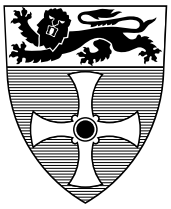
UNIVERSITY OF
NEWCASTLE



University of Newcastle upon Tyne

COMPUTING SCIENCE

A structural proof of the soundness of rely/guarantee rules

J. W. Coleman, C. B. Jones.

TECHNICAL REPORT SERIES

No. CS-TR-987 October, 2006

TECHNICAL REPORT SERIES

No. CS-TR-987 October, 2006

A structural proof of the soundness of rely/guarantee rules

Joey W. Coleman, Cliff B. Jones.

Abstract

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

Printed and published by the University of Newcastle upon Tyne,
Computing Science, Claremont Tower, Claremont Road,
Newcastle upon Tyne, NE1 7RU, England.

Bibliographical details

COLEMAN, J. W., JONES, C. B..

A structural proof of the soundness of rely/guarantee rules
[By] J. W. Coleman and C. B. Jones

Newcastle upon Tyne: University of Newcastle upon Tyne: Computing Science, 2006.

(University of Newcastle upon Tyne, Computing Science, Technical Report Series, No. CS-TR-987)

Added entries

UNIVERSITY OF NEWCASTLE UPON TYNE
Computing Science. Technical Report Series. CS-TR-987

Abstract

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

About the author

Joey earned a BSc (2001) in Applied Computer Science at Ryerson University in Toronto, Ontario. With that in hand he stayed on as a systems analyst in Ryerson's network services group. Following that he took a position at a post-dot.com startup as a software engineer and systems administrator. Having decided that research was likely more interesting than what he had been doing, Joey moved to Newcastle and earned a MPhil (2005) in Computing Science, and is currently working part-time on his PhD. He is involved primarily with the EPSRC "Splitting (Software) Atoms Safely" project, working on atomicity in software development methods. He is also involved in the RODIN project, working on methodology. Other associations include the DIRC project. His main interests lie in language design and semantics.

Cliff Jones is currently Professor of Computing Science and Project of the IRC on “Dependability of Computer-Based Systems”. He has spent more of his career in industry than academia. Fifteen years in IBM saw among other things the creation with colleagues in Vienna of VDM. Cliff is a fellow of the BCS, IEE and ACM. He Received a (late) Doctorate under Tony Hoare in Oxford in 1981 and immediately moved to a chair at Manchester University where he built a strong Formal Methods group which among other projects was the academic partner in the largest Alvey Software Engineering project (IPSE 2.5 created the "mural" theorem proving assistant). During his time at Manchester, Cliff had an SRC 5-year Senior Fellowship and spent a sabbatical at Cambridge with the Newton Institute event on "Semantics". Much of his research at this time focused on formal (compositional) development methods for concurrent systems. In 1996 he moved to Harlequin directing some 50 developers on Information Management projects and finally became overall Technical Director before leaving to re-join academia in 1999. Cliff's interests in formal methods have now broadened to reflect wider issues of dependability.

Suggested keywords

RELY/GUARANTEE,
SOUNDNESS,
STRUCTURAL OPERATIONAL SEMANTICS

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 and guarantee conditions have been used to record and reason about interference in ways which do indeed provide compositional development methods for such programs. This paper presents a new approach to justifying the soundness of rely/guarantee inference rules. The underlying concurrent language is defined by an operational semantics which allows fine-grained interleaving and nested concurrency; the proof that the rely/guarantee rules are consistent with that semantics (including termination) is by a structural induction. A lemma which relates the states which can arise from the extra interference that results from taking a portion of the program out of context is key to our ability to do the proof without having to perform induction over the computation history. This lemma also offers a way to understand some elusive expressibility issues around 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 [Apt81] "axiomatic semantics" for simple languages. Moreover, because the rules are "compositional" (see [Jon03a]), they can be used in the design process rather than just in post-facto proofs. Even for sequential programs, the rules used in the literature on VDM 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 recognizing the problems which result from undefined expressions. Section 2.2 expands on both of these points because they permeate the subsequent research on concurrency.

Finding compositional proof rules for concurrent programs proved more 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 it fails their final *Einmischungsfrei* proof obligation. Rely and guarantee conditions offer a way of documenting and reasoning about "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 characterized 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 is a lot written about rely/guarantee conditions,[1] but there is no convenient short summary (the encyclopedic [dR01] is neither short nor easy reading). It would thus seem useful to provide a reference point for the rules and methods we use. This is particularly timely because we are looking at new forms of "interference reasoning" in connection with "deriving specifications" (see for example [HJJ03,JHJ06]) and using rely/guarantee conditions in connection with our research on "splitting (software) atoms safely". Having decided to undertake this task, we believe that we have come up with a novel approach to the soundness proof.

The current paper provides an example of one particular set of rely/guarantee rules; an underlying operational semantics for the concurrent language; and a justification of the former with respect to the latter. The view here follows that of Tom Melham [CM92] and Tobias Nipkow [KNvO$^+$02]: the rules of an operational semantics – we use structural operational semantics (SOS) specifically– can be taken to provide an inductive definition of a relation ($\xrightarrow{s}$) over "configurations" (i.e. pairs of program texts and states). Results about programs could be

* A version of this paper without the termination proofs has been submitted to the Journal of Logic and Computation; please cite that version rather than this Technical Report.

[1] An annotated list of publications on rely/guarantee concepts can be found at http://homepages.cs.ncl.ac.uk/cliff.jones/home.formal/

proved *directly* in terms of this inference system. One can for example try to prove for some specific program S, pre condition P and post condition Q, that if P is true for some state, σ, and $(S, \sigma) \xrightarrow{s} (\mathbf{nil}, \sigma')$ where σ' is a final resulting state, then Q is true of the pair of states (σ, σ'). A proof of this sort can be presented in a natural deduction style (see Section 4 for more details) where, in addition to justifying steps using rules for logical operators, we also have the ability to justify steps by taking instances of SOS rules. A partial example might look like:

from $\cdots$

$\vdots$

α $\quad (mk\text{-}Assign(l, v), \sigma) \xrightarrow{s} (\mathbf{nil}, \sigma \dagger \{l \mapsto v\})$ Assn

$\vdots$

β $\quad (mk\text{-}If(\mathbf{true}, mk\text{-}Assign(l, v)), \sigma) \xrightarrow{s} (\mathbf{nil}, \sigma \dagger \{l \mapsto v\})$ If, α

$\vdots$

infer $\cdots$

We view the 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 with the longer proofs which might have been written directly in terms of that semantics). This 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.

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

1.1 Introducing rely/guarantee conditions

For the benefit of those unfamiliar with rely/guarantee concepts we provide a brief explanation. Program development using Floyd/Hoare 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 the state; and the execution of the program is represented in 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.

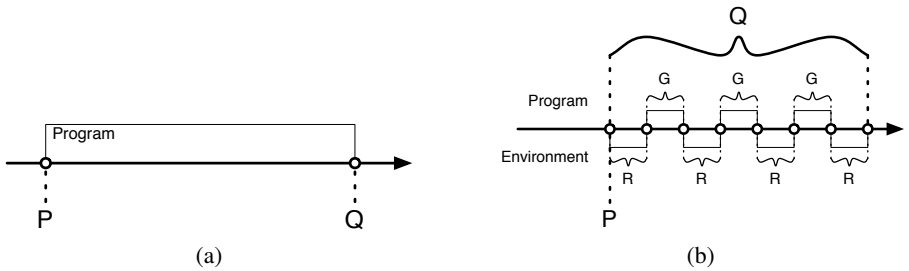


Fig. 1. (a) Pre/Post conditions and (b) Rely/Guarantee conditions

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 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.

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. 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 **true** to **false**) by one process or that a variable is unchanged when a flag is set. Another class of rely conditions which are useful in practice is that the value of a variable changes monotonically. Without locks, these latter conditions are intriguing and the example in Section 2 shows how careful choice of data representation can play a key part in such cases.

With Figure 1b in mind, then, the thrust of an R/G development lies in formalizing the behaviour of both the program and of its intended environment. Once the intended environment has been characterized in the rely condition, that condition can be used both in the proofs regarding the program and also by a 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.

There are of course decisions to be made in Floyd/Hoare 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) but such decisions are more numerous when there are four clauses to a specification. There are in fact several forms of rely/guarantee condition even where the idea is applied to shared-variable concurrency. We assume here that both rely and guarantee conditions are reflexive (it is possible that the state does not change) and transitive (they can cover multiple consecutive steps).

We assume that the conditions of a rely/guarantee specification satisfy certain constraints. First, pre and post conditions are "stable" under the interference that is indicated by the rely condition, giving us the axiom:

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

Second, the rely condition must satisfy a pair of related axioms so that the post condition is unaffected by interference, thus:

$$\textbf{RQ-ident}\ \frac{}{\overleftarrow{P} \wedge R \,|\, Q \ \Rightarrow\ Q}$$

$$\textbf{QR-ident}\ \frac{}{Q \,|\, R \ \Rightarrow\ Q}$$

Of the pair immediately above, the **QR-ident** rule allows for interference after a post condition has been established. The **RQ-ident** rule allows for the inclusion of interference from before the starting state, so long as the interference satisfies the rely condition and the earliest state also satisfies the pre condition.

There are further decisions to be made about whether the predicates written in $If/While$ statements are stable under interference but 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 are a useful way of thinking about a whole range of issues.

2 An example development

For this paper, we use as an example a problem which originated in [Owi75] and was tackled by rely/guarantee reasoning in [Jon81].[3]

[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.

[3] 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.

