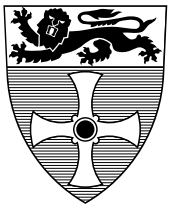
UNIVERSITY OF
NEWCASTLE



University of Newcastle upon Tyne

COMPUTING SCIENCE

Guaranteeing the soundness of rely/guarantee rules

J. W. Coleman and C. B. Jones.

TECHNICAL REPORT SERIES

No. CS-TR-955 March, 2006

TECHNICAL REPORT SERIES

No. CS-TR-955 March, 2006

Guaranteeing the soundness of rely/guarantee rules

Joey W. Coleman and Cliff B. Jones

Abstract

The challenges of finding compositional ways of (formally) developing concurrent programs are considerable. One way of tackling such design tasks is to deploy rely and guarantee conditions to record and reason about interference. This paper presents a new approach to justifying the soundness of rely/guarantee inference rules. The approach followed is to view a "structural operational semantics" as defining an inference system and to show that the proof rules used are valid proof tactics within that inference system. This leaves aside worries about completeness of the rely/guarantee rule set because one is always in a position to add new rules in the same way.

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..

Guaranteeing the soundness of rely/guarantee rules
[By] J. W. Coleman, 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-955)

Added entries

UNIVERSITY OF NEWCASTLE UPON TYNE
Computing Science. Technical Report Series. CS-TR-955

Abstract

The challenges of finding compositional ways of (formally) developing concurrent programs are considerable. One way of tackling such design tasks is to deploy rely and guarantee conditions to record and reason about interference. This paper presents a new approach to justifying the soundness of rely/guarantee inference rules. The approach followed is to view a “structural operational semantics” as defining an inference system and to show that the proof rules used are valid proof tactics within that inference system. This leaves aside worries about completeness of the rely/guarantee rule set because one is always in a position to add new rules in the same way.

About the author

Joey Coleman 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.while working as a Research Associate.

He is involved primarily with the RODIN project, working on methodology. Other associations include the DIRC project. His main interests lie in language design and semantics.

Cliff Jones is one of the Professors of Computing Science at Newcastle. Within the School of Computing Science he acts as Research Director.Currently his own major research project is the five university IRC on "Dependability of Computer-Based Systems" of which he is overall Project Director.

Cliff has actually 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 which is one of the better known "formal methods". After that time 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 a 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. Cliff is a Fellow of the Royal Academy of Engineering (FREng), ACM, BCS,
and IEE.

Suggested keywords

CONCURRENCY,
CONSISTENCY WITH RESPECT TO SOS,
FLOYD/HOARE RULES,
RELY/GUARANTEE CONDITIONS,
STRUCTURAL OPERATIONAL SEMANTICS

Guaranteeing the soundness of rely/guarantee rules*

Joey W. Coleman
Cliff B. Jones

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

Abstract. The challenges of finding compositional ways of (formally) developing concurrent programs are considerable. One way of tackling such design tasks is to deploy rely and guarantee conditions to record and reason about interference. This paper presents a new approach to justifying the soundness of rely/guarantee inference rules. The approach followed is to view a "structural operational semantics" as defining an inference system and to show that the proof rules used are valid proof tactics within that inference system. This leaves aside worries about completeness of the rely/guarantee rule set because one is always in a position to add new rules in the same way.

* A cut-down version of this paper was submitted to FM'06; please cite the conference version rather than this Technical Report.

1 Introduction

Floyd/Hoare rules provide a way of reasoning about non-interfering programs. For sequential programs, such rules are now well known; their soundness can be proved; and one can even obtain (relatively) complete [Apt81] "axiomatic semantics" for simple languages. One distinction from the standard literature is that VDM has always insisted on using post-conditions of two states but the initial set of inference rules [Jon80] were "unmemorable". Usable rules were proposed by Peter Aczel [Acz82] and are used in later VDM books such as [Jon90].

Finding compositional proof rules for concurrent programs proved more challenging (see below) but rely/guarantee conditions offer a way of documenting and reasoning about "interference". Various forms of such rules are in the literature and their soundness proved (see particularly [Pre03]).

The current paper provides an example of a set of such rely/guarantee rules; an underlying operational semantics; and a justification of the former with respect to the latter. The view taken here is that (following Tom Melham [CM92] and Tobias Nipkow [KNvO$^+$02]) the rules of an operational semantics can be taken to provide an inductive definition of a relation between pairs of pairs of program texts and states. Non-determinacy –including that from concurrency– forces one to think in terms of relations rather than functions. Results about programs could be proved directly in terms of this inference system. We view the rules for reasoning about rely/guarantee conditions as extra inference rules which have to be shown to be consistent with (longer) proofs directly in terms of the operational semantics. This view absolves us from concerns about completeness because one can just prove more rules as required. This is fortunate because rely/guarantee rules have to fit many different styles of concurrent programming and it is difficult to envisage a single canonical set.

There is a lot written about rely/guarantee conditions,[1] but there is no convenient short summary (the excellent [dR01] is neither short nor easy reading). It would 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]).

Thus this paper presents one version of a collection of rely/guarantee rules for reasoning about interference; a semantic model of a simplified, concurrent, shared-variable language; and outlines an approach by which one could give a justification of the formal rules with respect to the language model. To aid the reader's understanding we offer an example of a small concurrent program whose design can be explicated in terms of the aforementioned rules.

For the benefit of those less familiar with rely/guarantee concepts we have provided a brief explanation of their use. Program development using the Floyd/Hoare pre- and post-conditions can be visualised as shown in Figure 1a. The horizontal line represents the system state over time; P and Q are pre- and post-condition predicates, respectively; and the program's execution is represented in the box along the top of the diagram. This

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

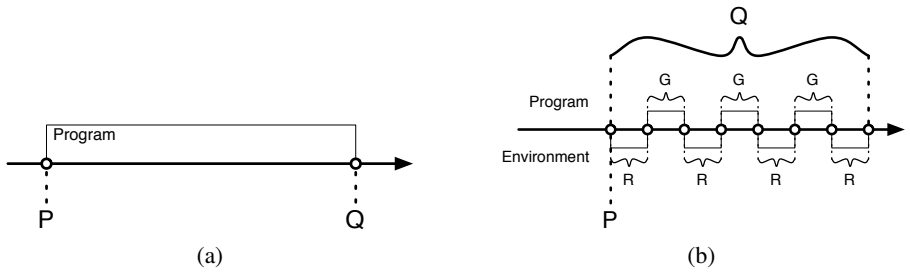


Fig. 1. (a) Pre-/Post-conditions and (b) Rely-/Guarantee-conditions

model is fine for isolated, sequential systems, but it assumes atomicity with respect to the program's environment, making it unsuitable for concurrent development.

