as they are presented. We can develop sound rules for both of the constructs, taking either or both courses; the applicability of a particular form of the rules to the development is what motivates the decision.

$ot \leftarrow$ **len** $v + 1$**;** $et \leftarrow$ **len** $v + 1$**;**
par
$\|$ ($oc \leftarrow 1$**;**
  while $(oc < ot \wedge oc < et)$ **do**
    if $oc < ot \wedge pred(v(oc))$ **then** $ot \leftarrow oc$ **fi;**
    $oc \leftarrow oc + 2$
  od)
$\|$ ($ec \leftarrow 2$**;**
  while $(ec < et \wedge ec < ot)$ **do**
    if $ec < et \wedge pred(v(ec))$ **then** $et \leftarrow ec$ **fi;**
    $ec \leftarrow ec + 2$
  od)
rap;
if $ot < et$ **then** $r \leftarrow ot$ **fi; if** $et < ot$ **then** $r \leftarrow et$ **fi**

Fig. 4. An implementation of FINDP

There are several interesting observations about the implementation of FINDP in Figure 4. The tests in both **while** statements (e.g. $(oc < ot \wedge oc < et)$) suffer interference from the other parallel thread; so one cannot take the (whole of) the test as an assumption for reasoning about the body. Rather than have a rule that makes such a fine distinction, we have chosen to repeat one conjunct in each thread (e.g. $oc < ot$) in the test of the **if** statement where (because they use variables that are "written to" only in the current thread) it is possible to carry the test into the pre condition of the body. The assignments (e.g. $oc \leftarrow oc + 2$) do not satisfy the "At Most One Assignment" rule[13] (i.e. only one shared variable per assignment) but are safe because, like $r \leftarrow r + 1$ earlier, there is no interference.

To see that this code satisfies the specifications of $SEARCH$ in Section 3.3, note that there is still a reference to a shared (changing) value in the test expression of the **while** but that the choice of the representation of t ensures the first conjunct of the guarantee condition; the argument for the second conjunct is similar. The post condition of $SEARCH$ follows by **While-I**.

Although the specification does not forbid us from checking every element of v even after we have found the minimum index that satisfies $pred$, we are trying to avoid doing so if possible. Given that the evaluation of $pred$ is expensive, one of the considerations in this design is how often we end up evaluating it — that is, how often we have to execute the loop body of either instance of $SEARCH$. Because of the representation chosen for t, the worst case only ends up with one extra evaluation of $pred$ for each $SEARCH$ block that does not find the minimum index. In most cases, this will not happen, but it can if the ot and et variables in the min expression are read just before being updated by the opposite parallel branch.

4 Soundness

For a sequential (non-concurrent) language, it is straightforward to establish that Floyd/Hoare-like rules are consistent with an operational (or denotational) semantics: early citations are [Lau71,Don76], and [Jon87] provides a soundness proof of the sequential rules of VDM based on a denotational description of the underlying language. There are even papers such as [Bro05] which undertake this with a denotational semantics (but without "power domains") for concurrency.

We need here to cope with concurrency –and its inherent non-determinacy– in the language description. The basic approach (like that in [KNvO$^+$02]) is to view the SOS rules as inference rules which make it possible to prove results about the $\xrightarrow{s}$ relation.

[13] This is occasionally referred to as "Reynold's rule" but see (9.32) and page 327 of [Sch97] for a more accurate attribution.

The general approach to –and challenges of– what needs to be proved can be explained without the complications of concurrency so a proof for the sequential subset of our language is *sketched* first for ease of understanding (the main steps of proof for the concurrent language are outlined in Sections 4.3–4.5; the detailed proofs are relegated to the technical report version of this paper [CJ07]).

4.1 The sequential case

The overall soundness result for a sequential language (assume that $Par \notin Stmt$ in Figure 2) would be that, under the assumption that we have a proof using inference rules in Appendix A (i.e. S **sim-sat** (P, Q)), if S is executed in a state for which $[\![P]\!](\sigma)$, then (a) the program cannot fail to terminate (i.e. the $\overset{s}{\longrightarrow}*$ relation must lead to $(\mathbf{nil}, \sigma')$ in a finite number of transitions); and (b) any state σ' for which $(S, \sigma) \overset{s}{\longrightarrow}* (\mathbf{nil}, \sigma')$ will be such that $[\![Q]\!](\sigma, \sigma')$.

The proofs for (a) and (b) above can be done by structural induction over the abstract syntax for $Stmt$:

$$\boxed{\textbf{Stmt-Indn}}\frac{\begin{array}{l}H(\mathbf{nil})\\ S \in Assign \vdash H(S)\\ H(sl) \wedge H(sr) \;\Rightarrow\; H(mk\text{-}Seq(sl, sr))\\ H(S) \;\Rightarrow\; H(mk\text{-}If(b, S))\\ H(S) \;\Rightarrow\; H(mk\text{-}While(b, S))\\ H(sl) \wedge H(sr) \;\Rightarrow\; H(mk\text{-}Par(sl, sr))\end{array}}{\forall S \in Stmt \cdot H(S)}$$

The assumption is made that all of the assignments have already been proven directly.[14] These proofs are, in [CJ07], laid out in a natural deduction format similar to that used in [Jon90,JJLM91,FL98].[15]

For the proofs in hand, the termination argument needs the correctness result to establish that the pre-condition of the second (sr) part of a Seq is satisfied. It is sound to prove correctness first since we only need to consider those final states that the model *can* reach; for a divergent computation there is no final state to consider.