2.1 The specification

We assume that we have a predicate over some arbitrary type, $pred\colon X \to \mathbb{B}$, that is expensive to evaluate. For reasons that anticipate concurrent versions of the program, $pred$ must be such that multiple instances can be executed in parallel without affecting one another; the mechanisms to ensure this are left unspecified for this paper.

The task is to find the least index i (to the vector v) such that $pred(v(i))$. A specification is presented in Figure 2 using the normal 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.

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 2.2). There is no constraint on the internal behaviour of this program from the guarantee condition. 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 $v(i)$; alternately, if there are no values in v for which $pred$ holds, then r will be precisely one greater than the length of v.[4] The final conjunct requires that the least such r is found. Notice the type of $r\colon \mathbb{N}_1$: 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 $v\colon X^*$
wr $r\colon \mathbb{N}_1$
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. 2. Specification of $FINDP$

2.2 Sequential aside

Apart from the rely/guarantee idea itself, there are aspects of VDM even as applied to sequential programs that do not fit with the "main stream" 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 pre-dates the christening of VDM and was made widely visible in [Jon80]. The form 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 characterization of the first attempt to give rules in [Jon80] as "unmemorable").

The other unusual feature of the VDM research 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 2. 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

[4] Of course, 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.

"variant function" [DS90].[5] (We mark the rules for sequential constructs with a prefix **sim-** (the parallel rules below have no prefix).) This might suggest a rule like:

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

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 2 only guarantees the $v(i)$ (and thus $pred(v(i))$) are defined for $i \in \{1..\textbf{len}\ v\}$, 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] in which everything would be safe here: the 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 $\neg$-$\vee$-I 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.[6] Thus the rule for 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 (being cautious about undefinedness) 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

Whose justification 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:

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

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:

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

[5] 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".

[6] In the language definition in Appendix A, 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 notion 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 in employing the rules during a (formal) development process. Proof rules for assignment statements are therefore of less interest than those for the combinators which combine 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.

2.3 Introducing parallelism

Rather than the sequential algorithm of Section 2.2, we actually have in mind a development (from the original specification in Figure 2) which uses parallel tasks to check subsets of the indices of v. It is possible to have such processes work entirely independently on a partition of the indices and, after their completion, 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 taking longer than the sequential solution (consider two processes, if there is only one index where $pred(v(i))$, 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.[7]

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 to copy the value of t into r.[8]

$t \leftarrow$ **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** of Appendix B. Here satisfaction of a specification is written $\{P, R\} \vdash S$ **sat** (G, Q). This form of writing satisfaction nicely separates the developer's assumptions from the constraints on the execution of the program. 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.

[7] This is actually the only interesting interference in FINDP; there is more in the parallel prime sieve development pointed to in Footnote 3.

[8] At this stage of development, we might be tempted to use r directly but t will actually be reified into an expression in Section 2.4. 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.

The most interesting of the rules in Appendix B is that 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] generalized to an arbitrary partition of the indices over n processes). So $SEARCHES$ could be implemented by

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

where $SEARCH$ can be specified 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 $\forall i \in ms \cdot i < t \Rightarrow \neg\, pred(v(i))$

(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.)

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 2.2 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 $\overleftarrow{t} \leq t$ safe. Ensuring $guar\text{-}SEARCHES$ is trivial because it is identically **true**. The combinations 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 $post\text{-}SEARCHES$.[9]

2.4 Decomposing $SEARCH$ and reifying t

There is however a problem hidden in the closing paragraph of the preceding section: updating the variable t which is shared between the two instances of $SEARCH$ and could lead to a race condition. An assignment such as $< t \leftarrow min(t, \cdots) >$ would need to be made "atomic" since the language of Appendix A permits interference during expression evaluation.[10] As pointed out in [Jon05], a useful strategy to avoid such problems is by choosing 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 Appendix A, 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 2.2). To illustrate these two possibilities, we choose the latter course for the while rule (i.e. **While-I** in Appendix B 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 $body$.

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

[10] 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 Appendix A; the use of the **if** construct avoids the need for min.

```
ot ← len v + 1; et ← len v + 1;
par
|| (oc ← 1;
   while (oc < ot ∧ oc < et) do
      if oc < ot ∧ pred(v(oc)) then ot ← oc fi;
      oc ← oc + 2
   od)
|| (ec ← 2;
   while (ec < et ∧ ec < ot) do
      if ec < et ∧ pred(v(ec)) then et ← ec fi;
      ec ← ec + 2
   od)
rap ;
if ot < et then r ← ot fi; if et < ot then r ← et fi
```

Fig. 3. Possible implementation of FINDP

There are several interesting observations about the possible implementation of FINDP as shown in Figure 3. 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[11] (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 2.3, 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 will end up evaluating it — that is, how often we have to execute the loop body of either $SEARCH$. 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. Most of the time it will not happen, but it can if the ot and et variables in the min expression are read just before being updated by the other parallel branch.

3 Semantics

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][12]. We of course avoid the Baroque excesses caused by using a "Grand State". We view the rules of the semantics as (inductively) defining a relation over configurations of program texts and states. It is necessary to go beyond functions over configurations and use *relations* in order to model non-determinism in general and concurrency in particular. As is made clear in the next section, we take this semantics as our only knowledge of the programming language.

The language that we are using has been kept deliberately small. It is defined in Appendix A. This description follows the "VDM tradition" of basing the semantics on an *abstract* syntax and restricting the class of programs in this syntax further using "context conditions".

The main semantic relation of the language was chosen to be over pairs of configurations. This symmetry between the type of the domain and range of the relation allows us to directly take the transitive closure of the semantic relation, giving us a convenient mechanism to talk about two configurations related by many steps of the semantics. The main semantic relation is denoted as $\xrightarrow{s}$ and its transitive closure as $\xrightarrow{s}*$.

[11] This is occasionally referred to as "Reynold's rule" but see (9.32) and page 327 of [Sch97] for a more accurate attribution.
[12] Republished as [Plo04b] — see also [Plo04a,Jon03b].

The language includes five statement constructs and **nil** to represent a completed statement, as well as a subsidiary expression construct. Assignment (to a scalar variable) is represented by the $Assign$ construct and is the only means to alter the state. Here, the only step of the Assignment which is atomic is the actual mutation of the state object. Expression evaluation is *non-atomic*; interference makes it possible that $(x+x) \neq 2x$. This design decision allows parallel statements to interfere directly with each other. Not surprisingly, this 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. 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 our example in Section 2) so the usual "else" branch has been omitted from the language.

Repetition is achieved with the $While$ construct, but 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 $While$.

The Seq construct provides sequential execution and its structure mimics that of a LISP-style cons-cell. The SOS rules steps through the sl field first and, when that is reduced to **nil**, sr is unwrapped. This behaviour means that any structure composed of Seq objects will be evaluated in order from left to right.

The Par construct offers interleaved parallel execution of two statements. The SOS rules have no inherent notion of fairness — the choice of which branch to follow is unspecified. The parallel execution of more than two statements can be achieved by nesting Par constructs and Par can be arbitrarily nested within other statements. This again somewhat complicates the proof of soundness of the rules of Appendix B but it is a useful freedom for writing realistic concurrent programs.

The **nil** statement acts as a *skip* or empty statement in this language, indicating that there is no computation to perform. At first glance it may seem that a simple self-assignment such as $x \leftarrow x$ would be equivalent to a statement which does nothing, but that overlooks the nature of the interference which this language allows. A self-assignment is not equivalent to doing nothing as it is possible that the value of a variable is changed by a parallel activity between x being read and written. The addition of **nil** to the $Stmt$ type has the nice 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.[13] 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 of Appendix A. 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.

The subsidiary type, $Expr$, is used by the $Assign$, If and $While$ constructs. It has its own semantic relation, $\xrightarrow{e}$, which models the process of expression evaluation. 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 any 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 later in this paper. The language has no notion of function or procedure calls and this lack is part of what keeps expressions side-effect free: adding procedure calls into expressions would make preserving this property difficult.

The language 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 in the state object at the start of execution. The state object in the language maps all variables to integer values, and extending this to include arrays is straightforward. To actually implement our example program this would be required; we have, however, omitted this as it adds nothing but complexity to the proofs.

4 Soundness

It is straightforward to write a consistency proof for a sequential (non-concurrent) language: early citations are [Lau71,Don76], and [Jon87] provides a soundness proof of the sequential rules of VDM based on a deno-

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

tational 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. The basic approach –like that in [KNvO$^+$02]– is to view the SOS rules as inference rules in an extended logic which makes it possible to prove results about about the relation $\xrightarrow{s}$. As indicated in Section 1, proofs of specific programs can be constructed directly in terms of the SOS; but –as shown in Section 2– it is convenient to use additional rules for reasoning about programs and we need to show that the rules in Appendix B are sound. The operational semantics is our starting point: anything additional that we wish to use in reasoning about programs has to be proven to be sound with respect to Appendix A.

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 proof for the concurrent language is outlined in Sections 4.1–4.3).

The overall soundness result for a sequential language (assume $Par \notin Stmt$ in Appendix A) would be that, under the assumption that we have a proof using inference rules in Appendix B (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 $\xrightarrow{s}*$ relation leads to $(\mathbf{nil}, \sigma')$ in a finite number of transitions); and (b) any state σ' for which $(S, \sigma) \xrightarrow{s}* (\mathbf{nil}, \sigma')$ will be such that $[\![Q]\!](\sigma, \sigma')$.

Soundness for assignment statements must be argued directly in terms of the underlying SOS rules for each assignment individually. These proofs are relatively direct for a specific program, however, as they involve only the particular assignment (relative to its associated rely condition).