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

Note the asymmetry between R and G: while the latter is a hard restriction on the system, the former does not constrain anything. It is the responsibility of the user of the system to ensure that it is run within 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 one thread's guarantee becomes part of the other threads' rely-conditions.

With Figure 1b in mind, then, the thrust of a R/G development lies in formalising assumptions about 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 then 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 becomes critical when reasoning about different branches of a program, or about the behavioural interaction of the two separately developed parallel programs.

2 An example development

We will use a problem which originated in [Owi75] and was tackled by rely/guarantee reasoning in [Jon81] as an example for this paper. This example, FINDP, is actually a little too simple to show the advantage (over Owicki/Gries) of compositional reasoning — the "prime sieve" [Jon96] is a more convincing example — but this is sufficient. We are assuming throughout that both rely and guarantee conditions are reflexive (we can stutter) and transitive (they can cover multiple steps should the other stutter).

2.1 The specification

We assume that we have a predicate, $pred: X \to \mathbb{B}$ that is expensive to evaluate. For reasons that anticipate concurrent version of the program, $pred$ must be free of side-effects and it must be re-entrant/thread-safe. 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 $pred(v(i))$.

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

As a brief explanation of $FINDP$'s specification, this program requires access to two variables: v and r. The former will only be read by $FINDP$, and the latter will be written to as well as read from. The precondition of this program allows for any starting state, and we are not constraining the visible behaviour of this program with the guarantee-condition. The rely-condition requires that the environment never changes v or r. Finally, the postcondition asserts that if r is a valid index into v, then $pred$ will hold on $v(i)$; alternately, if there are no values in v for with $pred$ holds, then r will be precisely one greater than the length of v.

2.2 Sequential aside

Apart from the rely/guarantee stuff itself, there are aspects of VDM which did not follow the "main stream" (although in some cases, others have moved towards the VDM position). VDM uses post conditions of two states (relations) and this means that standard Floyd/Hoare rules don't work; the form of the proof rules used here is as in the (first) 1986 edition of [Jon90]. The rule used for **while** in VDM ensures termination by requiring that the relation on the $body$ is well-founded; this appears more natural than Dijkstra's additional "variant function" [DS90]. VDM also deploys a "logic of partial functions" [BCJ84] (cf. $P \Rightarrow \delta_l(b)$ in **sim-While-I** ensures that the expressions of the logic (LPF) will be defined in the implementation language).

We are really interested in a parallel implementation but we could prove (using the rules justified in [Jon87]) a sequential implementation like:

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

This would use **sim-While-I** in Appendix B.1; which goes through with W:

$\overleftarrow{r} < r$

and P:

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

2.3 Introducing parallelism

We actually have in mind a development (from the specification in Section 2.1) like:

$t \leftarrow$ **len** $v + 1$;
$(SEARCH(\{i \in \textbf{inds}\, v \mid is\text{-}odd(i)\}) \parallel SEARCH(\{i \in \textbf{inds}\, v \mid \neg\, is\text{-}odd(i)\}))$;
$r \leftarrow t$

where the specification for $SEARCH$ is

$SEARCH(ms{:}\,\mathbb{N}\text{-}\textbf{set})$
rd $v{:}\,X^*$
wr $t{:}\,\mathbb{N}$
pre true
rely $ms \lhd v = ms \lhd \overleftarrow{v} \wedge t \leq \overleftarrow{t}$
guar $t \neq \overleftarrow{t} \;\Rightarrow\; t < \overleftarrow{t} \wedge pred(v(t))$
post $\forall i \in ms \cdot i < t \;\Rightarrow\; \neg\, pred(v(i))$

We use the key **Par-I** rule (with an "invariant" of $t \in \textbf{inds}\, v \;\Rightarrow\; pred(v(t))$) to show that the parallel statement satisfies

$SEARCHES$
rd $v{:}\,X^*$
wr $t{:}\,\mathbb{N}$
pre true
rely $v = \overleftarrow{v} \wedge t = \overleftarrow{t}$
guar $t \neq \overleftarrow{t} \;\Rightarrow\; t < \overleftarrow{t} \wedge pred(v(t))$
post $\forall i \in \{1..t-1\} \cdot \neg\, pred(v(i))$

and then **Seq-I** and **weaken** to show that

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

satisfies the specification of $FINDP$ in Section 2.1.

2.4 Decomposing $SEARCH$ and reifying t

We now firmly specialize the ms argument by allocating the even indices (of v) to one instance of $SEARCH$ and the odd ones to the other instance.[2] Doing this exposes the problem of updating the variable t which is shared between the two instances of $SEARCH$. An assignment like $< t \leftarrow min(t, \cdots) >$ would need to be flagged as "atomic" since the language of Appendix A permits interference during expression evaluation. As explained in [Jon05], it is a common strategy to avoid such problems by choosing suitable reifications of abstract variables. We choose to implement t as $min(ot, et)$. The code would look like:

[2] Up to now, we could have followed [Jon81] and generalise the previous step to allow an arbitrary number of instances of $SEARCH$.

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

To see that this satisfies the specification 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 representational choice for t, the worst case only ends up with one extra evaluation of $pred$ for each $SEARCH$ block that *doesn't* 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. It is straightforward for a sequential (non-concurrent) language to write such a consistency proof (see [Lau71,Don76]); there are even brave souls [Bro05] who try this with a denotational semantics (but without "power domains") for concurrency.

We take here an operational underpinning given in the form of a Structural Operational Semantics [Plo81,Plo04b,Plo04a,Jon03]. In particular, we view the rules of the semantics as (inductively) defining a relation over program texts and states[3]. For a sequential language, this is only one Lambda abstraction away from denotational semantics; the only failure of the "homomorphic rule" is for the While statement (which would need a least fixed point in a denotational description). It is necessary to take the usual function over (texts and) states to subsequent states and change it into a *relation* so that we can model non-determinism in general and concurrency in particular.

[3] We do, of course, avoid the Baroque excesses caused by using a "Grand State".