In the sequential case, it is easy to see that structural induction suffices for most of the argument. The only exception to this is the lemma for $While$: this is, in a technical sense, the most interesting proof since it requires the use of complete induction[16] over well-founded relation from **sim-While-I** in Section 3.2 (in addition to structural induction on the body of the $While$). We identify this transitive well-founded relation as $W \in \mathcal{P}(\Sigma \times \Sigma)$.The **sim-While-I** rule is also written such that all states that satisfy the pre condition of the rule, P, must be contained within either the domain or range of W (i.e. $P \subseteq \Sigma$). Finally, the equivalent of a "base case" for this inductive rule are those states which are not contained in the domain of W.

$$\boxed{W\text{-}Indn}\frac{\big(\forall a' \cdot W(a, a') \;\Rightarrow\; H(a')\big) \;\Rightarrow\; H(a)}{H(a)}$$

It is a consequence of the compositionality of the proof rules used in Section 3.2 that structural induction gets us so far in the sequential case; retaining this property in the concurrent language was one of our goals for the following proofs.

[14] Soundness for assignment statements must be argued directly in terms of the underlying SOS rules for each assignment individually.

[15] We follow the ideas in [Gen35] in reasoning about propositional operators and quantifiers with the aid of introduction and elimination rules. Our layout resembles that of Jáskowski in [Pra65]. These older references influenced [Gri81, Chapter 3] which in turn stimulated the use of natural deduction in [Jon90] and subsequent VDM publications.
It is perhaps surprising that [Gri81] confined the use of natural deduction to one chapter; having made progress on the presentation of bound variables, VDM has used the style extensively.

[16] As a reminder, complete induction over the integers is:

$$\boxed{\mathbb{N}\text{-}Indn}\frac{(\forall i < n \cdot H(i)) \;\Rightarrow\; H(n)}{H(n)}$$

4.2 The concurrent case

Even in the case of concurrency, we assume that whole programs are run without external interference (cf. Section 4.6), so the final result (Corollary 26) we need is that, when a program S has been shown to satisfy a pre/post condition (P, Q respectively) specification — that is

$$\{P, I\} \vdash S \textbf{ sat } (\textbf{true}, Q)$$

has been proved from the rules in Appendix A, it should hold that, for states σ where $[\![P]\!](\sigma)$ is true (a) the program cannot fail to terminate (i.e. the $\xrightarrow{s}*$ relation leads to $(\textbf{nil}, \sigma_f)$ in a finite number of transitions); and (b) any state σ' for which $(S, \sigma) \xrightarrow{s}* (\textbf{nil}, \sigma')$ will be such that $[\![Q]\!](\sigma, \sigma')$.

In order to state the more general property which admits interference, we need to show what it means to run a program with its interference being constrained by a relation. This is defined (with $Rely$ being relations over $(\Sigma \times \Sigma)$) as the transition relation

$\xrightarrow[\_]{r}: \mathcal{P}\left((Stmt \times \Sigma) \times Rely \times (Stmt \times \Sigma)\right)$

This new relation essentially introduces zero or more steps of interference between "ordinary" steps of the $\xrightarrow{s}$ transition.

$$\boxed{\text{A-R-Step}}\ \frac{[\![R]\!](\sigma, \sigma')}{(S, \sigma) \xrightarrow[R]{r} (S, \sigma')}$$

$$\boxed{\text{A-S-Step}}\ \frac{(S, \sigma) \xrightarrow{s} (S', \sigma')}{(S, \sigma) \xrightarrow[R]{r} (S', \sigma')}$$

Corollary 26 is an immediate consequence of Theorem 25 which reflects the possibility of interference. Where

$$\{P, R\} \vdash S \textbf{ sat } (G, Q)$$

has been proved, it should hold that, for states σ where $[\![P]\!](\sigma)$ is true (a) (S, σ) must terminate under $\xrightarrow[R]{r}*$; (b) for any state σ_f reached $[\![Q]\!](\sigma, \sigma_f)$; and (c) steps in the execution of S must not violate the guarantee condition G.

The first part of the proof concerns satisfaction of the post condition Q. As in the sequential case, this has to precede the argument about termination because we need to be able to conclude that the pre condition of the right part of a sequence is satisfied.

4.3 Respecting guarantee conditions

Unlike with the sequential language, here we also need to show that atomic state changes made by portions of programs are within the bounds given by the specified guarantee conditions which arise in the justification of a program. Strictly, the post condition and guarantee condition lemmas are mutually dependant but we present them as though they are independent because there is no technical difficulty in their combination and they are harder to read in the combined form. The dependencies show themselves in the proof that the sequence construct satisfies its guarantee condition, and in the proof that the parallel construct satisfies its post condition. The former proof requires that the post condition of the first part of the sequence is established before it can continue with the second part of the sequence. The latter proof is explained in the next section.

Were we to do the proofs in tandem, we could get away with talking about S satisfying (under appropriate assumptions) G and Q; because we separate the proofs, we need a way of writing claims about interference and choose:

$$\{P, R\} \models S \textbf{ within } G$$

which is the first point in the definition of $\{P, R\} \vdash S$ **sat** (G, Q) given in Section 3.3. This is actually a model-theoretic notion, and as such is independent of the definition given for **sat**; it is defined to be the equivalent of

$$\forall \cdot \sigma_0, \sigma_i, \sigma_j \in \Sigma \cdot \left([\![P]\!](\sigma_0) \wedge (S, \sigma_0) \xrightarrow[R]{r}* (S_i, \sigma_i) \xrightarrow{s} (S_j, \sigma_j)\right) \Rightarrow [\![G]\!](\sigma_i, \sigma_j)$$

Informally, this is amounts to the claim that the guarantee condition forms a bound within which all possible actions of the program must fit. We will frequently write S **within** G where P and R can be easily inferred from the context.

In the simplest case, we know that a completed program does nothing, and can therefore satisfy any guarantee, giving us:

$$\boxed{\textbf{nil}\text{-}\textit{within}}\ \frac{}{\{P, R\} \models \textbf{nil within } I}$$

This lemma can be derived directly from the semantics of our language and the definition of **within** above; intuitively, the simple fact that the statement **nil** cannot modify the state means that it satisfies the identity relation. Though obvious, this lemma is required to show the soundness of the parallel rule.