The proofs for (a) and (b) above can be done by structural induction over the abstract syntax for $Stmt$ (see **Stmt-Indn** in Appendix C.2), with the assumption that all of the assignments have already been proven directly. These proofs are presented in a natural deduction format similar to that used in [Jon90,JJLM91,FL98]; 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.)

Turning now to the actual proofs in hand, the termination proofs need 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, for correctness, 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 the termination proof is done mostly by structural induction. The exception to this is the lemma for $While$: this is, in a technical sense, the most interesting lemma as it requires the use of complete induction[14] over well-founded relation from **sim-While-I** in Section 2.2 (in addition to structural induction on the body of the $While$). We identify this transitive well-founded relation as $W \in \mathcal{P}(S \times S)$, where $S = (\mathbf{dom}\ W \cup \mathbf{rng}\ W) \wedge S \subseteq \Sigma$. The **sim-While-I** rule is also written such that all states that satisfy the rule's pre condition, P, must be contained within either the domain or range of W (i.e. $P \subseteq S$). 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 2.2 that these proofs can be done by structural induction. Retaining this property in the concurrent language was one of our goals for the following proofs.

Even in the case of concurrency, we assume that whole programs are run without interference, so the final result (Corollary 27) we need is that, when a program S has been shown to satisfy a pre/post condition (P, Q respectively) specification — that is

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

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

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

has been proved from the rules in Appendix B, 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 as the the transition relation $\xrightarrow[R]{r}$ in Appendix C.1, where R is a relation which constrains interference. This new relation essentially introduces zero or more steps of interference between "ordinary" steps of the $\xrightarrow{s}$ transition.

Corollary 27 is an immediate consequence of Theorem 26 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 will need to be able to conclude that the pre condition of the right part of a sequence is satisfied.

4.1 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 could be read as "given P and R, the program S will execute in the model such that its actions are within the bounds given by G". 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:

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

Though obvious, this axiom 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 Appendix A). 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 Appendix A). 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 B 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.

4.2 Correctness

The key correctness proofs (Lemmas 14–17 and Theorem 18) show that, under appropriate assumptions, the final states will satisfy Q. These proofs are given in detail in Appendix D. Thanks to Lemmas 6 and 7, most of them are only slightly more complex than in the sequential case. Since it is impractical to write the whole structural induction (over $Stmt$) as one proof, each of the Lemmas 14–17 list the induction hypothesis as a hypothesis; this is then expanded into a **from/infer** box because the inductive claim is not a simple predicate calculus expression.

It is important to realize the role of rely/guarantee conditions in these proofs. 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 B: at first sight, it may come as a shock to some[15] 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 2.4 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).

Lemma 16 uses complete induction over the W relation but there is a slight inconvenience (compare 7 with 8.2.2) in that one cannot assume that the W relation has its minimum element exactly when b is false; b could become false earlier (but not later). The trade-off of having to repeat part of the proof appears to us to be preferable to requiring the user of the While inference rule to achieve exact coincidence.

There is also a contortion in Lemma 16 due to the specific form of the H predicate that is used for the complete induction over the W relation. The predicate is actually an inference of the same form of the overall lemma, except that it is parameterized on the initial state. In the deduction this must be expanded as a **from/infer** box as it is not a simple predicate calculus expression. Circularity in the deduction — as the predicate almost assumes what we are trying to prove — is avoided as the state in which it is instantiated is that of the "next" iteration of the loop.

Also of interest is the Q proof is for Par (see Lemma 17). It is precisely here that the fact that the developer of a program using the R/G rules in Appendix B 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.

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.

4.3 Termination

The rules in Appendix B are intended to prove what is often called "total correctness": a correct program must always terminate if it is used as intended.[16] 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 Appendix A 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.

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

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

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 B, 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 B. Thus, it is the lemma about the $While$ construct which is most interesting (in exactly the same way as with the sequential language): Lemma 24 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 formalizing 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 (Theorem 25) just appeals to the lexicographic ordering of program texts for the statements other than $While$ and appeals to Lemma 16 for $While$.

4.4 Final theorem

Theorem 26 When

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

has been proved from the rules in Appendix B 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 18 and 25.

Corollary 27 If

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

has been proved from the rules in Appendix B 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 26.

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 to explore to what extent they come about because of the constraints of a complete machine-checked proof. Indeed, one possible further avenue for this research is to engineer a machine checked proof of our reasoning.

The most striking difference in the choices of language semantics is that we allow a much finer level of interference (indeed, to a non-HOL user, the embedding of whole statements as functions in the program and the way predicates are tested in [Pre01,Pre03] is surprising). Ours was not an arbitrary decision — we have argued elsewhere [Jon05] that assuming large atomic steps would make languages very expensive to implement. Other differences include the fact that we allow nested parallel statements and [Pre03] allows "await" statements.

The above 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). Another major difference with what is presented here is the fact that [Pre03] does not tackle termination (only addresses so-called "partial correctness"). 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. The source of the idea to use post conditions of single states would appear to be [Sti86] (who even uses predicates of single states for rely and guarantee conditions). This idea is not in the spirit of [Jon81,Jon83] 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].

A comparison with Ketil Stølen's PhD yields comments somewhat similar to those about Prensa Nieto's research. In[Stø90, Section 4.2.2] he makes clear that 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 but bases arguments on computation sequences (in fact, "potential computations"). Of course, Stølen is also facing the challenge of reasoning about his rules which have a wait condition to facilitate reasoning about progress; he also tackles the thorny issue of completeness but this interacts with the issue of "auxiliary variables" which is a topic discussed in Section 5.2 below.

5.2 Further work

There is as always much more to be done.

It was in fact difficulties with the heaviness of proofs using rely/guarantee conditions which led one of us to embark on constraining interference by using concurrent object based languages; this development is summarized 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 these proof have 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][17] 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.

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

As indicated, we are also interested in considering the requirements of machine checked proofs. In doing this, it would be worth examining again the 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.

Acknowledgments The technical content of this paper has benefited from discussions with Jon Burton, 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).

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.

[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.

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

[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.

[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.

[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.

[JHJ06] Cliff Jones, Ian Hayes, and Michael Jackson. Specifying systems that connect to the physical world. *Acta Informatica*, 2006. 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.

[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.

[Jon83] C. B. Jones. Specification and design of (parallel) programs. In *Proceedings of IFIP'83*, pages 321–332. North-Holland, 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.

[Jon05] C. B. Jones. An approach to splitting atoms safely. *Electronic Notes in Theoretical Computer Science, MFPS XXI, 21st Annual Conference of Mathematical Foundations of Programming Semantics*, pages 35–52, 2005.

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

[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. 75-251.

[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.

A The Base Language

A.1 Abstract Syntax

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

$Assign :: id : Id$
$\quad\quad\quad\quad e : Expr$

$Seq :: sl : Stmt$
$\quad\quad\quad sr : Stmt$

$If :: b : Expr$
$\quad\quad body : Stmt$

$While :: b : Expr$
$\quad\quad\quad body : Stmt$

$Par :: sl : Stmt$
$\quad\quad\quad sr : Stmt$

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

$Dyad :: op : + \mid - \mid < \mid = \mid > \mid \wedge \mid \vee$
$\quad\quad\quad\quad a : Expr$
$\quad\quad\quad\quad b : Expr$

A.2 Context Conditions

Auxiliary functions

$typeof : (Expr \times Id\text{-}\textbf{set}) \to \{\text{INT}, \text{BOOL}\}$
$typeof(e, ids) \triangleq$
cases e **of**
$e \in \mathbb{B} \to$ BOOL
$e \in \mathbb{Z} \to$ INT
$e \in ids \to$ INT
others cases $e.op$ **of**
$+ \to$ INT
$- \to$ INT
others BOOL
end
end

Expressions

$wf\text{-}Expr\colon (Expr \times Id\text{-}\mathbf{set}) \to \mathbb{B}$
$wf\text{-}Expr(e, ids) \triangleq e \in (ids \cup \mathbb{B} \cup \mathbb{Z})$
$wf\text{-}Expr(mk\text{-}Dyad(op, a, b), ids) \triangleq$
$\quad wf\text{-}Expr(a, ids) \wedge wf\text{-}Expr(b, ids) \wedge$
$\quad typeof(a, ids) = typeof(b, ids) \wedge$
$\quad op \in \{+, -, <, >\} \;\Rightarrow\; typeof(a, ids) = \textsc{Int} \wedge$
$\quad op \in \{\wedge, \vee\} \;\Rightarrow\; typeof(a, ids) = \textsc{Bool}$

Statements

$wf\text{-}Stmt\colon (Stmt \times Id\text{-}\mathbf{set}) \to \mathbb{B}$
$wf\text{-}Stmt(\mathbf{nil}, ids) \triangleq \mathbf{true}$
$wf\text{-}Stmt(mk\text{-}Assign(id, e), ids) \triangleq id \in ids \wedge typeof(e, ids) = \textsc{Int} \wedge wf\text{-}Expr(e, ids)$
$wf\text{-}Stmt(mk\text{-}Seq(sl, sr), ids) \triangleq wf\text{-}Stmt(sl, ids) \wedge wf\text{-}Stmt(sr, ids)$
$wf\text{-}Stmt(mk\text{-}If(b, s), ids) \triangleq$
$\quad typeof(b, ids) = \textsc{Bool} \wedge wf\text{-}Expr(b, ids) \wedge wf\text{-}Stmt(s, ids)$
$wf\text{-}Stmt(mk\text{-}While(b, s), ids) \triangleq$
$\quad typeof(b, ids) = \textsc{Bool} \wedge wf\text{-}Expr(b, ids) \wedge wf\text{-}Stmt(s, ids)$
$wf\text{-}Stmt(mk\text{-}Par(sl, sr), ids) \triangleq wf\text{-}Stmt(sl, ids) \wedge wf\text{-}Stmt(sr, ids)$

A.3 Semantic Objects

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

A.4 Semantic Rules

Expressions

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

Identifiers

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

Dyads

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

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

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

Statements

$\xrightarrow{s}\colon \mathcal{P}\left((Stmt \times \Sigma) \times (Stmt \times \Sigma)\right)$

Assign

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

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

Sequence

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

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

If

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

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

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

While

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

Parallel

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

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

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

B Inference Rules

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

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

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

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

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

The hypothesis $bottoms(W, \neg b)$ is taken to mean that any state that is not in the domain of W will cause any evaluation of b under interference constrained by R to be **false**. 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.

