SCHOOL OF COMPUTING SCIENCE

Constructing a Tractable Reasoning Framework upon a Fine-Grained Structural Operational Semantics

Thesis by

Joseph William Coleman

In partial fulfillment of the requirements
for the Degree of Doctor of Philosphy

January 2008

Abstract

The primary focus of this thesis is the semantic gap between a fine-grained structural operational semantics and a set of rely/guarantee-style development rules. The semantic gap is bridged by considering the development rules to be a part of the same logical framework as the operational semantics, and a set of soundness proofs show that the development rules, though making development easier for a developer, do not add any extra power to the logical framework as a whole. The soundness proofs given are constructed to take advantage of the structural nature of the language and its semantics; this allows for the addition of new development rules in a modular fashion.

The particular language semantics allows for very fine-grained concurrency. The language itself includes a construct for nested parallel execution of statements, and the semantics is written so that statements can interfere with each other between individual variable reads. The language also includes an atomic block construct for which the semantics is an embodiment of a form of software transactional memory.

The inclusion of the atomic construct helps illustrate the inherent expressive weakness present in the rely/guarantee rules with respect to termination properties. As such, two development rules are proposed for the atomic construct, one of which has serious restrictions in its application, and another for which the termination property does not hold.

Acknowledgments

The willingness of Cliff Jones to accept me as a student a second time in light of my tendency to generate pages of inscrutable formulae and proofs is astounding, and I am deeply grateful for it. His advice and enthusiastic technical discussions have been both beneficial and thoroughly enjoyable.

My wife, Chriss, has been wonderful, especially during the final stages of the write-up of this thesis. Her support, and her attempts to get my mind off the thesis, have been invaluable.

This work was supported in part by RODIN project (funded under the EU IST-6 programme) and the "Splitting (Software) Atoms Safely" project (funded by the UK EPSRC).

Contents

Constructing a Tractable Reasoning Framework upon a Fine-Grained Structural Operational Semantics

$\boxed{1}$ — Introduction

This thesis is concerned with the development of concurrent software as it is a task which has proven to be surprisingly difficult. Our aim, broadly, is to provide tractable reasoning tools which can be used in the development of concurrent software; tractable in this instance is taken to relate to the ease with which a software developer can apply these tools in their work. To this end we provide a rely/guarantee reasoning framework which is capable of dealing with fine-grained concurrency, and in particular, can reason about a language in which expressions are not evaluated in an atomic manner. The semantic model of the implementation language also includes a construct which is intended to provide explicit control over the granularity of execution, and the tools we provide must support that.

At a more abstract level, we are concerned with the soundness of the reasoning framework and the semantic gap between results derived in the framework and their validity in the language used by the software developer. To address this we aim to provide soundness proofs which use the semantic model of the development language directly. This approach is a marked improvement over soundness proofs which use another semantic model –such as Aczel traces [Acz82, dR01]– as that only "moves" the semantic gap from between the development rules and the semantic model of the implementation language to between the two semantic models.

We start from the position that formal and tractable methods are required for the development of concurrent software, and these methods must have practical application if they are to be interesting. Key to ensuring that these formal methods have applications –aside from tractability– is to reduce the semantic gap between the method and the programming language to which the method is applied. Tractable methods allow for systems to be designed by modelling their behaviour, and by iterative analysis of the model. Key to tractability is "compositionality" –see Section 3.1– which allows for reasoning about the individual elements of a model in isolation.

There are two main paradigms for the design of concurrent systems: those dependent explicitly on shared-variables for process interaction, and those dependent on message-passing; these are extremes of a continuum, of course, as there are systems that use both notions. Both of these paradigms allow for *interference* between processes, but they do so in different ways: shared-variable systems allow interaction through the values of shared variables; message-passing systems through the order and synchronization of messages. Neither paradigm is able to eliminate fully the problems that arise due to interference; it is possible to construct what is essentially a shared variable in message-passing systems, and to emulate message passing in a shared-variable system. This equivalence is well-known; Lauer and Needham demonstrated this in [LN79].

We concentrate on shared-variable systems, and the particular model used in this work is fixed by a fine-grained structural operational semantics. The adjective "fine-grained" is in reference to the smallest observable event in the semantic model; in particular, the model given in Chapter 2 is designed to mirror the behaviour of common concurrent languages.

To clarify the term "fine-grained" with an example, consider the pseudo-code fragment:

```
$x \leftarrow 1;$
$\big(x \leftarrow x + 1; x \leftarrow x + 1\big) \big\| \big(x \leftarrow x + x + x\big)$
```

There are at least four separate levels of granularity at which this code could be executed: each branch of the parallel executed as one step; each assignment as one step; assignments in two steps with expressions evaluated in one step; and assignments in multiple steps, with expressions taking at least one step per variable. We will consider each in turn.

At the coarsest level of granularity, with each branch of the parallel executed in one step, there are two possible final values for x: 5 and 9. The latter is a result of executing the left-hand branch first, and the former is a result of executing the right-hand branch first. This extremely coarse-grained interpretation is not particularly interesting as it allows no real opportunity for interference.

Statement-level granularity, with each assignment executing in a single step, allows for interference as execution of the statements in the parallel branches may interleave. The resulting values for x now include 7 in addition to 5 and 9, and the new value may occur when the right-hand branch executes between the two assignments in the left-hand branch.

Executing assignments as two steps gives a finer level of granularity, and adds 3 and 4 to the set of possible final values for x. The new values are obtained in the cases where an assignment has evaluated its expression before another assignment has finished; for example, if the first assignment of the left-hand branch and the assignment in the right-hand branch both evaluate their expressions –but not write the result to x– then the effect of one of the assignments will be lost.

The last interpretation –requiring at least one step per variable in the expression– has the finest granularity of this example, and adds 6 and 8 to the set of possible results for x. These results arise through the individual instances of variables in the expression being read in different overall states. It is possible, in this interpretation, for the right-hand branch to read x in such a way as to become $(x \leftarrow 1+2+3)$ simply by reading the first instance of x immediately, then each successive x after one of the assignments in the left-hand branch.

It should be noted that what is sometimes referred to as the "At Most One Assignments" rule –discussed in Section 9.2 of [Sch97]– does not allow for programs that we consider interesting. We are interested in programs that would violate this rule.

It is clear that the granularity of the model has a profound effect on the amount of interference that is possible during program execution. The semantic model in Chapter 2 allows the behaviour found in the fourth interpretation here, and so does the behaviour of common concurrent languages such as C and Java.

Using the semantic model of a language alone to reason about the behaviour found in fine-grained languages is possible, but it is not always simple. This provides the motivation for the use of development methods that can extend the model, in particular, rely/guarantee-style reasoning.

Rely/guarantee-style reasoning allows the behaviour and results of a program (or components thereof) to be reasoned about at an abstract level without needing to know the exact structure of the program. This is the style of framework we use in Chapter 3 to extend the semantic model of Chapter 2; this form of extension follows the ideas of [CM92] and [KNvO$^+$02].

The ability of rely/guarantee-style reasoning to compose and –in development– decompose programs is essential to the method's tractability. Most of the compositions (e.g. sequential and conditional execution) are straightforward, as the composed subprograms have no opportunities to interfere with each other. Parallel composition of programs, however, requires more care as the composed subprograms generally do interfere with each other. The characterization of the maximum interference between two processes allows us to determine whether or not it is safe to run two programs in parallel without needing to know any detail regarding how the programs execute. The *Comp-Par*, *Isolation-Par-L*, and *Isolation-Par-R* lemmas –covered briefly in Chapter 5– express this succinctly.

We present a second tool as a means of reasoning about the concurrent behaviour of programs in the form of an atomicity construct in the language model of Chapter 2 and accompanying development rules in Chapter 3. This atomicity construct is modelled on software transactional memory, which is useful as an alternative to locking protocols for resource management. The atomicity construct, as modelled, allows the contained program to assume that it is isolated from external interference; however, this benefit comes at the cost of potential restarts.

The extension of the semantic model of Chapter 2 with the rely/guarantee development rules in Chapter 3 has the potential for a meta-semantic gap which must be addressed. Indeed, the use of a development method –such as rely/guarantee development rules– with any semantic model has this issue. As a way of closing this meta-semantic gap, in Chapter 6 we use the lemmas in Chapter 5 to prove that each of our proposed development rules is sound with respect to our semantic model. Issues of logical completeness of the rely/guarantee development rules are deliberately untouched, except to note that the development rules presented herein are not complete, nor is completeness possible for the semantic model used here without losing the convergence property covered in Chapter 4, as well as the features that make rely/guarantee-style reasoning a tractable method.[1]

Much of this thesis is an elaboration of work done in [CJ07]; however, many of the details are improvements or refinements of the original ideas. The portions of this thesis that deal with software transactional memory are completely novel, as are the particulars of the convergence proofs. Also, a number of the rules in Chapter 3 have been improved relative to the versions in [CJ07].

Notational Conventions

This thesis does, for the most part, use the notational conventions and symbols common to VDM-style specification, especially as found in [Daw91]. Some the definitions of the less obvious operators are reproduced in Appendix E. Beyond this, however, there are some notational conventions that have been adopted in this work that are less common; a description of them follows in this section.

[1]See the discussion on "ghost variables" in Section 3.1.2.

Predicates in this work are denoted by the letter P, using subscripts where multiple predicates are necessary. Predicates are considered to be both a total function that maps the domain to the Boolean values, as well as the characteristic set containing all of the elements of the domain of the predicate for which the predicate maps to **true**. Thus, predicate application written as $[\![P]\!](\sigma)$ and the set membership test written as $\sigma \in [\![P]\!]$ have the same meaning (though we prefer the former to the latter). The use of Strachey brackets, $[\![\cdot]\!]$, around the predicate denotation arises from the common case where a predicate is defined in terms of an expression over individual identifiers in a state mapping.

The semantic model of the next chapter defines state objects in terms of a mapping from identifiers to values, and we use a shorthand notation in the predicate definition that has a variable identifier, say x, standing in for the application of the state mapping to the identifier, which would be $\sigma(x)$. Thus a predicate that is true for all states where the value x maps to is greater than zero might be written in definition as $P \triangleq x > 0$, and in use when applied to a state as either $[\![P]\!](\sigma)$ or $[\![x > 0]\!](\sigma)$.

Relations are pervasive in this work and typically denoted by the letters G, Q, R, and W where the relations range over state objects, and are often denoted as lettered arrows –such as $\xrightarrow{s}$– where it is a semantic transition relation. The semantic transition relations are defined in terms of inference rules; this is dealt with in Chapter 2. Relations over states are typically defined in terms of identifiers in the same manner as predicates. As with predicates, the notation for relations over states also treats the logical terms $[\![R]\!](\sigma, \sigma')$ and $(\sigma, \sigma') \in R$ as equivalent.

VDM post condition definitions tend to be over pairs of states, rather than the single-state post conditions found in standard Floyd/Hoare logics; related to this is the VDM convention of "hooking" identifiers that are to be given their value for the prior, left-hand state. Thus, if a relation is defined as $R \triangleq x < \overleftharpoon{x}$ –the value of x monotonically decreases– it can be taken to mean

$$[\![x < \overleftharpoon{x}]\!] \equiv \left\{(\overleftharpoon{\sigma}, \sigma) \in \Sigma \times \Sigma \;\middle|\; x \in \mathbf{dom}\,\overleftharpoon{\sigma} \wedge x \in \mathbf{dom}\,\sigma \wedge \sigma(x) < \overleftharpoon{\sigma}(x)\right\}$$

The standard map operators of VDM are used on relations where the definition is unambiguous in a relation context. Thus the **dom** and **rng** operators are used, but the override (†) operator is not. We define the field operator, **fld**, to be the union of the domain and range. Relational composition is denoted using the $\diamond$ glyph, and the usual definition is given in Appendix E.

Predicates may be used in the context of a relation directly. The denotation $[\![P]\!](\overleftharpoon{\sigma}, \sigma)$ is considered to be equivalent to $[\![P]\!](\sigma)$, and whole predicates may be hooked in a manner similar to identifiers, giving the denotation $[\![\overleftharpoon{P}]\!](\overleftharpoon{\sigma}, \sigma)$ equivalent meaning to $[\![P]\!](\overleftharpoon{\sigma})$. This allows logical statements that are a mixture of relations and predicates, such as $\overleftharpoon{P} \wedge R \;\Rightarrow\; P$ which is the equivalent of $[\![P]\!](\overleftharpoon{\sigma}) \wedge [\![R]\!](\overleftharpoon{\sigma}, \sigma) \;\Rightarrow\; [\![P]\!](\sigma)$.

The universal relation –which ranges over everything and includes every possible pair of things– is denoted simply as **true**, punning on the Boolean value.

The class of partial identity relations are denoted by the letter I, possibly with a meaningful subscript. An unadorned I is the complete identity relation. If there is a subscript on the glyph, such as I_s, the subscript must be a set of identifiers, and the relation is taken to range over state mappings. So the relation given as $[\![x = \overleftharpoon{x} \wedge y = \overleftharpoon{y}]\!]$ can be denoted as $I_{\{x,y\}}$, and in general a definition may be given as

$$I_s \equiv \left\{ (\overleftharpoon{\sigma}, \sigma) \in \Sigma \times \Sigma \,\middle|\, s \subseteq \textbf{dom}\, \overleftharpoon{\sigma} \wedge s \subseteq \textbf{dom}\, \sigma \wedge s \lhd \overleftharpoon{\sigma} = s \lhd \sigma \right\}$$

$\boxed{2}$ — Structural Operational Semantics

2.1 Introduction

The formalism chosen to present language semantics in this thesis is Structural Operational Semantics (SOS). The choice is motivated by a belief that the formalism presents a clear and concise description of a language's execution model that is directly useful to software designers.

Part of the justification for such a bold statement arises from the practice of using a SOS formalism: the emphasis is placed on what the language does –the execution model– rather than on the properties it exhibits.

Furthermore, the particular practice that SOS definitions have of using inference rules allows the model to be adapted to a reasoning framework in a direct manner — we will see more of this in Chapters 3 and 6.

The basic structure of a SOS model consists of four major parts: the abstract syntax; the context conditions; the semantic objects; and the semantic rules which define the transition relations.

The abstract syntax of a language gives the structure of all possible programs in the language by defining the "building blocks" that are used to construct a program. This is similar to the function that a BNF[1] specification serves, but without concern for the concrete syntax.

The abstract syntax does, of course, define a set of structures that can be a proper superset of all valid possible programs. To restrict this we have the context conditions which define the subset of possible structures which are valid programs. Context conditions are typically expressed as a predicate over the abstract syntax. It should be noted that context conditions have the same expressive power as the static analysis phase of a compiler, and essentially the same trade-offs between complexity and analytical ability.

The semantic objects provide the additional structures that are needed at "run-time"; these structures are involved in the dynamic behaviour of the language. These additions allow us to model the overall system of which the language is a part: this includes the abstract model of the memory store; anything else the language is able to manipulate; and structures which are simply a collection of simpler semantic objects. Thus, a tuple containing an entire description of a system at a specific point during execution –called a configuration– is also a semantic object.

Finally there are the transition relations as defined by their semantic rules. The transition relations are really the core of the whole model: they define the behaviour of a language by giving pairs of configurations. The pair of configurations represents an atomic step of the system, with respect to the system's observable behaviour. Because we use an unconstrained relation instead of a function, a given configuration may transition to several, possibly different, new configurations — this is critical for developing a model of

[1]Backus-Naur Form

non-deterministic language structures.

2.2 The Language

This section has two goals: first, to clarify and expand on the preceding introduction with a concrete example of a language definition; and second, to present the language that underlies the development rules and proofs that follow in Chapters 3 and 6. That said, this section only presents the relatively standard portion of the language: specifically, the construct that allows for arbitrarily large atomic actions is presented in Section 2.3 along with a general discussion of software transactional memory.

The language has been kept to a minimum of orthogonal elements as its purpose is to illustrate some of the more vexing problems faced by programmers using common multi-threaded programming languages. As such, many of the problems with interference can be expressed in this language in a simplified form; though issues such as aliasing, complex data structures, and so on are interesting, they are not required to show the basic issues involved in shared-variable concurrency languages and the development of programs therein.

To achieve this goal of illustrating the fundamental problems, this language is designed to allow a high degree of interference between parallel threads. As this language allows shared-variable concurrency, this necessitates expressions that are evaluated over multiple steps so that interference can be observed during evaluation. This, in turn, gives rise to a semantic model with very "small" transitions: each one does little on its own. Expression evaluation in languages like C, C++, and Java is very fine-grained: it is possible for the value of a variable to change during the evaluation of an expression that contains that variable. In pathological cases, this means that reading from the same variable twice in an expression has the potential to return two different results. The obvious result of this is that comparing a variable to itself –something which, intuitively, should give an equality result– may end up indicating that the variable is not equivalent to itself. Less obviously, this can generate hard-to-reproduce errors in a program due to variable read/write timing. The transitions in this language capture the sort of behaviour exhibited by languages like C, C++, and Java by allowing the system state to change between reads of a variable.

The concurrency model of the language is based on the notion of interleaving execution steps. A program in the language is a tree-like structure and the construct that allows parallel behaviour represents its concurrent sub-programs simply as branches. A single transition of the language semantics may only make changes at the leaves of a program's syntax tree. Upon encountering a parallel construct a non-deterministic choice is made as to which branch is followed in that transition; during the next transition, a fresh choice will be made as to which branch to follow. This non-determinism matches the model presented to the programmer in current languages as there is no general way to predict which thread will execute next.

On physical machines that do not permit simultaneous actions, the semantic model of the language maps closely to the actual execution models of current languages. On machines that do permit simultaneous actions –such as current multicore CPU machines– this semantic model fits as the physical machine still enforces consistency at the variable

```
$wf\text{-}Expr\colon (Expr \times Id\text{-}\mathbf{set}) \to \{\textsc{Bool}, \textsc{Int}, \textsc{Error}\}$
$wf\text{-}Expr(e, ids) \triangleq$
    cases $e$ of
                          $e \in \mathbb{B} \to \textsc{Bool}$
                          $e \in \mathbb{Z} \to \textsc{Int}$
              $e \in Id \wedge e \in ids \to \textsc{Int}$
        $mk\text{-}Dyad(op, left, right) \to$ let $ltype = wf\text{-}Expr(left, ids)$ in
                                   if $ltype = wf\text{-}Expr(right, ids) \wedge$
                                      $ltype \neq \textsc{Error}$
                                   then cases $(op, ltype)$ of
                                           $(+, \textsc{Int}) \to \textsc{Int}$
                                           $(-, \textsc{Int}) \to \textsc{Int}$
                                           $(<, \textsc{Int}) \to \textsc{Bool}$
                                           $(=, \textsc{Int}) \to \textsc{Bool}$
                                           $(>, \textsc{Int}) \to \textsc{Bool}$
                                          $(=, \textsc{Bool}) \to \textsc{Bool}$
                                          $(\wedge, \textsc{Bool}) \to \textsc{Bool}$
                                          $(\vee, \textsc{Bool}) \to \textsc{Bool}$
                                          others $\textsc{Error}$
                                          end
                                   else $\textsc{Error}$
    others $\textsc{Error}$
    end
```

Figure 2.1: Context condition for expressions.

read/write level. The resolution of two simultaneous writes to the same variable is simply modelled in the semantics by two writes happening consecutively.

Several things have been omitted from this language as they distract from the focus on shared-variable concurrency and software transactional memory. First, the memory store is global and static: there are no hiding mechanisms that would create local variables; nor are there any mechanisms to create or destroy variables during program execution. These mechanisms are not required for the rely/guarantee development that is done later in this work. Procedures and functions –callable program blocks with and without side-effects, respectively– are not a part of the language. Their introduction poses no serious difficulty, but does generate a lot of unnecessary bookkeeping. Finally, input and output from sources external to the program are simply not included as there is no use for them in the context of this work.

The remainder of this section will examine the language in detail by covering the expressions, then each statement in turn. As the constructs are covered, their interactions with the already-described portion of the language will also be noted.

Expressions in the language are represented by the $Expr$ type.

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

The most basic elements of an expression are the members of the Boolean and Integer sets. An expression that has been reduced to one of these elements is considered to be fully evaluated, and as such, is always well formed; the well-formedness function in Figure 2.1, $wf\text{-}Expr$, reflects this.

The well-formedness function is –unusually for a well-formedness function– not a predicate. It exploits the fact that all valid expressions are typed and, conversely, all invalid expressions do not, by definition, have a type. Thus the $\mathit{wf}\text{-}\mathit{Expr}$ function returns the type of an expression if the expression is well-formed, and ERROR otherwise. Note also that $\mathit{wf}\text{-}\mathit{Expr}$ is a total function as it is defined on all possible members of Expr. It is trivial to generate a predicate using $\mathit{wf}\text{-}\mathit{Expr}$ by wrapping it in a lambda expression:

$$\lambda e, \mathit{ids} \cdot (\mathit{wf}\text{-}\mathit{Expr}(e, \mathit{ids}) \neq \textsc{Error})$$

Identifiers in an expression, though terminal with respect to the structure of said expression, are only valid and well-formed if that particular identifier is in the context in which the expression will be evaluated. In terms of the context condition that means that the identifier must be a member of the set of valid identifiers, ids.

Assuming that the identifier is in ids, the context condition will return INT, indicating that not only is this a valid expression, but also that it should be considered an integer for expression evaluation. The well-formedness function returns INT for identifiers as the language semantics uses a memory store that only contains integer values. If, on the other hand, the identifier is not in ids, then the context condition will return ERROR.

The language semantics assumes a memory store that only contains integer values, and further, it is a simple mapping from identifiers to values. This drastic simplification of the memory store allows us to focus on actions, rather than values, when the store is involved in a construct's behaviour. In this work Σ will be used to represent the set of all possible stores, and variations on σ will represent specific stores.

$\Sigma = \mathit{Id} \xrightarrow{m} \mathbb{Z}$

The semantic transition for expressions, $\xrightarrow{e}$, is a relation between pairs and expressions, where the pair consists of an expression and a state. The transition represents simple expression reduction by either reading the value of an identifier from the store or performing operations on constant values.

$$\xrightarrow{e} : (\mathit{Expr} \times \Sigma) \times \mathit{Expr}$$

The semantics of an identifier in the language is encapsulated in a single rule of the expression transition relation, $\xrightarrow{e}$. This rule takes a pair consisting of an identifier and a store and simply applies the store to the identifier to retrieve a value. The important effect of this rule is that in this language variable reads are atomic, as the whole read is done in a single step.

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

The last component of the Expr type is the Dyad structure, representing binary operations.

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

The *op* field of the *Dyad* indicates which operation is to be done, and the *left* and *right* fields contain expressions which will yield the values which will be operated upon. This language deliberately contains a limited number of defined operations; adding more to the language is straightforward.

The portion of *wf-Expr* that deals with the *Dyad*s looks a bit involved, but is mostly bookkeeping to ensure that the subexpressions are both well-formed and of the correct type for the indicated operation. Note that the resulting type of an operation is solely dependent on which operation it is, and completely independent of the types of the subexpressions.

$$\boxed{\text{Dyad-L}}\ \frac{(\mathit{left}, \sigma) \xrightarrow{e} \mathit{left}'}{(\mathit{mk\text{-}Dyad}(\mathit{op}, \mathit{left}, \mathit{right}), \sigma) \xrightarrow{e} \mathit{mk\text{-}Dyad}(\mathit{op}, \mathit{left}', \mathit{right})}$$

$$\boxed{\text{Dyad-R}}\ \frac{(\mathit{right}, \sigma) \xrightarrow{e} \mathit{right}'}{(\mathit{mk\text{-}Dyad}(\mathit{op}, \mathit{left}, \mathit{right}), \sigma) \xrightarrow{e} \mathit{mk\text{-}Dyad}(\mathit{op}, \mathit{left}, \mathit{right}')}$$

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

The three rules required to define the evaluation of a *Dyad* exhibit some fascinating properties that are fundamental to the behaviour of the overall language. First, note that the choice between the Dyad-L and Dyad-R rules is non-deterministic. The non-deterministic choice between the left and right subexpressions means that the order in which the variables are read is unconstrained by the language semantics. In turn, this means that any proofs done using the semantics must not depend on any particular order of evaluation.

The elimination of the *Dyad* (and, thus, the actual performance of the indicated operation) cannot happen until both the *left* and *right* operands have been fully evaluated — as is shown in the antecedent of the Dyad-E rule. The rule is just a simple reduction of the *Dyad*, through the meaning of the operation (i.e. $[\![op]\!]$), to a final value.

An examination of the type signature of the expression transition reveals that expressions in this language cannot produce any side-effects as a result of evaluation. Preventing this is both an absence of side-effect producing constructs in the *Expr* type, as well as the fact that the transition does not have a store component on the right-hand side of the arrow, meaning that there is nowhere to denote a modified store.

There are eight kinds of statement in this language, and they are enumerated in the *Stmt* type.

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

Each of these kinds will be described in turn, but their explanation is presented best in the context of the whole system.