The only state changes caused by a program S come from the final step of executing assignments (see Assign-E in Section 2). It is relatively clear what S **within** G means for $S \in Assign$ (the qualification here is that, for example, $mk\text{-}Assign(x, x+1)$ **within** $\overleftarrow{x} < x$ only holds under some rely conditions R because of the fine grained semantics in Section 2). For composite statements S, S **within** G requires that G holds for every contained assignment.

The onus is on the user of the proof rules of Appendix A to show (using the SOS) that any assignments satisfy their associated **Assign-I**. Lemmas 1–4 are the separate cases (by the other types of $Stmt$) which justify Theorem 5 which follows by structural induction. Each of the proofs of the lemmas are straightforward precisely because the only way to change the values in the state is by assignment.

Having understood the notion of a piece of program respecting an interference constraint, we need to know that, under increased interference, there can never be fewer possible results. This is a monotonicity result on interference but the case we need in the subsequent proofs relates specifically to the observation that executing sl and sr in parallel will yield fewer possible resulting states than executing sl with the interference by which sr has been shown to be bounded (i.e. G_r). This is captured in Lemma 6 (and a symmetrical version for the right, Lemma 7). At first encounter, it might be easier to understand this result expressed using set comprehension. If, under the assumption of $\{P, R\} \models S$ **within** G, then

$$\left\{\sigma' \in \Sigma \;\middle|\; (mk\text{-}Par(sl, sr), \sigma) \xrightarrow[R]{r}{*} (mk\text{-}Par(sl', sr'), \sigma')\right\} \subseteq \left\{\sigma' \in \Sigma \;\middle|\; (sl, \sigma) \xrightarrow[R \vee G_r]{r}{*} (sl', \sigma')\right\}$$

This lemma is crucial to our ability to undertake the proofs in a compositional way using structural induction.

Further properties required in later lemmas are given in Lemmas 11 and 12.

4.4 Correctness

The key correctness proofs (Lemmas 13–16 and Theorem 17) show that, under appropriate assumptions, the final states will satisfy Q. The statements of all lemmas and theorems are given in Appendix B; their proofs are available in [CJ07]. Thanks to Lemmas 6 and 7, most of the proofs are only slightly more complex than in the sequential case.

It is important to realise the role of rely/guarantee conditions in the argument. To achieve separation of arguments about different "threads" in a program, there has to be a way of reasoning about a thread in isolation even though its execution can be interrupted (at a very fine grain) by other threads. Rely/guarantee conditions provide exactly this separation but introduce the need to show that the execution of a branch of a Par respects its guarantee condition.

The $While$ statement is one place where interference has a significant impact on the form of the rule in Appendix A: at first sight, it may come as a shock to some[17] that one can no longer assume (in general) that b is true for the $body$ proof but this is a direct consequence of (fine-grained) interference and it should be noted that the development in Section 3.4 uses a statement where such interference occurs. It is of course possible to justify alternative rules which cover the situation where b is stable under R (alternately, b is independent of R).

The proof of Lemma 15 uses complete induction over the W relation.

Also of interest is the proof for Par (see Lemma 16). It is precisely here that the fact that the developer of a program using the rely/guarantee rules in Appendix A has to prove their program correct under a wider assumption of interference than –in general– will arise from the actual concurrently executing program. This is where Lemmas 6 and 7 are key.

[17] Those who have actually done concurrent programming will be least surprised.

It is enlightening to compare where the proof challenge comes from for Seq and Par: in the former case, one needs to know that executing the first sl component establishes the pre condition of the second (sr); for concurrency, the proof effort is expended on establishing mutual respect of each component's rely condition.

4.5 Termination

The rules in Appendix A are intended to prove what is often called "total correctness": a correct program must always terminate if it is used as intended.[18] Unlike with sequential programs, the termination argument here cannot be by straight structural induction because of the interleaving of threads. Consider first how one might argue that programs terminate if there were no $While$ construct in the language. It is straightforward to define a lexicographic ordering over $Stmt$ such that all of the SOS rules in Section 2 reduce the program part of a configuration. Actually, most of the rules in our language genuinely reduce the depth of a $Stmt$ syntax; the only special case is in expressions where identifiers can be replaced by their values. Such a lexicographic reduction is clearly finite.

The presence of a $While$ construct potentially complicates the argument in two ways: textual expansion and the potential for blocking. First, it is obvious that a program might contain a non-terminating loop: the SOS rules would continually replace the offending instance of the non-terminating $While$ with a longer text (cf. While). Given that any program S for which $\{P, R\} \vdash P$ **sat** (G, Q) has a well-ordering (W) for each loop, such non-termination is ruled out.

The issue of blocking appears to be more subtle but is in fact an aspect of the same point. Within the language of Appendix A, one could write a program with a $While$ construct which "waited" on a flag to be set in a parallel thread but it would not be possible to prove that such a program satisfied a specification because there would be no well-founded W for such a blocking $While$. So although such a program conforms to the language description, it is not of concern to our soundness proof for the rules of Appendix A. Thus, it is the lemma about the $While$ construct which is most interesting (in exactly the same way as with the sequential language): Lemma 23 needs to use complete induction over the termination relations used in the proof of the their respective $While$ statements.

This last observation is interesting because of its connection with "fairness". A program which waited for another thread to unblock its "flag" would rely on fairness of the execution order of the SOS rules. We finesse this issue because of the need for the programmer to prove termination.

There are of course subtleties in formalising the argument above: one must remember that the need is to show divergence is impossible on any non-deterministic evaluation (not just that the evaluation *can* terminate). The final theorem on termination (Theorem 24) just appeals to the lexicographic ordering of program texts for the statements other than $While$ and appeals to Lemma 15 for $While$.

4.6 Final theorem

Theorem 25 When

$$\{P, R\} \vdash S \textbf{ sat } (G, Q)$$

has been proved from the rules in Appendix A it should hold that, when $[\![P]\!](\sigma)$ is true (S, σ) must terminate under $\xrightarrow[R]{r}$; and for any state σ_f reached $[\![Q]\!](\sigma, \sigma_f)$.