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

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

C Proof preparation

C.1 Augmented Semantics

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

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

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

C.2 Structural Induction

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

C.3 Notation

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

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

D Formal proofs

D.1 Respecting Guarantee Conditions

Lemma 1

$$\textbf{Seq-B}\ \frac{\{P, R\} \vdash sl\ \textbf{sat}\ (G, Q_{sl} \wedge P_{sr}) \quad \{P_{sr}, R\} \models sr\ \textbf{within}\ G}{\{P, R\} \models \textit{mk-Seq}(sl, sr)\ \textbf{within}\ G}$$

Proof

from $\{P, R\} \vdash \textit{mk-Seq}(sl, sr)\ \textbf{sat}\ (G, Q);\ [\![P]\!](\sigma_0);\ (\textit{mk-Seq}(sl, sr), \sigma_0) \xrightarrow[R]{r}\!* (\textbf{nil}, \sigma_f);\ \textit{IH-S}(sl);\ \textit{IH-S}(sr)$

1 $\{P, R\} \vdash sl\ \textbf{sat}\ (G, Q_{sl} \wedge P_{sr})$ — h, **Seq-I**

2 $\{P_{sr}, R\} \vdash sr\ \textbf{sat}\ (G, Q_{sr})$ — h, **Seq-I**

3 **from** $\{P_l, R_l\} \vdash sl\ \textbf{sat}\ (G_l, Q_l);\ [\![P_l]\!](\sigma_l);\ (sl, \sigma_l) \xrightarrow[R_l]{r}\!* (\textbf{nil}, \sigma_l')$

infer $\{P_l, R_l\} \models sl\ \textbf{within}\ G_l$ — IH-S(sl)

4 **from** $\{P_r, R_r\} \vdash sr\ \textbf{sat}\ (G_r, Q_r);\ [\![P_r]\!](\sigma_r);\ (sr, \sigma_r) \xrightarrow[R_r]{r}\!* (\textbf{nil}, \sigma_r')$

infer $\{P_r, R_r\} \models sr\ \textbf{within}\ G_r$ — IH-S(sr)

5 $\exists \sigma_i \in \Sigma \cdot (\textit{mk-Seq}(sl, sr), \sigma_0) \xrightarrow[R]{r}\!* (\textit{mk-Seq}(\textbf{nil}, sr), \sigma_i) \xrightarrow[R]{r}\!* (\textbf{nil}, \sigma_f)$ — h, $\xrightarrow[R]{r}\!*$

6 **from** $\sigma_i \in \Sigma$ **st** $(\textit{mk-Seq}(sl, sr), \sigma_0) \xrightarrow[R]{r}\!* (\textit{mk-Seq}(\textbf{nil}, sr), \sigma_i) \xrightarrow[R]{r}\!* (\textbf{nil}, \sigma_f)$

6.1 $(sl, \sigma_0) \xrightarrow[R]{r}\!* (\textbf{nil}, \sigma_i)$ — h6, *Lemma 8*

6.2 $\{P, R\} \models sl\ \textbf{within}\ G$ — h, 1, 3, 6.1

6.3 $[\![P_{sr}]\!](\sigma_i)$ — 1, 6.1, *Theorem 18*

6.4 $(sr, \sigma_i) \xrightarrow[R]{r}\!* (\textbf{nil}, \sigma_f)$ — h6, *Lemma 9*

6.5 $\{P_{sr}, R\} \models sr\ \textbf{within}\ G$ — 2, 4, 6.3, 6.4

infer $\{P, R\} \models \textit{mk-Seq}(sl, sr)\ \textbf{within}\ G$ — h6, 6.2, 6.5

infer $\{P, R\} \models \textit{mk-Seq}(sl, sr)\ \textbf{within}\ G$ — $\exists$-E(5, 6)

Lemma 2

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

Proof

from $\{P, R\} \vdash \textit{mk-If}(b, body)\ \textbf{sat}\ (G, Q);\ [\![P]\!](\sigma_0);\ (\textit{mk-If}(b, body), \sigma_0) \xrightarrow[R]{r}\!* (\textbf{nil}, \sigma_f);\ \textit{IH-S}(body)$

1 $\{P \wedge b, R\} \vdash body\ \textbf{sat}\ (G, Q)$ — h, **If-I**

2 **from** $\{P_b, R_b\} \vdash body\ \textbf{sat}\ (G_b, Q_b);\ [\![P_b]\!](\sigma_b);\ (body, \sigma_b) \xrightarrow[R_b]{r}\!* (\textbf{nil}, \sigma_b')$

infer $\{P_b, R_b\} \models body\ \textbf{within}\ G_b$ — IH-S(body)

3 $\exists v \in \mathbb{B}, \sigma_i \in \Sigma \cdot (\textit{mk-If}(b, body), \sigma_0) \xrightarrow[R]{r}\!* (\textit{mk-If}(v, body), \sigma_i) \xrightarrow[R]{r}\!* (\textbf{nil}, \sigma_f)$ — h, $\xrightarrow[R]{r}$

4 **from** $v \in \mathbb{B}, \sigma_i \in \Sigma$ **st** $(\textit{mk-If}(b, body), \sigma_0) \xrightarrow[R]{r}\!* (\textit{mk-If}(v, body), \sigma_i) \xrightarrow[R]{r}\!* (\textbf{nil}, \sigma_f)$

4.1 **from** $\neg\, v$

4.1.1 $[\![R]\!](\sigma_0, \sigma_f)$ — h4, h4.1, $\xrightarrow[R]{r}$

infer $\{P, R\} \models \textit{mk-If}(b, body)\ \textbf{within}\ G$ — h, 4.1.1

4.2 **from** v

4.2.1 $[\![R]\!](\sigma_0, \sigma_i)$ — h4, h4.1, $\xrightarrow[R]{r}$

4.2.2 $(body, \sigma_i) \xrightarrow[R]{r}\!* (\textbf{nil}, \sigma_f)$ — h4, h4.2, *Lemma 10*

4.2.3 $[\![P]\!](\sigma_i)$ — h, 4.2.1, **PR-ident**

4.2.4 $\{P \wedge b, R\} \models body\ \textbf{within}\ G$ — 1, 2, 4.2.2, 4.2.3

infer $\{P, R\} \models \textit{mk-If}(b, body)\ \textbf{within}\ G$ — h4, 4.2.1, 4.2.4

infer $\{P, R\} \models \textit{mk-If}(b, body)\ \textbf{within}\ G$ — $\vee$-E(h4, 4.1, 4.2)

infer $\{P, R\} \models \textit{mk-If}(b, body)\ \textbf{within}\ G$ — $\exists$-E(3, 4)

Lemma 3

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

Proof

from $\{P, R\} \vdash mk\text{-}While(b, body)$ **sat** (G, Q); $[\![P]\!](\sigma_0)$; $(mk\text{-}While(b, body), \sigma_0) \xrightarrow[R]{r}\!* (\textbf{nil}, \sigma_f)$; $IH\text{-}S(body)$

1	$\{P, R\} \vdash body$ **sat** (G, Q)	h, **While-I**
2	**from** $\{P_b, R_b\} \vdash body$ **sat** (G_b, Q_b); $[\![P_b]\!](\sigma_b)$; $(body, \sigma_b) \xrightarrow[R_b]{r}\!* (\textbf{nil}, \sigma'_b)$	
	infer $\{P_b, R_b\} \models body$ **within** G_b	IH-S(body)
3	$\exists v \in \mathbb{B}, \sigma_i \in \Sigma \cdot \left(\begin{array}{l} (mk\text{-}While(b, body), \sigma_0) \xrightarrow[R]{r}\!* \\ (mk\text{-}If(v, mk\text{-}Seq(body, mk\text{-}While(b, body))), \sigma_i) \xrightarrow[R]{r}\!* (\textbf{nil}, \sigma_f) \end{array} \right)$	h, $\xrightarrow[R]{r}$
4	**from** $v \in \mathbb{B}, \sigma_i \in \Sigma$ **st** [[3]]	
4.1	**from** $\neg\, v$	
4.1.1	$[\![R]\!](\sigma_0, \sigma_f)$	h4, h4.1, $\xrightarrow[R]{r}$
	infer $\{P, R\} \models mk\text{-}While(b, body)$ **within** G	h, 4.1.1
4.2	**from** v	
4.2.1	$[\![R]\!](\sigma_0, \sigma_i)$	h4, h4.1, $\xrightarrow[R]{r}$
4.2.2	$(body, \sigma_i) \xrightarrow[R]{r}\!* (\textbf{nil}, \sigma_f)$	h4, h4.2, *Lemma 10, 8*
4.2.3	$[\![P]\!](\sigma_i)$	h, 4.2.1, **PR-ident**
4.2.4	$\{P, R\} \models body$ **within** G	1, 2, 4.2.2, 4.2.3
	infer $\{P, R\} \models mk\text{-}While(b, body)$ **within** G	h4, 4.2.1, 4.2.4
	infer $\{P, R\} \models mk\text{-}While(b, body)$ **within** G	$\vee$-E(h4, 4.1, 4.2)
	infer $R \models mk\text{-}While(b, body)$ **within** G	2, **While-B**