The overall semantic transition in a SOS is a relation over configurations of the system, and this relation defines the entire behaviour of that system. A configuration is, in the case of this language, simply a pair consisting of a statement and a store. The relation for this language, $\xrightarrow{s}$, is symmetrical in that its "right-hand" object is also a configuration; the symmetry is not strictly necessary, but it does simplify the definition of the transitive

```
$wf\text{-}Stmt\colon (Stmt \times Id\text{-}\mathbf{set}) \to \mathbb{B}$
$wf\text{-}Stmt(stmt, ids) \triangleq$
    **cases** $stmt$ **of**
                               **nil** $\to$ **true**
              $mk\text{-}Assign(id, e) \to id \in ids \wedge wf\text{-}Expr(e, ids) = \textsc{Int}$
              $mk\text{-}Atomic(body) \to wf\text{-}Stmt(body, ids)$
                $mk\text{-}If(b, body) \to wf\text{-}Expr(b, ids) = \textsc{Bool} \wedge$
                                          $wf\text{-}Stmt(body, ids)$
           $mk\text{-}Par(left, right) \to wf\text{-}Stmt(left, ids) \wedge$
                                          $wf\text{-}Stmt(right, ids)$
           $mk\text{-}Seq(left, right) \to wf\text{-}Stmt(left, ids) \wedge$
                                          $wf\text{-}Stmt(right, ids)$
  $mk\text{-}STM(orig, \sigma_0, body, \sigma) \to wf\text{-}Stmt(orig, ids) \wedge$
                                          $wf\text{-}Stmt(body, ids) \wedge$
                                          $\mathbf{dom}\, \sigma_0 = \mathbf{dom}\, \sigma = ids$
            $mk\text{-}While(b, body) \to wf\text{-}Expr(b, ids) = \textsc{Bool} \wedge$
                                          $wf\text{-}Stmt(body, ids)$
    **others false**
    **end**
```

Figure 2.2: Context conditions for statements.

closure of the transition relation, which in turn makes it easier to reason about multiple transitions.

$Config = Stmt \times \Sigma$

$$\xrightarrow{s} : Config \times Config$$

The easiest/simplest statement is **nil** — pragmatically it is just the representation of "nothing left to do". The **nil** statement is a completed program; it has no internal structure of its own. By definition it is well formed, and the applicable portion of $wf\text{-}Stmt$ in Figure 2.2 reflects this. As it represents a completed program, there are no semantic rules for this statement (i.e. there is never a configuration composed of **nil** and some state that is in the domain of the transition relation).

The $Assign$ construct is the only other non-recursive construct (in the sense of not containing another statement) in the $Stmt$ type. Coincidentally, and conveniently, it is also the only construct which can directly modify the store.

$$\begin{aligned} Assign \;::\; id &: Id \\ e &: Expr \end{aligned}$$

The construct has two fields: id, which identifies the element in the domain of the store for which the range will be changed; and e, which contains the expression which, when evaluated, will yield the replacement value. The portion of the context condition for the $Assign$ construct in Figure 2.2 requires that the target identifier be within the set of valid identifiers, and that the expression be valid and yield an integer.

The design choice that this language exhibit fine-grained interference behaviours requires that there be two semantic rules to define the transitions that are possible from an

$Assign$ construct.

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

The first rule, Assign-Eval, models the process of expression evaluation within the construct. As such, it takes as an antecedent one transition step of the expression relation, using the initial expression and the current store as the left-hand parameter. The right-hand parameter –a reduced form of the initial expression– is used to form the overall right-hand system configuration by replacing the initial expression in the left-hand system configuration.

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

The second $Assign$ rule, Assign-E, is where the store modification actually happens. Once the expression has been reduced to a value it is no longer possible to perform a $\xrightarrow{e}$ transition on it. Thus, all that remains to be done is actually change the value of the store at the target identifier to the value of the expression.

The semantics of $Assign$ as presented result in a gap between the last *Assign-Eval* transition and the final *Assign-E* transition. The gap allows transitions that apply to other threads within the overall program to modify the memory store; this can lead to situations where the target variable is mutated long after the expression has completed evaluation.

The language does, of course, provide a construct for conditional execution.

$$\begin{aligned} If \ :: \quad b &: Expr \\ body &: Stmt \end{aligned}$$

The *If* construct is a basic one-branch conditional, allowing its body to be executed only in the case where the test, b, has evaluated to **true**. The portion of the context condition in Figure 2.2 requires that the test be both well-formed and evaluate to a Boolean value, and that the body be well-formed within the same context as the overall *If*.

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

Like the $Assign$ construct, the decision to have the language behave with a fine-grained semantics for expression evaluation means that the definition of the rules for an *If* construct includes a rule that evaluates the test expression one step at a time. It is, therefore, possible to interfere with the store between these evaluation steps.

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

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

Once the test has been completely evaluated to a Boolean value, it remains to reduce the whole *If* construct to either just the contained *body* if the test evaluated to **true**, or to

nil if the test evaluated to **false**.

Looping in this language is achieved through the *While* construct, which has the same structure as the *If* construct.

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

Not surprisingly, this construct has the same requirements in its portion of the context condition in Figure 2.2 as that of *If*: the test must be well formed and Boolean valued, and the body must also be well formed.

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

The semantic rule for the *While*, however, is of a very different form than those for *If*. The semantics of the while loop are almost completely dependent on that of the *If* construct. What the While rule does is rewrite the program, "unrolling" the loop once; the result forms the new configuration. The resultant *If* construct is evaluated normally: if the test evaluates to **true**, then the body is executed once and we have the original loop construct, ready to be unrolled and tested again; if the test evaluates to **false**, then the whole structure just reduces to **nil**. This rule is the simplest means of specifying the behaviour of a while loop in a fine-grained semantics.

As an aside, it is not possible to "infinitely expand" the *While* construct using this rule. When the construct has been unrolled once, the usual While rule cannot be applied again unless the test evaluates to true and execution of the body has completed. Once that has happened, however, applying the While rule again is precisely the correct thing to do.

Sequential composition of statements in the language is provided by the *Seq* construct.

$$\begin{array}{lrcl} \mathit{Seq} :: & \mathit{left} & : & \mathit{Stmt} \\ & \mathit{right} & : & \mathit{Stmt} \end{array}$$

There is nothing surprising in this construct's context condition: it simply requires that both the *left* and *right* contained statements are well formed. The semantic rules for this construct are straightforward.

$$\boxed{\textsf{Seq-Step}}\ \frac{(\mathit{left}, \sigma) \xrightarrow{s} (\mathit{left}', \sigma')}{(\mathit{mk\text{-}Seq}(\mathit{left}, \mathit{right}), \sigma) \xrightarrow{s} (\mathit{mk\text{-}Seq}(\mathit{left}', \mathit{right}), \sigma')}$$

$$\boxed{\textsf{Seq-E}}\ \frac{}{(\mathit{mk\text{-}Seq}(\mathbf{nil}, \mathit{right}), \sigma) \xrightarrow{s} (\mathit{right}, \sigma)}$$

The first, Seq-Step, just performs one step of the *left* statement and then wraps the result of that inside a *Seq* formed with the *right* statement from the original. Once the *left* statement has reduced to **nil**, the Seq-E rule then just unwraps the *right* statement, leaving it on its own. The *right* statement is then executed in the normal manner, but without the now-extraneous structure of the original *Seq* around it.

Parallel composition of statements is an important feature of the language and is supported via the *Par* construct.

$$\begin{array}{lrcl} Par & :: \quad left & : & Stmt \\ & right & : & Stmt \end{array}$$

Structurally, Par is almost exactly the same as Seq — the only difference is in the name. It should be no surprise that the context condition for this construct is the same as that for Seq.

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

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

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

The semantic rules for Par, though similar, define very different behaviour. The Par-L and Par-R rules have a similar effect as the Seq-Step rule in that they take a sub-statement, execute one step, then rebuild the construct using the newly reduced sub-statement and parts from the original. That there are two rules –one for each of the $left$ and $right$ fields of the Par– that could both be applied to a given Par as the source of both this construct's parallel behaviour and its non-determinacy. The simple fact that there are configurations to which both rules could be applied, and that the rules only execute a single step, is what allows the interference to play a role in the system.

Once both of the $left$ and $right$ sub-statements have reduced to **nil** then the whole Par construct can be reduced to **nil**. This is in contrast to the Seq construct, which eliminates the Seq as soon as the left sub-statement has finished.

The remaining constructs –$Atomic$ and STM– are covered in the next section, as they represent atomic blocks as implemented by a form of software transactional memory.

2.3 Software Transactional Memory

Software Transactional Memory (STM) can be loosely described as an attempt to adapt the database transactional model for use as a programming construct available for use in shared memory-based algorithms. Semantic models of atomic blocks based on STM mechanisms give systems that are conceptually simpler than lock-based models in terms of reasoning, and these models are a good fit for use with the Rely/Guarantee formalism we will examine in Chapter 3.

In general this transaction model allows for a set of state-modifying actions to be grouped together such that actions excluded from said group only see the results of that group's actions either all together or not at all. In a database environment, these actions may involve updates to several tables/rows. In a STM transaction, a more likely example could be the result of swapping the values of two variables.

Software transactional memory, then, as the name indicates, operates on blocks of memory — the system store. The necessary operations of STM are *commit* and *retry*; there is also the potential for *abort*, though the last is not treated in this work. There is also the ability to mark the beginning and end of a STM, of course, but that is really a structural notion rather than an operation.

Initialization of a transaction generally involves making a copy of the portions of the state on which the transaction depends. For a very conservative transaction system, this could potentially be the whole state. A more optimistic system would record only the values on which the transaction depends, and this could be shown to be behaviourally equivalent to the conservative model, given that the transaction would abort only on interference. A very optimistic system might do nothing at initialization, and only record values of variables at their first read.

The semantics for the STM construct in this language is a fairly conservative one in that it captures the values of all of the variables in the store before the body of the transaction begins execution. It is not maximally pessimistic, however, as it does allow the transaction to continue execution even when variables unrelated to the transaction are changed. It does, however, enforce a sort of temporal consistency over the values of the variables by not being so optimistic as to capture only the value of a variable at the point of first use.

At a high level of abstraction this model of the STM construct allows the user of the language to depend on the fiction of atomicity. Any program placed inside the STM construct will appear to produce all of its changes to the state in a single transition; however, at the operational level the semantics merely holds the alterations aside until the program is complete.

The *commit* operation ends the current, innermost transaction and allows the changes that have been made to the store to become visible to the containing scope. Other concurrently running transactions could, potentially, be invalidated by this operation if they depended on variables that had been modified by the transaction. At the point where a transaction commits, it does need to ensure that the changes it would make are still valid — this is done by checking that the variables on which that transaction depends have not changed in the interim.

Though the language in this work does not have an equivalent to the *abort* operation, the effect of the *abort* operation is useful to illustrate the *retry* operation which will follow. The *abort* operation is a sort of complement to *commit* in that it provides a means of exiting a transaction, but instead of allowing that transaction's changes to become visible, it just throws them away. The state after a transaction aborts is unaffected by an aborted transaction's exit, and is exactly the same as the state immediately prior to that exit. As this operation is an exit, it places control flow at the end of the transaction block.

Last of the operations, *retry*, is similar to *abort* in that it also throws away any changes that the transaction would have made; however, unlike *abort*, *retry* also restarts the transaction from its beginning. In one respect the *retry* operation can act like a loop, causing the transaction body to be executed multiple times. Unlike a loop, though, if the transaction is retried, there is no change to the external state due to execution of the body.

From this description it is clear that there are strict restrictions on the domains in which STM can be deployed. Roughly, it can only be deployed on things whose effects are fully reversible and concealable: usually this restriction allows only pure memory operations. Thus, blocks of actions that include input/output, human interaction, or message passing outside of the transaction are banned. The semantic model provided is restricted to pure memory operations, which allows us to focus on the immediate effects of this STM model.

Of the traditional ACID –Atomic, Consistent, Isolated, Durable– properties that are used in the database literature, the most important for STM systems is isolation. Through that property this mechanism can achieve the appearance of atomicity in its behaviour with respect to external processes; consistency relative to conditions on the behaviour of the process (see Chapter 3); and as much durability as is possible in a volatile memory store.

The isolation property is inherent in STM transactions as the very point of this mechanism is to hide the changes the transaction makes until it commits, as well as ensuring that the transaction body will not "see" external changes at all. This isolates the changes the transaction makes from its environment; in the other direction, the transaction is isolated from changes through the *retry* operation. In the STM language considered in the next Section, the *retry* operation is automatically triggered by an external change — this means that while the overall transaction will arguably become aware of the change, its reaction will be to restart the transaction's body to preserve the *appearance* of executing in a stable environment.

Related to the effect of *retry* and how it achieves isolation is the topic of interference. As with post conditions and guarantees, this is dealt with in detail in Chapter 3; however, described in terms of interference, a transaction reacts by restarting, and so the body of the transaction can effectively rely on there being no interference.

The STM model –in general and in the particular model used here– has the ability to nest STM structures within larger STM structures. This allows for a clean compositional model and eliminates any need for special rules in the semantics. Nesting STM structures is useful in the semantics as it is possible for a parallel construct to be placed within an atomic construct; though the contained parallel construct has no interference, the branches of the parallel construct may still interfere with each other.

2.3.1 Atomic/STM Language Elements

This section introduces the remaining constructs of the language, $Atomic$ and STM, which give a model of the operation of one form of software transactional memory used to implement atomic blocks.

$$Atomic :: body : Stmt$$

$$\begin{array}{rrcl} STM :: & orig & : & Stmt \\ & \sigma_0 & : & \Sigma \\ & body & : & Stmt \\ & \sigma & : & \Sigma \end{array}$$

Pragmatically, the STM construct is not intended to appear in a program that has not performed any execution at all, but for the sake of convenience, it has been included in the abstract syntax. The STM construct is very unusual in that it contains semantic objects –named σ_0 and σ– in its construction.

In the language model there is a fairly clear separation between the parts of the configuration that represent dynamic, changeable information about the system, and the parts that, though they generally reduce, can be considered static. To wit, the static part is the program text –the statement– and the dynamic part is the memory store. The design of the language model carefully keeps these two components separate, and this distinction is greatly helped by the lack of block-like scoping construct. The only point where there is an arguable violation of this separation is in the expressions, as their evaluation effectively records data about the previous states. The argument fails in that to extract this history information you also need to know the prior, unevaluated expressions which were reduced to the current construct. And, since a user of the language could –in all cases– have programmed the partially evaluated expression indirectly, it becomes moot.

The static/dynamic distinction is violated by the STM construct. It contains two elements of the type used to represent the memory store, which is inherently dynamic. Of these two elements, the first, σ_0, records the state of the memory store when the transaction is initialized. Admittedly, this value is never altered during the lifetime of the STM construct, but it is used in all of the rules specific to that construct, and it is, nonetheless, a dynamic object. The second, σ, is the delta mapping, and there is no question that this element does behave as a dynamic object — its purpose is to accumulate the dynamic changes to the state so that they may be isolated from the environment. It would be difficult for a user of this language to program a STM construct directly, though there is nothing that expressly forbids it.

The semantic model of this language just "bites the bullet" and includes the dynamic state as part of the (otherwise) static program text. Part of the justification is that the program text represents the work that remains to be done, and part is a practical observation on the difficulty of tying specific pieces of a dynamic state to a specific sub-tree of the program text.

The structure of the language as designed includes the STM construct as part of the $Stmt$ type. The alternative would have been to make the STM construct properly a semantic object rather than a part of the abstract syntax. An issue with this approach is the fact that to construct a configuration we then have to "lift" the statement type to include a semantic object. Notationally, that is rather awkward — it has to be taken as implicitly given that all of the fields that were just statements now have a broader type. Furthermore, lifting the statements to include semantic objects means that the context conditions have a narrower range of applicability — unnecessarily so.

Having noted this mixing of static and dynamic data in the STM language, we can move to the first of the rules that require it.

$$\boxed{\text{STM-Atomic}}\;\frac{}{(mk\text{-}Atomic(body), \sigma) \xrightarrow{s} (mk\text{-}STM(body, \sigma, body, \sigma), \sigma)}$$

The STM-Atomic rule is essentially just an initialization step, setting up the STM construct so that the language can actually execute the transaction. The body of the $Atomic$ construct and the initial state, σ, are used for the first two fields of the STM, establishing the original text of the transaction, and the state against which external changes will be checked. The last two fields are also filled with the original body and state, but these fields will change as the STM is reduced. The third field –holding the copy of the original body that will change– records the remaining work to be done as the transaction is executed. The last field is the delta mapping, and is used to record the changes to the original state which are made by the body as it is executed.

$$\boxed{\text{STM-Step}}\ \frac{(\mathit{Vars}(\mathit{orig}) \lhd \sigma_0) = (\mathit{Vars}(\mathit{orig}) \lhd \sigma) \qquad (\mathit{body}, \sigma_s) \xrightarrow{s} (\mathit{body}', \sigma_s')}{(\mathit{mk\text{-}STM}(\mathit{orig}, \sigma_0, \mathit{body}, \sigma_s), \sigma) \xrightarrow{s} (\mathit{mk\text{-}STM}(\mathit{orig}, \sigma_0, \mathit{body}', \sigma_s'), \sigma)}$$

A single step of a transaction has a surprising amount of bookkeeping to do at this level of semantic description. The first antecedent of *STM-Step* establishes that the variables the transaction depends upon have not changed since the beginning of that transaction; it does this by comparing the original state to the current state, but only on the variables found in the original body. The second antecedent provides the form of the result of *one* step of execution of the body, but to do so it must set the state in which the body executes to that of the delta mapping held by the STM construct, ignoring the external state entirely. In the resulting STM object the delta mapping is replaced by the target state provided by the second antecedent, and the body is replaced from the same source. The target state of the overall transition is unchanged from the source: this gives the construct the property of completely isolating the behaviour of the body of the STM object from its environment.

$$\boxed{\text{STM-E}}\ \frac{(\mathit{Vars}(\mathit{orig}) \lhd \sigma_0) = (\mathit{Vars}(\mathit{orig}) \lhd \sigma)}{(\mathit{mk\text{-}STM}(\mathit{orig}, \sigma_0, \textbf{nil}, \sigma_s), \sigma) \xrightarrow{s} (\textbf{nil}, \sigma \dagger (\mathit{Vars}(\mathit{orig}) \lhd \sigma_s))}$$

The STM-E rule represents the *commit* operation of a transaction. Its use is implicit, triggered by having a **nil** body, and only requires that none of the variables in the delta mapping (as restricted by the variables in the original body) have changed in the external state. If that antecedent holds, then the resulting configuration has a **nil** program text, the target state is the source state overwritten on the variables of the original body from the delta mapping. This causes the changes made by the execution of the body to become visible outside that transaction.

$$\boxed{\text{STM-Retry}}\ \frac{(\mathit{Vars}(\mathit{orig}) \lhd \sigma_0) \neq (\mathit{Vars}(\mathit{orig}) \lhd \sigma)}{(\mathit{mk\text{-}STM}(\mathit{orig}, \sigma_0, \mathit{body}, \sigma_s), \sigma) \xrightarrow{s} (\mathit{mk\text{-}Atomic}(\mathit{orig}), \sigma)}$$

The last rule specific to STM in this semantic model is the STM-Retry rule; it deals with the *retry* operation. A transaction should only need to retry if some part of the state on which it is dependent has been altered. This requirement is captured in the antecedent, allowing this rule to fire in precisely that case; note that STM-Step and STM-E cannot fire in those cases. When this rule does hold, however, we want the transaction to throw away all of the changes it would have made and revert to a configuration that is ready to attempt execution again. Thus, the right-hand configuration is now the original $Atomic$ construct that the STM object started out with.

This language has no inbuilt notion of error recovery. If the externally dependent state changes during a transaction, then the transaction *must* retry: there is no provision for the program to examine the external state dynamically and determine that it is harmless.

2.4 Constraints on the Semantic Model

The SOS model presented in this chapter conforms to a number of constraints in its design which are necessary for the construction of the rely/guarantee rules of Chapter 3 and their corresponding soundness proofs in Chapter 6.

The domain of the overall semantic relation includes all configurations which consist of a program which satisfies the $wf\text{-}Stmt$ predicate (Figure 2.2) and a suitable state,[2] and all of the configurations transitively reachable from the well-formed program configurations, minus those configurations whose program text is **nil**. Verifying that this is true for the semantic model given can be done by first checking that every construct has at least one rule, and then checking that the rules for each given construct cover the set of possible well-formed constructs. This constraint on the semantic relation has the effect of eliminating concerns about the definedness of the semantic relation in the context of this work.

As the overall semantic relation is essentially the union of its semantic rules we must take care to ensure that the individual rules do not conflict with each other. Though we support non-determinism in the model at the rule level by allowing multiple rules to apply to the same configuration –that is, we allow the domains of individual rules to overlap– we must be careful to ensure that no two rules allow the same transition. Allowing this leads to an ambiguity which would invalidate a step in the proofs of Chapter 6 as we depend on being able to identify the semantic rule to which a particular transition belongs.

There is a similar constraint to the ambiguity mentioned above which anticipates the augmented semantics introduced in Section 4.2: the semantic relation must not have any steps which do not alter the program component of a configuration. All of the transitions in the semantic model must alter the program component; introducing a transition which does not do so prevents the augmented semantics from being able to distinguish between program transitions and interference transitions.

The semantic rules which make up the semantic model used in this work can be grouped into two major themes: elimination rules, which replace the top-level construct in a configuration with either one of its components or **nil**; and wrapping rules, which wrap the behaviour of a component in the top-level construct. There are two rules –While and STM-Atomic– which do not fall into these categories, however: they replace the top-level construct with another construct entirely. In an unrestricted form rules which rewrite the program in the configuration can make reasoning about the semantic model extraordinarily difficult. The two rules mentioned here do so only in a superficial manner, however, and do not alter any of the components of the original construct; furthermore, they are only applicable in very constrained situations.

The addition of new constructs into the language poses no special problems so long as the accompanying semantic rules collectively cover the reachable instantiations of the construct (i.e. those accepted by the then-modified $wf\text{-}Stmt$ predicate and those transitively reachable from that initial set). Also –per the constraint on ambiguity– the newly

[2] I.e. a state with a domain which contains the set of all of the identifiers used in the program.

introduced rules must not produce any transition which is already in the semantic model. Removing a construct from the language requires that we check that, once again, all reachable remaining constructs are still in the domain of the now-reduced semantic relation. Modification of the structure or behaviour of a construct is essentially equivalent to removing the old form of the construct and then adding the new form of the construct.

There is an idiosyncrasy of the $While$ construct in this particular semantic model: the While rule gives the behaviour of the construct in terms of the If and Seq constructs. This design choice was made for the sake of clarity despite the dependency it introduces. Modelling the behaviour of the $While$ construct could have been done through the use of an auxiliary construct in much the same way as the behaviour of the $Atomic$ construct is given using the STM construct. Doing so would eliminate the dependence on constructs which are conceptually separate from the $While$, but would add little to the exposition. However, as the dependence is present in the model there is a practical dependency in the rest of the thesis in that results regarding the $While$ construct depend on the If and Seq remaining unaltered. Given that the semantic model is regarded as fixed in the rest of this work, this dependency is not a problem.

2.5 References

Structural Operational Semantics

McCarthy proposed abstract interpreters for programming languages in [McC63], and applied it to micro-ALGOL in [McC66]. The latter is contained in [Ste66], which also contains a fascinating record of the discussion that followed the presentation of the paper. The critical idea in this, however, is that language semantics can be given by describing the effect of the language's construct on a state — this is the essence of operational semantics. As micro-ALGOL is a deterministic language the abstract interpreter could be written as a simple recursive function; languages with non-deterministic constructs require more complex approaches. This notion of operational semantics as inspired by McCarthy was applied to PL/I in work by IBM Vienna group; the definitions gave rise to massive technical reports — the most useful overview is in the paper by Lucas and Walk [LW69]. Though an impressive piece of work, this semantics was described using VDL, and the notation did not lend itself to straightforward reasoning techniques.

The work in this chapter follows the tradition of structural operational semantics as introduced by Plotkin in what has come to be referred to as his Århus Notes [Plo81]. The Århus Notes have since been edited, corrected and republished in [Plo04b]. Plotkin's work is the basic starting point for SOS and the examples contained therein have an implicit emphasis on modelling the language rather than proving properties about the language.

The origins of SOS are recounted in [Plo04a] where the links between SOS and denotational semantics in the Scott-Strachey style [Sto77] are noted (among other connections). A shorter summary of the trends in language semantics can be found in Jones [Jon03b] which also includes a set of examples for modelling various language concepts in SOS. Jones also mentions the connection between the VDL semantics of PL/I mentioned above and its connection to McCarthy's abstract interpreters. The links between VDL and the denotational semantic aspects of VDM are described in [Jon01].

There are two notable texts that introduce SOS: the first by Riis Nielson and Nielson is [RNN07] (which is an updated version of [RNN92]), and the second by Winskel is [Win93]. The former is arguably the better introduction to SOS, but both suffer –for the current purpose– from the fact that the texts as a whole are more interested in program correctness and verification than they are in modelling. This focus allows the texts to also cover denotational and axiomatic semantics in an integrated way, but unfortunately ends up using very coarse-grained examples which are outside of the area where SOS really shines.

Software Transactional Memory

Software transactional memory has its roots in hardware transactional memory architectures, in particular work by Herlihy and Moss [HM93]. The first proposal to implement transactional memory in software was by Shavit and Touitou in [ST95]. Shavit and Touitou specifically extend Herlihy and Moss' work to the software domain and propose a detailed pseudo-code implementation for it. Unfortunately, as their implementation was intended to be done on top of an existing language, there are no real semantic underpinnings to their work. Their implementation also required that the set of objects affected by the STM transaction was statically declared.

Later work by Herlihy et al. [HLMS03] gives the first implementation of dynamic STM, removing the need to declare the transactional objects in advance. The objects affected by the STM in their work must be controlled by a container object, however, which does leave some limits in place. The implementation of their STM mechanism is done by a software library –rather than integrated into the language– which means that the actual semantics of the STM is defined by the code rather than a proper formal description.

Parallel work by Harris and Fraser [HF03] gives a STM mechanism which appears to be integrated into the language through use of Hoare's conditional critical regions [Hoa72]. This version of STM is intriguing as it adds a language construct to the Java language, and uses a modified compiler and virtual machine to actually implement the STM mechanism. The lack of formal semantics, however, makes it difficult to see precisely how the mechanism works.