Proof Follows immediately from Theorems 17 and 24.

Corollary 26 If

$$\{P, I_\Sigma\} \vdash S \textbf{ sat } (\textbf{true}, Q)$$

has been proved from the rules in Appendix A it should hold that, when $[\![P]\!](\sigma)$ is true (S, σ) must terminate under $\xrightarrow{s}$; and for any state σ_f reached $[\![Q]\!](\sigma, \sigma_f)$

Proof This is an immediate corollary of Theorem 25.

[18] The termination proofs are thus important but are omitted in [Pre03] which only tackles "partial correctness" — see Section 5.1.

5 Conclusions

5.1 Related work

The most relevant piece of related research is certainly [Pre01] (see [Pre03] for an overview) which provides an Isabelle/HOL proof of two related results. The differences are interesting and we hope in the future to explore to what extent they come about because of the constraints of a complete machine-checked proof.

We review here a representative sample of proofs of soundness of rely/guarantee rules. Jones' own [Jon81] presents an argument that the form of rely/guarantee rules contained there are sound with respect to an operational semantics. This proof is far from formal and is made opaque by being based on a VDL-style [LW69] operational semantics (it is worth remembering –as an extenuating circumstance– that Plotkin's Århus notes on SOS [Plo81] were not written until 1981). The proof in [Jon81] is based on an argument over all computations even though the state of a computation is viewed in VDL as a (rather Baroque) tree.

An obvious comparison of our work is with Ketil Stølen's PhD: in[Stø90, §4.2.2] he makes clear that, like Prensa Nieto, he also assumes that expression evaluation –including that in tests– is atomic (whereas we avoid this assumption). The semantic model in [Stø90] is based on a transition relation between configurations and he bases arguments on computation sequences (in fact, "potential computations"). Of course, Stølen is facing the additional challenge of reasoning about rules which have a wait condition to facilitate proofs about progress; he also tackles completeness for his rules — this interacts with the thorny issue of of "auxiliary variables" which is a topic discussed in Section 5.2 below.

Finally, we move back to Leonor Prensa Nieto's recent proofs (she discusses [Xu92,XdH97] which she points out follow a similar proof approach to that in [Pre01] although Xu Qiwen's "is presented in a conventional pencil and paper style").

Some of the differences between [Pre01] and the current paper derive from the choices of language semantics.

- Here we allow a much finer level of interference (indeed, the embedding of whole statements as functions in the program and the way predicates are tested in [Pre01,Pre03] might seem natural to a HOL user but is surprising to anyone familiar with semantics based on a more conventional (abstract) syntax). Ours was not an arbitrary decision — we have argued elsewhere [Jon07] that assuming large atomic steps would make languages very expensive to implement.
- Whereas we allow nested parallel statements, Prenso Nieto does not: she observes that her decision to define a "parallel program" as a list of (un-nested) "component programs" which do not allow any concurrency is to simplify her proofs. Our Lemma 6 shows that coping with more global thread switching is possible.
- [Pre03] does allow "await" statements in the same way that Owicki did in [Owi75]. They would present no fundamental problems for us but would introduce a termination problem whose finessing we comment on in Section 4.5.

The language decisions obviously affect the proof rules used. One surprise in [Pre03] is the decision to use post conditions which are predicates of the final state only (rather than relations between initial and final states).[19] Another major difference with what is presented here is the fact that the soundness proof in [Pre01] does not tackle termination (it only addresses so-called "partial correctness").

The proof of soundness for her chosen rules in [Pre03, see §4.2.3] is based on "computations". That having been said, we believe that both approaches could benefit from the other and we are in the process of following up on this. We hope to investigate whether our Lemma 6 can be used to simplify a machine checked proof of rules which cope with fine-grained interference and nested parallel constructs.

The view that rules like those in [Jon96] are proved as needed contrasts starkly with that of providing a (complete) axiomatic semantics for a language. The current authors note that the only non-trivial language for which this has been done is "Turing" [H$^+$88]. Our view absolves us from concerns about completeness because one can prove more rules as required. This is fortunate because rely/guarantee rules have to fit many different styles of concurrent programming (depending on the intimacy of interference employed) and it is difficult to envisage a single canonical set.

[19] The source of the idea to use post conditions of single states would appear to be [Sti86] (which even uses predicates of single states for rely and guarantee conditions). This idea is not in the spirit of [Jon81,Jon83a] which views all such assertions as relations over pairs of states. Stirling was attempting completeness proofs and the simplification there is understandable but the counter-intuitive coding of, for example, variables retaining their values reduces the usability of the excellent combination of approaches in [Din00].

5.2 Further work

There is as always much more to be done. We have, of course, used the rely/guarantee rules on other examples. It is worth noting that the only interesting interference in FINDP is on setting the one shared variable (originally t; then reified). This means that FINDP is actually a little too simple to show the advantage (over Owicki/Gries) of compositional reasoning –the "prime sieve" of [Jon96] is a more convincing example– but FINDP is shorter and illustrates most aspects of rely/guarantee rules.

It would be useful to compile collections of rules (like those in Appendix A) which are tuned to different "patterns" of concurrent programming. In particular, it would be interesting to look at rules for assignment statements that embody different interference assumptions.

We are also interested in considering the requirements of machine checked proofs (our proofs in [CJ07] are presented in a rather formal "natural deduction" style but have not been machine generated). In doing this, it would be worth examining again the soundness proofs in [Jon87] (or in detail with the Technical Report version thereof [Jon86]) where we gave a (relational) denotational semantics (the basic proof tool there was fixed point induction). Although that work was based on a sequential (non-concurrent) language and is in a denotational setting, it is clear that reasoning explicitly about relations avoids having to pull out explicitly (name) many individual states.