Lemma 4

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

Proof

from $\{P, R\} \vdash mk\text{-}Par(sl, sr)$ **sat** (G, Q); $[\![P]\!](\sigma_0)$; $(mk\text{-}Par(sl, sr), \sigma_0) \xrightarrow[R]{r}\!* (\textbf{nil}, \sigma_f)$; $IH\text{-}S(sl)$; $IH\text{-}S(sr)$

1	$\{P, R \vee G_{sr}\} \vdash sl$ **sat** (G_{sl}, Q_{sl})	h, **Par-I**
2	$\{P, R \vee G_{sl}\} \vdash sr$ **sat** (G_{sr}, Q_{sr})	h, **Par-I**
3	$G_{sl} \vee G_{sr}\ \Rightarrow\ G$	h, **Par-I**
4	**from** $\{P_l, R_l\} \vdash sl$ **sat** (G_l, Q_l); $[\![P_l]\!](\sigma_l)$; $(sl, \sigma_l) \xrightarrow[R_l]{r}\!* (\textbf{nil}, \sigma'_l)$	
	infer $\{P_l, R_l\} \models sl$ **within** G_l	IH-S(sl)
5	**from** $\{P_r, R_r\} \vdash sr$ **sat** (G_r, Q_r); $[\![P_r]\!](\sigma_r)$; $(sr, \sigma_r) \xrightarrow[R_r]{r}\!* (\textbf{nil}, \sigma'_r)$	
	infer $\{P_r, R_r\} \models sr$ **within** G_r	IH-S(sr)
6	$\exists \sigma_i \in \Sigma \cdot (mk\text{-}Par(sl, sr), \sigma_0) \xrightarrow[R]{r}\!* (mk\text{-}Par(\textbf{nil}, \textbf{nil}), \sigma_i) \xrightarrow[R]{r}\!* (\textbf{nil}, \sigma_f)$	h, $\xrightarrow[R]{r}\!*$
7	**from** $\sigma_i \in \Sigma$ **st** $(mk\text{-}Par(sl, sr), \sigma_0) \xrightarrow[R]{r}\!* (mk\text{-}Par(\textbf{nil}, \textbf{nil}), \sigma_i) \xrightarrow[R]{r}\!* (\textbf{nil}, \sigma_f)$	
7.1	$(sl, \sigma_0) \xrightarrow[R \vee G_{sr}]{r}\!* (\textbf{nil}, \sigma_i)$	h6, *Lemma 6*
7.2	$\{P, R \vee G_{sr}\} \models sl$ **within** G_{sl}	h, 1, 4, 7.1
7.3	$(sr, \sigma_0) \xrightarrow[R \vee G_{sl}]{r}\!* (\textbf{nil}, \sigma_i)$	h6, *Lemma 7*
7.4	$\{P, R \vee G_{sl}\} \models sr$ **within** G_{sr}	h, 2, 5, 7.1
7.5	$[\![R]\!](\sigma_i, \sigma_f)$	h7, $\xrightarrow[R]{r}$
	infer $\{P, R\} \models mk\text{-}Par(sl, sr)$ **within** G	3, h7, 7.2, 7.4, 7.5
	infer $\{P, R\} \models mk\text{-}Par(sl, sr)$ **within** G	$\exists$-E(5, 6)

Theorem 5 For any $S \in Stmt$ for which $\{P, R\} \vdash S$ **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

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

D.2 Monotonicity Under Interference

Lemma 6

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

Lemma 7

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

Lemma 8

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

Lemma 9

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

Lemma 10

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

D.3 Further lemmas on respects

Lemma 11

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

Lemma 12

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

D.4 Correctness Proofs

Prior to showing the soundness proofs relative to correctness, it would be well to explain an idiom that is commonly used throughout. This pattern shows up frequently:

from $(S, \sigma_0) \xrightarrow[R]{r}* (\textbf{nil}, \sigma_f)$

$\vdots$

α $\quad \exists S' \in Stmt, \sigma_i \in \Sigma \cdot (S, \sigma_0) \xrightarrow[R]{r}* (S', \sigma_i) \xrightarrow[R]{r}* (\textbf{nil}, \sigma_f)$ h, $\xrightarrow[R]{r}$

β $\quad$ **from** $S' \in Stmt, \sigma_i \in \Sigma$ **st** $(S, \sigma_0) \xrightarrow[R]{r}* (S', \sigma_i) \xrightarrow[R]{r}* (\textbf{nil}, \sigma_f)$

$\quad\quad \vdots$

$\quad\quad$ **infer** *some-deduction* contents of β

γ $\quad$ *some-deduction* $\exists$-$E(\alpha,\beta)$

$\vdots$

infer $\cdots$

The purpose of this pattern is to perform variable binding in a relational context. Line α asserts the existence of some intermediate configuration between the first and last given in the overall hypothesis (and includes the first and last as well), and the hypothesis of line β has the same form and effectively binds variables for the contents of that **from/infer** block. Line γ uses $\exists$-E to promote the result of line β so that it can be used elsewhere in the deduction.

Lemma 14 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 $[\![Q]\!](\sigma_0, \sigma_f)$

Proof

from $\{P, R\} \vdash mk\text{-}Seq(sl, sr)$ **sat** (G, Q); $[\![P]\!](\sigma_0)$; $(mk\text{-}Seq(sl, sr), \sigma_0) \xrightarrow[R]{r}* (\textbf{nil}, \sigma_f)$; $IH\text{-}S(sl)$; $IH\text{-}S(sr)$

1 $\quad \{P, R\} \vdash sl$ **sat** $(G, Q_{sl} \wedge P_{sr})$ h, **Seq-I**

2 $\quad \{P_{sr}, R\} \vdash sr$ **sat** (G, Q_{sr}) h, **Seq-I**

3 $\quad Q_{sl} \,|\, Q_{sr} \;\Rightarrow\; Q$ h, **Seq-I**

4 $\quad$ **from** $\{P_l, R_l\} \vdash sl$ **sat** (G_l, Q_l); $[\![P_l]\!](\sigma_l)$; $(sl, \sigma_l) \xrightarrow[R]{r}* (\textbf{nil}, \sigma_l')$

$\quad$ **infer** $[\![Q_l]\!](\sigma_l, \sigma_l')$ IH-S(sl)

5 $\quad$ **from** $\{P_r, R_r\} \vdash sr$ **sat** (G_r, Q_r); $[\![P_r]\!](\sigma_r)$; $(sr, \sigma_r) \xrightarrow[R]{r}* (\textbf{nil}, \sigma_r')$

$\quad$ **infer** $[\![Q_r]\!](\sigma_r, \sigma_r')$ IH-S(sr)

6 $\quad \exists \sigma_i \in \Sigma \cdot (mk\text{-}Seq(sl, sr), \sigma_0) \xrightarrow[R]{r}* (mk\text{-}Seq(\textbf{nil}, sr), \sigma_i) \xrightarrow[R]{r}* (\textbf{nil}, \sigma_f)$ h, $\xrightarrow[R]{r}$

7 $\quad$ **from** σ_i **st** $(mk\text{-}Seq(sl, sr), \sigma_0) \xrightarrow[R]{r}* (mk\text{-}Seq(\textbf{nil}, sr), \sigma_i) \xrightarrow[R]{r}* (\textbf{nil}, \sigma_f)$

7.1 $\quad\quad (sl, \sigma_0) \xrightarrow[R]{r}* (\textbf{nil}, \sigma_i)$ h7, *Lemma 8*

7.2 $\quad\quad [\![Q_{sl} \wedge P_{sr}]\!](\sigma_0, \sigma_i)$ h, 1, 4, 7.1

7.3 $\quad\quad [\![P_{sr}]\!](\sigma_i)$ 7.2

7.4 $\quad\quad (sr, \sigma_i) \xrightarrow[R]{r}* (\textbf{nil}, \sigma_0)$ h7, *Lemma 9*

7.5 $\quad\quad [\![Q_{sr}]\!](\sigma_i, \sigma_f)$ 2, 5, 7.3, 7.4

7.6 $\quad\quad [\![Q_{sl} \,|\, Q_{sr}]\!](\sigma_0, \sigma_f)$ 7.2, 7.5

$\quad$ **infer** $[\![Q]\!](\sigma_0, \sigma_f)$ 3, 7.6

infer $[\![Q]\!](\sigma_0, \sigma_f)$ $\exists$-E(7, 8)

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

Proof

from $\{P, R\} \vdash \mathit{mk\text{-}If}(b, \mathit{body})$ **sat** (G, Q); $[\![P]\!](\sigma_0)$; $(\mathit{mk\text{-}If}(b, \mathit{body}), \sigma_0) \xrightarrow[R]{r}* (\mathbf{nil}, \sigma_f)$; $\mathit{IH\text{-}S}(\mathit{body})$