The base language that we are using is defined in Appendix A[4] and has been kept deliberately simple. It has six main statement constructs and **nil** to represent a completed statement, as well as a subsidiary expression construct. Variable assignment is represented by the $Assign$ construct and is the only means to alter the state. Assignment in this language is only atomic for the actual mutation of the state object. Expression evaluation is non-atomic and allows situations where $(x + x) \neq 2x$. This is a deliberate design decision as it allows parallel statements to directly interfere with each other in a way which would permit efficient implementations. Conditional execution of statements 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.

Looping is afforded by the $While$ construct, but our language description gives the behaviour for this construct indirectly. The SOS rule that specifically deals with this construct rewrites the program text in terms of an If that contains a sequence with the loop body and the original $While$.

The Seq construct handles sequential execution, and its structure mimics that of a LISP-style cons-cell. The SOS rules step through the sl field first, and when it is reduced to **nil**, the sr is unwrapped by the SOS rules[5].

The Par construct represents 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.

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 at the level of the whole text requires making non-deterministic choices all the way down to a leaf statement (even –see below– to a leaf operand of an expression).

We have included a construct named $DecStmt$ which does nothing relative to the operational semantics of the language. This construct is, however, critical to the form of our proofs because of the assertions in its r and g fields. These fields provide annotations about the rely- and guarantee-conditions for the contained statement. This construct effectively provides modularity for the proofs — anything that runs in parallel with a $DecStmt$ can safely remain ignorant of the actual content of that $DecStmt$.

The subsidiary type, $Expr$, is used by the $Assign$, If, and $While$ constructs. It has its own relation, $\xrightarrow{e}$, which models the process of expression evaluation. Expression evaluation cannot cause side effects as there is no mechanism to mutate the state.

The base 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 initialised in the state object at the start of execution.

[4] This description follows the "VDM tradition" of basing the semantics on an *abstract* syntax and restricting the class further using "context conditions".

[5] This behaviour means that any structure composed of Seq objects will be still evaluated from left to right.

As presented, the base language contains no reference to any external environment — it has been given as a closed system. As such, the final configuration of a system in this language is of the form $(\mathbf{nil}, \sigma)$. Some of the initial designs of this language included a rule that explicitly modelled environmental actions that conformed to the rely of a given program. This was removed to simplify the proofs and partially due to the observation that it is simpler to model the environment as a program-like object running in parallel with the actual program (i.e. $mk\text{-}Par(\mathbf{Program}, \mathbf{Environment})$).

4 Soundness

The overall approach to the proof can, however, be seen without the complications of concurrency and these simpler proofs are sketched here for ease of understanding; comments are provided on the extra issues that have had to be resolved for the concurrent language.[6]

The overall soundness result is that, under the assumption that we have a proof using inference rules in Appendix B (i.e. $\vdash S$ **sat** (P, Q)), if S is executed in a state for which $[\![P]\!](\sigma)$, then (the program cannot fail to terminate, and) any state σ' which can be reached by $\xrightarrow{s*}$ will be such that $[\![Q]\!](\sigma, \sigma')$. Both of these proofs can be done by structural induction over the abstract syntax for $Stmt$ (see **Stmt-Indn** in Appendix B.3). Interestingly, the termination proofs need the correctness lemmas; so we do correctness first. The choice of order is safe: 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.

It is important to realise the role of rely/guarantee conditions in these proofs. To achieve separation of arguments about different "threads" is 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 (in the full concurrency proofs) the need to show that the execution of a $DecStmt$ respects the rely condition (see Section 4.3).

4.1 Correctness

We first need to establish a link between the meaning of an expression (respectively, predicate) and its evaluation in the SOS.[7]

Lemma 1 Under suitable conditions $(e, \sigma) \xrightarrow{e} v$ iff $[\![e]\!](\sigma) = v$. We will also cite this lemma when e is embedded in a ($Assign$, If, or $While$) statement.

Proof The issue here is really the "conditions": because of the potential presence of undefined terms, $[\![e]\!]$ might be defined in LPF [BCJ84] while an implementation would fail to deliver a value. We simply ensure that no such expressions are left at the end of development (cf. use of $\delta(e)$ in the rules for **sim-If-I** and **sim-While-I**). The other issue

[6] To carry over as much as possible of the argument to the concurrent case, we use here a "small step" semantics although "large step" would be possible for a non-interfering language.

[7] This is one issue which is not tackled in [Pre03] — see Section 5.

	from $mk\text{-}If(b, th)$ **sat** (P, Q); $[\![P]\!](\sigma)$; $(mk\text{-}If(b, th), \sigma) \xrightarrow{s*} (\textbf{nil}, \sigma')$	
1	$[\![b]\!](\sigma) \in \mathbb{B}$	h, wf-Stmt
2	**from** $\neg\, [\![b]\!](\sigma)$	
2.1	$(mk\text{-}If(b, th), \sigma) \xrightarrow{s*} (mk\text{-}If(\textbf{false}, th), \sigma)$	h, h2, *L1*, If-Eval
2.2	$\sigma' = \sigma$	h, h2, 2.1, IF-E-F
	infer $[\![Q]\!](\sigma, \sigma')$	h, h2, 2.2, **sim-If-I**
3	**from** $[\![b]\!](\sigma)$	
3.1	**from** th **sat** (P_{th}, Q_{th}); $[\![P_{th}]\!](\sigma_{th})$; $(th, \sigma_{th}) \xrightarrow{s*} (\textbf{nil}, \sigma'_{th})$	
	infer $[\![Q_{th}]\!](\sigma_{th}, \sigma'_{th})$	IH
3.2	th **sat** $(P \wedge b, Q)$	h, **sim-If-I**
3.3	$[\![P \wedge b]\!](\sigma)$	h, h3
3.4	$(mk\text{-}If(b, th), \sigma) \xrightarrow{s*} (mk\text{-}If(\textbf{true}, th), \sigma)$	h, h3, *L1*, If-Eval
3.5	$(mk\text{-}If(\textbf{true}, th), \sigma) \xrightarrow{s*} (th, \sigma)$	If-E-T
3.6	$(th, \sigma) \xrightarrow{s*} (\textbf{nil}, \sigma')$	h, SOS model
	infer $[\![Q]\!](\sigma, \sigma')$	h, h3, 3.1, 3.2, 3.3, 3.6
	infer $[\![Q]\!](\sigma, \sigma')$	1, $\vee$-E(2, 3)

Fig. 2. Proof of correctness of *If*

is interference during expression evaluation: with the level of granularity that we are allowing here, it is frequently invalid to think about a single state for such expression evaluation. This requires care in formulation of the inference rules in Appendix B.2.