The paper that has had the most influence on the $Atomic/STM$ semantics in this chapter is [HMPJH05]. Critically to this thesis, the work of Harris et al. is the first to propose STM as a fully integrated language construct — all but one of the other papers propose implementing STM as a software library in the language. As there is a proper operational semantics in [HMPJH05] (albeit in a slightly different style than the SOS used here), a brief comparison between their STM design and the one used in this chapter follows. It should be noted, first, that their semantics has been written in the context of the Haskell language, which, being a purely functional language, imposes a different set of constraints than are present in the imperative language presented in this chapter.

The first major difference between the two semantic models is that theirs is presented at a much coarser level. Not only are transitive steps used in their semantic rules directly, but their semantics explicitly disallows the interleaving of the steps of an atomic action with steps from other threads.

Transactional variables in their semantics must be of a designated type — this is due, in part, to the functional nature of Haskell and its treatment of mutable variables. In the semantic model of this chapter all variables are transactional, and thus usable in a STM transaction without any need for any special designation.

The modelling of STM retry events is fundamentally different between the two semantics: their model uses an explicit mechanism which is available to the programmer, and the model in this chapter only allows implicit retry transitions. Their model requires that a STM transaction not execute if the initial state is such that it would result in a retry; this appears in their model as an absence of a rule that allows a transition to a retry, and their implementation is essentially required to tentatively execute the transaction to find out if it would retry. The model in this chapter, however, allows STM retries as an implicit result of the state in the STM construct and the overall state becoming inconsistent relative to the transaction's dependent variables.

That their semantic model requires that a STM transaction not execute if it result in a retry means that their model incorporates a form of angelic non-determinism in the way threads are chosen for the next transition. We wholly avoid this in our semantic model, by explicitly giving a semantic mechanism that allows transactions to be interleaved; it may be that the implementation of their semantic model follows an approach that is similar to our semantic model, but this is speculation.

Finally, two things that their semantics includes (where our model does not) are a notion of alternation between possible STM transactions and a mechanism for dealing with fault handling. The first –alternation– is a fascinating approach but adoption here would require invasive changes to our semantic model. The latter is necessary in their work due to Haskell's fault handling system, but is completely out of scope in this thesis.

In addition to the material above, there are a few papers that focus primarily on the implementation of STM in various contexts [MSS05, CH05, MSH$^+$06]. While the first of these –[MSS05]– gave a sense of some of the trade-offs faced in the design of potential implementations of STM, none of the three had much to offer to the design of the semantics for the $Atomic/STM$ construct in this chapter.

3 — Rely/Guarantee Conditions

3.1 Introduction

This section presents an overview of the rely/guarantee framework used in this work, and discusses some of the implications of the particular framework used. Section 3.2 presents a set of possible rules to cover the constructs presented in the language of Chapter 2, and some of those constructs are given multiple rules to cover slightly different situations. That a construct might have multiple rules is completely within the spirit of the rely/guarantee method; furthermore, it is recognized that it is simply not possible to create an exhaustive set of rules that covers all possible situations.

3.1.1 Overview

The rely/guarantee method is a set of attitudes towards the development of software that grew out of the need to reason about the interference a program must tolerate, both from external sources and from within the program itself. It arises, as discussed in Section 3.3 out of Hoare Logic, VDM, Owicki/Gries' work, and directly from Jones' thesis.

Very much at the centre of this is the desire to be able to reason compositionally about concurrent programs and this entails reasoning about the interference that different, parallel, parts will generate. This is accomplished through relations that pair the states from immediately before and after an atomic action in the system. These relations are elements of a specification that distinguishes between the behaviour a program must conform to, and behaviour that the environment is assumed to exhibit.

To explain how rely/guarantee reasoning works, we will start with the computational model. Though left implicit in the previous chapters for the sake of clarity, it helps here.

From the description of the semantic model in Chapter 2, we can consider computation as a sequence of steps (or transitions), as shown in Figure 3.1. In this figure, the line represents a notion of time; each circle on the line represents a particular system configuration; and the "jumping" arrows are steps of the system. The diagram is best interpreted as a single possible computation from a given starting configuration.

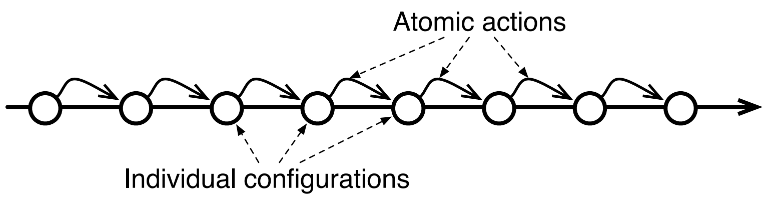


Figure 3.1: Abstract model of computation.

Each successive configuration on the bold line in this figure differs from the prior ones (even though they are represented as circles without any differentiating internal structure).

Some will differ in changes to the state, some by changing the remaining program text, and some on both points. All steps, however, do cause some form of change.

We now distinguish the steps as belonging to one of two categories: those of the program, and those of the environment. This is shown in Figure 3.2, with the program steps above the line, and the environment steps below the line.

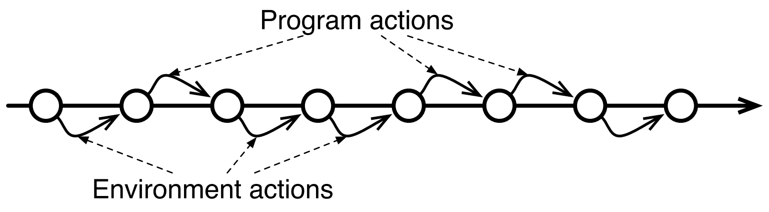


Figure 3.2: Abstract model of computation distinguishing interference.

The words “program” and “environment” are used as labels when we consider a system. These labels are simply shorthand, allowing a quick means of referring to either the portion of the system that we are developing at that point (the program) or the portion of the whole system about which we are making assumptions (the environment). It should be understood that, if we consider a closed system containing only a multi-threaded program, when we focus on a single thread of execution, it becomes the program and the rest of the threads are considered the environment. In this sense there is a pleasant symmetry present, dependent only on the viewpoint from which the whole system is considered.

Neither of the figures here have more than a single arrow between any pair of consecutive configurations. This omission is deliberate to match the semantic model of Chapter 2: the semantic model allows only one transition between a pair of configurations, thus constraining the computational model.

There may be occasions where there is more than one possible action that may follow a given configuration. In these cases a single action will be chosen non-deterministically from those possible, and the system will proceed from the following configuration.

The semantic model of Chapter 2 uses VDM-style record notation to describe the structure of a language’s abstract syntax, and VDM operators and pattern matching notation to manipulate these constructs in the rule definitions. It should come as little surprise, then, that the rely/guarantee specifications use a notation similar to VDM’s operation specifications.

The template in Figure 3.3 gives the general form of a rely/guarantee specification. There are eight elements to the specification; the first, of course, names the specified system. The **rd** and **wr** keywords give lists of the variables that this program will (respectively) read from and write to. Following that is the **pre** keyword, which gives the system’s pre condition. The **rely** and **guar** keywords give the rely and guarantee conditions, and the **post** keyword gives the post condition.

It is worth noting that the **rd** and **wr** keywords affect the actual guarantee and post conditions: they are constraints on the overall behaviour of the specified program. As a consistency constraint on rely/guarantee specifications, we assume that if a variable is

SYSTEMNAME
rd $x{:}\,X, y{:}\,Y, \ldots$
wr $s{:}\,S, t{:}\,T, \ldots$
pre P
rely R
guar G
post Q

Figure 3.3: A template for rely/guarantee specifications.

not mentioned in either a **rd** or **wr** keyword then it neither affects nor is affected by any program that conforms to the specification.

If the name SYSTEMNAME from Figure 3.3 is allowed as a stand-in for the eventual system/program it represents, we can write the corresponding "specification statement" [Mor88] as $(P, R) \vdash \mathit{SystemName}\ \textbf{sat}\ (G, Q)$. This specification statement could be read as "the pre condition, P, and rely condition, R, entail that the program S will satisfy the behaviour guarantee condition, G, and the post condition, Q". As usual, unpacking a logical sentence is a bit of a mouthful — it's unlikely that it will be expanded again.

There are a few constraints on the elements of a rely/guarantee specification which follow from its roots in VDM. First, the pre condition must be total, and thus defined for all possible states, and the post condition must be total over the domain characterized by the pre condition. The rely and guarantee condition must be total over the domain which can be found by starting with the domain characterized by the pre condition and taking the transitive closure of the union of the rely and guarantee relations; stated more directly, the domain of the rely and guarantee must be equivalent, and must be a superset (or equal to) the union of their ranges.

It should also be noted that Figure 3.3 does not presuppose any particular logical formalism by its structure. This work follows the usual approach of rely/guarantee work and uses the logic of partial functions (LPF) –a weakening of first-order predicate calculus– so that undefinedness may be dealt with in a reasonable manner. The use of LPF in this work is primarily intended to allow the use of prior work in VDM and rely/guarantee reasoning, such as [BFL$^+$94].

The rely/guarantee framework has limits on its expressive range — this is implicit in the fact that we will later prove a set of development rules in the framework sound relative to a semantic model, but that we also maintain that completeness relative to that model is not practically feasible.

In particular the interpretation of the rely and guarantee conditions as outer bounds on behaviour means that these conditions tend to be imprecise. Even given a condition that contains precisely and only the behaviour desired we must still deal with the fact that the condition is reflexive. This reflexivity means that the conditions cannot be used directly to show that some behaviour occurs. To give a trivial example, consider a condition that constrains a variable to decrease only monotonically. Using this condition alone to reason about the variable does not allow us to conclude that the variable will change at all: the reflexive nature of the condition implies only that the variable *may* change in value. To conclude that it *must* change value one is required to use either the semantics directly, or use the post condition of the specification of which the condition is a part. The former approach can be cumbersome –and one reason why we are investigating the use of rely/guarantee at all– and the latter approach is of no use if you need to know that the variable has changed at a given point during the specification, rather than after it has completed.

3.1.2 Implications

The basic approach to the rely/guarantee rules used here is to treat them as potential components of a usable development framework. As such, the desire is for rules that fit the particular situation at hand, rather than forcing all developments to use a fixed, immutable set of rules.

As a direct consequence of this philosophy, it turns out that the rules of a rely/guarantee framework are highly adaptable. Admittedly, there is a trade-off between the restrictiveness of the rules against their ease of use. The more restrictions that are in a rule's antecedents, the easier it becomes to prove soundness. Conversely, the less restrictions, the easier it becomes to prove that a given specification is valid. It's an interesting trade-off, but it plays well on the developer's actual knowledge of the system, allowing the developer's informal assumptions to be incorporated into the formal design.

So, not only is it possible to create more rules for this language at need in a straightforward manner, but it is also possible to generate rules for any language that has an accurate SOS model.

Behavioural specifications –as given by the rely and guarantee conditions– are a matter of specifying the set of possible actions that a process may effect. There are two things that are difficult to specify through the use of the transitive and reflexive relations that the rely/guarantee framework prefers: first is an "order" on the actions in the relation (such that some things precede others); and second is encoding the notion that some action *must* happen.

These expressive weaknesses in the rely/guarantee framework can be circumvented through the use of "ghost" or "auxiliary" variables. Ghost variables are similar to variables held in the system's state, except that they are completely inaccessible to the system, and they never appear in any final implementation of the specification. Unrestricted use of ghost variables, however, destroys the compositional nature of rely/guarantee specifications, even given the reflexive and transitive restrictions enforced in this particular framework. All of the work in this thesis assumes that ghost variables are not used.

In particular, it is possible to use a ghost variable to record the history of the computation; given this history one can then encode a notion of what will happen next into a specification. Doing so, however, destroys the compositional nature of the rely and guarantee conditions, and leaves anyone who wishes to use the specification in the uncomfortable position of doing their analysis more in the spirit of Owicki's work than in the spirit of rely/guarantee. Once a significant portion of the program's structure has been encoded into the specification, what was the purpose of specifying the program in the first place?

The rely and guarantee conditions are considered to be outer bounds on behaviour, and this particular framework assumes that all of these conditions are reflexive. If we imagine a very trivial program that never changes the state, then this means that *all* guarantee conditions are outer bounds for this program. In the other direction, the minimum or tightest relation that describes a trivial program that executes in one step and *does* alter the state must still include the identity relation.

For any program, its specification's guarantee conditions must include the possibility that the program will do nothing, even in cases where it is possible to prove that it always does something. The class of programs that are always modifying the state, however, is extremely limited given the semantics of Chapter 2: they consist solely of single assignment statements with a constant expression. Every other possible program in that semantic model includes at least one step which does not modify the state.

One of the contributions of the Coleman/Jones joint paper that we are particularly happy with is the insight contained in what are the *Isolation-Par-L* and *Isolation-Par-R* lemmas in this work.[1] These lemmas allow a means of reasoning about a small fragment of a program, simply by composing the behaviour of all that surrounds it into the overall rely condition for the fragment. Conversely, if our desire is to reason about the effect of a program's actions without requiring the detail of the program, we can replace that program by its guarantee condition in a manner reminiscent of Morgan's specification statements.

An advantage afforded by the rely/guarantee method is the ability to abstract a program to its specification and reason about that program purely in terms of the conditions in that specification. As an example –indeed, one that is critical later in this thesis– it is possible to reason about one branch of a parallel construct as though it were an isolated program so long as the guarantee of the other branch is taken as part of the rely condition of the first branch. This is done without any regard to the actual implementation of that other branch.

The rely/guarantee rules as we use them are intended to be an extension to the logical framework provided by the basic SOS model. Due to this, and because it is intended that the rules be used to shorten otherwise tedious proofs based solely on the SOS definition, the rely/guarantee rules must be proven to be sound with respect to the SOS model.

The requirement for the proof, then, is that if some property of a program is deduced due to the use of rely/guarantee rules, then that property must also be deducible through the SOS model directly.

3.1.3 In Context

The particular rely/guarantee framework that is used in this work makes several assumptions about the predicates and relations involved in the specifications and, thus, used in the development rules.

The rely and guarantee conditions are always reflexive, even where not explicitly made to be so. The reason behind this is both to allow for program steps that do not modify the state, and to always allow the possibility of an environment step that does not modify the state. Directly related to the reflexivity requirement, the rely and guarantee conditions are

[1] See Section 5.2.6.

also required to be transitive. This allows for either of the program or the environment to take multiple consecutive steps and have them considered as a single step. This simplifies the logic required in the proofs as there is no mechanism to distinguish between an environmental step that does nothing and no environmental step at all.

The pre condition of a specification must be robust relative to the rely condition. Formally, this is expressed as

$$\overleftharpoon{P} \wedge R \;\Rightarrow\; P$$

and is the *PR-ident* lemma: given a single state that satisfies the pre condition, any state that follows it by interference (i.e. the rely condition holds between the states) must also satisfy the pre condition. This holds only for the semantic, instantaneous predicate satisfaction, not under multiple-state evaluation as discussed in Section 3.2.

Finally, if the post condition holds between two states, we require that further interference that conforms to the rely condition will only result in states that still satisfy the post condition. Formally this is expressed as

$$Q \diamond R \;\Rightarrow\; Q$$

and is the *QR-ident* lemma: the rely condition acts as a right-hand identity with respect to the post condition.

These assumptions are not required by rely/guarantee frameworks in general. However, during the development of the rules presented in this work (and during the development of the rules in [CJ07]) these assumptions played a role in nearly every rule. Though they could have been included as antecedents in every rule, they have been "factored out" here.

3.2 The Development Rules

The purpose of this section is to present a set of useful rely/guarantee rules that are intended to be used in software development. Except where otherwise noted, these rules are proven sound with respect to the semantic model of the language of Chapter 2 in Chapter 6.

The set of development rules given here is not complete with respect to the semantics, and no attempt is made to gain completeness with respect to the semantic definition. This is a deliberate choice as one of the desired properties of a program developed with rely/guarantee rules is that it terminate, and it is certainly possible to use the semantics alone to prove things about non-terminating programs. Alternative formulations of the development rules for the constructs are presented to demonstrate the adaptability of the overall rely/guarantee framework to specific situations.

Before tackling the rules, however, a brief reminder is in order. The assertion $(P, R) \vdash S$ **psat** (G, Q) means that given that the pre and rely conditions, P and R, are valid, the program, S, will behave as specified in the guarantee and post conditions, G and Q. However, Q will only be satisfied for those cases where the execution of the program actually terminates. The stronger assertion $(P, R) \vdash S$ **sat** (G, Q) includes the conditions of the previous assertion, but has the additional property that the program will always terminate when run in an environment that satisfies the (P, R) assumption.

The first inference rule, *Weaken*, does not actually have a corresponding language construct, but is useful nonetheless.

$$\boxed{\text{Weaken}}\ \frac{\begin{array}{c}(P, R) \vdash S\ \textbf{sat}\ (G, Q)\\ P' \Rightarrow P\\ R' \Rightarrow R\\ G \Rightarrow G'\\ Q \Rightarrow Q'\end{array}}{(P', R') \vdash S\ \textbf{sat}\ (G', Q')}$$

The purpose of this rule is to allow a program that has been developed to satisfy a given specification to satisfy a "weaker" specification. In the case of the assumptions –the pre and rely conditions– weaker means admitting to a smaller range of states. For the pre condition, the set of states that satisfy the weaker pre condition will be a subset of those that satisfy the original pre condition. For the rely condition, the set of pairs of states that satisfy the weaker rely condition will be a subset of those that satisfy the original rely condition. So, to weaken a specification, one thing that you can do is make the assumptions more restrictive.

For the constraints –the guarantee and post conditions– "weaker" means less restrictive conditions. The set of states admitted by the original guarantee (post) condition must also be admitted by the weaker guarantee (respectively, post) condition.

Moving on to the first of the rules that actually do correspond to a language construct, we shall start with the rules for sequential composition.

$$\boxed{\text{Seq-raw-I}}\ \frac{\begin{array}{c}(P_l, R_l) \vdash \mathit{left}\ \textbf{sat}\ (G_l, Q_l)\\ (P_r, R_r) \vdash \mathit{right}\ \textbf{sat}\ (G_r, Q_r)\\ Q_l \Rightarrow P_r\end{array}}{(P_l, R_l \wedge R_r) \vdash \mathit{mk\text{-}Seq}(\mathit{left}, \mathit{right})\ \textbf{sat}\ (G_l \vee G_r, Q_l \diamond Q_r)}$$

Sequential composition is represented by the Seq construct, and the *Seq-raw-I* rule is a very direct –if naïve– way of encoding its properties in rely/guarantee terms. The first two antecedents presume the rely/guarantee-style development of the left and right subprograms and identify the associated conditions therein. The third antecedent gives the only interdependence between the left and right subprograms, namely, that the post condition of the left subprogram must establish the pre condition of the right subprogram. The consequent of the rule, then, follows from combinations of the conditions of the subprograms: P_l is the overall pre condition as it executes first; $R_l \wedge R_r$ as the overall rely condition as the overall environment must be suitable for both subprograms; $G_l \vee G_r$ as the overall guarantee condition as the combined behaviour is all of the behaviours of both subprograms; and finally, $Q_l \diamond Q_r$ as the overall post condition as the overall effect of the two subprograms will be precisely the composition of its parts.

Unfortunately, the *Seq-raw-I* rule does not obviously lend itself to the kind of top-down, decompositional development that is preferred by VDM-style development methods. It is possible, however, to use the *Weaken* rule to transform *Seq-Raw-I* into *Seq-I* below.

$$\boxed{\text{Seq-I}}\ \frac{\begin{array}{c}(P, R) \vdash \mathit{left}\ \mathbf{sat}\ (G, Q_l \wedge P_r)\\ (P_r, R) \vdash \mathit{right}\ \mathbf{sat}\ (G, Q_r)\\ Q_l \diamond Q_r \;\Rightarrow\; Q\end{array}}{(P, R) \vdash \mathit{mk\text{-}Seq}(\mathit{left}, \mathit{right})\ \mathbf{sat}\ (G, Q)}$$

To see how this works, we define $P \triangleq P_l$, $R \triangleq (R_l \wedge R_r)$, and $G \triangleq (G_l \vee G_r)$; and set $Q_l \triangleq (Q'_l \wedge P_r)$ (justified by $Q_l \;\Rightarrow\; P_r$ in *Seq-raw-I*). This allows the implications in the antecedents of the *Weaken* rule to be satisfied for the *left* subprogram, allowing the first antecedent of *Seq-raw-I* to become the first antecedent of *Seq-I* (modulo renaming). It must still be verified that the *left* subprogram actually does establish P_r, but this is unavoidable; this also makes obsolete the third antecedent of *Seq-raw-I*. The definitions also allow us to use the *Weaken* rule to obtain the second antecedent of *Seq-I* from the second antecedent of *Seq-raw-I* in a similar manner. We add the third antecedent of *Seq-I* by simply instantiating the fifth antecedent of *Weaken*. The consequent of *Seq-I* is obtained by substitution of the definitions, and a trivial application of *Weaken* using the third antecedent.

The full proofs of both of these rules are in Chapter 6 and Appendix D, though for the remaining constructs we will not show the naïve (raw) versions of the rules, as they are less suitable to rely/guarantee development.

Moving on to a more general combinator of programs we come to parallel composition.

$$\boxed{\text{Par-I}}\ \frac{\begin{array}{c}(P, R \vee G_r) \vdash \mathit{left}\ \mathbf{sat}\ (G_l, Q_l)\\ (P, R \vee G_l) \vdash \mathit{right}\ \mathbf{sat}\ (G_r, Q_r)\\ G_l \vee G_r \;\Rightarrow\; G\\ \overleftharpoon{P} \wedge Q_l \wedge Q_r \wedge (R \vee G_l \vee G_r)^* \;\Rightarrow\; Q\end{array}}{(P, R) \vdash \mathit{mk\text{-}Par}(\mathit{left}, \mathit{right})\ \mathbf{sat}\ (G, Q)}$$

The two subprograms share an interesting symmetry that is shown in the first two antecedents: they both must tolerate the interference generated by the other, and so the rely condition of each includes the guarantee of the other. Since both of the subprograms will be starting at the same point, they have the same pre condition. However, since the two subprograms are likely to do different things, their constraints differ from each other independently. The third antecedent combines the behaviours of the subprograms into a single guarantee condition, and the final antecedent combines all of the conditions of the subprograms to imply the overall post condition. Of the terms in the conjunct in the fourth antecedent, the last allows for parallel programs where the constraints on behaviour are necessary –in addition to the post conditions of the left and right branches– to obtain the post condition.

The explanation of the final antecedent in the *Par-I* rule is helped by an example; we will take a brief look at the decomposition in Chapter 7 of SEARCHES into SEARCH(odds) and SEARCH(evens). The Q term of that final antecedent corresponds to *post*-SEARCHES in this case and it is given as

$$\Big(t = \overleftharpoon{t} \vee (t < \overleftharpoon{t} \wedge \mathit{pred}(v(t)))\Big) \wedge \Big(\forall i \in \{1..t\text{-}1\} \cdot \neg\, \mathit{pred}(v(i))\Big)$$

The Q_l and Q_r terms in the final antecedent correspond to $post$-$\textsc{Search}(odds)$ and $post$-$\textsc{Search}(evens)$; taking is as $odds$ or $evens$ in turn, $post$-$\textsc{Search}(is)$ is given as

$$\forall i \in is \cdot (i < t \;\Rightarrow\; \neg\, pred(v(i)))$$

From this parameterized definition it is clear that an assertion of the form $Q_l \wedge Q_r \;\Rightarrow\; Q$ would not be valid in this case: though the second parenthesized term of $post$-$\textsc{Searches}$ is satisfied, the first is not. The addition of the guarantee conditions from $\textsc{Search}(odds)$ and $\textsc{Search}(evens)$ is required to satisfy the first parenthesized term of $post$-$\textsc{Searches}$. The guarantee conditions $guar$-$\textsc{Search}(is)$ –using is as above– are given as

$$t = \overleftharpoon{t} \vee (t < \overleftharpoon{t} \wedge pred(v(t)))$$

which clearly satisfies the first parenthesized term of $post$-$\textsc{Searches}$. This example is covered in more detail in Chapter 7.

The rules thus far have dealt with constructs whose behaviour is independent of the system's state object. The remaining constructs' behaviour is at least partially dependent on the state object which causes some difficulty when reasoning about expressions in a fine-grained semantics at a logical level.

The logical expression which we use to define predicates in our specifications are either true or false relative to a single state: we talk about whether or not individual states satisfy a predicate. At the logical level this "evaluation" is not affected by the possibility of interference in the semantics.

Expressions at the level of the language semantics –such as the tests of the *If* and *While* constructs– are profoundly affected by interference. In a language with expression evaluation semantics that are coarser than we use in Chapter 2 we would have a direct equivalence between the value obtained from both logical and semantic evaluation of an expression. This is referred to as single-state evaluation as it appears that semantic evaluation took place using the same state object during the entire expression evaluation.

Given the fine-grained semantics of our language, however, we know that expressions do not have to be evaluated using a single state, but rather using a succession of states that may change from one to the next as program execution proceeds. Naturally, this is referred to as multiple-state evaluation; and, as the states used to evaluate the expression may differ in value for a given variable, this gives rise to the seemingly nonsensical situation where the semantic evaluation of $x < x$ may result in a true value.

In effect this difference between single- and multiple-state evaluation means that an expression in the language cannot usually be lifted to its equivalent logical expression for use in the development rules. If it can be shown that the semantic evaluation of the expression will happen in such a way as to always give the appearance of single-state evaluation, however, then the expression may be used in both contexts. The simplest –and most restrictive– condition under which multiple-state evaluation will appear to be single-state evaluation is when interference as denoted by the rely condition is constrained to be the identity relation on the relevant variables.