1	b **indep** R	h, **If-I**
2	$\{P \wedge b, R\} \vdash \mathit{body}$ **sat** (G, Q)	h, **If-I**
3	$\overleftarrow{P} \wedge \overleftarrow{\neg\, b} \;\Rightarrow\; Q$	h, **If-I**
4	**from** $\{P_b, R_b\} \vdash \mathit{body}$ **sat** (G_b, Q_b); $[\![P_b]\!](\sigma_b)$; $(\mathit{body}, \sigma_b) \xrightarrow[R]{r}* (\mathbf{nil}, \sigma'_b)$	
	infer $[\![Q_b]\!](\sigma_b, \sigma'_b)$	IH-S(body)
5	$\exists v \in \mathbb{B}, \sigma_i \in \Sigma \cdot (\mathit{mk\text{-}If}(b, \mathit{body}), \sigma_0) \xrightarrow[R]{r}* (\mathit{mk\text{-}If}(v, \mathit{body}), \sigma_i) \xrightarrow[R]{r}* (\mathbf{nil}, \sigma_f)$	h, $\xrightarrow[R]{r}$
6	**from** $v \in \mathbb{B}, \sigma_i \in \Sigma$ **st** $(\mathit{mk\text{-}If}(b, \mathit{body}), \sigma_0) \xrightarrow[R]{r}* (\mathit{mk\text{-}If}(v, \mathit{body}), \sigma_i) \xrightarrow[R]{r}* (\mathbf{nil}, \sigma_f)$	
6.1	**from** $\neg\, v$	
6.1.1	$[\![R]\!](\sigma_0, \sigma_i)$	h6, h6.1, $\xrightarrow[R]{r}$
6.1.2	$[\![R]\!](\sigma_i, \sigma_f)$	h6, h6.1, $\xrightarrow[R]{r}$
6.1.3	$[\![R]\!](\sigma_0, \sigma_f)$	6.1.1, 6.1.2
6.1.4	$[\![\neg\, b]\!](\sigma_0)$	1, h6, h6.1, 6.1.3
6.1.5	$[\![\overleftarrow{\neg\, b}]\!](\sigma_0, \sigma_f)$	6.1.4
6.1.6	$[\![\overleftarrow{P} \wedge \overleftarrow{\neg\, b}]\!](\sigma_0, \sigma_f)$	h, 6.1.5
	infer $[\![Q]\!](\sigma_0, \sigma_f)$	3, 6.1.6
6.2	**from** v	
6.2.1	$[\![R]\!](\sigma_0, \sigma_i)$	h6, $\xrightarrow[R]{r}$
6.2.2	$[\![P]\!](\sigma_i)$	h, 6.2.1, **PR-ident**
6.2.3	$(\mathit{body}, \sigma_i) \xrightarrow[R]{r}* (\mathbf{nil}, \sigma_f)$	2, h6, h6.2, *Lemma 10*
6.2.4	$[\![Q]\!](\sigma_i, \sigma_f)$	2, 4, 6.2.2, 6.2.3
	infer $[\![Q]\!](\sigma_0, \sigma_f)$	h, 6.2.4, **RQ-ident**
	infer $[\![Q]\!](\sigma_0, \sigma_f)$	$\vee$-E(h6, 6.1, 6.2)
	infer $[\![Q]\!](\sigma_0, \sigma_f)$	$\exists$-E(5,6)

Lemma 16 Given $\{P, R\} \vdash \mathit{mk\text{-}While}(b, \mathit{body})$ **sat** (G, Q) and providing body behaves according to its specification, for all $\sigma_0, \sigma_f \in \Sigma$ such that $[\![P]\!](\sigma_0)$, $(\mathit{mk\text{-}While}(b, \mathit{body}), \sigma_0) \xrightarrow[R]{r}* (\mathbf{nil}, \sigma_f)$ it must follow that $[\![Q]\!](\sigma_0, \sigma_f)$.

Please note that in the following proof line 8.2.3.2.5 is the assumption of the complete induction step. As such, it should be written:

$$\left(\begin{array}{l} \{P, R\} \vdash \mathit{mk\text{-}While}(b, \mathit{body}) \textbf{ sat } (G,\ W^* \wedge P \wedge \neg\, b); \\ [\![P]\!](\sigma_2); \\ (\mathit{mk\text{-}While}(b, \mathit{body}), \sigma_2) \xrightarrow[R]{r}* (\mathbf{nil}, \sigma_f); \\ \mathit{IH\text{-}S}(\mathit{body}) \end{array}\right) \textbf{ gives } [\![Q]\!](\sigma_2, \sigma_f)$$

but space constraints have forced us to use the shorthand that is present instead. Nonetheless, this forms the H predicate in the *W-Indn* rule, and expands to a **from/infer** box as is done on line 8.2.3.2.5.1.

Proof

from $\left(\begin{array}{l}\{P, R\} \vdash \mathit{mk\text{-}While}(b, \mathit{body})\ \textbf{sat}\ (G,\ W^* \wedge P \wedge \neg\, b);\ [\![P]\!](\sigma_0);\\ g(\mathit{mk\text{-}While}(b, \mathit{body}), \sigma_0) \xrightarrow[R]{r}\!* (\textbf{nil}, \sigma_f);\ \mathit{IH\text{-}S}(\mathit{body})\end{array}\right)$

1	$\mathit{bottoms}(W, \neg\, b)$	h, **While-I**
2	$\mathit{twf}(W)$	h, **While-I**
3	$\{P, R\} \vdash \mathit{body}\ \textbf{sat}\ (G,\ W \wedge P)$	h, **While-I**
4	$\overleftarrow{\neg\, b} \wedge R\ \Rightarrow\ \neg\, b$	h, **While-I**
5	$R\ \Rightarrow\ W^*$	h, **While-I**
6	**from** $\{P_b, R_b\} \vdash \mathit{body}\ \textbf{sat}\ (G_b, Q_b);\ [\![P_b]\!](\sigma_b);\ (\mathit{body}, \sigma_b) \xrightarrow[R_b]{r}\!* (\textbf{nil}, \sigma'_b)$	
	infer $[\![Q_b]\!](\sigma_b, \sigma'_b)$	IH-S(body)
7	**from** $\sigma_0 \notin \textbf{dom}\ W$	
7.1	$[\![\neg\, b]\!](\sigma_0)$	h, 1, h7
7.2	$[\![R]\!](\sigma_0, \sigma_f)$	4, 7.1, $\xrightarrow[R]{r}$
7.3	$[\![\neg\, b]\!](\sigma_0, \sigma_f)$	4, 7.1, 7.2
7.4	$[\![W^*]\!](\sigma_0, \sigma_f)$	5, 7.2
7.5	$[\![P]\!](\sigma_0, \sigma_f)$	h, 7.2, **PR-ident**
	infer $[\![W^* \wedge P \wedge \neg\, b]\!](\sigma_0, \sigma_f)$	7.3, 7.4, 7.5
8	**from** $\sigma_0 \in \textbf{dom}\ W$	
8.1	$\exists v \in \mathbb{B}, \sigma_1 \in \Sigma \cdot \left(\begin{array}{l}(\mathit{mk\text{-}While}(b, \mathit{body}), \sigma_0) \xrightarrow[R]{r}\\ (\mathit{mk\text{-}If}(b, \mathit{mk\text{-}Seq}(\mathit{body}, \mathit{mk\text{-}While}(b, \mathit{body}))), \sigma_0) \xrightarrow[R]{r}\!*\\ (\mathit{mk\text{-}If}(v, \mathit{mk\text{-}Seq}(\mathit{body}, \mathit{mk\text{-}While}(b, \mathit{body}))), \sigma_1) \xrightarrow[R]{r}\!*\\ (\textbf{nil}, \sigma_f)\end{array}\right)$	h, $\xrightarrow[R]{r}$
8.2	**from** $v \in \mathbb{B}, \sigma_1 \in \Sigma$ **st** [[8.1]]	
8.2.1	$[\![R]\!](\sigma_0, \sigma_1)$	h8.2, $\xrightarrow[R]{r}$
8.2.2	**from** $\neg\, v$	
8.2.2.1	$[\![R]\!](\sigma_1, \sigma_f)$	4, h8.2.2, $\xrightarrow[R]{r}$
8.2.2.2	$[\![R]\!](\sigma_0, \sigma_f)$	8.2.1, 8.2.2.1
8.2.2.3	$[\![W^*]\!](\sigma_0, \sigma_f)$	5, 8.2.2.2
8.2.2.4	$[\![P]\!](\sigma_0, \sigma_f)$	h, 8.2.2.2, **PR-ident**
8.2.2.5	$[\![\neg\, b]\!](\sigma_f)$	4, 8.2.2.1
	infer $[\![W^* \wedge P \wedge \neg\, b]\!](\sigma_0, \sigma_f)$	8.2.2.3, 8.2.2.4, 8.2.2.5
8.2.3	**from** v	
8.2.3.1	$\exists \sigma_2 \in \Sigma \cdot \left(\begin{array}{l}(\mathit{mk\text{-}If}(v, \mathit{mk\text{-}Seq}(\mathit{body}, \mathit{mk\text{-}While}(b, \mathit{body}))), \sigma_1)\\ \xrightarrow[R]{r}\!* (\mathit{mk\text{-}While}(b, \mathit{body}), \sigma_2) \xrightarrow[R]{r}\!* (\textbf{nil}, \sigma_f)\end{array}\right)$	h8.2, h8.2.3, $\xrightarrow[R]{r}$
8.2.3.2	**from** $\sigma_2 \in \Sigma$ **st** [[8.2.3.1]]	
8.2.3.2.1	$(\mathit{body}, \sigma_1) \xrightarrow[R]{r}\!* (\textbf{nil}, \sigma_2)$	h8.2.3, h8.2.3.2, *Lemmas 10, 8, 9*
8.2.3.2.2	$[\![P]\!](\sigma_1)$	h, 8.2.1, **PR-ident**
8.2.3.2.3	$[\![W \wedge P]\!](\sigma_1, \sigma_2)$	3, 6, 8.2.3.2.1, 8.2.3.2.2
8.2.3.2.4	$[\![W \wedge P]\!](\sigma_0, \sigma_2)$	h, 8.2.1, 8.2.3.2.3, **RQ-ident**
8.2.3.2.5	**from** $[\![W]\!](\sigma_0, \sigma_2)\ \Rightarrow\ \left(h[\sigma_2/\sigma_0]\ \textbf{gives}\ [\![Q]\!](\sigma_2, \sigma_f)\right)$	
8.2.3.2.5.1	**from** $\{P, R\} \vdash \mathit{mk\text{-}While}(b, \mathit{body})\ \textbf{sat}\ (G,\ W^* \wedge P \wedge \neg\, b);\ [\![P]\!](\sigma_2);$ $(\mathit{mk\text{-}While}(b, \mathit{body}), \sigma_2) \xrightarrow[R]{r}\!* (\textbf{nil}, \sigma_f);\ \mathit{IH\text{-}S}(\mathit{body})$	
	infer $[\![W^* \wedge P \wedge \neg\, b]\!](\sigma_2, \sigma_f)$	8.2.3.2.4, h8.2.3.2.5
8.2.3.2.5.2	$[\![W^* \wedge P \wedge \neg\, b]\!](\sigma_2, \sigma_f)$	h, 8.2.3.2.4, h8.2.3.2, 8.2.3.2.5.1
	infer $[\![W^* \wedge P \wedge \neg\, b]\!](\sigma_0, \sigma_f)$	8.2.3.2.4, 8.2.3.2.5.2
	infer $[\![W^* \wedge P \wedge \neg\, b]\!](\sigma_0, \sigma_f)$	**W-Indn**(8.2.3.2.5)
	infer $[\![W^* \wedge P \wedge \neg\, b]\!](\sigma_0, \sigma_f)$	$\exists$-E(8.2.3.1, 8.2.3.2)
	infer $[\![W^* \wedge P \wedge \neg\, b]\!](\sigma_0, \sigma_f)$	$\vee$-E(h8.2, 8.2.2, 8.2.3)
	infer $[\![W^* \wedge P \wedge \neg\, b]\!](\sigma_0, \sigma_f)$	$\exists$-E(8.1, 8.2)
	infer $[\![W^* \wedge P \wedge \neg\, b]\!](\sigma_0, \sigma_f)$	$\vee$-E(7, 8)