One base case for the induction over $Stmt$ –that for **nil**– is trivial and embedded in Theorem 1; the base case for assignments might look like:

Lemma 2 Given $mk\text{-}Assign(id, expr)$ **sat** (P, Q), for any $\sigma \in \Sigma$ such that $[\![P]\!](\sigma)$, if $(mk\text{-}Assign(id, expr), \sigma) \xrightarrow{s} (\textbf{nil}, \sigma')$ then $[\![Q]\!](\sigma, \sigma')$.

In fact, there is no proof rule for assignments offered in Appendix B as they tend to be very specialised.

We'll look at one example of induction and chose *If* because it shows the need to use Lemma 1; however, we pass over the next lemma without proof.

Lemma 3 Given $mk\text{-}Seq(sl, sr)$ **sat** (P, Q), for any $\sigma \in \Sigma$ such that $[\![P]\!](\sigma)$, if $(mk\text{-}Seq(sl, sr), \sigma) \xrightarrow{s} (\textbf{nil}, \sigma')$ then $[\![Q]\!](\sigma, \sigma')$ providing sl and sr behave according to their specifications.

A non-interfering conditional rule (recall that we have no else clause) would be **sim-If-I** as in Appendix B.1.

Lemma 4 Given $mk\text{-}If(b, th)$ **sat** (P, Q), for any $\sigma \in \Sigma$ such that $[\![P]\!](\sigma)$, if $(mk\text{-}If(b, th), \sigma) \xrightarrow{s} (\textbf{nil}, \sigma')$ then $[\![Q]\!](\sigma, \sigma')$ providing th behaves according to its specification.

Proof in Figure 2.[8]

[8] Notice that the induction hypothesis has to be given as a **from/infer** box because it is not a simple implication.

This is one place where interference has a significant impact on the form of the rely/guarantee rule in Appendix B.2: at first sight, it comes as a shock that one can no longer (in general) assume that b is true for the th proof but this is a direct consequence of interference and it should be noted that the development in Section 2 uses a statement where such interference occurs. (It is of course possible to justify other rules which cover the situation where b is stable under R.)

The rule **sim-While-I** in Appendix B.2 is similar to that for If.

Lemma 5 Given $mk\text{-}While(b, body)$ **sat** (P, Q), for any $\sigma \in \Sigma$ such that $[\![P]\!](\sigma)$ if $(mk\text{-}While(b, body) \xrightarrow{s} (\textbf{nil}, \sigma')$ then $[\![Q]\!](\sigma, \sigma')$ providing $body$ behaves according to its specification.

The proof of this lemma is elided from the conference version of the paper but the essential interest (in complete induction) is explored in the proof of *L12* in Figure 3.

Moving now to the concurrent part of the language, we show the lemmas regarding concurrency[9].

Lemma 6 Given $mk\text{-}DecStmt(r, body, g)$ **sat** (P, Q), for any $\sigma \in \Sigma$ such that $[\![P]\!](\sigma)$ if $(mk\text{-}DecStmt(r, body, g), \sigma) \xrightarrow{s} (\textbf{nil}, \sigma')$ then $[\![Q]\!](\sigma, \sigma')$ providing $body$ behaves according to its specification.

Lemma 7 Given $mk\text{-}Par(sl, sr)$ **sat** (P, Q), for any $\sigma \in \Sigma$ such that $[\![P]\!](\sigma)$ if $(mk\text{-}Par(sl, sr), \sigma) \xrightarrow{s} (\textbf{nil}, \sigma')$ then $[\![Q]\!](\sigma, \sigma')$ providing sl, sr behave according to their specifications.

The final theorem just uses the lemmas on the constructs (tying all of the loose ends on appeals to induction hypotheses).

Theorem 1 For any $st \in Stmt$ for which st **sat** (P, Q), for any $\sigma \in \Sigma$ such that $[\![P]\!](\sigma)$ if $(st, \sigma) \xrightarrow{s} (\textbf{nil}, \sigma')$ then $[\![Q]\!](\sigma, \sigma')$.

4.2 Termination

The issue here is that it has to be shown that divergence is impossible on any non-deterministic evaluation (not just that the evaluation *can* terminate).[10] The predicate $terminates$ indicates that there can be no infinite sequence of $\xrightarrow{s}$ (respectively $\xrightarrow{e}$) reductions. Proving results about statements which contain expressions needs the following lemma.

Lemma 8 For any $e \in Expr$ and suitable $\sigma \in \Sigma$, $terminates(e, \sigma, \xrightarrow{e})$ with $v \in Value$. We will also cite this lemma when e is embedded in a ($Assign$, If, or $While$) statement.

Proof This follows by structural induction over the abstract syntax of $Expr$ (using **Expr-Indn** of Appendix B.3): by inspection of Appendix A every step of $\xrightarrow{e}$ reduces

[9] Please note that the **sim-*** rules are actually insufficient to prove these lemmas; an upcoming journal version of this paper will have full proofs.

[10] These termination proofs are interesting but are omitted in [Pre03] which only tackles "partial correctness" — see Section 5.