The next two rules both correspond to the construct for conditional execution. Two rules are presented here (and proven sound in Chapter 6) as there are two extremes to the situations in which an *If* construct might be used. The cases revolve around whether or not it is important for the logical meaning of the test of the construct to remain true during the execution of the body of the conditional.

$$\boxed{\text{If-b-I}}\ \frac{\begin{array}{c}(P \wedge b, R) \vdash \mathit{body}\ \mathbf{sat}\ (G, Q) \\ R \Rightarrow I_{Vars(b)} \\ \overleftarrow{P} \wedge \overleftarrow{\neg b} \Rightarrow Q\end{array}}{(P, R) \vdash \mathit{mk\text{-}If}(b, \mathit{body})\ \mathbf{sat}\ (G, Q)}$$

In *If-b-I* the situation is that the test, b, remains true until the body of the *If* acts to alter it; this is captured in the first antecedent: including b in the pre condition of the first antecedent has the effect of requiring that the rely condition, R, cannot change the state in such a way that any single state will cause the un-interfered-with evaluation of b to be false. To close the hole of multiple-state evaluation, the second antecedent requires that the actions of the environment be equivalent to the identity relation as far as the free variables in the test are concerned. The last antecedent handles the case where the test evaluates to false: here the post condition must be trivially fulfilled by the pre condition and the negation of the test.

The *If-I* rule is similar to *If-b-I* but differs in that it does not require that the test be stable under interference.

$$\boxed{\text{If-I}}\ \frac{\begin{array}{c}(P, R) \vdash \mathit{body}\ \mathbf{sat}\ (G, Q) \\ \overleftarrow{P} \Rightarrow Q\end{array}}{(P, R) \vdash \mathit{mk\text{-}If}(b, \mathit{body})\ \mathbf{sat}\ (G, Q)}$$

With this rule, it must be the case that the post condition is satisfied in any state which differs only by interference from those states that satisfy the pre condition. The body of the *If* must also satisfy the post condition, if executed, but its execution is completely optional.

Unlike the *If* construct, there is only one rule for the *While* construct as it combines the features of the two *If* rules with respect to the conditional expression.

$$\boxed{\text{While-I}}\ \frac{\begin{array}{c}\mathit{well\text{-}founded}(W) \\ \mathit{bottoms}(W) \subseteq [\![\neg (b_s \wedge b_u)]\!] \\ R \Rightarrow W^* \wedge I_{Vars(b_s)} \\ \mathit{SingleSharedVar}(b_u, R) \\ \overleftarrow{\neg (b_s \wedge b_u)} \wedge R \Rightarrow \neg (b_s \wedge b_u) \\ (P \wedge b_s, R) \vdash \mathit{body}\ \mathbf{sat}\ (G, W \wedge P)\end{array}}{(P, R) \vdash \mathit{mk\text{-}While}(b_s \wedge b_u, \mathit{body})\ \mathbf{sat}\ (G, W^* \wedge P \wedge \neg (b_s \wedge b_u))}$$

This development rule is the most complex of the ones presented in this work, but it is flexible enough to be applied to a large range of the applications of the *While* construct.

First let us consider the W relation as it is pivotal to the termination argument for the *While-I* rule. The W relation is well-founded over states –as indicated by the first antecedent and the fact that W is a part of the post conditions– which means that it give a transitive but irreflexive ordering over states. The irreflexive property of W means that there is a definite notion of progress that can be used in a termination proof. The second antecedent adds the constraint that the W-minimal states –the bottoms of W– are contained in the set of states which cause the $While$ construct's conditional to evaluate to false under single-state evaluation. This allows the termination proof to argue that every iteration of the body of the $While$ construct results in a state that is either a bottom element of W (and will thus terminate) or is, at least, closer to a bottom element than the state prior to the iteration. Part of the third antecedent establishes that interference cannot "reverse" the progress of the $While$ as given by the W relation. It may appear that the $R \Rightarrow W^*$ portion of the third antecedent is implicitly true due to the *QR-ident*, but unfortunately, that is not the case: it may be possible to create a state that is still related through W relative to some initial state, but such that the new state is further from the bottoms of W.

Noting that we have only required that the bottoms of W be contained within a set characterized by single-state evaluation, we must constrain the $While$ construct's conditional so that it may be considered to have been evaluated as though it were done in a single state (even though we may not know *which* state). This is done in two parts mirroring the way in which the conditional has been split. The first part, b_s represents the stable portion of the conditional expression, and the second part, b_u represents the unstable portion. The stable portion can be considered as though it were evaluated in a single state as the $R \Rightarrow I_{vars(b_s)}$ portion of the third antecedent isolates it from interference entirely. Using the third antecedent in this manner has the effect of also allowing the stable portion of the conditional expression to be used as a part of the pre condition for the body of the $While$ construct: this was a key motivation behind splitting the conditional in the first place.

That the unstable portion of the conditional expression may be considered to have evaluated as though it were in a single state requires the fourth antecedent. This antecedent is primarily a syntactic constraint on the form of b_u relative to the rely condition. Specifically, there may only be a single appearance of a variable in b_u whose behaviour under interference is *not* equivalent to the identity relation. This constraint means that the only state which is critical to the evaluation of b_u is the one in which the sole shared variable is read.

The fifth antecedent adds a further constraint so that we can be certain that when a state is reached wherein the conditional will evaluate to false then all subsequent states will also be such that the conditional will still evaluate to false. It is important to note that this antecedent is not relevant unless the evaluation of the conditional expression can be considered to have happened in a single state; otherwise the general case of multiple-state evaluation renders this antecedent useless for analyzing the behaviour of a program. This antecedent frees us from having to worry about the loop terminating before satisfying its post condition. The loop may still terminate before reaching a state in the bottoms of W –even with this antecedent– but if it does so we know that the conditional would not have evaluated to true again.

The last antecedent is the specification satisfaction assertion on the body of the loop. As mentioned, splitting the conditional allows us to add the stable portion of the conditional expression to the pre condition of the body. The post condition of the body requires that two things hold: first, that some progress is made as a result of its execution, thus the W rather than W^* term; and also that the initial pre condition holds in case the body is executed again; thus the P term. The stable portion of the conditional is, of course, not required to still hold as a result of executing the body.

The next construct –assignment to variables in the state– will not in general have a rule that easily fits all situations. However, there is one case which can be formalized directly, and it provides some insight into the kind of direct reasoning that must be done with assignments.

$$\boxed{\text{Assign-I}}\ \frac{\begin{array}{c} R \;\Rightarrow\; I_{Vars(e)\cup\{id\}} \\ G = \{(\sigma, \sigma \dagger \{id \mapsto [\![e]\!](\sigma)\}) \mid \sigma \in \Sigma\} \cup I \\ Q = \{(\sigma, \sigma') \mid \sigma, \sigma' \in \Sigma \wedge \sigma'(id) = [\![e]\!](\sigma)\} \end{array}}{(P, R) \vdash mk\text{-}Assign(id, e)\ \mathbf{sat}\ (G, Q)}$$

Conceptually, the rule covers a simple situation: those cases where it can be shown that the rely condition is an identity over the target variable and those variables in the assignment's expression; or, simply, those cases where there is no interference to worry about. This only covers a rather restricted set of possible situations in which an $Assign$ construct might be found; however, we anticipate that in the general case constructs will have to be proven directly in terms of the semantic model.

The second and third antecedents of this rule give the guarantee and post conditions in the form of set comprehensions over pairs of states; this turned out to be the most direct way of specifying the conditions such that they could be used in the proofs of Chapter 6 and Appendix D. The guarantee condition is the identity relation combined with a relation that has the right-hand state such that its value on the target identifier is that of the evaluated expression; this gives a guarantee that allows for state mutations the $Assign$ construct would perform in given single-state evaluation. The post condition is similar to the guarantee, but is carefully constructed so that interference on variables that are not in the expression is permitted.

Thus far we have covered the "regular" constructs in the language. The remaining constructs are $Atomic$ and STM — these constructs are far more difficult to develop a good formalism for as they violate many of the usual expectations.

One of the interesting effects of the $Atomic/STM$ pair of constructs is that the overall effect of the body of the construct is seen by the environment to have happened all in one step. Setting aside concerns regarding termination for a moment, this suggests the following development rule.

$$\boxed{\text{Atomic-psat-I}}\ \frac{\begin{array}{c} (P, I) \vdash body\ \mathbf{psat}\ (\mathbf{true}, Q') \\ Q' \;\Rightarrow\; G \\ \overleftarrow{P} \wedge R \diamond Q' \diamond R \;\Rightarrow\; Q \end{array}}{(P, R) \vdash mk\text{-}Atomic(body)\ \mathbf{psat}\ (G, Q)}$$

The rule is given in terms of partial satisfaction –**psat**– which does not require termination. The first antecedent notes that the body may assume that there will be no interference while it runs and gives the body no behavioural constraints as it allows a guarantee condition of **true**. The post condition of the body does not necessarily have to be the same as that of the overall atomic block; this simplifies reasoning about the program given the second and third antecedents. The second antecedent requires that the post condition of the body conforms to the behavioural specification of the overall atomic block. The third post condition explicitly places the body's post condition in the effective computation as observed from the atomic block and asserts that it must satisfy the overall post condition.

That the post conditions of the body and the overall atomic block are not the same has advantages in the proofs of Chapter 6, but it also makes reasoning during program design easier. Specifically, it allows for the post condition of the body to be more precise than the overall post condition of the construct; for example, it is possible to require that the body increments a variable by an exact amount while the overall post condition only requires that the variable be increased.

In the limited situation where the atomic block is being used for read isolation –rather than to protect against interfering writes– the following development rule is useful and allows for full satisfaction.

$$\boxed{\text{Atomic-I}}\ \frac{\begin{array}{c}(P, I) \vdash \mathit{body}\ \textbf{sat}\ (\textbf{true}, Q') \\ Q' \Rightarrow G \\ R \Rightarrow I_{Vars(body)} \\ \overleftarrow{P} \wedge R \diamond Q' \diamond R \Rightarrow Q\end{array}}{(P, R) \vdash \mathit{mk\text{-}Atomic}(\mathit{body})\ \textbf{sat}\ (G, Q)}$$

The main change relative to *Atomic-psat-I* in *Atomic-I* is the addition of the third antecedent. This antecedent constrains interference so that the variables in the body are not altered by the environment. Looking at the semantic model, this implies that the STM-Retry transition cannot occur as the comparison of the restricted domains between the initial and current external states will never be unequal. And without the possibility of a STM-Retry transition, the termination of the atomic block is solely dependent on the body.

3.3 References

The obvious primary source –in the context of this work– on the rely/guarantee method of reasoning is Jones' DPhil thesis [Jon81]. Shorter introductions that followed soon after the thesis are [Jon83a, Jon83b], and they are more accessible both in terms of length and ease of acquisition. These works propose rely and guarantee conditions as an extension to a method which uses specification, data reification and operation decomposition in a manner that is typical of VDM development.

The use of rely/guarantee reasoning is illustrated with a case study in a paper by Collette and Jones [CJ00]. The development contained therein is well motivated and rigorously done at the level of specification. However, though their example is developed all the way to pseudo-code of an imperative language, their work does not deal with the connection between the development rules and the language semantics.

Another example of rely/guarantee development is examined in [Jon96], here working with a concurrent object-oriented language. This paper presents –in addition to the rely/guarantee development– a non-rely/guarantee transformation method of developing concurrent programs, but notes that the transformational technique has considerable weaknesses relative to rely/guarantee reasoning.

Considering the problem of reasoning about concurrency in general, de Roever's impressive [dR01] goes into exhaustive detail on many methods for reasoning about the development of concurrent programs, in both compositional and non-compositional ways. Included for each method covered are the method's history, its relation to other methods, examples of its use, and proofs regarding the soundness of the formal elements of the method. It should be noted that the soundness proofs for the rely/guarantee method that are present in de Roever's work are based on the use of Aczel traces [Acz82]. This is an entirely different basis for soundness than is used in the proof of Chapter 6.

An outline of the early history of reasoning methods suitable for concurrency is given in [Jon03a]. Some highlights of the work mentioned in that work follow here.

The work of Hoare in [Hoa69] sets forth a method for developing sequential programs in a formal manner. Hoare-style axioms have had a profound influence on subsequent methods of formal development and their influence can be seen in Owicki's thesis [Owi75] and a subsequent paper of Owicki and Gries [OG76]. The Owicki-Gries methods attempts to tackle interference directly in the formal development of concurrent programs. Its reliance on global reasoning to give the final interference-freedom proof, however, renders their method distinctly non-compositional and has the risk of requiring the user to redevelop earlier stages so as to discharge that final proof. The influence of Hoare-style axioms can also be seen in the VDM method [Jon90]. Jones' thesis [Jon81] works in what is essentially the same logical framework as VDM, and placing rely/guarantee concepts within this allows interference to be reasoned about using inference rules in a manner similar to Hoare-style axioms.

More recent work on rely/guarantee-style reasoning includes Stølen's thesis [Stø90] and subsequent papers [Stø91b, Stø91a]. Stølen's work extends Jones' rely/guarantee framework by adding a predicate which allows for reasoning about the conditions under which a program may block. Other pieces of recent work on rely/guarantee-style reasoning include [Sti86, Xu92, Col94, Bue00, BS01], as noted in Jones' annotated bibliography [Jon07]. However, none of these treat the gap between the language semantics and the development rules in the same manner as this thesis. Finally, as noted in Chapter 1, this work is an expansion of work done in [CJ07].

$\boxed{4}$ — Proof Methodology

The material in this chapter has been written –as much as possible– to be clear without needing to reference the proofs of Chapter 6; despite this, much of it remains embedded in the context of the proofs.

4.1 Logical Tools

The purpose of this section is to introduce the format and inference system that the lemmas of Chapter 5 and proofs of Chapter 6 are presented in. As part of this process, the more interesting techniques are covered and, where appropriate, the inference rule that corresponds to our use of the technique is given.

4.1.1 Natural Deduction

Natural deduction, rather than a form of the sequent calculus, is used as the overall system of inference in this work. The canonical reference on natural deduction is Prawitz's monograph *Natural Deduction* [Pra65], which is a formalization of natural deduction using Gentzen's proof-theoretic semantics [Sza69]. Natural deduction can be loosely characterized as the use of introduction and elimination rules as applied directly to the hypotheses and goal in an effort to find a path of reasoning between them. Any specific deduction in the system references the previous deductions and assumptions that justify the step.

In particular, the proofs of Chapter 6 are presented in a variant of the boxed linear notation sometimes referred to as "Fitch-style". The appellation "Fitch-style" arises from the work of F. B. Fitch, in particular [Fit52]. The choice to use this format is simply to keep things stylistically consistent with the proofs encountered in [Jon90], [JJLM91], and [BFL$^+$94].

An example proof is given in Figure 4.1, showing how natural deduction is used to prove that implication distributes over logical conjunction; for simplicity, all terms are assumed to be defined.

```
from $A \Rightarrow (B \wedge C)$
1     from $A$
1.1       $B \wedge C$                                   $\Rightarrow$-E(h, h1)
      infer $B$                                        $\wedge$-E(1.1)
2     $A \Rightarrow B$                                    $\Rightarrow$-I(1)
3     from $A$
3.1       $B \wedge C$                                   $\Rightarrow$-E(h, h3)
      infer $C$                                        $\wedge$-E(3.1)
4     $A \Rightarrow C$                                    $\Rightarrow$-I(3)
infer $(A \Rightarrow B) \wedge (A \Rightarrow C)$                 $\wedge$-I(2, 4)
```

Figure 4.1: An example natural deduction proof of the distribution of implication over logical conjunction.

Assumptions are introduced by **from** keywords, which start a block giving an implicit scope in which the assumption is valid. The overall **from/infer** block represents entailment from the assumption and prior deductions to the conclusion, and can be used as such directly in later deductions. The scoping effect of the **from/infer** block allows for a certain degree of modularity with these blocks, something that is exploited to split up the soundness proofs.

The basic set of inference rules that are used in this work come from both [Jon90] and [BFL^{+}94].

4.1.2 Structural Induction

As the language definition is given using structural operational semantics, it should come as no surprise that the logical tool that ties the proofs together is that of structural induction. The basic idea here is that if some property is entailed by every subclass of a particular structure, then it is entailed by all members of the structure.

For the language of Chapter 2, and given H as the desired property, the particular instantiation of the structural induction principle is

$$\boxed{\text{Stmt-Indn}}\ \frac{\begin{array}{c} H(\mathbf{nil}) \\ S \in \mathit{Assign} \vdash H(S) \\ H(\mathit{left}) \wedge H(\mathit{right}) \vdash H(\mathit{mk\text{-}Seq}(\mathit{left}, \mathit{right})) \\ H(\mathit{body}) \vdash H(\mathit{mk\text{-}If}(b, \mathit{body})) \\ H(\mathit{body}) \vdash H(\mathit{mk\text{-}While}(b, \mathit{body})) \\ H(\mathit{left}) \wedge H(\mathit{right}) \vdash H(\mathit{mk\text{-}Par}(\mathit{left}, \mathit{right})) \\ H(\mathit{body}) \vdash H(\mathit{mk\text{-}Atomic}(\mathit{body})) \end{array}}{\forall S \in \mathit{Stmt} \cdot H(S)}$$

The first two antecedents of the *Stmt-Indn* rule represent the base cases of the language structure as neither the **nil** statement nor any $Assign$ construct have subcomponents that are in $Stmt$. The remaining antecedents represent the inductive step of the structural induction: showing that the property holds on the composite structure on the basis that the property holds on its components.

4.1.3 Name Binding and Quantifiers

Name binding in the proofs is pervasive, and is used to solve two problems. The first problem is trivial, and does not –in the uses here– represent anything remotely profound: these uses are a substitutional shorthand, and the variables named in them are considered to be fresh, that is, otherwise unused. The second category of use is to provide a means of accessing a specific –but arbitrary– element of a set (e.g. the domain or range of a relation). This latter category requires the use of quantifiers, and its need comes about due to the difference between relations in general versus their restricted form as functions.

The uses of equalities as definitions in the proofs appear in two forms: the first is in the **from** line of a **from/infer** box; the second is as a regular numbered line in a proof where the justification is given as “definition”. Both forms may introduce new variables and their types are determined implicitly by the structure of the objects in the equality and the types of the variables that are already known. For example, if we know that C_i is a configuration in the semantic model, the equality $C_i = (S_i, \sigma_i)$ introduces two new variables –S_i and σ_i– that denote, respectively, a Stmt object and a state, and together they comprise a configuration. The use of the flat equality for name binding is done carefully so as to ensure that any newly introduced variables could be replaced with the appropriate accessor functions using the previously known variables without any change in the meaning of the proofs.

The equality is not, however, a suitable mechanism to handle the relations that are used heavily throughout this work. In the simple case of functions we can be certain that $f(x)$ –a function, f, applied to the parameter x– will always denote the same value. With relations in general this is not true: given a relation, R, and a fixed value, x, it is not the case in $[\![R]\!](x, y)$ that y always denotes the same value. In this latter example, y could denote many different values, and using the relation naïvely can result in separate instances of y denoting different values. We have a need, then, to bind y to a single one of those values, but in a manner such that while y could denote any of the values, the specific value is consistent through a collection of instances of y. Thus we use quantifiers to achieve this.

The first quantified form of name binding –and the less complex form that we use– uses universal introduction. We use this method when the fact that a property holds over all members of a set is deduced. Once we know the set that we are interested in, we just use the set membership operator in the **from** line of a **from/infer** box. For example, we may use $C_v \in C^v$ in the **from** line of a **from/infer** box, where C^v is already known to be a set. This binds C_v to be an arbitrary element of C^v and all uses of C_v within that **from/infer** box then refer to the same element in a consistent manner. If C_v was already known before this particular **from/infer** box, then uses of C_v within the box are considered to reference the new C_v introduced in the hypothesis of the box, and the previously known C_v is not accessible within the box. The actual introduction of the universal quantifier uses the same set as used in the **from/infer** box, and binding is handled in the usual way for a universal quantification; the binding survives in the quantifier as the inferred property of the **from/infer** box uses the reference to the bound variables directly.

As an inference rule, universal introduction has the form

$$\boxed{\forall\text{-I}}\,\frac{x \in X \vdash H(x)}{\forall x \in X \cdot H(x)}$$

and we take the $x \in X$ portion of the antecedent to be a binding in our proofs. As the antecedent of the $\forall$-I rule is in the form of a sequent, this means it is translated into a **from/infer** box in a natural deduction proof, thus giving the use of the $\forall$-I rule the following pattern

```
     ⋮
n    from $x \in X$
         ⋮
     infer $H(x)$
m    $\forall x \in X \cdot H(x)$                    ∀-I(n)
     ⋮
```

Note, also, that the set denoted in the pattern as X will usually have been defined earlier as a flat equality; this is just the typical usage in the proofs rather than by any necessity.

The second quantified form of name binding in the proofs uses existential elimination to bind variables temporarily so as to derive a more general property. Existential elimination is a generalization of disjunction elimination, and the latter can be shown to be a special case of the former. Disjunction elimination may be given as an inference rule in the following form

$$\boxed{\vee\text{-E}}\;\frac{\begin{array}{c}H_1 \vee H_2\\ H_1 \vdash H\\ H_2 \vdash H\end{array}}{H}$$

where the various H and H_i are self-contained properties. If H_1 and H_2 can be parameterized such that $H'(x_1) = H_1$ and $H'(x_2) = H_2$ then we have a means of writing the $\vee$-E rule in a more general manner. Specifically, the first antecedent becomes $\exists x \in \{x_1, x_2\} \cdot H'(x)$ and the last two antecedents, together, become $y \in \{x_1, x_2\}, H'(y) \vdash H$ with the constraint that y is an arbitrary value in $\{x_1, x_2\}$. The first new antecedent is critical as it establishes that H' holds for at least one of the values, and the use of the existential quantifier requires that the subject set is not empty. The second new antecedent simply means that if H' holds on any arbitrary parameter in the set then H must hold. Finally, we can replace the two-element set, $\{x_1, x_2\}$ with an arbitrary set, X, and we arrive at an inference rule that embodies existential elimination.

$$\boxed{\exists\text{-E}}\;\frac{\begin{array}{c}\exists x \in X \cdot H'(x)\\ y \in X, H'(y) \vdash H\end{array}}{H}\quad y \text{ is arbitrary}$$

When using this rule in practice to infer H from $H'(y)$ it requires some care to ensure that the inference only depends on properties which all elements of X have; properties specific to elements of a strict subset of X cannot be used unless there is an alternate means of making the inference for elements of the complement of that subset.

The name binding in the ∃-E inference rule is primarily in the second antecedent, specifically in the $y \in X$ portion of the left of the sequent. The implicit binding in the first antecedent is self-contained and not an issue. In a natural deduction proof the second antecedent translates into a **from/infer** box, giving rise to the following pattern for using the rule

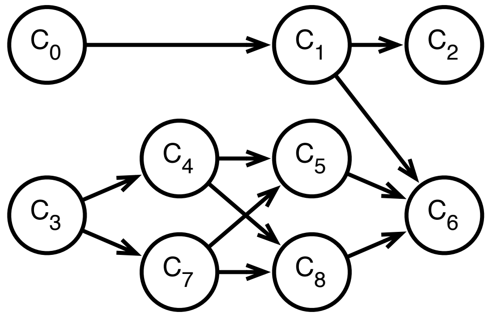


Figure 4.2: Example graph of computational paths.

```
     ⋮
n    $\exists x \in X \cdot H'(x)$
m    **from** $x \in X$ **st** $H'(x)$
         ⋮
     **infer** $H$
p    $H$                                      ∃-E(n,m)
     ⋮
```

This usage pattern in the proofs uses the same symbol –x– in both of the lines n and m despite the fact that the ∃-E rule uses different symbols. Practically speaking, however, this is not a problem as the meta-semantic model ensures that the uses of x in lines n and m do not interfere with one another. It is worth noting that the actual property given by H' in the proofs of Chapter 6 tends to be trivial as the motivation behind using this rule is to define the subject set over which x ranges.

4.1.4 Pinch Sets

During the development of the proofs for the partial satisfaction properties it became apparent that a given computation can usually be thought about in terms of phases. For instance, the assignment construct has two phases: first, the evaluation of the expression takes place; and second, the alteration of the state happens based on the fully evaluated expression.

These phases are about the semantic transitions: activities such as the evaluation of an expression happen during the transition between two configurations in the semantics. Thus, to define the boundary between two phases the simplest thing to do is define the set of configurations that lie between the phases. This set is called a pinch set.

The explanation of the defining property of pinch sets is given here with reference to the graph in Figure 4.2. This graph is of the possible computational paths from a set of initial configurations to a set of final configurations. The graph has been simplified to the minimum essential features; please note that a direct correspondence between the graph and the semantics of Chapter 2 is not intended. The intention of the graph is to illustrate the problems which must be dealt with to provide a usable definition for pinch sets.

Each labelled node in Figure 4.2 represents a configuration, and the arcs represent transitions. The graph is acyclic which is critical to the property definition and required to match the situation found in the partial satisfaction properties where termination is assumed. A cycle in this graph would mean that a configuration may transition to itself, and execution of a corresponding program would not be guaranteed to terminate.

We can see that C_0 and C_3 are minimal in this graph, as they have no transitions leading to them; by a similar argument, C_2 and C_6 are maximal. These two pairs are, respectively, the start and end configurations of all computations in the graph. A pinch set in the graph is a set of configurations such that all computations from a minimal element to a maximal element must pass through at least one of the configurations of the pinch set.

This definition immediately suggests two trivial pinch sets: the first is the set of minimal elements, and the second is the set of maximal elements. These two sets are not all that useful, however, so we shall leave them aside. A non-trivial pinch set could consist of C_1, C_4, and C_7, for which it is not hard to see that it is impossible to find a path in the graph from a minimal element to a maximal element that does not include one of the members of this pinch set.