Lemma 17 Given $\{P, R\} \vdash \mathit{mk\text{-}Par}(sl, sr)$ **sat** (G, Q) and providing sl, sr behave according to their specifications, for any $\sigma_0, \sigma_f \in \Sigma$ such that $[\![P]\!](\sigma_0)$, $(\mathit{mk\text{-}Par}(sl, sr), \sigma_0) \xrightarrow[R]{r}\!* (\textbf{nil}, \sigma_f)$ it must follow that $[\![Q]\!](\sigma_0, \sigma_f)$

Proof:

from $\{P, R\} \vdash \mathit{mk\text{-}Par}(sl, sr)$ **sat** (G, Q); $[\![P]\!](\sigma_0)$; $(\mathit{mk\text{-}Par}(sl, sr), \sigma_0) \xrightarrow[R]{r}\!* (\textbf{nil}, \sigma_f)$ $\mathit{IH\text{-}S}(sl)$; $\mathit{IH\text{-}S}(sr)$

1	$[\![\overleftarrow{P}]\!](\sigma_0, \sigma_f)$	h, definition of hook
2	$\{P, R \vee G_{sr}\} \vdash sl$ **sat** (G_{sl}, Q_{sl})	h, **Par-I**
3	$\{P, R \vee G_{sl}\} \vdash sr$ **sat** (G_{sr}, Q_{sr})	h, **Par-I**
4	$\overleftarrow{P} \wedge Q_l \wedge Q_r \wedge (R \vee G_{sl} \vee G_{sr})^* \;\Rightarrow\; Q$	h, **Par-I**
5	$\{P, R \vee G_{sr}\} \models sl$ **within** G_{sl}	2, *Theorem 5*
6	$\{P, R \vee G_{sl}\} \models sr$ **within** G_{sr}	3, *Theorem 5*
7	$\exists sr' \in \mathit{Stmt}, \sigma_i \in \Sigma \cdot (\mathit{mk\text{-}Par}(sl, sr), \sigma_0) \xrightarrow[R]{r}\!* (\mathit{mk\text{-}Par}(\textbf{nil}, sr'), \sigma_i) \xrightarrow[R]{r}\!* (\textbf{nil}, \sigma_f)$	h, $\xrightarrow[R]{r}$
8	**from** $sr' \in \mathit{Stmt}, \sigma_i \in \Sigma$ **st** $(\mathit{mk\text{-}Par}(sl, sr), \sigma_0) \xrightarrow[R]{r}\!* (\mathit{mk\text{-}Par}(\textbf{nil}, sr'), \sigma_i) \xrightarrow[R]{r}\!* (\textbf{nil}, \sigma_f)$	
8.1	$(sl, \sigma_0) \xrightarrow[R \vee G_{sr}]{r}\!* (\textbf{nil}, \sigma_i)$	6, h8, *Lemma 6*
8.2	**from** $\{P_l, R_l\} \vdash sl$ **sat** (G_l, Q_l); $[\![P_l]\!](\sigma_l)$; $(sl, \sigma_l) \xrightarrow[R_l]{r}\!* (\textbf{nil}, \sigma'_l)$	
	infer $[\![Q_l]\!](\sigma_l, \sigma'_l)$	IH-S(sl)
8.3	$[\![Q_{sl}]\!](\sigma_0, \sigma_i)$	h, 2, 8.1, 8.2
8.4	$(\textbf{nil}, \sigma_i) \xrightarrow[R \vee G_{sr}]{r}\!* (\textbf{nil}, \sigma_f)$	6, h8, *Lemma 6*
8.5	$[\![(R \vee G_{sr})^*]\!](\sigma_i, \sigma_f)$	8.4, **nil-within**, $\xrightarrow[R \vee G_{sr}]{r}$
	infer $[\![Q_{sl}]\!](\sigma_0, \sigma_f)$	2, 8.3, 8.5, **QR-ident**
9	$[\![Q_{sl}]\!](\sigma_0, \sigma_f)$	$\exists$-E(7, 8)
10	$[\![Q_{sr}]\!](\sigma_0, \sigma_f)$	a symmetrical argument to 7–9 about sr
11	$\exists \sigma_i \in \Sigma \cdot (\mathit{mk\text{-}Par}(sl, sr), \sigma_0) \xrightarrow[R]{r}\!* (\mathit{mk\text{-}Par}(\textbf{nil}, \textbf{nil}), \sigma_i) \xrightarrow[R]{r}\!* (\textbf{nil}, \sigma_f)$	h, $\xrightarrow[R]{r}$
12	**from** $\sigma_i \in \Sigma$ **st** $(\mathit{mk\text{-}Par}(sl, sr), \sigma_0) \xrightarrow[R]{r}\!* (\mathit{mk\text{-}Par}(\textbf{nil}, \textbf{nil}), \sigma_i) \xrightarrow[R]{r}\!* (\textbf{nil}, \sigma_f)$	
12.1	$(sl, \sigma_0) \xrightarrow[R \vee G_{sr}]{r}\!* (\textbf{nil}, \sigma_i)$	6, h8, *Lemma 6*
12.2	$[\![(R \vee G_{sl} \vee G_{sr})^*]\!](\sigma_0, \sigma_i)$	h, 2, 12.1, *Lemma 11*
12.3	$[\![R]\!](\sigma_i, \sigma_f)$	h12, $\xrightarrow[R]{r}$
	infer $[\![(R \vee G_{sl} \vee G_{sr})^*]\!](\sigma_0, \sigma_f)$	12.2, 12.3
13	$[\![(R \vee G_{sl} \vee G_{sr})^*]\!](\sigma_0, \sigma_f)$	5 or 6, *Lemma 11*
infer $[\![Q]\!](\sigma_0, \sigma_f)$		1, 4, 9, 10, 13

Theorem 18 For any $st \in \mathit{Stmt}$ 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 $[\![Q]\!](\sigma, \sigma')$.
Proof Straightforward structural induction using Lemmas 14–17.

D.5 Termination Proofs

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

1. (expression types)

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

2. (internal expressions)

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

3. (**nil** at bottom)

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

4. (expression evaluation)

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

5. (internal statements)

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

6. (containment)

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

7. (transitivity)

$$\exists S'' \in \mathit{Stmt} \cdot S' < S'' \land S'' < S \ \Rightarrow \ S' < S$$

Lemma 19 The evaluation of an expression always reduces –over repeated steps– to a value.
Proof Done in cases by the SOS rules:
1. Id-E reduces an identifier in Id to a value in $\mathbb{Z}$.
2. Dyad-L reduces the left parameter of the $Dyad$, induction brings us to a value.
3. Dyad-R is symmetrical to Dyad-L.
4. Dyad-E reduces a $Dyad$ containing only values to a value.

Lemma 20 The execution of any $s \in Assign$ always reduces –over repeated steps– to the statement **nil**.
Proof Done in cases by the SOS rules:
1. Assign-Eval reduces the expression in s, induction brings it to a value.
2. Assign-E reduces s to **nil** when the contained expression is a value.