It was in fact difficulties with the heaviness of proofs using rely/guarantee conditions which led Jones to embark on constraining interference by using concurrent object based languages; this development is summarised in [Jon96]. The fact remains that if one wishes to use "racy" interference, something like rely/guarantee proofs appear to be required.

We feel that constructing the proofs has sharpened our understanding of the expressiveness of rely and guarantee conditions. It was clear from the beginning that expressing interference via a relation was weak in the sense that there are things one would want to say that cannot be expressed. The standard way of achieving the sort of completeness result in [Stø90][20] is to employ "auxiliary" (or "ghost") variables. They essentially make it possible to encode in variables information about the flow of control. The insight which comes from the proofs here is derived from Lemmas 6 and 7 which make precise the limited expressiveness of the relation intended to capture a rely condition. We would like to take this idea forward to look at controlled extensions to the language used for recording and reasoning about interference.

Acknowledgements The technical content of this paper has benefited from discussions with Jon Burton, Ian Hayes, Tony Hoare and Leonor Prensa Nieto. The authors gratefully acknowledge funding for their research from EPSRC (DIRC project and "Splitting (Software) Atoms Safely") and the EU IST-6 programme (for RODIN). The authors are indebted to the journal's anonymous referees whose comments helped us clarify the earlier draft.

References

[Acz82] P. Aczel. A note on program verification. (private communication) Manuscript, Manchester, January 1982.

[Apt81] K. R. Apt. Ten years of Hoare's logic: A survey – part I. *ACM Transactions on Programming Languages and Systems*, 3:431–483, 1981.

[BCJ84] H. Barringer, J. H. Cheng, and C. B. Jones. A logic covering undefinedness in program proofs. *Acta Informatica*, 21:251–269, 1984.

[BH88] Rod Burstall and Furio Honsell. A natural deduction treatment of operational semantics. Technical Report ECS-LFCS-88-69, LFCS, University of Edinburgh, 1988.

[Bro05] Stephen Brookes. Retracing the semantics of CSP. In Ali E. Abdallah, Cliff B. Jones, and Jeff W. Sanders, editors, *Communicating Sequential Processes: the First 25 Years*, volume 3525 of *LNCS*. Springer-Verlag, 2005.

[CJ00] Pierre Collette and Cliff B. Jones. Enhancing the tractability of rely/guarantee specifications in the development of interfering operations. In G. Plotkin, C. Stirling, and M. Tofte, editors, *Proof, Language and Interaction*, chapter 10, pages 275–305. MIT Press, 2000.

[CJ07] J. W. Coleman and C. B. Jones. Guaranteeing the soundness of rely/guarantee rules (revised). Technical Report CS-TR-??, School of Computing Science, University of Newcastle, 2007.

[CM92] J. Camilleri and T. Melham. Reasoning with inductively defined relations in the HOL theorem prover. Technical Report 265, Computer Laboratory, University of Cambridge, August 1992.

[Dij76] E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.

[Din00] Jürgen Dingel. *Systematic Parallel Programming*. PhD thesis, Carnegie Mellon University, 2000. CMU-CS-99-172.

[20] Such a result was first sketched by Ruurd Kuiper in [Kui83].

[Don76] J. E. Donahue. *Complementary Definitions of Programming Language Semantics*, volume 42 of *Lecture Notes in Computer Science*. Springer-Verlag, 1976.

[dR01] W. P. de Roever. *Concurrency Verification: Introduction to Compositional and Noncompositional Methods*. Cambridge University Press, 2001.

[DS90] Edsger W Dijkstra and Carel S Scholten. *Predicate Calculus and Program Semantics*. Springer-Verlag, 1990. ISBN 0-387-96957-8, 3-540-96957-8.

[FL98] John Fitzgerald and Peter Gorm Larsen. *Modelling systems: practical tools and techniques in software development*. Cambridge University Press, 1998.

[Gen35] G. Gentzen. Untersuchungen über das logische Schliessen. *Matematische Zeitschrift*, 39:176–210, 405–431, 1935. Available as Investigations into Logical Deduction, Chapter 3 of The Collected Papers of Gerhard Gentzen (ed.) M. E. Szabo.

[Gri81] D. Gries. *The Science of Programming*. Springer-Verlag, 1981.

[GS96] David Gries and Fred B. Schneider. *A Logical Approach to Discrete Math*. Springer-Verlag, second edition, 1996.

[H+88] Richard C. Holt et al. *The Turing Programming Language: Design and Defintion*. Prentice Hall International, 1988.

[HJJ03] Ian Hayes, Michael Jackson, and Cliff Jones. Determining the specification of a control system from that of its environment. In Keijiro Araki, Stefani Gnesi, and Dino Mandrioli, editors, *FME 2003: Formal Methods*, volume 2805 of *LNCS*, pages 154–169. Springer Verlag, 2003.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 583, October 1969.

[JHJ07] Cliff Jones, Ian Hayes, and Michael Jackson. Specifying systems that connect to the physical world. *LNCS*, 2007. Submitted.

[JJLM91] C. B. Jones, K. D. Jones, P. A. Lindsay, and R. Moore. *mural: A Formal Development Support System*. Springer-Verlag, 1991. ISBN 3-540-19651-X.

[Jon72] C. B. Jones. Formal development of correct algorithms: an example based on Earley's recogniser. In *SIGPLAN Notices, Volume 7 Number 1*, pages 150–169. ACM, January 1972.

[Jon80] C. B. Jones. *Software Development: A Rigorous Approach*. Prentice Hall International, 1980. ISBN 0-13-821884-6.

[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.

[Jon83a] C. B. Jones. Specification and design of (parallel) programs. In *Proceedings of IFIP'83*, pages 321–332. North-Holland, 1983.

[Jon83b] C. B. Jones. Tentative steps toward a development method for interfering programs. *ACM Transactions on Programming Languages and Systems*, 5(4):596–619, 1983.