Let us now give a formal statement of the defining property of pinch sets. First, we define C^i to be the set of all reachable configurations for the computation that we are interested in. Relative to the graph, C^i is simply all of the configurations in the diagram, $C^i \triangleq \{C_0..C_8\}$, and, given that this example is not precisely the semantics of Chapter 2, the semantic relation will be represented by a blank transition arrow, $\longrightarrow$, and is defined by the arcs present in Figure 4.2. Given this, C^p is a pinch set if and only if

$$\forall i \in C^i \cdot \left[\begin{array}{l} i \in C^p \ \vee \\ \big(\forall i' \in \{i'' \in C^i \mid i \longrightarrow i''\} \cdot \exists p \in C^p \cdot i' \longrightarrow\!\!* \ p\big) \ \vee \\ \big(\forall i' \in \{i'' \in C^i \mid i'' \longrightarrow i\} \cdot \exists p \in C^p \cdot p \longrightarrow\!\!* \ i'\big) \end{array} \right]$$

This property is read as saying that for all reachable configurations in the computation one of the following must hold:

1. the configuration is in the pinch set; *or*
2. all immediate successors of the configuration are either in the pinch set or have a successor that is in the pinch set; *or*
3. all immediate predecessors of the configuration are either in the pinch set or have a predecessor that is in the pinch set.

That the set consisting of C_1, C_4, and C_7 satisfies the above definition can be verified by looking in turn at the individual configurations in the graph. The minimal configurations satisfy the second term as the set of their immediate successors is the pinch set. Configurations C_1, C_4, and C_7 satisfy the first term, of course, as they are the pinch set. Next, configurations C_2, C_5, and C_8 satisfy the third term as the set of their immediate predecessors is the pinch set. And, last, C_6 also satisfies the third term through the transitivity of the semantic relation.

Now let us examine a set that is not a pinch set to see why it fails to satisfy the property given above. Consider a proposed pinch set consisting of configurations C_1, C_5, and C_7 — this does not comprise a pinch set as the transition between C_4 and C_8 allows for a path that does not include any of the configurations from the proposed pinch set. In terms of the logical definition, we can see that C_4 trivially fails both of the first and third terms of the property, and more importantly, also fails the second term. This last is a little more difficult to see, as C_4 does have one successor, C_5, that is in the proposed pinch set; however, the other successor, C_8, is neither in the pinch set nor does it have a successor that is. As the term requires that all of the immediate successors lead to a pinch set element, the second term is not satisfied due to C_8. A similar set of arguments apply to C_8 with respect to the third term of the property.

It should be clear from the definition and the examples that a superset of a pinch set remains a pinch set.

4.1.5 Well-founded Induction

Well-founded induction is a form of induction that depends upon a well-founded relation to provide the set over which the induction ranges. By way of analogy, the familiar strong (or complete) induction uses the "less-than" relation, <, and the natural numbers as the inductive set.

Briefly, using H as the desired property, natural number induction can be given as an inference rule in the form

$$\boxed{\mathbb{N}\text{-Indn}}\ \frac{\left[\forall i \in \{i' \in \mathbb{N} \mid i' < n\} \cdot H(i)\right] \vdash H(n)}{H(n)}$$

The rule is read to mean that so long as $H(n)$ can be deduced from the assumption that $H(i)$ is true for all lesser values i, then $H(n)$ must be true for any arbitrary n. It must be the case that $H(0)$ is true on its own merits, as the universal quantification in the antecedent is vacuously true.

Well-founded induction generalizes strong induction over natural numbers in that, unlike the natural numbers, a well-founded relation does not, in general, require that there is a single minimal element. The particular rule we use is a bit different than the one for natural numbers as the "direction" of the well-founded relations that we use is opposite than it is for less than; specifically, where smaller numbers are to the left with the less than relation, states closer to the minimal states are written to the left of the well-founded relation that we use in the development rules.

Given a well-founded relation, W, and the desired property, H, the rule for well-founded induction is

$$\boxed{\text{W-Indn}}\ \frac{\begin{array}{c} \mathit{well\text{-}founded}(W) \\ \left[\forall \sigma' \in \{\sigma'' \in \textbf{fld}\ W \mid [\![W]\!](\sigma, \sigma'')\} \cdot H(\sigma')\right] \vdash H(\sigma) \end{array}}{H(\sigma)}$$

As with the $\mathbb{N}$-*Indn* rule, this rule is read to mean that so long as $H(\sigma)$ can be deduced from the assumption that $H(\sigma')$ holds on all values of σ' closer to the minimal elements of

W than σ, then $H(\sigma)$ must be true for any arbitrary σ. And, once again, H must be true of all of the minimal elements of W on its own merits as the quantification in the antecedent is vacuously true for the minimal elements.

Applying this to the proofs on the While construct for satisfaction of the post condition and for convergence is somewhat more complicated. This is explained in more detail in Chapter 6, but brief descriptions follow here.

The W relation of the rule is essentially the W relation of the *While-I* development rule, except that it is implicitly lifted to be a well-founded relation over whole configurations rather than just over states. The W relation is applied over the configurations between successive iterations of the loop body, so that the well-founded ordering is over configurations consisting of a static While construct, and a state component. Minimal elements of the relation are those configurations for which the state component will cause the While's test expression to evaluate –under interference– to false. That the test expression evaluates under interference is described in the discussion regarding multiple-state evaluation in Section 3.2 (in particular, page 35). The *While-I* development rule is carefully defined to ensure that the behaviour of test evaluation is consistent with the requirements imposed by well-founded induction.

For the proof of the satisfaction of the post condition, the property H is that execution starting from the pre condition-satisfying state will satisfy the post condition of the While specification. That post condition includes the reflexive closure of the well-founded relation as well as the pre condition; this implies that the post condition –and thus the H property– will be true in the vacuous case, and from that the induction builds.

In the case of the convergence proof, the H property is that execution from the given state always converges upon a configuration that has a **nil** statement component. As the minimal elements of the well-founded relation are those states that force the loop to terminate, and all iterations of the body result in a state that is closer to the minimal elements, the induction follows directly.

4.1.6 Proof by Contradiction

Using absurdity –proof by contradiction– in the proofs is, perhaps surprisingly, the easiest mechanism available to show that certain rules in the semantics cannot be applied. The mechanism is only used in one proof: convergence of the $\mathit{Atomic/STM}$ construct in Section 6.4.2. It is, however, the most direct way of reaching the conclusion.

Two inference rules are used to allow a proof by contradiction in this work, and both are variants of the usual rules that deal with absurdity. The first allows the introduction of an absurdity constant; as the law of the excluded middle does not hold in the presiding logical framework, the first antecedent is there to ensure that the proposition, H, is properly defined (eliminating the third value). The remaining two antecedents assume that the proposition, H, is both valid and invalid at the same time; the three antecedents together suggest that one of the hypotheses that allowed their deduction is an absurdity.

$$\boxed{\text{⅄-I}}\ \frac{\begin{array}{c}\delta(H)\\ H\\ \neg H\end{array}}{\text{⅄}}$$

Absurdity is denoted by a special glyph, ⅄, rather than **false** to ensure that there is no ambiguity between the semantic **false** value, the evaluated language value **false**, and a logical contradiction.

To put this absurdity constant to work –as it is unlikely that the absurdity is, itself, the target of the overall proof– we need a rule to eliminate it.

$$\boxed{\text{⅄-E}}\ \frac{\begin{array}{c}\delta(H)\\ \neg H \vdash \text{⅄}\end{array}}{H}$$

As with the introduction rule, absurdity elimination requires that the proposition is defined. Once that is covered, if the negation of the proposition leads to an absurdity, then the proposition itself must be true.

4.2 Augmented Semantics

It is clear that the system within which a program will actually be executed will include interference from sources other than the program, and that the basic operational semantics of Chapter 2 does not include a mechanism to model this interference. This section explores the two methods used in this work to bring external interference into the semantic model.

The rely/guarantee model of a system is, at the abstract level, just an interleaving of actions which conform to either the guarantee or the rely condition. The semantic model of the language is –abstractly, still– just the sequence of program steps. Assuming that the program was developed using the rely/guarantee rules, then all of those semantic model program steps correspond to the rely/guarantee model steps that conform to the guarantee condition. That implies that the semantic model presented previously is only a partial model of the systems that we are interested in, and that we need to augment the semantics to allow for steps based on external interference.

This augmentation is done in two ways: one form of the augmented semantics provides an explicit rule to incorporate the changes made by interference, and does so in such as way as to always distinguish the interference steps from the program steps. This distinguishing semantics fully implements the model assumed by the rely/guarantee framework. The second form is the merging semantics, which incorporates interference by perturbing the state component at the start and end of each program step. The three sets of semantics have different characteristics which make one or another more useful for specific proofs; however, they all describe the same system by taking a different perspective on interference.

4.2.1 The Distinguishing Augmented Semantics

The first of the two augmented semantic models is a very simple extension of the basic language semantics. It consists of two rules that define the relation $\xrightarrow[\_]{r}$. The relation is parameterized by a rely condition, so the relation for a specific rely condition, R is written $\xrightarrow[R]{r}$.

The first rule of this semantic model is a transitive wrapper that includes the entire $\xrightarrow{s}$ relation of the semantics from Chapter 2.

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

The A-S-Step rule means that all of the program steps of the basic semantics are a part of the distinguishing semantics. As the rule does the inclusion at the level of individual steps, the fine-grained concurrency already in the basic semantics is preserved and extended so that interference operates at the same granularity.

The second rule introduces interference steps into this semantic model.

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

The A-R-Step rule does not cause any change to the program text as this rule does not represent a program step. Instead, this rule changes the state component in a manner that will satisfy the rely condition.

These two rules jointly define the distinguishing version of the augmented semantics. The choice between the two rules is non-deterministic: it is certainly possible in this semantic model to completely execute the program from start to finish without any interference whatsoever. This does correspond to the rely/guarantee model of execution, though the likelihood of this happening —which is not addressed by either model— is very low.

An alternative, equivalent, definition of the distinguishing semantics can be given in terms of a union of the basic semantic relation and the rely condition lifted to configurations.

$$\xrightarrow[R]{r} \equiv \xrightarrow{s} \cup \{((S,\sigma),(S,\sigma')) \mid S \in \mathit{Stmt} \land (\sigma,\sigma') \in R\}$$

This definition is less useful in the proofs, however, as it is convenient to be able to reference the A-S-Step and A-R-Step directly.

It should come as no surprise that if the rely condition is the identity relation, then this semantic model becomes nearly equivalent to that of the basic semantic model — certainly the set of reachable configurations from any given initial configuration are the same for both models. The difference between the models is that this augmentation allows identity transitions: a transition from a configuration to itself. In that respect, the set of pairs of configurations —the semantic relation itself— of the augmented semantics is a strict superset of that of the basic semantics.

The inclusion of identity transitions in the distinguishing semantics leads to the unfortunate consequence that –though harmless in this context– means this semantic model includes infinite sequences of configurations that differ only by their state components. The context that renders this side-effect harmless is the overall framework assumption that execution starvation never occurs due to external interference. For the soundness proofs of the partial satisfaction properties, infinite sequences can be dealt with as these properties only apply to a subset of possible computations. In the proofs of the convergence property, we require a different semantic model.

4.2.2 The Merging Augmented Semantics

The second augmented semantics is, if anything, even simpler than the distinguishing semantics. It consists of a single rule which defines the semantic relation, $\xrightarrow[\_]{m}$, and is parameterized by a rely condition. As with the previous semantic relation, this semantic relation with a specific rely condition, R, is written $\xrightarrow[R]{m}$. This rule combines the two rules in the distinguishing semantics, merging their effects.

$$\boxed{\text{M-Step}}\ \frac{\begin{array}{c}[\![R]\!](\sigma_0, \sigma_1)\\ (S, \sigma_1) \xrightarrow{s} (S', \sigma_2)\\ [\![R]\!](\sigma_2, \sigma_f)\end{array}}{(S, \sigma_0) \xrightarrow[R]{m} (S', \sigma_f)}$$

The M-Step rule is just a basic semantic step that allows, directly, for interference to happen immediately before and after the program step. In terms of relational composition,

$$\xrightarrow[\_]{m} \equiv \big(\text{A-R-Step} \diamond \text{A-S-Step} \diamond \text{A-R-Step}\big)$$

that is, the merging semantics is equivalent to the specific composition of the rules from the distinguishing semantics.

This semantic model matches that assumed by the rely/guarantee framework: despite the appearance that interference steps happen nearly twice as often as program steps, recall that the rely condition is both reflexive and transitive. The latter means that two (or more) consecutive interference steps will be "seen" by the program as a single step. The former means that, in a sense, interference does not always have to alter the state: it could leave it unchanged.

The merging semantics does have two major differences relative to the distinguishing semantics: first, all transitions in this semantics must alter the textual component of the configuration; and second, it is no longer possible to distinguish the actions of the environment from those of the program (as the name indicates).

The feature in this semantic model that all transitions must alter the textual component of the configuration means that the infinite sequences of interference steps that are possible in the distinguishing semantics are not possible in the merging semantics. It should be noted that the merging semantic model only has this property because the basic semantic model does: if the basic semantics allowed transitions that did not alter the textual component, this would not be true. The net effect of this property (and the primary reason this model exists) is to simplify the convergence proofs — to use the distinguishing semantic model in the convergence proof one would have to formalize the assumption that external interference never causes execution starvation. Unfortunately, this property also makes the proofs of the partial satisfaction properties very difficult, as they require the ability to distinguish between program steps and interference; thus there are two augmented semantic models.

Merging the interference steps into the program steps limits the type of properties that can be used over pairs of state components. The only ones that are necessary in the required proofs, however, relate to the satisfaction of the post condition of (sub)programs. From that it is possible to use well-founded induction with the *While* construct, and to show that the pre condition of the right-hand component of a sequence construct is satisfied by the resultant states from execution of the left-hand component.

Restrict the set of transitions in the distinguishing semantics to those with differing textual components, and you have the basic semantics. You also have a subset of the merging semantics: precisely the case where the rely condition is equivalent to the identity relation.

Every multiple-step computation that is possible in the merging semantics is also possible in the distinguishing semantics (with “fill-in” steps). However, every finite multiple-step computation in the augmented semantics is also possible in the merging semantics, barring those that never use the A-S-Step rule. The result of this near equivalence is that for every non-trivial program that we are interested in we can treat any result in one of the augmented semantics as applying to both semantics.

4.3 Property Definitions

For the rely/guarantee rules to be considered sound with respect to the operational semantics, we must prove that two properties hold: first, that the partial correctness property holds, and second that a program developed using these rules will, during execution, converge upon the **nil** statement. The partial correctness property, in turn, depends on two properties: that all of the steps performed by the program conform to the guarantee condition, and that the post condition is satisfied by all pairs of initial and final states due to the program execution.

4.3.1 Partial Correctness

The partial correctness properties of a program relative to its specification deal with the states the program reaches under the assumption that the program does terminate. Traditionally, for sequential systems, this meant only that a program would produce, at termination, a set of states that satisfied the post condition of the specification. This requirement stands in the rely/guarantee framework, but partial satisfaction in this context requires that the program’s behaviour also satisfies the guarantee condition. This section defines and discusses these two properties that define partial satisfaction in the context of a rely/guarantee framework.

The partial correctness properties are defined in terms of the distinguishing augmented semantics; it would be possible to give an equivalent set of definitions in terms of the merging semantics, but the soundness proofs would be more complex.

Within

The $Within$ property is an assertion about the behaviour of a program, looking at the changes the program makes to the state component from the source to target configuration of each indivisible semantic transition. As changes to the state component made by interference are not due to the program itself, they are not directly of concern to this property, however, their affect on the actions of the program is, and so we use the distinguishing semantics to incorporate interference into our reasoning.

At its base, $Within$ is defined over single semantic transitions. It is then extended to multiple transitions, and then abstractions of the notation are added to allow for assumptions about the environmental context to be used. There are three versions of the $Within$ property as applied in different contexts. The first, denoted $Within_1$, applies only to pairs of configurations related through a single semantic transition. The second is denoted $Within_m$ and is applied to pairs of related configurations where all of the single steps between the two configurations satisfy the $Within_1$ property. The third is denoted $Within_s$ and is used to indicate that all computations from a given statement will behave according to the specified guarantee condition.

A pair of configurations related through a single semantic transition satisfies the definition of $Within$ if and only if

1. the transition is due to the A-S-Step rule and the state components of the source and target configurations together satisfy the guarantee condition; or

2. the transition is due to the A-R-Step rule.

These two cases are represented in the form of inference rules for direct use in the proofs of Chapter 6. The *Within-Prog* rule corresponds to the first case of the definition, and the *Within-Rely* rule corresponds to the second.

$$\boxed{\text{Within-Rely}}\ \frac{(C, C') \in \text{A-R-Step}}{Within_1(C, C', G)}$$

$$\boxed{\text{Within-Prog}}\ \frac{\begin{array}{c}((S,\sigma),(S',\sigma')) \in \text{A-S-Step}\\ [\![G]\!](\sigma,\sigma')\end{array}}{Within_1((S,\sigma),(S',\sigma'),G)}$$

It is interesting to note that, for this definition, the nature of the rely condition is wholly irrelevant. It can also be seen from *Within-Rely* that any interference transition will satisfy the definition of $Within$ for all valid guarantee conditions: thus interference is not directly relevant to the definition of $Within$.

A lemma that follows directly from the definitional inference rules is a "weakening" lemma that allows the replacement of the guarantee condition with another, more permissive, condition.

$$\boxed{\text{Within-Weaken}}\;\frac{\begin{array}{c}G \Rightarrow G' \\ \mathit{Within}_1(C, C', G)\end{array}}{\mathit{Within}_1(C, C', G')}$$

This lemma is defined on single-step transitions, as are the definitions; because of this, the lemma allows for free replacement of the rely condition in the semantic transition arrow. It is not hard to see that this is valid: if the rely condition is relevant to the transition, then the transition must be due to the A-R-Step rule, and is therefore subject to the *Within-Rely* portion of the definition. Otherwise the rely condition has no affect on the transition, in which case the transition derives from the A-S-Step rule, and the pair of state components must have satisfied the guarantee condition.

Having a definition that deals with single-step transitions is useful, but what is needed for the proofs is a means of extending this property to multiple-step transitions.

$$\boxed{\text{Within-Multi}}\;\frac{\begin{array}{c}C^{ij} = \{(C_i, C_j) \mid C_0 \xrightarrow[R]{r}\!* \, C_i \xrightarrow[R]{r} C_j \xrightarrow[R]{r}\!* \, C_f\} \\ \forall (C_i, C_j) \in C^{ij} \cdot \mathit{Within}_1(C_i, C_j, G)\end{array}}{\mathit{Within}_m(R, C_0, C_f, G)}$$

The essence of the definition of Within over multiple steps is stated as an inference rule, *Within-Multi*. The definition itself is not surprising: a multiple-step transition satisfies the definition of Within if and only if every possible intermediate single-step transition individually satisfies the definition of Within.

Stepping through the antecedent in turn, the first fixes the overall multiple-step transition that is of interest. The second antecedent defines the set of all intermediate pairs of configurations that are related by a single step of the semantic relation. It is important for this set to include all of the configurations that are related both to the overall source and target configurations. Finally, the third antecedent universally quantifies over all of the intermediate pairs of configurations to make sure that they all conform to the single-step definition of Within.

Given this notion of Within over multiple-step transitions, the *Within-Weaken* lemma can be altered to give a form valid for multiple-step transitions as well. The assumption in the single-step lemma is that the rely condition can be essentially ignored; for the multiple-step lemma, because Within_m is sensitive to all transitions in all of the computational paths between the source and target configurations, and since the rely condition is a partial cause of there being multiple paths in the first place[1], the multiple-step lemma requires an additional constraint.

$$\boxed{\text{Within-Weaken-Multi}}\;\frac{\begin{array}{c}R' \Rightarrow R \\ G \Rightarrow G' \\ \mathit{Within}_m(R, C, C', G)\end{array}}{\mathit{Within}_m(R', C, C', G')}$$

The extra constraint –the first antecedent in *Within-Weaken-Multi*– restricts the interference allowed by the rely condition in the consequent to be wholly contained within

[1]With non-deterministic constructs being the other cause.

the bounds of the interference allowed by the rely condition in the third antecedent. The essence of this restriction is that it reduces the number of computational paths between the source and target configurations. This reduction is a subset of the set of paths allowed by the original rely condition, and, given that the original set satisfied the definition of Within, the smaller set must as well.

The last form of the Within property $-\mathit{Within}_s-$ may be obtained through the use of the *Within-Abstract* rule.

$$\boxed{\text{Within-Abstract}}\ \frac{\forall \sigma, \sigma' \in \Sigma \cdot \left(\begin{array}{l} [\![P]\!]\sigma \wedge (S,\sigma) \xrightarrow[R]{r}\!\!* (\mathbf{nil}, \sigma') \\ \quad \Rightarrow\ \mathit{Within}_m(R, (S,\sigma), (\mathbf{nil},\sigma'), G) \end{array} \right)}{\mathit{Within}_s(P, R, S, G)}$$

Given a statement and a pre condition-satisfying state, if it is known that all terminating computations satisfy the Within_m property for a specific guarantee condition, then the Within_s property holds for that statement. As this only applies to terminating computations, the target configuration of the computation is omitted from the Within_s property entirely.

The statement-based Within_s property may be used to infer a Within_m property by using the *Within-Concrete* rule.

$$\boxed{\text{Within-Concrete}}\ \frac{\begin{array}{c} \mathit{Within}_s(P, R, S, G) \\ [\![P]\!](\sigma) \\ (S,\sigma) \xrightarrow[R]{r}\!\!* (\mathbf{nil}, \sigma') \end{array}}{\mathit{Within}_m(R, (S,\sigma), (\mathbf{nil},\sigma'), G)}$$

The antecedents of this rule require the Within_s property, a state which satisfies the contained pre condition, and a terminating computation starting from the contained statement and pre condition-satisfying state. Given these elements the Within_m property follows directly.

Post Condition Satisfaction

The second property required for a statement to satisfy the partial satisfaction property is that the post condition holds over the pair of initial and final states from all terminating computations starting with the given statement. This assumes that the initial state satisfies the specification's pre condition, but is not concerned with any intermediate states between the initial and final states.

This requirement is not complex, and comes directly from the general class of Floyd-Hoare rule systems. The major difference in the rely guarantee framework used here (and used in most rely/guarantee frameworks) is the insistence on post conditions of two states.

4.3.2 Convergence

The convergence of a sequence of configurations to a specific set of configurations is a surprisingly difficult property to define in an accurate, meaningful way for a rely/guarantee framework. The framework is predisposed to dealing with events that might possibly happen; convergence is about events that must definitely happen.

Unlike the properties of partial satisfaction –which are essentially state-based– convergence is based on the program text. For total correctness we need to know that all computational paths from a given set of starting configurations will, in a finite number of transitions, reach a configuration with a **nil** statement.

Though the desired property –convergence to a **nil** configuration– is sufficient, with partial satisfaction, to show total correctness, the proofs become very difficult to write using it. Thus a more general property is required that allows for convergence on an arbitrary set of configurations, which is characterized by a set of statements. The convergence property is only concerned with the statement component of configurations; the state component is not relevant to this property. Furthermore, as we are operating within a rely/guarantee framework, the environmental context in which a statement will be executed is of definite importance. The environmental context must therefore be a part of the definition.

First, we will restrict our definition of $Converges$ to the merging semantics. The potential infinitely recurring interference that is a part of the distinguishing semantics is not actually a part of the systems that we are interested in, as we assume that (execution) resource starvation never occurs. And, as this property is not directly concerned with the state components in the target configurations, distinguishing the program steps from the environment steps is unnecessary.

There are two forms of the $Converges$ property as used in the formal proofs. The first, $Converges_c$, deals with convergence starting from a specific configuration, and is embodied in the *Conv-I* inference rule.

$$\boxed{\text{Conv-I}}\ \frac{\begin{array}{c} C^i = \{C_i \mid C_0 \xrightarrow[R]{m}{*}\ C_i\} \\ C^f = \{(S_f, \sigma_f) \in \mathit{Config} \mid S_f \in \mathit{Set}_f\} \\ \forall C_i \in C^i \cdot \left(\exists C_f \in C^f \cdot (C_i \xrightarrow[R]{m}{*}\ C_f) \lor (C_f \xrightarrow[R]{m}{*}\ C_i)\right) \end{array}}{\mathit{Converges}_c(C_0, R, \mathit{Set}_f)}$$

A convergence property in the form $Converges_c(C_0, R, Set_f)$ is read: "All computations from the configuration C_0, allowing for interference bounded by R, will always reach a configuration with a textual component that is a member of Set_f in a finite number of steps".

The second form of the convergence property is $Converges_s$ which deals with convergence starting from a given statement and states satisfying a pre condition. It is possible to use the *Conv-Abstract* inference rule to obtain a $Converges_s$ property from a quantified $Converges_c$ property.

$$\boxed{\text{Conv-Abstract}}\ \frac{\forall \sigma \in \Sigma \cdot [\![P]\!](\sigma)\ \Rightarrow\ \mathit{Converges}_c((S, \sigma), R, \mathit{Set}_f)}{\mathit{Converges}_s(S, P, R, \mathit{Set}_f)}$$

Thus, $Converges_s(S, P, R, Set_f)$ is read: "Configurations formed from the initial statement, S, and states satisfying the pre condition, P, when executed in an environment where interference is bounded by the rely condition, R, will always reach a configuration with a textual component that is a member of the set of statements, Set_f".