	from $S = mk\text{-}While(b, body)$; S **sat** $(P, P \wedge \neg\, b \wedge (W \vee I_\Sigma))$; $[\![P]\!](\sigma)$	
1	$[\![b]\!](\sigma) \in \mathbb{B}$	h, wf-Stmt
2	$S' = mk\text{-}Seq(body, S)$	definition
3	$(S, \sigma) \xrightarrow{s} (mk\text{-}If(b, S'), \sigma)$	2, While
4	**from** $\neg\, [\![b]\!](\sigma)$	
4.1	$(mk\text{-}If(b, S'), \sigma) \xrightarrow{s*} (mk\text{-}If(\mathbf{false}, S'), \sigma)$	h4, *L1*
4.2	$(mk\text{-}If(\mathbf{false}, S'), \sigma) \xrightarrow{s} (\mathbf{nil}, \sigma)$	If-E-F
	infer $terminates(S, \sigma, \xrightarrow{s})$	3, 4.1, 4.2
5	**from** $[\![b]\!](\sigma)$	
5.1	$(mk\text{-}If(b, S'), \sigma) \xrightarrow{s*} (mk\text{-}If(\mathbf{true}, S'), \sigma)$	h5, *L1*
5.2	$(mk\text{-}If(\mathbf{true}, S'), \sigma) \xrightarrow{s} (S', \sigma)$	If-E-T
5.3	**from** $body$ **sat** (P_b, Q_b); $[\![P_b]\!](\sigma_b)$	
	infer $terminates(body, \sigma_b, \xrightarrow{s})$	h5, IH
5.4	$body$ **sat** $(P \wedge b, P \wedge W)$	**sim-While-I**
5.5	$terminates(body, \sigma, \xrightarrow{s})$	h, h5, 5.4, 5.3
5.6	$(S', \sigma) \xrightarrow{s*} (mk\text{-}Seq(\mathbf{nil}, S), \sigma')$	5.5, Seq-Step
5.7	$[\![W]\!](\sigma, \sigma') \wedge [\![P]\!](\sigma')$	5.4, 5.6, *L5*
5.8	$(mk\text{-}Seq(\mathbf{nil}, S), \sigma') \xrightarrow{s} (S, \sigma')$	Seq-E
5.9	**from** $[\![P]\!](\sigma_w)$; $[\![W]\!](\sigma, \sigma_w)$	
	infer $terminates(S, \sigma_w, \xrightarrow{s})$	h, h5.9, **W-Indn**
5.10	$terminates(S, \sigma', \xrightarrow{s})$	5.7, 5.9
	infer $terminates(S, \sigma, \xrightarrow{s})$	3, 5.1, 5.2, 5.6, 5.8, 5.10
	infer $terminates(S, \sigma, \xrightarrow{s})$	3, $\vee$-E(1, 4, 5)

Fig. 3. Proof of termination of While

either the tree itself or substitutes a $Value$ for an identifier; this provides a well-founded order and guarantees termination.

Turning now to statements: one of the key points of our language is that most of its SOS rules are reductive. The only rule that is not a reduction is While.

The base case for **nil** is trivial and embedded in Theorem 2; that for assignments is

Lemma 9 For any $s \in Assign$ and suitable $\sigma \in \Sigma$, $terminates(s, \sigma, \xrightarrow{s})$.

Lemma 10 For any $mk\text{-}Seq(sl, sr)$ and suitable $\sigma \in \Sigma$, providing sl and sr terminate, $terminates(mk\text{-}Seq(sl, sr, \xrightarrow{s})$.

The proof for conditional is similar (but uses Lemma 8).

Lemma 11 For any $mk\text{-}If(b, th)$ and suitable $\sigma \in \Sigma$, providing th terminates, then $terminates(mk\text{-}If(b, th), \sigma, \xrightarrow{s})$.

The interesting termination proof is of course for the while statement.

Lemma 12 For any $mk\text{-}While(b, body)$ and suitable $\sigma \in \Sigma$, providing $body$ terminates, $terminates(mk\text{-}While(b, body), \sigma, \xrightarrow{s})$.

Proof See Figure 3. This is exactly where we need "complete induction" [11] over the (well founded) loop relation. For some transitive well-founded relation $W \in \mathcal{P}(\Sigma \times \Sigma)$[12]

$$\boxed{W\text{-}Indn}\frac{(\forall \sigma \in \{\sigma \mid (\sigma, \sigma') \in W\} \cdot H(\sigma)) \;\Rightarrow\; H(\sigma')}{\forall \sigma \in \Sigma \cdot H(\sigma)}$$

This ensures that the loop will always terminate (which is what we need to prove: that the sematics cannot make endless $\xrightarrow{s}$ transitions, so long as the proof rule has been used and the pre-condion is true).

In the interfering case, one needs to show that R still respects W. There is a more subtle issue on fairness which is not addressed here.

Moving now to the concurrent part of the language.

Lemma 13 For any $mk\text{-}DecStmt(r, body, g)$ and suitable $\sigma \in \Sigma$, providing $body$ terminates, $terminates(mk\text{-}DecStmt(r, body, g), \sigma, \xrightarrow{s})$.

The important issue with rely and guarantee conditions is addressed in Section 4.3 below

Lemma 14 For any $mk\text{-}Par(sl, sr)$ and suitable $\sigma \in \Sigma$, providing sl, sr terminate, $terminates(mk\text{-}Par(sl, sr), \sigma, \xrightarrow{s})$.

The final theorem just appeals to the lemmas on the constructs (tying all of the loose ends on appeals to IH).

Theorem 2 For any $S \in Stmt$ and suitable $\sigma \in \Sigma$, $terminates(S, \sigma, \xrightarrow{s})$.

4.3 Interference

As can be seen, the issue of interference affects relatively few places in the above argument; where it does impinge, the effect is delicate and requires careful thought. The inability to carry the information about the tested expression b in both If and $While$ statements has been touched on in Section 4.1. This is a consequence of our decision to permit rather fine grained interference; we feel this is necessary to facilitate realistic implementation.

Turning now to the place where one has to reason with respect to interference itself, the behavioural proofs only need to demonstrate that all of the changes to σ that the program makes will conform to the guarantee. This becomes interesting as it is much easier to do than it is to verify correctness. In our particular language, this means that the only construct which is directly affected is $Assign$. There are no other constructs that mutate σ.

The process of verifying an $Assign$ against its guarantee is less simple than we might like. The initial phase of the Assign — that of expression evaluation — trivially satisfies the guarantee as it is only reading from the state. While it is doing so, however,

[11] The complete induction equivalent of induction over the integers is:

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

[12] Here the equivalent of a "zero case" is $e \notin \mathbf{dom}\ W$.

it is possible for the environment to mutate the state, meaning that $(e, \sigma) \xrightarrow{e*} [\![e]\!](\sigma)$ does not hold. Once the first phase has finished there is a set of possible evaluations of the expression; that set, and the state at the completion of evaluation, are what must be used to check if the $Assign$ conforms to its guarantee.

One side-effect that interference has on the proofs is fairly profound; where in the sequential version we need to prove correctness before termination, when we consider interference, we find that we cannot prove either of those without having proved the behavioural correctness first. To give an example, if we need to ensure that, for a While loop, R respects W (as was pointed out earlier), we also need to know that the guarantees of any part of the program running parallel to that while also respect W.

5 Conclusions

Of the related work, the most relevant comparison is certainly with [Pre03] which provides an Isabelle/HOL proof of a related result. 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. (We are the first to confess that there are points in our proofs which would have to be written more pedantically to achieve machine checking: for example, step 4.1 of Figure 3 should strictly observe that non-termination is impossible which requires a "continuation-like" implication.) The most striking difference in our choice of language 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 is counter-intuitive). This is not an arbitrary decision — we have argued elsewhere [Jon05] that many attempts to simplify proofs by assuming large atomic steps make languages very expensive to implement. Other differences include the fact that we allow nested parallel statements; and [Pre03] uses "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, there is we believe that both approaches could benefit from the other and we are in the process of following up on this.