[Jon86] C. B. Jones. Program specification and verification in VDM. Technical Report UMCS 86-10-5, University of Manchester, 1986. Extended version of [Jon87] (includes the full proofs).

[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.

[Jon90] C. B. Jones. *Systematic Software Development using VDM*. Prentice Hall International, second edition, 1990. ISBN 0-13-880733-7.

[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] C. B. Jones. Wanted: a compositional approach to concurrency. In A. McIver and C. Morgan, editors, *Programming Methodology*, pages 1–15. Springer Verlag, 2003.

[Jon03b] Cliff B. Jones. Operational semantics: concepts and their expression. *Information Processing Letters*, 88(1-2):27–32, 2003.

[Jon07] C. B. Jones. Splitting atoms safely. *Theoretical Computer Science, in press*, 2007. doi:10.1016/j.tcs.2006.12.029.

[KNvO+02] Gerwin Klein, Tobias Nipkow, David von Oheimb, Leonor Prensa Nieto, Norbert Schirmer, and Martin Strecker. Java source and bytecode formalisations in Isabelle. 2002.

[Kui83] Ruurd Kuiper. On completeness of an inference rule for parallel composition, 1983. (private communication) Manuscript, Manchester.

[Lau71] P. E. Lauer. *Consistent Formal Theories of the Semantics of Programming Languages*. PhD thesis, Queen's University of Belfast, 1971. Printed as TR 25.121, IBM Lab. Vienna.

[LW69] P. Lucas and K. Walk. *On The Formal Description of PL/I*, volume 6 of *Annual Review in Automatic Programming Part 3*. Pergamon Press, 1969.

[OG76] S. S. Owicki and D. Gries. An axiomatic proof technique for parallel programs I. *Acta Informatica*, 6:319–340, 1976.

[O'H07] Peter O'Hearn. Resources, concurrency and local reasoning. *(accepted for publication in) Theoretical Computer Science*, 2007.

[Owi75] S. Owicki. *Axiomatic Proof Techniques for Parallel Programs*. PhD thesis, Department of Computer Science, Cornell University, 1975.

[Plo81] G. D. Plotkin. A structural approach to operational semantics. Technical report, Aarhus University, 1981.

[Plo04a] Gordon D. Plotkin. The origins of structural operational semantics. *Journal of Logic and Algebraic Programming*, 60–61:3–15, July–December 2004.

[Plo04b] Gordon D. Plotkin. A structural approach to operational semantics. *Journal of Logic and Algebraic Programming*, 60–61:17–139, July–December 2004.

[Pra65] Dag Prawitz. *Natural Deduction: a Proof-Theoretical Study*. Dover publications, 1965.

[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.

[Sch97] Fred B. Schneider. *On concurrent programming*. Graduate Texts in Computer Science. Springer-Verlag New York, Inc., New York, NY, USA, 1997.

[Sti86] C. Stirling. A compositional reformulation of Owicki-Gries' partial correctness logic for a concurrent while language. In *ICALP'86*. Springer-Verlag, 1986. LNCS 226.

[Stø90] K. Stølen. *Development of Parallel Programs on Shared Data-Structures*. PhD thesis, Manchester University, 1990. Available as UMCS-91-1-1.

[XdH97] Q. Xu, W.-P. de Rpever, and J. He. The rely-guarantee method for verifying shared variable concurrent programs. *Formal Aspects of Computing*, 9(2):149–174, 1997.

[Xu92] Qiwen Xu. *A Theory of State-based Parallel Programming*. PhD thesis, Oxford University, 1992.

A Inference Rules

$$\textbf{weaken}\ \frac{\begin{array}{l}\{P, R\} \vdash S\ \textbf{sat}\ (G, Q)\\ P' \ \Rightarrow\ P\\ R' \ \Rightarrow\ R\\ G \ \Rightarrow\ G'\\ Q \ \Rightarrow\ Q'\end{array}}{\{P', R'\} \vdash S\ \textbf{sat}\ (G', Q')}$$

$$\textbf{Par-I}\ \frac{\begin{array}{l}\{P, R \vee G_r\} \vdash sl\ \textbf{sat}\ (G_l, Q_l)\\ \{P, R \vee G_l\} \vdash sr\ \textbf{sat}\ (G_r, Q_r)\\ G_l \vee G_r \ \Rightarrow\ G\\ \overleftarrow{P} \wedge Q_l \wedge Q_r \wedge (R \vee G_l \vee G_r)^* \ \Rightarrow\ Q\end{array}}{\{P, R\} \vdash mk\text{-}Par(sl, sr)\ \textbf{sat}\ (G, Q)}$$

$$\textbf{While-I}\ \frac{\begin{array}{l}bottoms(W, \neg\, b)\\ twf(W)\\ \{P, R\} \vdash body\ \textbf{sat}\ (G, W \wedge P)\\ \overleftarrow{\neg\, b} \wedge R \ \Rightarrow\ \neg\, b\\ R \ \Rightarrow\ W^*\end{array}}{\{P, R\} \vdash mk\text{-}While(b, body)\ \textbf{sat}\ (G, W^* \wedge P \wedge \neg\, b)}$$

Note that W in the **While-I** rule is both transitive and well-founded (twf) over states and stable under R, but W should not be reflexive. The latter property is a side effect of/enforced by W being part of the post condition. The use of the transitive closure of W to imply the post condition is needed to add reflexivity in the case where the overall $While$ does nothing and the initial and final states are identical.