Unsurprisingly, a $Converges_c$ property may be obtained from a $Converges_s$ property simply by supplying a suitable state.

$$\boxed{\text{Conv-Concrete}}\ \frac{[\![P]\!](\sigma) \qquad Converges_s(S, P, R, Set_f)}{Converges_c((S, \sigma), R, Set_f)}$$

The *Converges* property may be weakened by enlarging the set of final statements: if the starting point always reached an element of the smaller set, it will still always reach an element of the larger one. This is embodied in the ***Conv-Weaken*** inference rule.

$$\boxed{\text{Conv-Weaken}}\ \frac{R' \Rightarrow R \qquad Set_f \subseteq Set_f' \qquad Converges_c(C, R, Set_f)}{Converges_c(C, R', Set_f')}$$

There are several interesting things that this notation allows, depending on the form of the set of final statements. The simplest is that, if the target set of statements is the singleton set containing **nil**, then this property is equivalent to the usual notion of program termination; specifically, this is the property we desire for total correctness.

Similar to the singleton set containing **nil**, if the empty set is used as the target set then of all of the configurations formed from the initial statement and a pre condition-satisfying state, none of them are in the domain of the semantic relation. The empty set used as a target set implies that all of the possible initial configurations have halted. This does not imply computational divergence: that follows from the negation of the convergence property with a singleton set containing **nil**. This places the *Converges* property in a fairly myopic position: though it may assert that all computational paths must reach a configuration with a textual component that is a member of the target set, it asserts nothing about what computation may follow that configuration. In fact, in a pathological case – such as with a looping construct– it is entirely possible that a computation path may pass successively through many configurations containing an element of the target set.

Another interesting property of the *Converges* notation is that the set of target statements does not have to be minimal: a *Converges* assertion with a set of target statements that is equivalent to the set of all possible statements is a tautology for any combination of valid initial statements, pre conditions, and rely conditions. This is the case even though most of the statements in the set of all possible statements are completely unreachable from any given initial statement. The requirement of this property is that all paths from the initial statement will reach at least one element of the set of target statements. This particular feature of the property allows the use of target sets that contain statements with a quantified component where the quantification includes statements that cannot be reached (but, for the sake of showing termination, make no difference to the argument). For example, it allows for a target set with a variable quantified over the Boolean values in the test expression field of the *If* constructs, and allows that to be useful even in situations where it is known that one of the Boolean values cannot be reached.

Lastly, if the initial statement is an element of the target set, the *Converges* property is trivially true regardless of the environmental context. Having the initial statement in the target set means that all possible configurations that can be formed from that statement trivially satisfy the requirement that the computation must reach a configuration with a statement component that is a member of the target set. The convergence property is considered over the transitive closure of the semantic relation, thus the reflexive step of a configuration onto itself is included.

All this leads to the question of what allows for the deduction of the convergence property. The straightforward –though profoundly expensive, not to mention generally impossible– means of doing this would be to simply simulate all possible executions of the given program to see what happens.

It is fairly obvious, just from looking at the rule definitions in the operational semantics, that all of the constructs in the language –except While and the $\mathit{Atomic}/\mathit{STM}$ pair– have reduction-based semantics, and as long as there is an element in the target set that is a reduction of the initial statement, then the convergence property can be proven. For these statements the proofs are essentially structural: either the construct reduces directly based on the applicable rules, or it reduces because the applicable rules reduce a component.

Convergence properties on the While construct generally have one of two forms: either convergence to **nil** or convergence to a partially evaluated unfolding of the While construct. In the proofs in Chapter 6 we require the use of the latter to prove the former: at the very least a While construct must become an If construct before it can reach a **nil**.

Arguments about the convergence of the Atomic construct to **nil** have a problem that appears to be similar to that of the While construct's iterative nature. It is possible to show that an Atomic construct must converge on a STM construct of the form given by the STM-Atomic rule, but it is difficult to assert that any particular successor configuration must be reached. The key here is that, due to the nature of interference, it may be that the STM-Retry rule can be applied at nearly any point, and also –again due to the nature of interference– that since the STM-Retry rule may never be applied. As interference is represented as a relation and the A-R-Step rule does not provide a mechanism for selecting any particular state for the target configuration, this means that the behaviour of the STM construct is profoundly non-deterministic when evaluated with an arbitrary rely condition. So, then, the key is to somehow make the argument that the STM-Retry rule cannot be applied during one of the attempts to execute the STM construct. With the very tight constraint that the *Atomic-I* development rule adds –namely, that the rely condition cannot modify variables that are in the Atomic construct's body (which would, in turn, trigger the STM-Retry rule)– this argument can be made trivially.

Knowing the conditions under which we can definitely argue convergence for the While construct and $\mathit{Atomic}/\mathit{STM}$ pair, we can use structural induction once again to gain a complete convergence argument for a whole program. This is not the only possible means to argue convergence, however: showing that the semantic relation is, itself, well-founded with configurations formed from the target set as minimal elements is another mechanism. Indeed, that is precisely the argument that will be used in Section 6.4 to show that any assignment statement must terminate.

$\boxed{5}$ — Preparatory Lemmas

The purpose of this chapter is threefold: introduce the lemmas that are used in the formal soundness proofs of Chapter 6 and Appendix D; give a justification for their validity, as formal proofs are not provided; and provide a motivation for their existence and use. As such, this chapter is more modular than those prior, and each subsection can nearly be taken on its own. All of these lemmas are listed together in Appendix C, immediately prior to the proofs.

5.1 Miscellaneous

The lemmas in this section do not have a common underlying theme except that they express properties held by the semantics –direct and augmented– of the language in Appendix A.

5.1.1 Rely-Trivial

For any multiple-step transition between a pair of configurations this lemma concludes that their textual components must conform to the rely condition so long as all of the individual steps are a result of the specified rules. The specific rely condition is given in the first antecedent, and the rules are named in the second.

$$\boxed{\text{Rely-Trivial}}\ \frac{\begin{array}{c}(S,\sigma)\xrightarrow[R]{r}{*}\,(S',\sigma')\\ ((S,\sigma),(S',\sigma'))\in\left(\begin{array}{l}\text{A-R-Step}\cup\text{Assign-Eval}\cup\text{STM-Atomic}\cup\\ \text{If-Eval}\cup\text{If-T-E}\cup\text{If-F-E}\cup\text{Seq-E}\cup\\ \text{STM-Step}\cup\text{STM-Retry}\cup\text{Par-E}\cup\text{While}\end{array}\right)^{*}\end{array}}{[\![R]\!](\sigma,\sigma')}$$

All of the rules named in the second antecedent have one of two properties: either they never modify the state component, or, for the rules that do alter the state component, they do so explicitly according to the rely condition.

In the case of the base semantics, the antecedent will not contain the A-R-Step rule and the rest of the rules all leave the state component unmodified. As the identity relation is a subset of all possible rely conditions in this rely/guarantee framework, the consequent is trivially satisfied.

For the augmented semantics that distinguishes interference, $\xrightarrow[R]{r}$, we have the same partitioning of the rules as with the base semantics. All of the rules except A-R-Step trivially satisfy the consequent. The A-R-Step is now relevant, however, the consequent of this lemma is the antecedent of that semantic rule, so we have trivial conformance once more.

Finally, for the augmented semantics that wraps interference around each rule step, it is simply a case of noting that the change to the state component for the rules which would otherwise act as an identity is equivalent to the relational composition $R \diamond I \diamond R$. And that, of course, is precisely equivalent to R.

By far the most frequent use of this lemma is to show that the only changes to the state from the beginning of execution were caused by interference, and that in turn allows for the assertion that the pre condition still holds despite those changes.

5.1.2 Sequential-Effect

This lemma is valid under all three semantic relations, though only the $\xrightarrow[I]{r}\!*$ version is shown here. This lemma requires that there be no interference –the rely condition is the identity relation– which allows this result to be applicable under the distinguishing augmented semantics, as shown, as well as both the basic semantics in Chapter 2 and the merging augmented semantics.

$$\boxed{\text{Sequential-Effect}}\ \frac{(S, \sigma) \xrightarrow[I]{r}\!* (S', \sigma')}{\sigma' = \sigma \dagger (\mathit{Vars}(S) \lhd \sigma')}$$

The conclusion of this lemma gives the relation between the source and target states, indicating how they differ. Naturally, this lemma is used to characterize the effect that the body of an $\mathit{Atomic}/\mathit{STM}$ has on the state so that the effect can be shown to satisfy its guarantee and post conditions.

The basic observation is that a program can only change a given variable if the variable is named in the program's syntax. Conversely, if the program never names a given variable, then that variable cannot be changed by the program; and, given no interference, that variable will remain unchanged over the execution of that program.

5.1.3 Frame-Rule

This lemma is directly inspired by the work on separation logic and the axiom of the same name in that framework. The pragmatic intent of this lemma is the same: for any given interference-free computation, only the variables in the source program text are relevant to computation; all of the variables that are not in the program text can be of any value.

$$\boxed{\text{Frame-Rule}}\ \frac{\begin{array}{c}(S_0, \sigma_0) \xrightarrow[I]{r}\!* (S_1, \sigma_1)\\ (\mathit{Vars}(S_0) \lhd \sigma_0) = (\mathit{Vars}(S_0) \lhd \sigma_0')\\ (\mathit{Vars}(S_0) \lhd \sigma_1) = (\mathit{Vars}(S_0) \lhd \sigma_1')\\ (\mathit{Vars}(S_0) ⩤ \sigma_0) = (\mathit{Vars}(S_0) ⩤ \sigma_1)\\ (\mathit{Vars}(S_0) ⩤ \sigma_0') = (\mathit{Vars}(S_0) ⩤ \sigma_1')\end{array}}{(S_0, \sigma_0') \xrightarrow[I]{r}\!* (S_1, \sigma_1')}$$

More concretely, it is possible to have arbitrary values in the variables not named in the initial program text without any affect on the final result of the computation. The first antecedent gives a 'reference' computation; the following two antecedents ensure that the values of the variables named in the initial program text are the same in both pairs of states; and the last two antecedents ensure that all of the variables not named in the program text are unchanged through the program execution.

The restriction of this rule to interference-free computations –where the rely condition is the identity relation– may be more restrictive than is strictly necessary; it is, however, sufficient for the purposes of the formal proofs in Chapter 6. The interference-free constraint is what motivates the last two antecedents; allowing for interference would require that the last two antecedents take interference into account, and it is very likely that the rely condition would have to imply the identity over the variables named in the body of the program text. This does merit future work, and is noted in the final chapter of this work.

As with the *Sequential-Effect* lemma, the *Frame-Rule* lemma is used primarily to reason about the effects of an $Atomic/STM$ body in the surrounding context while proving the guarantee and post conditions.

5.1.4 Single-Eval-Assign and Single-Eval-If

In Chapter 3 the issue of multiple-state evaluation of expressions is discussed, and most of the development rules are carefully designed to avoid having to depend directly on evaluated variables because of this. The *Assign-I* and *If-b-I* rules, however, cannot avoid this issue, and so are designed to require a context that enforces interference-free behaviour from the environment on the variables in the expressions they depend on.

$$\boxed{\text{Single-Eval-Assign}}\ \frac{\begin{array}{c} R \Rightarrow I_{Vars(e)} \\ (mk\text{-}Assign(id, e), \sigma) \xrightarrow[R]{r}{*}\ (mk\text{-}Assign(id, v), \sigma') \\ v \in \mathbb{Z} \end{array}}{v = [\![e]\!](\sigma) = [\![e]\!](\sigma')}$$

$$\boxed{\text{Single-Eval-If}}\ \frac{\begin{array}{c} R \Rightarrow I_{Vars(e)} \\ (mk\text{-}If(b, body), \sigma) \xrightarrow[R]{r}{*}\ (mk\text{-}If(v, body), \sigma') \\ v \in \mathbb{B} \end{array}}{v = [\![b]\!](\sigma) = [\![b]\!](\sigma')}$$

Because of this restriction on the interference context, there are two lemmas that can relate the final evaluated value to the semantic value as denoted under any state out of a set of states that differ only by the permitted interference. In both of these lemmas we require, as the first antecedent, that the rely condition implies the identity relation with respect to the variables in the construct's expression. Combined with the semantic transition, this allows us to conclude that the evaluated value will be the same as the semantic meaning of the expression in either of the states denoted in the semantic transition. Note that the final antecedent of both of the lemmas –restricting the type of v to be either an integer or Boolean– matches the context conditions in the language semantics; in particular, the restriction in *Single-Eval-Assign* corresponds to states that only allow integers in their range.

The observation that motivates the lemma is simple: if nothing changes the variables that an expression depends on, then the evaluation of that expression will be the same no matter how much (or how little) interference occurs.

These lemmas are used to establish that the pre condition holds at the start of the execution of the body of an *If* construct developed using *If-b-I* and that the change to the state from an $Assign$ construct developed with *Assign-I* conforms to the form of the guarantee and post conditions.

5.1.5 While-interstices-pre and While-interstices-psat

These two lemmas depend on the partial satisfaction properties on the body of the $While$ loop. This restricts the use of the lemmas to contexts which have established the **psat** property on the body; structural induction in the proofs, for example. The lemmas are concerned with the states in the interstices between iterations of the body; an interstice, in this context, is a configuration between successive iterations of a $While$ loop's body.

The meaning of the first lemma, *While-interstices-pre*, is that the pre condition must hold on the interstitial states.

$$\boxed{\text{While-interstices-pre}}\ \frac{\begin{array}{c}[\![P]\!](\sigma)\\ (P \wedge b_s, R) \vdash \mathit{body}\ \textbf{psat}\ (G, W \wedge P)\\ R \ \Rightarrow\ W^* \wedge I_{Vars(b_s)}\\ wh = mk\text{-}While(b_s \wedge b_u, \mathit{body})\\ (wh, \sigma) \xrightarrow[R]{r}\!\!* (mk\text{-}Seq(\mathit{body}, wh), \sigma')\end{array}}{[\![P \wedge b_s]\!](\sigma')}$$

There are two cases to consider for this lemma: those where the target configuration of the last antecedent is reached without iterating through the body, and those where the body is iterated through at least once. Note that in both cases the form of the target configuration implies that the conditional of the $While$ must have evaluated to **true**, thus giving us the fact that b_s holds for σ'. If the target configuration is reached without iterating through the body then the pre condition holds on σ' due to the *PR-ident* lemma. In the second case –where the body has been iterated through at least once– we gain the fact that the pre condition holds for σ' through the post condition of the body and the *PR-ident* lemma.

The second lemma, *While-interstices-psat*, is very similar to the first, but is concerned with the post condition of the body over pairs of states when it is known that the body has been executed at least once. Because of the “at least once” requirement, this lemma is defined in terms of the merging semantics rather than the distinguishing semantics.

$$\boxed{\text{While-interstices-psat}}\ \frac{\begin{array}{c}[\![P]\!](\sigma)\\ (P \wedge b_s, R) \vdash \mathit{body}\ \textbf{psat}\ (G, W \wedge P)\\ R \ \Rightarrow\ W^* \wedge I_{Vars(b_s)}\\ wh = mk\text{-}While(b_s \wedge b_u, \mathit{body})\\ C^w = \{(wh, \sigma') \mid (wh, \sigma) \xrightarrow[R]{m}\!\!* (wh, \sigma') \wedge \sigma \neq \sigma'\}\end{array}}{\forall (wh, \sigma') \in C^w \cdot [\![W \wedge P]\!](\sigma, \sigma')}$$

This lemma uses the same first four antecedents as *While-interstices-pre*, but instead of a simple semantic transition in the fifth antecedent, defines the set of all reachable configurations that have the same textual component as the initial configuration. The conclusion, then, is that the post condition of the body must hold between the initial state and all states in the set defined in the fifth antecedent. This conclusion relies on three things: the fact that the W relation in the post condition is transitive; that the P term of the post condition is a predicate, and thus only applied to the right-hand state given the application rules used in this work; and the fact that interference does not affect the post condition due to the *QR-ident* lemma.

5.2 Composition and Isolation

The set of lemmas in this section state properties about how subcomponents of programs are related to the larger context. The composition lemmas allow a property of a subprogram to be lifted into a property of that subprogram in its larger context. The isolation lemmas go in the other direction, and allow a property of the larger program context to be asserted on a subprogram.

All of these lemmas are in the same spirit as the principle of structural induction itself; they merely specialize that for the specific program constructs.

It should come as no surprise that all of these lemmas are valid for both of the augmented relations. Only the $\xrightarrow[R]{r}$ form is shown here, however.

5.2.1 Isolation-If and Isolation-Seq-R

The first lemma here allows the body of an *If* construct to be treated separately from the enclosing construct when it is known that the expression will evaluate to a **true** value. The first antecedent of this rule ensures that the second transition in the second antecedent is not a reflexive step; i.e. it ensures that S' is a program text that actually follows from S.

$$\boxed{\text{Isolation-If}}\ \frac{\begin{array}{c} S' \in \mathit{If} \;\Rightarrow\; S'.\mathit{body} \neq S \\ (\mathit{mk\text{-}If}(b, S), \sigma_0) \xrightarrow[R]{r}{*}\, (\mathit{mk\text{-}If}(\mathbf{true}, S), \sigma_i) \xrightarrow[R]{r}{*}\, (S', \sigma_j) \end{array}}{(S, \sigma_0) \xrightarrow[R]{r}{*}\, (S', \sigma_j)}$$

The initial configuration of the consequent uses σ_0 as its state because it is possible that there was no interference during the evaluation of the test and unwrapping of the *If* construct. In this case the body would start in the same state as the overall *If* construct. Any other state that the body might start in will only differ from σ_0 by interference, and it is entirely possible within the augmented semantics for interference to happen before any change is made to the body text.

$$\boxed{\text{Isolation-Seq-R}}\ \frac{\begin{array}{c} S' \in \mathit{Seq} \;\Rightarrow\; S'.\mathit{right} \neq S \\ (\mathit{mk\text{-}Seq}(\mathbf{nil}, S), \sigma) \xrightarrow[R]{r}{*}\, (S', \sigma') \end{array}}{(S, \sigma) \xrightarrow[R]{r}{*}\, (S', \sigma')}$$

The *Isolation-Seq-R* is analogous to *Isolation-If* in that it strips off a structural wrapping that never modifies the state. Like an *If* construct for which the test will evaluate to **true**, the execution of a sequence with an empty left-hand component is effectively equivalent to just the execution of the right-hand component.

5.2.2 Seq-Equiv

The basic observation behind this lemma is that a sequence construct is just a passive wrapper around the left-hand component. To wit, not only can you take any computation, wrap a sequence around it, and get the same behaviour over the states, but you can also take the portion of a sequence executed by the Seq-Step rule –i.e. only the execution of the left-hand component of the sequence– and strip off the sequence construct while preserving the behaviour over the states. This lemma formalizes the notion that there is a sort of "equivalence modulo structure" in behaviour between a sequence and its left-hand component.

$$\boxed{\text{Seq-Equiv}}\;\frac{(S, \sigma) \xrightarrow[R]{r}\!\!* (S', \sigma')}{(\mathit{mk\text{-}Seq}(S, \mathit{right}), \sigma) \xrightarrow[R]{r}\!\!* (\mathit{mk\text{-}Seq}(S', \mathit{right}), \sigma')}$$

It is worth noting that the right-hand component of the sequence is free in this lemma — nothing whatsoever is said of its behaviour here. This lemma is restricted to the behaviour of the left-hand component; the tree-like structure of the program text prevents the need to worry about the right-hand component.

5.2.3 Isolation-While

This lemma derives from a combination of *Isolation-If*, *Seq-Equiv* and *Isolation-While*; unusually for these lemmas, the presentation of the semantic transition is done as a set membership test on the transitive closure of several semantic rules.

$$\boxed{\text{Isolation-While}}\;\frac{\begin{array}{c} \mathit{wh} = \mathit{mk\text{-}While}(b, \mathit{body}) \\ \tau = ((\mathit{mk\text{-}If}(\mathbf{true}, \mathit{mk\text{-}Seq}(\mathit{body}, \mathit{wh})), \sigma), (\mathit{wh}, \sigma')) \\ \tau \in \big(\text{A-R-Step} \cup \text{If-T-E} \cup \text{Seq-Step} \cup \text{Seq-E}\big)^* \end{array}}{(\mathit{body}, \sigma) \xrightarrow[R]{r}\!\!* (\mathbf{nil}, \sigma')}$$

The antecedents present what is really one execution of the body of a *While* construct: the semantic rules chosen in the third antecedent are the only rules that can be applied to the intermediate program texts that arise from the source configuration — the exclusion of the While rule means that the actual "looping" step is precluded. The first two antecedents are simply definitions to allow the lemma to fit within the width of a page.

This lemma is specifically adapted to all three sets proofs on the *While* construct, as they are long enough without having to apply all three of the lemmas this one derives from.

5.2.4 Isolation-STM

This is the first of the lemmas in this section that actually needs to manipulate the rely condition for the augmented semantics. As the purpose and effect of the *Atomic*/*STM* construct is to execute its body in isolation from any and all outside interference, the conclusion of this lemma uses the identity relation to represent the allowed interference.

$$\boxed{\text{Isolation-STM}}\ \frac{(\mathit{mk\text{-}STM}(\mathit{orig}, \sigma_0, S, \sigma_s), \sigma) \xrightarrow[R]{r}\!\!* (\mathit{mk\text{-}STM}(\mathit{orig}, \sigma_0, S', \sigma_s'), \sigma')}{(S, \sigma_s) \xrightarrow[I]{r}\!\!* (S', \sigma_s')}$$

The other thing of note is that the consequent uses the states contained in the STM construct to the exclusion of the state-components of the antecedent's source and target configurations. Otherwise, this lemma is just a formal statement of what the STM construct does.

5.2.5 Comp-Par

This lemma's conclusion is essentially semantic in nature, but requires the use of the Within property to justify it. The aim of this lemma is to lift a property of the subcomponents of the parallel to the overall parallel.

$$\boxed{\text{Comp-Par}}\ \frac{\begin{array}{c} [\![P]\!](\sigma) \\ \mathit{Within}_s(P, R \vee G_r, \mathit{left}, G_l) \\ \mathit{Within}_s(P, R \vee G_l, \mathit{right}, G_r) \\ (\mathit{left}, \sigma) \xrightarrow[R \vee G_r]{r}\!\!* (\mathit{left}', \sigma') \\ (\mathit{right}, \sigma) \xrightarrow[R \vee G_l]{r}\!\!* (\mathit{right}', \sigma') \end{array}}{(\mathit{mk\text{-}Par}(\mathit{left}, \mathit{right}), \sigma) \xrightarrow[R \vee G_l \vee G_r]{r}\!\!* (\mathit{mk\text{-}Par}(\mathit{left}', \mathit{right}'), \sigma')}$$

The last pair of antecedents establish the form of the consequent: they define the textual subcomponents, state, and the rely conditions. Since these two antecedents have their states in common they restrict the computations from the initial subcomponents to the successor states that both can reach. The combination of the rely conditions in the consequent is a simple disjunction as there is no means to restrict it further without more knowledge of the actual rely conditions.

The first antecedent restricts the initial state, ensuring that the next two antecedents are actually relevant. Those two antecedents –the second and third– establish that the behaviour of each subcomponent is within the interference that their opposite subcomponent is able to tolerate. This behavioural constraint is what allows the parallel construct to be composed of the two subcomponents at all.

The lemma itself is only used in the convergence proof of the parallel construct. There the widening of the interference is tolerable, as the convergence property is essentially structural in nature.

5.2.6 Isolation-Par-L and Isolation-Par-R

These lemmas are key to being able to do the guarantee and post proofs using natural deduction and structural induction. As noted at the definition of the *Par-I* development rule in Section 3.2, the interference seen by one branch of a parallel construct is the combination of the interference seen by the overall parallel construct plus interference that arises from the behaviour of the other branch. We can use this knowledge to isolate one of the branches from the parallel construct by using the guarantee from the other branch and manipulating the rely condition given to the augmented semantic relation.

$$\boxed{\text{Isolation-Par-L}}\ \frac{\begin{array}{c}[\![P]\!](\sigma)\\ (P, R \vee G_r) \vdash \mathit{left}\ \textbf{sat}\ (G_l, Q_l)\\ (P, R \vee G_l) \vdash \mathit{right}\ \textbf{sat}\ (G_r, Q_l)\\ (\mathit{mk}\text{-}\mathit{Par}(\mathit{left}, \mathit{right}), \sigma) \xrightarrow[R]{r}{*}\ (\mathit{mk}\text{-}\mathit{Par}(\mathit{left}', \mathit{right}'), \sigma')\end{array}}{(\mathit{left}, \sigma) \xrightarrow[R \vee G_r]{r}{*}\ (\mathit{left}', \sigma')}$$

$$\boxed{\text{Isolation-Par-R}}\ \frac{\begin{array}{c}[\![P]\!](\sigma)\\ (P, R \vee G_r) \vdash \mathit{left}\ \textbf{sat}\ (G_l, Q_l)\\ (P, R \vee G_l) \vdash \mathit{right}\ \textbf{sat}\ (G_r, Q_l)\\ (\mathit{mk}\text{-}\mathit{Par}(\mathit{left}, \mathit{right}), \sigma) \xrightarrow[R]{r}{*}\ (\mathit{mk}\text{-}\mathit{Par}(\mathit{left}', \mathit{right}'), \sigma')\end{array}}{(\mathit{right}, \sigma) \xrightarrow[R \vee G_l]{r}{*}\ (\mathit{right}', \sigma')}$$

Specifically, we know from the antecedent that the left- and right-hand branches of the parallel will behave according to their respective guarantee conditions, assuming interference that is a combination of the overall rely and the behaviour of the other branch. Combine that with knowledge of the overall parallel construct's successor configurations, and we can then reason about the successor configurations of one branch on its own. The rely on the consequent of both of these lemmas is expanded with the guarantee condition of the other branch to incorporate the activity of the other branch into the consequent's target state component.

5.3 Behavioural

The lemmas related to the definition of *Within* are all straightforward. As the definition of *Within* is only given for the $\xrightarrow[R]{r}$ augmented semantics, these lemmas are only valid for reasoning in that context. The overall results do hold in general, however, and could be adapted for the merging semantics. These lemmas are used primarily in the guarantee proofs.