It is worth comparing what is presented here with the soundness proofs in [Jon87] (or in detail with the Technical Report version thereof [Jon86]): there we gave a (relational) denotational semantics.

We have draft proofs of the lemmas/theorems in this paper and expect to publish them in a journal version of this paper once all of the details of the R/G extensions are captured. Various optional excursions into rules for assignments etc. could be considered. The first author expects then to consider predicates over continuously varying quantities used in [HJJ03]. As indicated, we are also interested in considering the requirements of machine checked proofs.

Acknowledgments The authors gratefully acknowledge funding for their research from EPSRC (DIRC project and "Splitting (Software) Atoms Safely") and the EU IST-6 pro-

gramme (for RODIN). The technical content of this paper has benefited from discussions with Jon Burton; its creation was inspired by Tony Hoare's complaint that there was no convenient paper on rely/guarantee rules (we hope that the fuller version of this compressed conference submission will fit the bill).

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: theFirst 25 Years*, volume 3525 of *LNCS*. Springer-Verlag, 2005.

[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.

[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.

[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 *Lecture Notes in Computer Science*, pages 154–169. Springer Verlag, 2003.

[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.

[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.

[Jon03] 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.

[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.

[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.

[Pre03] Leonor Prensa Nieto. The rely-guarantee method in Isabelle/HOL. In *Proceedings of ESOP 2003*, volume 2618 of *LNCS*. Springer-Verlag, 2003.

A Base Language

A.1 Abstract Syntax

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

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

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

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

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

$DecStmt :: r : \Sigma \times \Sigma \to \mathbb{B}$
$\quad\quad s : Stmt$
$\quad\quad g : \Sigma \times \Sigma \to \mathbb{B}$

$Par :: dl : [DecStmt]$
$\quad\quad dr : [DecStmt]$

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

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

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**
$e \in \mathbb{B}$: BOOL,
$e \in \mathbb{Z}$: INT,
$e \in ids$: INT,
$mk\text{-}Dyad(+, \_, \_)$: INT,
$mk\text{-}Dyad(-, \_, \_)$: INT,
$mk\text{-}Dyad(\textbf{min}, \_, \_)$: INT,
$mk\text{-}Dyad(\textbf{max}, \_, \_)$: INT,
$mk\text{-}Dyad(<, \_, \_)$: BOOL,
$mk\text{-}Dyad(=, \_, \_)$: BOOL,
$mk\text{-}Dyad(>, \_, \_)$: BOOL
end

Expressions

$wf\text{-}Expr: (Expr \times Id\text{-}\textbf{set}) \to \mathbb{B}$
$wf\text{-}Expr(e, ids) \triangleq e \in (ids \cup \mathbb{B} \cup \mathbb{Z})$
$wf\text{-}Expr(mk\text{-}Dyad(op, a, b), ids) \triangleq$
$\quad typeof(a, ids) = \text{INT} \wedge typeof(b, ids) = \text{INT} \wedge wf\text{-}Expr(a, ids) \wedge wf\text{-}Expr(b, ids)$

Statements

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

A.3 Semantic Objects

$\Sigma = Id \xrightarrow{m} Value$
$\xrightarrow{e}\colon \mathcal{P}((Expr \times \Sigma) \times Expr)$
$\xrightarrow{s}\colon \mathcal{P}((Stmt \times \Sigma) \times (Stmt \times \Sigma))$

A.4 Semantic Rules

Expressions

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

Assign

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

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

Sequence

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

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

If

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

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

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

While

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

DecStmt

$$\text{DecStmt-Step}\ \frac{(s,\sigma) \xrightarrow{s} (s',\sigma')}{(mk\text{-}DecStmt(r,s,g),\sigma) \xrightarrow{s} (mk\text{-}DecStmt(r,s',g),\sigma')}$$

$$\text{DecStmt-E}\ \frac{}{(mk\text{-}DecStmt(r,\textbf{nil},g),\sigma) \xrightarrow{s} (\textbf{nil},\sigma)}$$

Parallel

$$\text{Par-L}\ \frac{(dl,\sigma) \xrightarrow{s} (dl',\sigma')}{(mk\text{-}Par(dl,dr),\sigma) \xrightarrow{s} (mk\text{-}Par(dl',dr),\sigma')}$$

$$\text{Par-R}\ \frac{(dr,\sigma) \xrightarrow{s} (dr',\sigma')}{(mk\text{-}Par(dl,dr),\sigma) \xrightarrow{s} (mk\text{-}Par(dl,dr'),\sigma')}$$

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

B Compositional Inference Rules

We have not presented a rule for the assignment in either subsection as we prefer to reason about it directly in terms of the SOS rules.

B.1 Sequential (simple) Rules

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

$$\textbf{sim-If-I}\ \frac{\begin{array}{l} th\ \textbf{sat}\ (P \wedge b, Q) \\ P \ \Rightarrow\ \delta_l(b) \\ P \wedge \neg\, b \wedge I_\Sigma \ \Rightarrow\ Q \end{array}}{mk\text{-}If(b, th)\ \textbf{sat}\ (P, Q)}$$

$$\textbf{sim-While-I}\ \frac{\begin{array}{l} body\ \textbf{sat}\ (P \wedge b, P \wedge W) \\ P \ \Rightarrow\ \delta_l(b) \end{array}}{mk\text{-}While(b, body)\ \textbf{sat}\ (P, P \wedge \neg\, b \wedge (W \vee I_\Sigma))}$$

$$\textbf{sim-DecStmt-I}\ \frac{body\ \textbf{sat}\ (P, Q)}{mk\text{-}DecStmt(r, body, g)\ \textbf{sat}\ (P, Q)}$$

$$\textbf{sim-Par-I}\ \frac{\begin{array}{l} dl\ \textbf{sat}\ (P_{dl}, Q_{dl}) \\ dr\ \textbf{sat}\ (P_{dr}, Q_{dr}) \\ P \ \Rightarrow\ P_{dl} \wedge P_{dr} \\ Q_{dl} \wedge Q_{dr} \ \Rightarrow\ Q \end{array}}{mk\text{-}Par(dl, dr)\ \textbf{sat}\ (P, Q)}$$