In general, a hypothesis of the form $bottoms(W, P)$, where W is a transitive, well-founded relation and P is a predicate, indicates that the predicate is true for all elements that are *not* in the domain of W. Thus the inference rule

$$\textbf{bottoms}\ \frac{\begin{array}{l}bottoms(W, P)\\ \sigma \notin \textbf{dom}\ W\end{array}}{[\![P]\!](\sigma)}$$

gives the form of the property of $bottoms$ that we use in the proofs. The hypothesis $bottoms(W, \neg\, b)$, then, is used to help ensure that once the $While$ loop has reached a state that is not in the domain of the relation W, the $While$ loop will consequently have its test expression evaluate to false, and terminate the loop.

$$\textbf{If-I}\ \frac{\begin{array}{l} b\ \textbf{indep}\ R \\ \{P \wedge b, R\} \vdash body\ \textbf{sat}\ (G, Q) \\ \overleftarrow{P} \wedge \overleftarrow{\neg\, b}\ \Rightarrow\ Q \end{array}}{\{P, R\} \vdash mk\text{-}If(b, body)\ \textbf{sat}\ (G, Q)}$$

The hypothesis b **indep** R is taken to mean that the evaluation of the expression b is unaffected by interference constrained by R.

$$\textbf{Seq-I}\ \frac{\begin{array}{l} \{P, R\} \vdash sl\ \textbf{sat}\ (G, Q_{sl} \wedge P_{sr}) \\ \{P_{sr}, R\} \vdash sr\ \textbf{sat}\ (G, Q_{sr}) \\ Q_{sl} \mid Q_{sr}\ \Rightarrow\ Q \end{array}}{\{P, R\} \vdash mk\text{-}Seq(sl, sr)\ \textbf{sat}\ (G, Q)}$$

$$\textbf{Assign-I}\ \frac{\textbf{true}}{\{\delta(e), I_{FV(e)}\} \vdash mk\text{-}Assign(v, e)\ \textbf{sat}\ (I_{comp(v)}, v = \overleftarrow{e})}$$

Note that $FV(e)$ denotes the set of free variables in the expression e. Thus, $I_{FV(e)}$ denotes the relation that acts as an identity on the variables free in the expression e. The guarantee condition in the above rule, $I_{comp(v)}$, denotes the relation that acts as an identity on all variables except v.

B Formal statements of all lemmas

The proofs of these results are in the [CJ07].

B.1 Respecting Guarantee Conditions

Lemma 1

$$\textbf{Seq-B}\ \frac{\begin{array}{l} \{P, R\} \vdash sl\ \textbf{sat}\ (G, Q_{sl} \wedge P_{sr}) \\ \{P_{sr}, R\} \models sr\ \textbf{within}\ G \end{array}}{\{P, R\} \models mk\text{-}Seq(sl, sr)\ \textbf{within}\ G}$$

Lemma 2

$$\textbf{If-B}\ \frac{\{P \wedge b, R\} \models body\ \textbf{within}\ G}{\{P, R\} \models mk\text{-}If(b, body)\ \textbf{within}\ G}$$

Lemma 3

$$\textbf{While-B}\ \frac{\{P, R\} \models body\ \textbf{within}\ G}{\{P, R\} \models mk\text{-}While(b, body)\ \textbf{within}\ G}$$

Lemma 4

$$\textbf{Par-B}\ \frac{\begin{array}{l} \{P, R \vee G_{sr}\} \models sl\ \textbf{within}\ G_{sl} \\ \{P, R \vee G_{sl}\} \models sr\ \textbf{within}\ G_{sr} \\ G_{sl} \vee G_{sr}\ \Rightarrow\ G \end{array}}{\{P, R\} \models mk\text{-}Par(sl, sr)\ \textbf{within}\ G}$$

Theorem 5 For any $S \in Stmt$ for which $\{P, R\} \vdash S\ \textbf{sat}\ (G, Q)$, for any $\sigma \in \Sigma$ such that $[\![P]\!](\sigma)$ for any transition $(S', \sigma') \xrightarrow{s} (S'', \sigma'')$ which is reachable from (S, σ), it follows that $[\![G]\!](\sigma', \sigma'')$. Thus

$$\textit{Theorem resp}\ \frac{\{P, R\} \vdash S\ \textbf{sat}\ (G, Q)}{\{P, R\} \models S\ \textbf{within}\ G}$$

B.2 Monotonicity under interference

Lemma 6

$$\boxed{\textit{MonoIntf-r}}\ \frac{sr \textbf{ within } G_r \quad (mk\text{-}Par(sl, sr), \sigma) \xrightarrow[R]{r}* (mk\text{-}Par(sl', sr'), \sigma')}{(sl, \sigma) \xrightarrow[R \vee G_r]{r}* (sl', \sigma')}$$

Lemma 7

$$\boxed{\textit{MonoIntf-l}}\ \frac{sl \textbf{ within } G_l \quad (mk\text{-}Par(sl, sr), \sigma) \xrightarrow[R]{r}* (mk\text{-}Par(sl', sr'), \sigma')}{(sr, \sigma) \xrightarrow[R \vee G_l]{r}* (sr', \sigma')}$$

Lemma 8

$$\boxed{\textit{Isolation-r}}\ \frac{(mk\text{-}Seq(sl, sr), \sigma) \xrightarrow[R]{r}* (mk\text{-}Seq(sl', sr), \sigma')}{(sl, \sigma) \xrightarrow[R]{r}* (sl', \sigma')}$$

Lemma 9

$$\boxed{\textit{Isolation-l}}\ \frac{(mk\text{-}Seq(\textbf{nil}, sr), \sigma) \xrightarrow[R]{r}* (sr', \sigma')}{(sr, \sigma) \xrightarrow[R]{r}* (sr', \sigma')}$$

Lemma 10

$$\boxed{\textit{Isolation-If}}\ \frac{(mk\text{-}If(\textbf{true}, body), \sigma) \xrightarrow[R]{r}* (body', \sigma')}{(body, \sigma) \xrightarrow[R]{r}* (body', \sigma')}$$

B.3 Further lemmas on within

Lemma 11

$$\boxed{\textit{RG-Holds}}\ \frac{\{P, R\} \models S \textbf{ within } G \quad (S, \sigma) \xrightarrow[R]{r}* (S', \sigma')}{[\![(R \vee G)^*]\!](\sigma, \sigma')}$$