5.3.1 Within-Relation

The *Within* property is concerned with whether or not transitions conform to a guarantee condition. This means that a transition –single- or multiple-step– that has the *Within* property is comprised of pairs of states which satisfy either the guarantee or rely condition; extending this specifically to multiple-step transitions, this means that the transition is a sequence of pairs of states that satisfy either the guarantee or rely condition.

$$\boxed{\text{Within-Relation}}\ \frac{\mathit{Within}_m(R, (S, \sigma), (S', \sigma'), G)}{[\![(R \vee G)^*]\!](\sigma, \sigma')}$$

The net effect of a series of transitions that satisfy either the guarantee or rely condition is a pair of states –the source and target state– that satisfy the transitive closure of the disjunction of the guarantee and rely conditions. This lemma just allows for the direct deduction of the relational satisfaction from a Within_m property.

5.3.2 Within-Rely-Trivial

The purpose of this lemma is to dispose of pairs of configurations which both have **nil** as their textual component. Any sequence of configurations with **nil** as the textual component for all of them will trivially behave according to the guarantee condition: as there is no program to execute at this point, there is no behaviour.

$$\boxed{\textsf{Within-Rely-Trivial}}\ \frac{(\mathbf{nil}, \sigma) \xrightarrow[R]{r}{*}\ (\mathbf{nil}, \sigma')}{\mathit{Within}_m(R, (\mathbf{nil}, \sigma), (\mathbf{nil}, \sigma'), G)}$$

The lemma derives from *Within-Rely*, *Within-Multi* and the observation that all configurations with **nil** as their textual component are only in the domain of the A-R-Step semantic rule. That semantic rule ensures that the textual component of all successor configurations will remain **nil**. All of the individual steps, then, trivially satisfy the *Within-Rely* definition as the antecedent of *Within-Rely* is precisely that the steps be within A-R-Step. Knowing that, the *Within-Multi* definition simply lifts that Within result to the whole sequence.

5.3.3 Within-Prog-Cor

This lemma is of the same spirit as *Within-Rely-Trivial* in that it captures a set of related configurations which trivially conform to any guarantee condition. The purpose of this lemma is to capture those program steps that do not change the state component.

$$\boxed{\textsf{Within-Prog-Cor}}\ \frac{(C, C') \in \left(\begin{array}{l}\textsf{A-R-Step} \cup \textsf{Assign-Eval} \cup \textsf{STM-Atomic} \cup \\ \textsf{If-Eval} \cup \textsf{If-T-E} \cup \textsf{If-F-E} \cup \textsf{Seq-E} \cup \\ \textsf{STM-Step} \cup \textsf{STM-Retry} \cup \textsf{Par-E} \cup \textsf{While}\end{array}\right)}{\mathit{Within}_1(C, C', G)}$$

For the semantic rules listed in the antecedent we know that they will always leave the state unchanged, or they will explicitly modify it according to the rely condition. For the former case we have trivially satisfied the guarantee condition, giving us the Within property according to the *Within-Prog* definition. For the latter, we gain the Within property according to the *Within-Rely* definition, just as the *Within-Rely-Trivial* lemma does.

Unlike *Within-Rely-Trivial*, this lemma is given over a single step rather than over many steps. A version of this lemma could be written for multiple steps, deriving from an application of *Within-Multi*, but none of the proofs require such a lemma.

5.3.4 Within-Equiv

The *Within-Equiv* lemma is intended to allow the Within_1 property on a pair of configurations to be transferred to another pair of configurations. This lemma is used in the proofs to lift the Within_1 property on a step of a construct's sub-component to the step taken by the overall construct.

$$\boxed{\text{Within-Equiv}}\ \frac{(S_0,\sigma_0)\xrightarrow[R]{r}(S_1,\sigma_1)\quad (S_0',\sigma_0)\xrightarrow[R]{r}(S_1',\sigma_1)\quad (S_0=S_1)\ \Leftrightarrow\ (S_0'=S_1')\quad \mathit{Within}_1((S_0,\sigma_0),(S_1,\sigma_1),G)}{\mathit{Within}_1((S_0',\sigma_0),(S_1',\sigma_1),G)}$$

Even given that the state elements of the pairs are the same, there is a catch to this transfer of the Within_1 property. It matters whether or not the transition between the two pairs of configurations was due to the A-R-Step or A-S-Step semantic rule; the same rule must have been the cause of both transitions. The third antecedent captures this requirement as, for single steps, the mechanism to distinguish between the two semantic rules is to check the textual components of the source and target configurations for equality.

5.3.5 Within-Concat

As soon as we start reasoning about multiple-step transitions using the Within_m property it quickly becomes useful to be able to concatenate the properties together over adjacent multiple-step transitions.

$$\boxed{\text{Within-Concat}}\ \frac{C^i=\{C_i \mid C_0 \xrightarrow[R]{r}{*}\ C_i \xrightarrow[R]{r}{*}\ C_f\}\quad C^p\subseteq C^i\quad \forall C_i\in C^i\cdot\left(\exists C_p\in C^p\cdot C_i\xrightarrow[R]{r}{*}\ C_p \vee C_p\xrightarrow[R]{r}{*}\ C_i\right)\quad \forall C_p\in C^p\cdot \mathit{Within}_m(R,C_0,C_p,G)\wedge \mathit{Within}_m(R,C_p,C_f,G)}{\mathit{Within}_m(R,C_0,C_f,G)}$$

The key to this lemma –both here and in its use in the proofs– is to note that C^p is a pinch set.[1] Knowing this, the lemma comes down to simply verifying that every computational path on either side of the pinch set has the Within_m property.

5.4 Convergence

These lemmas are based on the convergence properties defined in Section 4.3.2 and are only defined for reasoning on the $\xrightarrow[\_]{m}$ augmented semantics though the results are valid more generally.

5.4.1 Conv-Concat

This lemma gives the conditions under which two *Converges* assertions may be combined. A frequent strategy used in the convergence proofs is to prove that a statement will always reach a set of statements that are part of the way to **nil**, and then show that each statement in that set will, in turn, converge on **nil**.

[1]See Section 4.1.4.

$$\boxed{\text{Conv-Concat}}\ \frac{\begin{array}{c} \mathit{Converges}_c(C_0, R, \mathit{Set}_0) \\ C^i = \{(S_i, \sigma_i) \mid C_0 \xrightarrow[R]{m}{*}\, (S_i, \sigma_i) \wedge S_i \in \mathit{Set}_0\} \\ \forall C_i \in C^i \cdot \mathit{Converges}_c(C_i, R, \mathit{Set}_f) \end{array}}{\mathit{Converges}_c(C_0, R, \mathit{Set}_f)}$$

The first antecedent gives the initial configuration and the set of target statements which it will reach. The second antecedent defines the set of all reachable configurations from the initial configuration for which each configuration in the set has a textual component from the first target set. Finally, the last antecedent ensures that each of the configurations given in the second antecedent will converge on a statement in the final target set. The last antecedent essentially verifies the concatenation by brute force, but it fits perfectly in the convergence proofs in Chapter 6.

5.4.2 Conv-Wrap-Seq, Conv-Wrap-Par, and Conv-Wrap-STM

These three lemmas are intended to lift a convergence result on a construct's component to a convergence result on the construct itself. The first of these deals with the Seq construct.

$$\boxed{\text{Conv-Wrap-Seq}}\ \frac{\begin{array}{c} \mathit{Converges}_c((\mathit{left}, \sigma), R, \mathit{Set}_f) \\ \mathit{Set}'_f = \{\mathit{mk\text{-}Seq}(S, \mathit{right}) \mid S \in \mathit{Set}_f\} \end{array}}{\mathit{Converges}_c((\mathit{mk\text{-}Seq}(\mathit{left}, \mathit{right}), \sigma), R, \mathit{Set}'_f)}$$

This lemma is simple: given a convergence result on some statement, it is possible to wrap that statement in a Seq construct and have an analogous convergence result still hold if the target set is altered by wrapping all of its elements in a Seq construct.

The next lemma –for the Par construct– has somewhat more restrictive antecedents than that for the Seq construct.

$$\boxed{\text{Conv-Wrap-Par}}\ \frac{\begin{array}{c} (P, R \vee G_r) \vdash \mathit{left}\ \textbf{psat}\ (G_l, Q_l) \\ (P, R \vee G_l) \vdash \mathit{right}\ \textbf{psat}\ (G_r, Q_r) \\ \mathit{Converges}_c((\mathit{left}, \sigma), R \vee G_r, \mathit{Set}_l) \\ \mathit{Converges}_c((\mathit{right}, \sigma), R \vee G_l, \mathit{Set}_r) \\ \mathit{Set}_f = \{\mathit{mk\text{-}Par}(S_l, S_r) \mid S_l \in \mathit{Set}_l \wedge S_r \in \mathit{Set}_r\} \end{array}}{\mathit{Converges}_c((\mathit{mk\text{-}Par}(\mathit{left}, \mathit{right}), \sigma), R, \mathit{Set}_f)}$$

The partial satisfaction antecedents on the left and right components are required to ensure that they conform to their rely and guarantee conditions. The last antecedent generates a target set that is essentially the Cartesian production of the left and right target sets. The conclusion weakens the contained rely condition directly, as it is always possible to reduce the amount of possible interference in a convergence result.

The last of these lemmas applies to the STM construct in the restricted situation where a retry transition cannot happen.

$$\boxed{\text{Conv-Wrap-STM}}\ \frac{\begin{array}{c} C_0 = (\mathit{mk\text{-}STM}(\mathit{body}, \sigma_0, \mathit{body}, \sigma_0), \sigma) \\ \mathit{Converges}_c((\mathit{body}, \sigma_0), I, \mathit{Set}_f) \\ \forall \tau \in \{(C_i, C_j) \mid C_0 \xrightarrow[R]{m}{*}\, C_i \xrightarrow[R]{m} C_j\} \cdot \tau \notin \text{STM-Retry} \end{array}}{\mathit{Converges}_c(C_0, R, \{S \in \mathit{STM} \mid S.\mathit{body} \in \mathit{Set}_f\})}$$

The first antecedent simply names the initial configuration from which we wish to infer a convergence property; it must contain a STM construct that has not yet started execution. The second antecedent is a convergence result on the body of the STM construct, giving the convergence result that we wish to lift to the STM. The last antecedent ensures that none of the transitions –as represented by pairs of configurations– that follow from the source configuration are due to the STM-Retry semantic rule. The conclusion uses a target set which is, by its sparse definition, a strict superset of the actual minimal target set, as the set includes elements which are unreachable given the semantics.

$\boxed{6}$ — Proving Soundness

6.1 Overall Theorems

There are two major theorems that, together, constitute the proof of soundness of the rely/guarantee rules of Chapter 3 with respect to the language semantics of Chapter 2. The first of these theorems is defined by the *psat-I* rule in Figure 6.1, and establishes the properties that are required for partial satisfaction. The second theorem is defined by the *sat-I* rule in Figure 6.1, and establishes what is necessary to lift a partial satisfaction result to complete satisfaction.

$$\boxed{\text{psat-I}}\ \frac{\begin{array}{c} \mathit{Within}_s(P, R, S, G) \\ \forall \sigma, \sigma' \in \Sigma \cdot \Big([\![P]\!](\sigma) \wedge (S, \sigma) \xrightarrow[R]{r}{*}\ (\mathbf{nil}, \sigma') \Big) \ \Rightarrow\ [\![Q]\!](\sigma, \sigma') \end{array}}{(P, R) \vdash S \ \mathbf{psat}\ (G, Q)}$$

$$\boxed{\text{sat-I}}\ \frac{\begin{array}{c} (P, R) \vdash S \ \mathbf{psat}\ (G, Q) \\ \mathit{Converges}_s(S, P, R, \{\mathbf{nil}\}) \end{array}}{(P, R) \vdash S \ \mathbf{sat}\ (G, Q)}$$

Figure 6.1: Theorem PSAT and Theorem SAT embodied as inference rules.

The proof for the partial satisfaction theorem is shown in Figure 6.2. The proof is a deduction of the properties of Within and post condition satisfaction –described in Section 4.3.1– from the assumption that the rely/guarantee development rules were used to develop the program represented by S and the assumption that the program and a pre condition satisfying state may terminate. It is important to include the second assumption: it is used to constrain the proof to only those computations that converge on the **nil** statement. We are not interested in those computations that do no converge, as in those cases there is no final state for the post condition to be applied to.

The overall structure of the proof in Figure 6.2 (and the lemmas it depends upon) is based on the use of structural induction; the sub-components of the language constructs are assumed to the have desired properties, and the overall construct is shown to have the desired property based on these and the construct's semantics.

The first element of the hypothesis in the Theorem PSAT proof and all of the partial satisfaction lemma proofs gives the specification statement of the program in terms of the **sat** property instead of the **psat** property. The proofs as given work equally well with either –and are stronger as **psat** versions– but the latter approach would have required a set of parallel versions of the rules based solely on **psat** rather than **sat**.

The main strategy of this proof is to show that for all specific computations –i.e. all computations from a specific initial configuration to a specific final configuration– that the behavioural and post condition properties hold. Now that we know that the properties hold

from $(P, R) \vdash S$ **sat** (G, Q); $\Pi = \{(\sigma, \sigma') \in \Sigma \times \Sigma \mid [\![P]\!](\sigma) \wedge (S, \sigma) \xrightarrow[R]{r}{*} (\textbf{nil}, \sigma')\}$

1 **from** $(\sigma_0, \sigma_f) \in \Pi$

1.1 $S \in \mathit{Stmt}$ — h

1.2 **from** $S \in \mathit{Assign}$
infer $\mathit{Within}_m(R, (S, \sigma_0), (\textbf{nil}, \sigma_f), G) \wedge [\![Q]\!](\sigma_0, \sigma_f)$ — h, h1, h1.2, *Assign-Within*, *Assign-Post*

1.3 **from** $S \in \mathit{Atomic}$
infer $\mathit{Within}_m(R, (S, \sigma_0), (\textbf{nil}, \sigma_f), G) \wedge [\![Q]\!](\sigma_0, \sigma_f)$ — h, h1, h1.3, *Atomic-Within*, *Atomic-Post*

1.4 **from** $S \in \mathit{If}$
infer $\mathit{Within}_m(R, (S, \sigma_0), (\textbf{nil}, \sigma_f), G) \wedge [\![Q]\!](\sigma_0, \sigma_f)$ — h, h1, h1.4, *If-Within*, *If-Post*

1.5 **from** $S \in \mathit{Par}$
infer $\mathit{Within}_m(R, (S, \sigma_0), (\textbf{nil}, \sigma_f), G) \wedge [\![Q]\!](\sigma_0, \sigma_f)$ — h, h1, h1.5, *Par-Within*, *Par-Post*

1.6 **from** $S \in \mathit{Seq}$
infer $\mathit{Within}_m(R, (S, \sigma_0), (\textbf{nil}, \sigma_f), G) \wedge [\![Q]\!](\sigma_0, \sigma_f)$ — h, h1, h1.6, *Seq-Within*, *Seq-Post*

1.7 **from** $S \in \mathit{While}$
infer $\mathit{Within}_m(R, (S, \sigma_0), (\textbf{nil}, \sigma_f), G) \wedge [\![Q]\!](\sigma_0, \sigma_f)$ — h, h1, h1.7, *While-Within*, *While-Post*

infer $\mathit{Within}_m(R, (S, \sigma_0), (\textbf{nil}, \sigma_f), G) \wedge [\![Q]\!](\sigma_0, \sigma_f)$ — $\vee$-E(1.1–1.7)

2 $\forall (\sigma_0, \sigma_f) \in \Pi \cdot \mathit{Within}_m(R, (S, \sigma_0), (\textbf{nil}, \sigma_f), G)$ — $\forall$-I(1)

3 $\mathit{Within}_s(P, R, S, G)$ — h, 2, *Within-Abstract*

4 $\forall (\sigma_0, \sigma_f) \in \Pi \cdot [\![Q]\!](\sigma_0, \sigma_f)$ — $\forall$-I(1)

infer $\mathit{Within}_s(P, R, S, G) \wedge \forall (\sigma_0, \sigma_f) \in \Pi \cdot [\![Q]\!](\sigma_0, \sigma_f)$ — $\wedge$-I(3,4)

Figure 6.2: Proof of Theorem PSAT.

for all of the pairs of initial and final configurations, we can generalize the properties to all computations for that statement given the environmental context.

The individual boxed inferences that establish the properties per language construct, lines 1.2–1.7, invoke lemmas which are logically a part of the overall proof. Splitting the contents of each **from/infer** box out into lemmas avoids the difficulty involved in reading and writing a proof that would have spanned a good dozen pages. Because of this split, and the fact that these lemmas are essentially a part of the overall proof, the lemmas have additional hypotheses not shown in lines 1.2–1.7. The additional hypotheses are of two kinds: the first are just binding definitions, and could be substituted throughout each of the lemmas without any real change in semantics; the second is a hypothesis to indicate that the lemma is actually a part of a structural induction, and these appear as $\mathit{IH}\text{-}S$ predicate, indicating the Induction Hypothesis on partial Satisfaction. The $\mathit{IH}\text{-}S$ predicate is applied to all of the subcomponents of the language construct the lemma is working with.

The proof of the complete satisfaction theorem is shown in Figure 6.3. This proof deduces that a program developed with the rely/guarantee rules will always converge during execution to a configuration containing **nil** as its textual component. This proof assumes that the partial satisfaction properties are valid, and makes the further assumption that there is at least one state which satisfies the pre condition of the specification. There are no assumptions about the final configuration, as the task of this proof is to prove that they exist

	from $(P, R) \vdash S$ **sat** $(G, Q);\ \Pi = \{\sigma \in \Sigma \mid [\![P]\!](\sigma)\};\ \Pi \neq \{\,\}$	
1	**from** $\sigma_0 \in \Pi$	
1.1	$S \in \mathit{Stmt}$	h
1.2	**from** $S \in \mathit{Assign}$	
	infer $\mathit{Converges}_c((S, \sigma_0), R, \{$**nil**$\})$	h, h1, h1.2, *Assign-Converges*
1.3	**from** $S \in \mathit{Atomic}$	
	infer $\mathit{Converges}_c((S, \sigma_0), R, \{$**nil**$\})$	h, h1, h1.3, *Atomic-Converges*
1.4	**from** $S \in \mathit{If}$	
	infer $\mathit{Converges}_c((S, \sigma_0), R, \{$**nil**$\})$	h, h1, h1.4, *If-Converges*
1.5	**from** $S \in \mathit{Par}$	
	infer $\mathit{Converges}_c((S, \sigma_0), R, \{$**nil**$\})$	h, h1, h1.5, *Par-Converges*
1.6	**from** $S \in \mathit{Seq}$	
	infer $\mathit{Converges}_c((S, \sigma_0), R, \{$**nil**$\})$	h, h1, h1.6, *Seq-Converges*
1.7	**from** $S \in \mathit{While}$	
	infer $\mathit{Converges}_c((S, \sigma_0), R, \{$**nil**$\})$	h, h1, h1.7, *While-Converges*
	infer $\mathit{Converges}_c((S, \sigma_0), R, \{$**nil**$\})$	$\vee$-E(1.1–1.7)
2	$\forall \sigma \in \Pi \cdot \mathit{Converges}_c((S, \sigma), R, \{$**nil**$\})$	$\forall$-I(1)
	infer $\mathit{Converges}_s(S, P, R, \{$**nil**$\})$	h, 2, *Conv-Abstract*

Figure 6.3: Proof of Theorem SAT.

and are unavoidable.

The overall structure of this proof is almost exactly the same as for Theorem PSAT, and the contents of the **from/infer** box have been split out into lemmas in exactly the same manner, and for exactly the same reason. The hypothesis representing structural induction in the lemmas is represented by the $\mathit{IH}\text{-}T$ predicate, as it is the Induction Hypothesis on Termination. This proof does depend on Theorem PSAT for its validity: it is not uncommon in the lemmas to conclude something that follows from applying Theorem PSAT to a computation involving a subcomponent of the language construct involved in the lemma.

6.2 Partial Satisfaction Behavioural Lemmas

All of the lemmas in this section follow essentially the same pattern: define the set of all intermediate transitions between the source and target configurations of the hypothesis; use case analysis on the semantic rules which govern the possible transitions to deduce that all transitions of each possible semantic rule satisfy the single-step Within property; introduce a universal quantifier on that property; then, finally, promote that to a multiple-step Within property.

Establishing the single-step Within property on most of the transitions can be done using trivial applications of the *Within-Rely* and *Within-Prog-Cor* lemmas. This comes about due to the fact that most of the semantic rules do not alter the state component of the configuration in any way; most of the work for single-step transitions involves showing a structural dependence on a sub-component.

Of the lemmas for the language constructs not explicitly described in a following subsection we will touch on a few interesting points here. The *Seq-Within* lemma uses existential elimination and the post condition of a left-hand component to discharge the pre condition on the latter half of the proof, allowing structural induction to infer the behavioural

$\boxed{\textit{Assign-Within}}$

from	$(P, R) \vdash \textit{mk-Assign}(id, e)$ **sat** (G, Q); $[\![P]\!](\sigma_0)$; $(\textit{mk-Assign}(id, e), \sigma_0) \xrightarrow[R]{r}{*} (\textbf{nil}, \sigma_f)$	
1	$R \Rightarrow I_{Vars(e) \cup \{id\}}$	h, *Assign-I*
2	$G = \{(\sigma, \sigma \dagger \{id \mapsto [\![e]\!](\sigma)\}) \mid \sigma \in \Sigma\} \cup I$	h, *Assign-I*
3	$T^i = \{(C_1, C_2) \mid (\textit{mk-Assign}(id, e), \sigma_0) \xrightarrow[R]{r}{*} C_1 \xrightarrow[R]{r} C_2 \xrightarrow[R]{r}{*} (\textbf{nil}, \sigma_f)\}$	definition
4	**from** $(C_1, C_2) \in T^i$	
4.1	$(C_1, C_2) \in (\text{A-R-Step} \cup \text{Assign-Eval} \cup \text{Assign-E})$	h4, inspection of $\xrightarrow[R]{r}$
4.2	**from** $(C_1, C_2) \in \text{A-R-Step}$	
	infer $\textit{Within}_1(C_1, C_2, G)$	h4.2, *Within-Rely*
4.3	**from** $(C_1, C_2) \in \text{Assign-Eval}$	
	infer $\textit{Within}_1(C_1, C_2, G)$	h4.3, *Within-Prog-Cor*
4.4	**from** $(C_1, C_2) \in \text{Assign-E}$; $C_1 = (S_1, \sigma_1)$; $C_2 = (S_2, \sigma_2)$	
4.4.1	$S_1 \in \textit{Assign}$	h4.4, Assign-E
4.4.2	$S_1.id = id$	h4, 4.4.1, inspection of $\xrightarrow[R]{r}$
4.4.3	$S_1.e \in \mathbb{Z}$	h4.4, 4.4.1, Assign-E
4.4.4	$\sigma_2 = \sigma_1 \dagger \{id \mapsto S_1.e\}$	h4.4, Assign-E
4.4.5	$S_1.e = [\![e]\!](\sigma_0) = [\![e]\!](\sigma_1)$	1, h4, 4.4.1, 4.4.2, 4.4.3, *Single-Eval-Assign*
4.4.6	$[\![G]\!](\sigma_1, \sigma_2)$	2, 4.4.4, 4.4.5
	infer $\textit{Within}_1(C_1, C_2, G)$	h4.4, 4.4.6, *Within-Prog*
	infer $\textit{Within}_1(C_1, C_2, G)$	$\vee$-E(4.1–4.4)
5	$\forall (C_1, C_2) \in T^i \cdot \textit{Within}_1(C_1, C_2, G)$	$\forall$-I(4)
infer	$\textit{Within}_m(R, (\textit{mk-Assign}(id, e), \sigma_0), (\textbf{nil}, \sigma_f), G)$	3, 5, *Within-Multi*

Figure 6.4: Proof of the behavioural lemma for the $Assign$ construct.

property of the right-hand component of the sequence.

The *If-Within* lemma has to split cases corresponding to the computational paths at the point immediately after the elimination rules of the *If* construct; this requires constructing a pinch set that contains the reachable configurations with **nil** or the body of the *If* in the textual component of the configuration. With that accomplished, the proof can proceed in much the same manner as *Seq-Within*, but with the addition of a case distinction on the textual component in the latter half. The *If-Within* lemma is only given using the *If-I* development rule; trivial changes along the same lines as those for the post condition lemma *If-Post* are all that is needed for a version of *If-Within* using the *If-b-I* development rule.

The *Par-Within* lemma is pleasantly simple, with the case distinctions on the left-hand and right-hand branches of the parallel using symmetric arguments. Further, the argument for a step of one of the branches satisfying the single-step $Within$ property follows directly from the application of structural induction and the *Within-Weaken* rule.

6.2.1 Assign

The assignment construct has the interesting property that for this language it is ultimately the source of all modifications from the program to the state component between any two configurations. Happily, it also has one of the more straightforward proofs, and so it is used here to show the general structure of the behavioural lemmas.

Figure 6.4 contains the full proof of the lemma; as mentioned earlier, the entire lemma should be considered as though it were substituted into Theorem PSAT in Figure 6.2. At the topmost level, this lemma allows the conclusion that all computational paths between two specific configurations satisfy a guarantee condition.

Looking first at just the top level lines, there are five intermediate steps. Lines 1 & 2 import the required antecedents from the *Assign-I* development rule, giving us the constraint imposed upon the rely condition by the development rule as well as the definition of the guarantee condition that the assignment's behaviour must match.