B.2 Concurrent Rules

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

$$\textbf{If-I}\ \frac{\begin{array}{l} th\ \textbf{sat}\ (P, R, G, Q) \\ (P \,|\, R) \ \Rightarrow\ Q \end{array}}{mk\text{-}If(b, th)\ \textbf{sat}\ (P, R, G, Q)}$$

$$\textbf{While-I}\ \frac{\begin{array}{l} body\ \textbf{sat}\ (P' \wedge b, R, G, Q' \wedge W \wedge (\neg\, b \vee P)) \\ P \ \Rightarrow\ P' \\ (R - I) \subseteq W \\ (\neg\, b) \,|\, R \ \Rightarrow\ \neg\, b \end{array}}{mk\text{-}While(b, body)\ \textbf{sat}\ (P, R, G, Q \wedge \neg\, b \wedge (W \vee R))}$$

Note that W in the **While-I** rule is both transitive and well-founded over states.

$$\textbf{DecStmt-I}\ \frac{\begin{array}{l} body\ \textbf{sat}\ (P, r, g, Q) \\ R\ \Rightarrow\ r \\ g\ \Rightarrow\ G \end{array}}{mk\text{-}DecStmt(r, body, g)\ \textbf{sat}\ (P, R, G, Q)}$$

$$\textbf{Par-I}\ \frac{\begin{array}{l} sl\ \textbf{sat}\ (P, R \vee G_{sr}, G_{sl}, Q_{sl}) \\ sr\ \textbf{sat}\ (P, R \vee G_{sl}, G_{sr}, Q_{sr}) \\ G_{sl} \vee G_{sr}\ \Rightarrow\ G \\ P \wedge Q_{sl} \wedge Q_{sr} \wedge (R \vee G_{sl} \vee G_{sr})^*\ \Rightarrow\ Q \end{array}}{mk\text{-}Par(sl, sr)\ \textbf{sat}\ (P, R, G, Q)}$$

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

B.3 Induction rules

$$\textbf{Expr-Indn}\ \frac{\begin{array}{l} n \in (\mathbb{Z} \mid \mathbb{B}) \vdash H(n) \\ id \in \sigma\ \Rightarrow\ H(id) \\ H(a) \wedge H(b)\ \Rightarrow\ H(mk\text{-}Dyad(op, a, b)) \end{array}}{\forall e \in Expr \cdot H(e)}$$

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

$$\textbf{W-Indn}\ \frac{(\forall \sigma \in \{\sigma \mid (\sigma, \sigma') \in W\} \cdot H(\sigma))\ \Rightarrow\ H(\sigma')}{\forall \sigma \in \Sigma \cdot H(\sigma)}$$

C Termination Proofs

Proof of L9 (Assign)

	from $S = mk\text{-}Assign(id, e)$; S **sat** (P, Q); $[\![P]\!](\sigma)$	
1	$(e, \sigma) \xrightarrow{e*} v$	h, *L8*
2	$(S, \sigma) \xrightarrow{s*} (mk\text{-}Assign(id, v), \sigma)$	h, 1, Assign-Eval
3	$(mk\text{-}Assign(id, v), \sigma) \xrightarrow{s} (\textbf{nil}, \sigma')$	Assign-E
	infer $\models terminates(S, \sigma, \xrightarrow{s})$	2, 3

Proof of L10 (Sequence)

	from $S = mk\text{-}Seq(sl, sr)$; S **sat** (P, Q); $[\![P]\!](\sigma)$	
1	**from** sl **sat** (P_{sl}, Q_{sl}); $[\![P_{sl}]\!](\sigma_{sl})$	
	infer $terminates(sl, \sigma_{sl}, \xrightarrow{s})$	IH
2	sl **sat** $(P, Q_l \wedge P_r)$	h, **sim-Seq-I**
3	$terminates(sl, \sigma, \xrightarrow{s})$	h, 1, 2
4	$(S, \sigma) \xrightarrow{s*} (mk\text{-}Seq(\textbf{nil}, sr), \sigma')$	h, 3, Seq-Step
5	$[\![P_r]\!](\sigma')$	*L3*, 4, **sim-Seq-I**
6	$(mk\text{-}Seq(\textbf{nil}, sr), \sigma') \xrightarrow{s} (sr, \sigma')$	Seq-E
7	**from** sr **sat** (P_{sr}, Q_{sr}); $[\![P_{sr}]\!](\sigma_{sr})$	
	infer $terminates(sr, \sigma_{sr}, \xrightarrow{s})$	IH
8	sr **sat** (P_r, Q_r)	h, **sim-Seq-I**
9	$terminates(sr, \sigma', \xrightarrow{s})$	5, 7, 8
	infer $\models terminates(S, \sigma, \xrightarrow{s})$	4, 6, 9

Proof of L11 (If)

	from $S = mk\text{-}If(b, th)$; S **sat** (P, Q); $[\![P]\!](\sigma)$	
1	**from** $\neg[\![b]\!](\sigma)$	
1.1	$(S, \sigma) \xrightarrow{s*} (mk\text{-}If(\textbf{false}, th), \sigma)$	h1, *L1*
1.2	$(mk\text{-}If(\textbf{false}, th), \sigma) \xrightarrow{s} (\textbf{nil}, \sigma)$	If-E-F
	infer $terminates(S, \sigma, \xrightarrow{s})$	1.1, 1.2
2	**from** $[\![b]\!](\sigma)$	
2.1	$(S, \sigma) \xrightarrow{s*} (mk\text{-}If(\textbf{true}, th), \sigma)$	h2, *L1*
2.2	$(mk\text{-}If(\textbf{true}, th), \sigma) \xrightarrow{s} (th, \sigma)$	If-E-T
2.3	**from** $th \in Stmt$; th **sat** (P_{th}, Q_{th}); $[\![P_{th}]\!](\sigma_{th})$	
	infer $terminates(th, \sigma_{th}, \xrightarrow{s})$	IH
2.4	th **sat** $(P \wedge b, Q)$	**sim-If-I**
2.5	$[\![P \wedge b]\!](\sigma)$	h,h2
2.6	$terminates(th, \sigma, \xrightarrow{s})$	2.3, 2.4, 2.5
	infer $terminates(S, \sigma, \xrightarrow{s})$	2.1, 2.2, 2.6
	infer $terminates(S, \sigma, \xrightarrow{s})$	$\vee$-E, 1, 2