Lemma 12

$$\boxed{\textit{Lemma Par-wrap}}\ \frac{\{P, R\} \models sr \textbf{ within } G}{\{P, R\} \models mk\text{-}Par(\textbf{nil}, sr) \textbf{ within } G}$$

B.4 Correctness proofs

Lemma 13 Given $\{P, R\} \vdash mk\text{-}Seq(sl, sr)$ **sat** (G, Q) and providing sl and sr behave according to their specifications, for all $\sigma_0, \sigma_f \in \Sigma$ such that $[\![P]\!](\sigma_0)$, $(mk\text{-}Seq(sl, sr), \sigma_0) \xrightarrow[R]{r}* (\textbf{nil}, \sigma_f)$ it must follow that $[\![Q]\!](\sigma_0, \sigma_f)$

Lemma 14 Given $\{P, R\} \vdash mk\text{-}If(b, body)$ **sat** (G, Q) and providing $body$ behaves according to its specification, for all $\sigma_0, \sigma_f \in \Sigma$ such that $[\![P]\!](\sigma_0)$, $(mk\text{-}If(b, body), \sigma_0) \xrightarrow[R]{r}* (\textbf{nil}, \sigma_f)$ it must follow that $[\![Q]\!](\sigma_0, \sigma_f)$

Lemma 15 Given $\{P, R\} \vdash mk\text{-}While(b, body)$ **sat** (G, Q) and providing $body$ behaves according to its specification, for all $\sigma_0, \sigma_f \in \Sigma$ such that $[\![P]\!](\sigma_0)$, $(mk\text{-}While(b, body), \sigma_0) \xrightarrow[R]{r}* (\textbf{nil}, \sigma_f)$ it must follow that $[\![Q]\!](\sigma_0, \sigma_f)$.

Lemma 16 Given $\{P, R\} \vdash mk\text{-}Par(sl, sr)$ **sat** (G, Q) and providing sl, sr behave according to their specifications, for any $\sigma_0, \sigma_f \in \Sigma$ such that $[\![P]\!](\sigma_0)$, $(mk\text{-}Par(sl, sr), \sigma_0) \xrightarrow[R]{r}* (\textbf{nil}, \sigma_f)$ it must follow that $[\![Q]\!](\sigma_0, \sigma_f)$

Theorem 17 For any $st \in Stmt$ for which $\{P, R\} \vdash st$ **sat** (G, Q), for any $\sigma \in \Sigma$ such that $[\![P]\!](\sigma)$ if $(st, \sigma) \xrightarrow{s} (\textbf{nil}, \sigma')$ then $[\![Q]\!](\sigma, \sigma')$.

B.5 Termination proofs

The definition of the lexicographical ordering, $<$, given $S, S' \in (Stmt - While)$:

1. (expression types)

   $\forall v \in (\mathbb{B} \mid \mathbb{Z} \mid \mathbb{Z}^+), c \in (Id \mid Dyad) \cdot v < c$

2. (internal expressions)

   $\forall E, E' \in Dyad \cdot E.op = E'.op \wedge E.b = E'.b \wedge E'.a < E.a \;\Rightarrow\; E' < E$
   $\forall E, E' \in Dyad \cdot E.op = E'.op \wedge E.a = E'.a \wedge E'.b < E.b \;\Rightarrow\; E' < E$

3. (**nil** at bottom)

   $S \in Stmt \wedge S \neq \textbf{nil} \;\Rightarrow\; \textbf{nil} < S$

4. (expression evaluation)

   $$\forall e, e' \in Expr, s \in Stmt \cdot \begin{pmatrix} (S = mk\text{-}Assign(id, e) \wedge S' = mk\text{-}Assign(id, e')) \vee \\ (S = mk\text{-}If(e, s) \wedge S' = mk\text{-}If(e', s)) \end{pmatrix} \wedge e' < e \;\Rightarrow\; S' < S$$

5. (internal statements)

   $$\forall s, s', sl, sr \in Stmt \cdot \begin{pmatrix} (S = mk\text{-}Seq(s, sr) \wedge S' = mk\text{-}Seq(s', sr)) \vee \\ (S = mk\text{-}Par(s, sr) \wedge S' = mk\text{-}Par(s', sr)) \vee \\ (S = mk\text{-}Par(sl, s) \wedge S' = mk\text{-}Par(sl, s')) \end{pmatrix} \wedge s' < s \;\Rightarrow\; S' < S$$

6. (containment)

   $\forall b \in Expr, sl \in Stmt \cdot S < mk\text{-}Seq(sl, S) \wedge S < mk\text{-}If(b, S)$

7. (transitivity)

   $\exists S'' \in Stmt \cdot S' < S'' \wedge S'' < S \;\Rightarrow\; S' < S$

Lemma 18 The evaluation of an expression always reduces –over repeated steps– to a value.

Lemma 19 The execution of any $s \in Assign$ always reduces –over repeated steps– to the statement **nil**.

Lemma 20 The execution of any $s \in Seq$ always reduces –over repeated steps– to the statement **nil**.

Lemma 21 The execution of any $s \in If$ always reduces –over repeated steps– to the statement **nil**.

Lemma 22 The execution of any $s \in Par$ always reduces –over repeated steps– to the statement **nil**.

Lemma 23 Given $S \in While$ such that $S = mk\text{-}While(b, body)$, $\{P, R\} \vdash S$ **sat** $(R, W^* \wedge P \wedge \neg b)$, and suitable $\sigma \in \Sigma$, providing $body$ reduces to **nil** and W is a well-founded order over Σ, then S reduces to **nil**.

Theorem 24 For any $S \in Stmt$ such that $\{P, R\} \vdash S$ **sat** (G, Q) and suitable $\sigma \in \Sigma$, then the termination predicate $reaches((S, \sigma), Stmt\text{-}Term, \xrightarrow[R]{r})$ holds.