Lemma 21 The execution of any $s \in Seq$ always reduces –over repeated steps– to the statement **nil**.
Proof Done in cases by the SOS rules:
1. Seq-Step reduces the left-hand statement in s, induction brings it to **nil**.
2. Seq-E reduces s to the right-hand statment in s, and induction reduces that to **nil**.

Lemma 22 The execution of any $s \in If$ always reduces –over repeated steps– to the statement **nil**.
Proof Done in cases by the SOS rules:
1. If-Eval reduces the expression in s, induction brings it to a boolean value.
2. If-T-E reduces s when the expression is the value **true** to the body statement, and induction reduces that to **nil**.
3. If-F-E reduces s when the expression is the value **false** directly to **nil**.

Lemma 23 The execution of any $s \in Par$ always reduces –over repeated steps– to the statement **nil**.
Proof Done in cases by the SOS rules:
1. Par-L reduces the left-hand statement in s, induction brings it to **nil**.
2. Par-R is symmetrical to Par-L.
3. Par-E reduces s directly to **nil** when both component statements are **nil**.

Lemma 24 Given $S \in \mathit{While}$ such that $S = \mathit{mk\text{-}While}(b, \mathit{body})$, $\{P, R\} \vdash S$ **sat** $(R,\ W^* \land P \land \neg\, b)$, and suitable $\sigma \in \Sigma$, providing body reduces to **nil** and W is a well-founded order over Σ, then S reduces to **nil**.

$$\mathit{NilP} \triangleq \lambda c{:}\ \mathit{Config} \cdot c(1) = \textbf{nil}$$

$$\mathit{reaches} : \mathit{Config} \times \mathit{CPred} \times \mathit{SemRel} \to \mathbb{B}$$
$$\mathit{reaches}(c, \mathit{CP}, \mathcal{T}) \ \triangleq \ \forall cs \in \mathit{inf\text{-}seqs}(c, \mathcal{T}) \cdot \exists i \in \mathbb{N}_1 \cdot \mathit{CP}(cs(i))$$

$$\mathit{inf\text{-}seqs} : \mathit{Config} \times \mathit{SemRel} \to \mathit{Config}^\infty$$
$$\mathit{inf\text{-}seqs}(c, \mathcal{T}) \ \triangleq \ \{cs \mid cs \in \mathit{Config}^\infty \land \textbf{hd}\ cs = c \land \forall i \in \mathbb{N}_1 \cdot \mathcal{T}(cs(i), cs(i+1))\}$$

$$\mathit{CPred}{:}\ \mathit{Config} \to \mathbb{B}$$
$$\mathit{SemRel}{:}\ \mathcal{P}\,(\mathit{Config} \times \mathit{Config})$$
$$\mathit{Config}{:}\ \mathit{Stmt} \times \Sigma$$

Proof:

from $\{P, R\} \vdash \mathit{mk\text{-}While}(b, \mathit{body})$ **sat** $(G,\ W^* \wedge P \wedge \neg\, b);\ [\![P]\!](\sigma_0);\ \mathit{IH\text{-}T}(\mathit{body})$

1 $\mathit{bottoms}(W, \neg\, b)$ — h, **While-I**

2 $\mathit{twf}(W)$ — h, **While-I**

3 $\{P, R\} \vdash \mathit{body}$ **sat** $(G,\ W \wedge P)$ — h, **While-I**

4 $\overleftarrow{\neg\, b} \wedge R \ \Rightarrow\ \neg\, b$ — h, **While-I**

5 $R \ \Rightarrow\ W^*$ — h, **While-I**

6 **from** $\{P_b, R_b\} \vdash \mathit{body}$ **sat** $(G_b,\ Q_b);\ [\![P_b]\!](\sigma_b)$

infer $\mathit{reaches}((\mathit{body}, \sigma_b), \mathit{NilP}, \xrightarrow[R]{r})$ — IH-T(body)

7 $\exists \sigma_1 \in \Sigma \cdot \left(\begin{array}{l} (\mathit{mk\text{-}While}(b, \mathit{body}), \sigma_0) \xrightarrow[R]{r}{*} (\mathit{mk\text{-}While}(b, \mathit{body}), \sigma_1) \\ \xrightarrow[R]{r} (\mathit{mk\text{-}If}(b, \mathit{mk\text{-}Seq}(\mathit{body}, \mathit{mk\text{-}While}(b, \mathit{body}))), \sigma_1) \end{array} \right)$ — h, While, $\xrightarrow[R]{r}$

8 **from** $\sigma_1 \in \Sigma$ **st** [[7]]

8.1 $\mathit{seq} = \mathit{mk\text{-}Seq}(\mathit{body}, \mathit{mk\text{-}While}(b, \mathit{body}))$ — definition

8.2 $\exists v \in \mathbb{B} \cdot (\mathit{mk\text{-}If}(b, \mathit{seq}), \sigma_1) \xrightarrow[R]{r}{*} (\mathit{mk\text{-}If}(v, \mathit{seq}), \sigma_2)$ — h8, 8.1, $\xrightarrow[R]{r}$, *Lemma 22.1*

8.3 **from** $v \in \mathbb{B}$ **st** [[8.2]]

8.3.1 **from** $\neg\, v$

8.3.1.1 $\mathit{reaches}((\mathit{mk\text{-}If}(v, \mathit{seq}), \sigma_2), \mathit{NilP}, \xrightarrow[R]{r})$ — h8.3, h8.3.1, *Lemma 22.3*

infer $\mathit{reaches}((\mathit{mk\text{-}While}(b, \mathit{body}), \sigma_0), \mathit{NilP}, \xrightarrow[R]{r})$ — h8, h8.3, 8.3.1.1, $\xrightarrow[R]{r}$

8.3.2 **from** v

8.3.2.1 $\exists \sigma_3 \in \Sigma \cdot (\mathit{mk\text{-}If}(v, \mathit{seq}), \sigma_2) \xrightarrow[R]{r}{*} (\mathit{mk\text{-}While}(b, \mathit{body}), \sigma_3)$ — h8.3, h8.3.2, $\xrightarrow[R]{r}$, IH-T(body)

8.3.2.2 **from** $\sigma_3 \in \Sigma$ **st** [[8.3.2.1]]

8.3.2.2.1 $[\![W \wedge P]\!](\sigma_0, \sigma_3)$ — h, 3, 6, h8, h8.3, h8.3.2, h8.3.2.2, **PR-ident**, $\xrightarrow[R]{r}$, *Thm 18*

8.3.2.2.2 **from** $[\![W]\!](\sigma_0, \sigma_3)\ \Rightarrow\ \left(h[\sigma_3/\sigma_0] \textbf{ gives } \mathit{reaches}((\mathit{mk\text{-}While}(b, \mathit{body}), \sigma_3), \mathit{NilP}, \xrightarrow[R]{r}) \right)$

8.3.2.2.2.1 **from** $\{P, R\} \vdash \mathit{mk\text{-}While}(b, \mathit{body})$ **sat** $(G,\ W^* \wedge P \wedge \neg\, b); [\![P]\!](\sigma_3); \mathit{IH\text{-}T}(\mathit{body})$

infer $\mathit{reaches}((\mathit{mk\text{-}While}(b, \mathit{body}), \sigma_3), \mathit{NilP}, \xrightarrow[R]{r})$ — 8.3.2.2.1, h8.3.2.2.2

8.3.2.2.2.2 $\mathit{reaches}((\mathit{mk\text{-}While}(b, \mathit{body}), \sigma_3), \mathit{NilP}, \xrightarrow[R]{r})$ — h, 8.3.2.2.1, 8.3.2.2.2.1

infer $\mathit{reaches}((\mathit{mk\text{-}While}(b, \mathit{body}), \sigma_0), \mathit{NilP}, \xrightarrow[R]{r})$ — 8.3.2.2.1, 8.3.2.2.2.2

infer $\mathit{reaches}((\mathit{mk\text{-}While}(b, \mathit{body}), \sigma_0), \mathit{NilP}, \xrightarrow[R]{r})$ — **W-Indn**(8.3.2.2.2)

infer $\mathit{reaches}((\mathit{mk\text{-}While}(b, \mathit{body}), \sigma_0), \mathit{NilP}, \xrightarrow[R]{r})$ — $\exists$-E(8.3.2.1, 8.3.2.2)

infer $\mathit{reaches}((\mathit{mk\text{-}While}(b, \mathit{body}), \sigma_0), \mathit{NilP}, \xrightarrow[R]{r})$ — $\vee$-E(h8.3, 8.3.1, 8.3.2)

infer $\mathit{reaches}((\mathit{mk\text{-}While}(b, \mathit{body}), \sigma_0), \mathit{NilP}, \xrightarrow[R]{r})$ — $\exists$-E(8.2, 8.3)

infer $\mathit{reaches}((\mathit{mk\text{-}While}(b, \mathit{body}), \sigma_0), \mathit{NilP}, \xrightarrow[R]{r})$ — $\exists$-E(7, 8)

Theorem 25 For any $S \in \mathit{Stmt}$ such that $\{P, R\} \vdash S$ **sat** (G, Q) and suitable $\sigma \in \Sigma$, then the termination predicate $\mathit{reaches}((S, \sigma), \mathit{Stmt\text{-}Term}, \xrightarrow[R]{r})$ holds.

Proof Observe that every rule of the SOS except While always transitions in such a way that the $<$-ordering is maintained; with suitable conditions, the *While* construct's body conforms to a transitive, well-founded ordering over states that eventually eliminates the *While*.