Proof of L12 (While)

	from $S = mk\text{-}While(b, body);\ S$ **sat** $(P, P \wedge \neg\, b \wedge (W \vee I_\Sigma));\ [\![P]\!](\sigma)$	
1	$[\![b]\!](\sigma) \in \mathbb{B}$	h, wf-Stmt
2	$S' = mk\text{-}Seq(body, S)$	definition
3	$(S, \sigma) \xrightarrow{s} (mk\text{-}If(b, S'), \sigma)$	2, While
4	**from** $\neg\, [\![b]\!](\sigma)$	
4.1	$(mk\text{-}If(b, S'), \sigma) \xrightarrow{s*} (mk\text{-}If(\textbf{false}, S'), \sigma)$	h4, *LI*
4.2	$(mk\text{-}If(\textbf{false}, S'), \sigma) \xrightarrow{s} (\textbf{nil}, \sigma)$	If-E-F
	infer $terminates(S, \sigma, \xrightarrow{s})$	3, 4.1, 4.2
5	**from** $[\![b]\!](\sigma)$	
5.1	$(mk\text{-}If(b, S'), \sigma) \xrightarrow{s*} (mk\text{-}If(\textbf{true}, S'), \sigma)$	h5, *LI*
5.2	$(mk\text{-}If(\textbf{true}, S'), \sigma) \xrightarrow{s} (S', \sigma)$	If-E-T
5.3	**from** $body$ **sat** $(P_b, Q_b);\ [\![P_b]\!](\sigma_b)$	
	infer $terminates(body, \sigma_b, \xrightarrow{s})$	h5, IH
5.4	$body$ **sat** $(P \wedge b, P \wedge W)$	**sim-While-I**
5.5	$terminates(body, \sigma, \xrightarrow{s})$	h, h5, 5.4, 5.3
5.6	$(S', \sigma) \xrightarrow{s*} (mk\text{-}Seq(\textbf{nil}, S), \sigma')$	5.5, Seq-Step
5.7	$[\![W]\!](\sigma, \sigma') \wedge [\![P]\!](\sigma')$	5.4, 5.6, *L5*
5.8	$(mk\text{-}Seq(\textbf{nil}, S), \sigma') \xrightarrow{s} (S, \sigma')$	Seq-E
5.9	**from** $[\![P]\!](\sigma_w);\ [\![W]\!](\sigma, \sigma_w)$	
	infer $terminates(S, \sigma_w, \xrightarrow{s})$	h, h5.9, **W-Indn**
5.10	$terminates(S, \sigma', \xrightarrow{s})$	5.7, 5.9
	infer $terminates(S, \sigma, \xrightarrow{s})$	3, 5.1, 5.2, 5.6, 5.8, 5.10
	infer $terminates(S, \sigma, \xrightarrow{s})$	3, $\vee$-E(1, 4, 5)

Proof of L13 (DecStmt)

	from $S = mk\text{-}DecStmt(r, body, g);\ S$ **sat** $(P, Q);\ [\![P]\!](\sigma)$	
1	**from** $body$ **sat** $(P_b, Q_b);\ [\![P_b]\!](\sigma_b)$	
	infer $terminates(body, \sigma_b, \xrightarrow{s})$	IH
2	$body$ **sat** (P, Q)	h, **sim-DecStmt-I**
3	$terminates(body, \sigma, \xrightarrow{s})$	h, 1, 2
4	$(S, \sigma) \xrightarrow{s*} (mk\text{-}DecStmt(r, \textbf{nil}, g), \sigma')$	3, DecStmt-Step
5	$(mk\text{-}DecStmt(r, \textbf{nil}, g), \sigma') \xrightarrow{s} (\textbf{nil}, \sigma')$	DecStmt-E
	infer $terminates(A, \sigma, \xrightarrow{s})$	4, 5

Proof of L14 (Parallel)

	from $S = mk\text{-}Par(dl, dr)$; S **sat** (P, Q); $[\![P]\!](\sigma)$	
1	**from** dl **sat** (P_{dl}, Q_{dl}); $[\![P_{dl}]\!](\sigma_{dl})$	
	infer $terminates(dl, \sigma_{dl}, \xrightarrow{s})$	IH
2	dl **sat** (P_l, Q_l)	h, **sim-Par-I**
3	$terminates(sl, \sigma, \xrightarrow{s})$	h, 1, 2
4	**from** dr **sat** (P_{dr}, Q_{dr}); $[\![P_{dr}]\!](\sigma_{dr})$	
	infer $terminates(dr, \sigma_{dr}, \xrightarrow{s})$	IH
5	dr **sat** (P_r, Q_r)	h, **sim-Par-I**
6	$terminates(dr, \sigma, \xrightarrow{s})$	h, 4, 5
7	$(S, \sigma) \xrightarrow{s*} (mk\text{-}Par(\mathbf{nil}, \mathbf{nil}), \sigma')$	Par-Step-*, 3, 6
8	$(mk\text{-}Par(\mathbf{nil}, \mathbf{nil}), \sigma') \xrightarrow{s} (\mathbf{nil}, \sigma')$	Par-E
	infer $terminates(S, \sigma, \xrightarrow{s})$	7, 8

Proof of Theorem T2 (Termination)

	from $S \in Stmt$; S **sat** (P, R, G, Q); $[\![P]\!](\sigma)$	
1	**from** $S = \mathbf{nil}$	
1.1	$(\mathbf{nil}, \sigma) \notin \mathbf{dom}\ \xrightarrow{s}$	def'n $\xrightarrow{s}$
	infer $terminates(S, \sigma, \xrightarrow{s})$	h, h1, 1.1
2	**from** $S \in Assign$	
	infer $terminates(S, \sigma, \xrightarrow{s})$	h, h2, **L9**
3	**from** $S \in Seq$	
	infer $terminates(S, \sigma, \xrightarrow{s})$	h, h3, **L10**
4	**from** $S \in If$	
	infer $terminates(S, \sigma, \xrightarrow{s})$	h, h4, **L11**
5	**from** $S \in While$	
	infer $terminates(S, \sigma, \xrightarrow{s})$	h, h5, **L12**
6	**from** $S \in DecStmt$	
	infer $terminates(S, \sigma, \xrightarrow{s})$	h, h6, **L13**
7	**from** $S \in Par$	
	infer $terminates(S, \sigma, \xrightarrow{s})$	h, h7, **L14**
	infer $terminates(S, \sigma, \xrightarrow{s})$	h, Stmt-Indn(1...7)