The first few steps where the antecedents of the development rule are imported into the proof are common to all of the lemma proofs in this chapter. Each of these proofs is valid only for the particular development rule it is associated with; as such, it may be argued that the antecedents of the development rule properly belong in the overall hypothesis of the proof. We place the antecedents on numbered lines –rather than in the hypothesis– as this has the stylistic advantage that it is possible to refer to the antecedents directly.

Line 3 is just a definition, and could be substituted into lines 4 and 5 without any change to the meaning and validity of the proof. This line defines the name T^i to refer to the set of all intermediate pairs of configurations related by one transition, where all of the configurations are between the source and target configurations of the hypothesis. The set T^i gains all intermediate transitions through careful use of a combination of the regular semantic arrow and the transitive closure of the semantic arrow. To give an example, consider that in the extreme case it could be that C_1 is the initial configuration and that C_f is the final configuration; this can only happen if the initial configuration is able to reduce to **nil** in one step, but it is possible.

The set of pairs defined in line 3 has an additional, useful property: the set of all possible configurations which are collected as C_1 form a pinch set, as does the set of all possible configurations collected as C_2. Combined with the knowledge that a pair in set T^i is a valid possible transition, we may now non-deterministically choose not only individual configurations, but also transitions.

The pinch set defined in line 3 is then used in line 4 to infer that all of the transitions in that set satisfy the single-step $Within$ property. Line 5 is the natural next step of introducing a universal quantifier based on the inference of line 4, and that leads to the final inference, giving the multiple-step $Within$ property.

Looking inside the **from/infer** box numbered as line 4, then, the deduction is essentially a case distinction. Line 4.1 identifies the three cases that we are concerned with: transitions based on the *A-R-Step*, *Assign-Eval*, and *Assign-E* semantic rules. Note that these three semantic rules are the only ones that apply; no other semantic rule applies to any of the transitions identified in the set of line 3. Line 4.2 is a trivial inference that is repeated through all of the behavioural lemmas: any step that is caused by interference –represented by the A-R-Step semantic rule– automatically conforms to the single-step $Within$ property due to the *Within-Rely* lemma. Line 4.3 makes a similar deduction to line 4.2: certain semantic rules, including *Assign-Eval*, trivially satisfy the definition of $Within$ by way of the *Within-Prog* lemma. These semantic rules do not alter the state between their source and target configurations, and since all guarantee conditions are reflexive, configurations related by these transitions trivially satisfy any guarantee condition; this is recorded in the *Within-Prog-Cor* lemma.

Box 4.4 addresses the core of the lemma for assignments: as the only rule for this construct that does something non-trivial, it is, naturally, where all the work is. Lines 4.4.1 through 4.4.4 establish the detailed structure of the components of the source and target configurations of this transition. This allows 4.4.5 to give the relationship between the evaluated value present in the transition's source configuration and the semantic meaning of the original expression relative to the state component in the transition's source configuration. Specifically, because of the restriction placed on the rely condition by the *Assign-I* development rule and imported into the lemma in line 1, it can be inferred that the evaluated value and the semantic meaning of the expression are equivalent. Line 4.4.6 infers that the state components of the transition's source and target configuration do satisfy the guarantee condition, on the basis of the prior two lines and the definition of the guarantee condition given in line 2. Satisfaction of the guarantee condition then allows this **from/infer** block to conclude that this step does satisfy the single-step Within property by way of *Within-Prog*.

6.2.2 While

The behavioural lemma for the While construct, surprisingly, does not directly require the use of induction to deal with the construct's iterative nature. This is because the top-level of the proof is similar to that of all of the behavioural lemmas: line 1 imports the only relevant antecedent from the *While-I* rule; line 2 binds a name, T^i to the set of pairs of configurations related by a single transition that are between the source and target configurations of the hypothesis; line 3 is a **from/infer** box that deduces the single-step Within property from any pair of configurations in T^i; line 4 introduces a universal quantification based on lines 2 & 3; and the final **infer** generalizes line 4 into a multiple-step Within property. Inside line 3, lines 3.1–3.3 do the work of setting up the case distinction and dealing with the trivial cases.

It is in line 3.4 that the main argument of this proof rests. The hypothesis of line 3.4 selects the pairs in the set T^i where the configurations are related by a Seq-Step transition, and for the sake of convenience, respectively binds the names C_1 and C_2 to the source and target configurations of that transition.

All of the steps that are made by the body of the While construct are "masked" by the Seq-Step rule; this situation arises due to the fact that the body is executed as a left-hand component of a Seq construct. This follows directly from the SOS definition in Chapter 2, and it can also be seen that Seq-Step transitions that follow a While construct are all due to execution of the body.

Use of structural induction allows us to infer that all of the transitions made executing the body will satisfy the single-step Within property. However, before making that inference, we must show that the body's execution started in a state that satisfied the pre condition. To this end, line 3.4.1 picks out the configuration that starts the particular iteration of the body that C_1 & C_2 are a part of. Line 3.4.2 is the deduction from a configuration marked by line 3.4.1 that the transition from C_1 to C_2 does satisfy the single-step

$\boxed{\textit{While-Within}}$

from $(P, R) \vdash \textit{mk-While}(b, \textit{body})$ **sat** (G, Q); $[\![P]\!](\sigma_0)$; $C_0 = (\textit{mk-While}(b, \textit{body}), \sigma_0)$;
$C_f = (\mathbf{nil}, \sigma_f)$; $C_0 \xrightarrow[R]{r}{}^{*} C_f$; $\textit{IH-S}(\textit{body})$

1 $(P \wedge b_s, R) \vdash \textit{body}$ **sat** $(G, W \wedge P)$ — h, *While-I*

2 $T^i = \{(C_1, C_2) \mid C_0 \xrightarrow[R]{r}{}^{*} C_1 \xrightarrow[R]{r} C_2 \xrightarrow[R]{r}{}^{*} C_f\}$ — definition

3 **from** $(C_1, C_2) \in T^i$

3.1 $(C_1, C_2) \in (\text{A-R-Step} \cup \text{While} \cup \text{If-Eval} \cup \text{If-T-E} \cup \text{If-F-E} \cup \text{Seq-E} \cup \text{Seq-Step})$ — h3, inspection of $\xrightarrow[R]{r}$

3.2 **from** $(C_1, C_2) \in \text{A-R-Step}$
infer $\textit{Within}_1(C_1, C_2, G)$ — h3.2, *Within-Rely*

3.3 **from** $(C_1, C_2) \in (\text{While} \cup \text{If-Eval} \cup \text{If-T-E} \cup \text{If-F-E} \cup \text{Seq-E})$
infer $\textit{Within}_1(C_1, C_2, G)$ — h3.3, *Within-Prog-Cor*

3.4 **from** $(C_1, C_2) \in \text{Seq-Step}$; $C_1 = (S_1, \sigma_1)$; $C_2 = (S_2, \sigma_2)$

3.4.1 $\exists C_a \in \textit{Config}, \sigma_a \in \Sigma \cdot$ — h3, h3.4, inspection of $\xrightarrow[R]{r}$
$C_a = (\textit{mk-Seq}(\textit{body}, \textit{mk-While}(b, \textit{body})), \sigma_a)$
$\wedge\ C_0 \xrightarrow[R]{r}{}^{*} C_a \xrightarrow[R]{r}{}^{*} C_1 \xrightarrow[R]{r} C_2$
$\wedge\ (C_a, C_1) \in (\text{A-R-Step} \cup \text{Seq-Step})^*$

3.4.2 **from** $C_a \in \textit{Config}, \sigma_a \in \Sigma$ **st**
$C_a = (\textit{mk-Seq}(\textit{body}, \textit{mk-While}(b, \textit{body})), \sigma_a)$
$\wedge\ C_0 \xrightarrow[R]{r}{}^{*} C_a \xrightarrow[R]{r}{}^{*} C_1 \xrightarrow[R]{r} C_2$
$\wedge\ (C_a, C_1) \in (\text{A-R-Step} \cup \text{Seq-Step})^*$

3.4.2.1 $[\![P \wedge b_s]\!](\sigma_a)$ — h, 1, h3.4.2, *While-interstices-pre*

3.4.2.2 $(\textit{body}, \sigma_a) \xrightarrow[R]{r}{}^{*} (S_1.\textit{left}, \sigma_1) \xrightarrow[R]{r} (S_2.\textit{left}, \sigma_2)$ — h3.4, h3.4.2, *Seq-Equiv*

3.4.2.3 $\textit{Within}_1((S_1.\textit{left}, \sigma_1), (S_2.\textit{left}, \sigma_2), G)$ — h, 1, 3.4.2.1, 3.4.2.2, IH-S(body)

infer $\textit{Within}_1(C_1, C_2, G)$ — 3.4.2.3, *Within-Equiv*

infer $\textit{Within}_1(C_1, C_2, G)$ — $\exists$-E(3.4.1,3.4.2)

infer $\textit{Within}_1(C_1, C_2, G)$ — $\vee$-E(3.1–3.4)

4 $\forall (C_1, C_2) \in T^i \cdot \textit{Within}_1(C_1, C_2, G)$ — $\forall$-I(3)

infer $\textit{Within}_m(R, C_0, C_f, G)$ — 2, 4, *Within-Multi*

Figure 6.5: Proof of the behavioural lemma for the *While* construct.

Within property, and the **infer** of line 3.4 uses existential elimination to infer the single-step *Within* property regardless of the particular configuration that started this iteration of the body.

Examining the deduction inside line 3.4.2, we see line 3.4.2.1 establishes that the pre condition and the stable portion of the test both hold at the beginning of the current iteration of the body through use of the *While-interstices-pre* lemma. As noted in Section 5.1.5, given that the partial satisfaction property holds on execution of the body, it can be seen that the pre condition must hold between iterations of the body due to the fact that the pre condition is a part of the body's post condition.

Line 3.4.2.2 also satisfies one of the required elements for structural induction, using the *Seq-Equiv* lemma to give the semantic relation relative to the body from the hypothesis of line 3.4.2. With that, we can apply structural induction to show that the single-step transition holds on the configurations that follow from the body in lines 3.4.2.2, and the **infer** of line 3.4.2 lifts that result using *Within-Equiv* to apply to the configurations C_1 and C_2.

6.2.3 Atomic/STM

The overall structure of the behavioural proof for the $Atomic/STM$ construct is, of course, similar to the rest of the behavioural lemmas. There are two things that make this construct unusual: first, like the assignment construct, this construct is responsible for modifying the state component of a configuration; and second, due to the manipulation that this construct performs on state components, the body of the construct will execute as though there is no external interference.

The hard work of this proof (Figure 6.6) is done in the **from/infer** box of line 4.4, but a few points need to be made first to give some context for this proof. Line 4.3 addresses every program step that is not the final elimination step of the construct, and this is done through the corollary of the *Within-Prog* definition. This fits –despite potentially state-modifying actions in the body during STM-Step transitions– as the construct carefully isolates any state modifications made by the body from everything outside of the construct. Thus, even though the construct's body may have altered the state, the top-level configuration does not have those modifications applied to its state; instead the changes are accumulated within the STM construct until a STM-Retry or STM-E transition.

The inference of line 4.4 takes advantage of this context in several ways; first however, let us investigate the structure of this **from/infer** box. As with the proof of the $While$ construct in the previous subsection, lines 4.4.2 and 4.4.3 set up an existential elimination, as we need to identify the start of the particular attempt to execute the body that this STM-E transition follows. Line 4.4.1 helps with the setup of the existential elimination, but only in so far as it gives obvious constant values to portions of S_1 and S_2.

The **from/infer** box of line 4.4.3 uses a pinch set containing the start of the particular execution attempt of which the transition between C_1 and C_2 is a part. Once we have identified the start of the current execution attempt we can then deduce that changes in the states at the commit step (i.e. the STM-E transition) do conform to the guarantee condition.

To show that the STM-E transition satisfies the behavioural constraint –the **infer** of line 4.4.3– we need to establish that the two states involved in the transition –σ_1, σ_2– satisfy the guarantee condition. This is inferred on line 4.4.3.14, on the basis that the transition's states satisfy the body's post condition (line 4.4.3.13), and the knowledge from the *Atomic-psat-I* rule that the body's post condition satisfies the guarantee condition (line 2). Getting to the deduction that the states of the STM-E transition satisfy the body's post condition is a bit tricky, however.

In line 4.4.3.12 we deduce that the target state component of the STM-E transition is equivalent to the source state component overwritten by a portion –defined by the variables in the body of the $Atomic$– of the accumulated state, σ_s, in the STM construct in the source configuration; this comes directly from the hypotheses of lines 4.4 and 4.4.3, and the definition of the STM-E rule. Combine that with the deduction of line 4.3.11 and line 4.4.3.13 follows by substitution.

Line 4.4.3.11 is the deduction that the body's post condition is satisfied by the pair of states given as σ_1 and σ_1 overwritten by the part of σ_s restricted to the variables in the body. This deduction is possible due to the use of structural induction, in particular from

$\boxed{\textit{Atomic-Within}}$

from	$(P, R) \vdash \textit{mk-Atomic}(\textit{body})$ **psat** (G, Q); $[\![P]\!](\sigma_0)$; $C_0 = (\textit{mk-Atomic}(\textit{body}), \sigma_0)$; $C_f = (\textbf{nil}, \sigma_f)$; $C_0 \xrightarrow[R]{r}{}^{*} C_f$; $\textit{IH-S}(\textit{body})$	
1	$(P, I) \vdash \textit{body}$ **sat** $(\textbf{true}, Q')$	h, *Atomic-psat-I*
2	$Q' \Rightarrow G$	h, *Atomic-psat-I*
3	$T^i = \{(C_1, C_2) \mid C_0 \xrightarrow[R]{r}{}^{*} C_1 \xrightarrow[R]{r} C_2 \xrightarrow[R]{r}{}^{*} C_f\}$	definition
4	**from** $(C_1, C_2) \in T^i$	
4.1	$(C_1, C_2) \in (\textsf{A-R-Step} \cup \textsf{STM-Atomic} \cup \textsf{STM-Step} \cup \textsf{STM-Retry} \cup \textsf{STM-E})$	h4, inspection of $\xrightarrow[R]{r}$
4.2	**from** $(C_1, C_2) \in \textsf{A-R-Step}$	
	infer $\textit{Within}_1(C_1, C_2, G)$	h4.2, *Within-Rely*
4.3	**from** $(C_1, C_2) \in (\textsf{STM-Atomic} \cup \textsf{STM-Step} \cup \textsf{STM-Retry})$	
	infer $\textit{Within}_1(C_1, C_2, G)$	h4.3, *Within-Prog-Cor*
4.4	**from** $(C_1, C_2) \in \textsf{STM-E}$; $C_1 = (S_1, \sigma_1)$; $C_2 = (S_2, \sigma_2)$	
4.4.1	$S_1 \in \textit{STM} \wedge S_1.\textit{body} = \textbf{nil} \wedge S_2 = \textbf{nil}$	h4.4, $\textsf{STM-E}$
4.4.2	$\exists C_a \in \textit{Config}, \sigma_a, \sigma_s \in \Sigma \cdot$ $C_0 \xrightarrow[R]{r}{}^{*} C_a \xrightarrow[R]{r}{}^{*} C_1 \xrightarrow[R]{r} C_2$ $\wedge\ C_a = (\textit{mk-STM}(\textit{body}, \sigma_a, \textit{body}, \sigma_a), \sigma_a)$ $\wedge\ S_1 = (\textit{mk-STM}(\textit{body}, \sigma_a, \textbf{nil}, \sigma_s), \sigma_1)$ $\wedge\ (C_a, C_1) \in (\textsf{A-R-Step} \cup \textsf{STM-Step})^*$	h4, h4.4, inspection of $\xrightarrow[R]{r}$
4.4.3	**from** $C_a \in \textit{Config}, \sigma_a, \sigma_s \in \Sigma$ **st** $C_0 \xrightarrow[R]{r}{}^{*} C_a \xrightarrow[R]{r}{}^{*} C_1 \xrightarrow[R]{r} C_2$ $\wedge\ C_a = (\textit{mk-STM}(\textit{body}, \sigma_a, \textit{body}, \sigma_a), \sigma_a)$ $\wedge\ S_1 = (\textit{mk-STM}(\textit{body}, \sigma_a, \textbf{nil}, \sigma_s), \sigma_1)$ $\wedge\ (C_a, C_1) \in (\textsf{A-R-Step} \cup \textsf{STM-Step})^*$	
4.4.3.1	$(C_0, C_a) \in (\textsf{A-R-Step} \cup \textsf{STM-Atomic} \cup \textsf{STM-Step} \cup \textsf{STM-Retry})$	h4.4.3, inspection of $\xrightarrow[R]{r}$
4.4.3.2	$[\![R]\!](\sigma_0, \sigma_a)$	4.4.3.1, *Rely-Trivial*
4.4.3.3	$[\![R]\!](\sigma_a, \sigma_1)$	h4.4.3, *Rely-Trivial*
4.4.3.4	$[\![P]\!](\sigma_1)$	h, 4.4.3.2, 4.4.3.3, *PR-ident*
4.4.3.5	$(\textit{body}, \sigma_a) \xrightarrow[I]{r} (\textbf{nil}, \sigma_s)$	h4.4.3, *Isolation-STM*
4.4.3.6	$\sigma_s = \sigma_a \dagger (\textit{Vars}(\textit{body}) \lhd \sigma_s)$	4.4.3.5, *Sequential-Effect*
4.4.3.7	$(\textit{body}, \sigma_a) \xrightarrow[I]{r} (\textbf{nil}, \sigma_a \dagger (\textit{Vars}(\textit{body}) \lhd \sigma_s))$	4.4.3.5, 4.4.3.6
4.4.3.8	$[\![I_{\textit{Vars}(\textit{body})}]\!](\sigma_a, \sigma_1)$	h4.4.3, $\textsf{STM-E}$
4.4.3.9	$(\textit{Vars}(\textit{body}) \lhd \sigma_a) = (\textit{Vars}(\textit{body}) \lhd \sigma_1)$	4.4.3.8
4.4.3.10	$(\textit{body}, \sigma_1) \xrightarrow[I]{r} (\textbf{nil}, \sigma_1 g \dagger (\textit{Vars}(\textit{body}) \lhd \sigma_s))$	4.4.3.7, 4.4.3.9, *Frame-Rule*
4.4.3.11	$[\![Q']\!](\sigma_1, \sigma_1 \dagger (\textit{Vars}(\textit{body}) \lhd \sigma_s))$	h, 1, 4.4.3.4, 4.4.3.10, IH-S(body)
4.4.3.12	$\sigma_2 = \sigma_1 \dagger (\textit{Vars}(\textit{body}) \lhd \sigma_s)$	h4.4, h4.4.3, $\textsf{STM-E}$
4.4.3.13	$[\![Q']\!](\sigma_1, \sigma_2)$	4.4.3.11, 4.4.3.12
4.4.3.14	$[\![G]\!](\sigma_1, \sigma_2)$	2, 4.4.3.13
	infer $\textit{Within}_1(C_1, C_2, G)$	h4.4, h4.4.3, 4.4.3.14, *Within-Prog*
	infer $\textit{Within}_1(C_1, C_2, G)$	$\exists$-E(4.4.2,4.4.3)
	infer $\textit{Within}_1(C_1, C_2, G)$	$\vee$-E(4.1–4.4)
5	$\forall (C_1, C_2) \in T^i \cdot \textit{Within}_1(C_0, C_f, G)$	$\forall$-I(4)
infer	$\textit{Within}_m(R, C_0, C_f, G)$	3, 5, *Within-Multi*

Figure 6.6: Proof of the behavioural lemma for the $\textit{Atomic/STM}$ construct.

the computation in line 4.4.3.10. To satisfy the required elements for structural induction, we use line 1 to give the specification; line 4.4.3.4 for the state satisfying the pre condition; and the computation in line 4.4.3.10. That σ_1 satisfies the pre condition is established by the use of *PR-ident* following lines 4.4.3.1–4.4.3.3; line 4.4.3.10 is a little unusual, however.

Taking line 4.4.3.10 as a sub-goal, it can be thought of as producing a computation of a specific form so that the post condition of the body can be deduced to hold on the states of the configurations in that computation. This line corresponds to the pragmatic intent of the $Atomic/STM$ construct in the language of Chapter 2: it should behave as though the entire execution of the body happened in the state immediately prior to the block committing its results.

The justification for line 4.4.3.10 is the *Frame-Rule* lemma, on the basis of lines 4.4.3.7 and 4.4.3.9. The frame rule lemma is essentially the observation that the only portion of the state that is in any way relevant to a computation is the portion restricted to those variables involved in the computation. Negatively, any variable not involved in the computation is irrelevant and cannot affect the computation.[1] Thus, line 4.4.3.7 gives us a computation of the body starting with the σ_a state, and line 4.4.3.9 concludes that the σ_a and σ_1 states have the same value for all of the variables in the body. Line 4.4.3.9 is a direct consequence of the fact that the identity relation (restricted to the variables contained in the body) holds between σ_a and σ_1, as concluded in line 4.4.3.8. That, in turn, is a direct consequence of the STM-E rule and the hypothesis of line 4.4.3.

Line 4.4.3.7 is derived by substitution from lines 4.4.3.5 and 4.4.3.6, of which the latter is actually a direct consequence of the former. In particular, line 4.4.3.5 is result of applying the *Isolation-STM* to the computation in the hypothesis of line 4.4.3, which gives the equivalent computation of the body, free from the $Atomic/STM$ that surrounds it. Line 4.4.3.6 follows by the *Sequential-Effect* lemma, which applies to interference-free computations, giving an equivalence between the source state component and the target state component. This results in line 4.4.3.7 being of a suitable form to easily apply the frame rule lemma in 4.4.3.10.

Recapitulating, then, moving from the inside out, we start with line 4.4.3.5 as the computation of the body on its own. Lines 4.4.3.6 and 4.4.3.7 transform that into a form suitable for transposition by the frame rule lemma in line 4.4.3.10. This transposition is supported by the restricted equivalence developed in lines 4.4.3.8 and 4.4.3.9. We can then deduce that the body's post condition is satisfied in line 4.4.3.11, given the deduction of the pre condition in lines 4.4.3.1–4.4.3.4. This is rewritten in line 4.4.3.13 to apply to the required states through line 4.4.3.12. Finally, line 4.4.3.14 gives the desired property that the guarantee condition is satisfied by σ_1 and σ_2, and the single-step $Within$ property is a direct consequence of that.

[1] See Section 5.1.3 for more about the *Frame-Rule* lemma.

6.3 Partial Satisfaction Post Condition Lemmas

The lemmas that show that the post condition is satisfied by programs developed using the rely/guarantee development rules all use existential elimination as their major deduction tool. This differs from the behavioural lemmas described previously and the convergence lemmas described in the next section as the basic nature of the post condition is different than the behavioural property and convergence. Satisfaction of the post condition on a specific –but arbitrary– pair of states, if true, is true regardless of the intermediate configurations that the computation passed through on the way. This is in stark contrast to the behavioural property over multiple steps, which, if true, is true *because* the single-step property holds over all intermediate steps; and contrasts with the convergence property which, like the behavioural property, asserts that a starting point always reaches a configuration of a certain form.

Thus, satisfaction of the post condition allows us to show that it holds over a pair of configurations given some arbitrary intermediate configuration, and then generalize it to show that it must hold regardless of which intermediate configuration is chosen. We use existential elimination to define a pinch set that all computations must pass through, and then reason from elements of that pinch set. As some configurations may disallow certain types of sub-computation,[2] the associated branches in the proof become vacuous (but not false!) in those situations.

Noting all of this, it must still be said that the lemmas of this section are still similar to the lemmas for corresponding constructs in a sequential system. Interference needs to be accounted for in these proofs, but the way in which structural induction has been used means that, for the most part, interference is not the primary concern. What is novel here is the use of the language structure to "push" the interference around so that we only need to consider it where it actually makes a difference.

Two particular lemmas will be looked at in detail in the following subsections: the first is *While-Post*, as it uses well-founded induction; and the second is *Atomic-Post* as it involves the titular construct of this thesis. Interesting elements of the remaining lemma proofs –found in Appendix D– are detailed here.

The *Assign-Post* lemma is very similar to *Assign-Within* as it uses the strategy of matching the states in the computation to the definition of a relation, though it is the post condition in this case. This lemma is, however, much simpler than its behavioural counterpart as it does not need to be concerned with the intermediate steps of the construct. The existential elimination usage in this lemma forms a good, basic example of how it is used in the lemmas. The pinch set created by the existential quantifier in line 3 is actually based around the Assign-E transition. All assignment statements will, eventually reach an assignment statement with an integer as their expression; it is not possible in this language for the construct to do otherwise. We do not, however, know which integer the assignment will eventually reach, so we must not assume more detail other than that it will be an integer (that much detail is justified only from the language's well-formedness predicate,

[2]For example, in some situations the evaluation of the test in an *If* construct may always be false, precluding the execution of the body.

$wf\text{-}Stmt$). Strictly speaking, line 3 does not define a set directly — if the quantifying predicate is used in a set comprehension, however, it defines the pinch set in question here. All of the reasoning that then follows in line 4 must be valid for all members of this set: in a sense, we are doing universal quantification over that set, however, as we do not require the members of that set in the conclusion, existential elimination is the appropriate tool.

Two proofs are provided in the appendix for the If construct: one applies to developments that use the *If-I* rule, the other to those using the *If-b-I* rule. Taken together, both of these proofs justify the *If-Post* lemma: as long as the program was developed using one of the If construct rules, then one of these deductions will hold. The primary difference between the two deductions is that the one for *If-b-I* requires a more constrained pre condition to hold before structural induction can be applied to the body's actions. Otherwise both proofs are straightforward deductions.

The *Seq-Post* lemma hardly warrants mentioning: it is, far and away, the simplest of all of the deductions in Appendix D.

Of the lemmas not described in detail, *Par-Post* is probably the most interesting as it is for a construct which the traditional non-concurrent proof methods simply do not address. Despite this, the deduction for this lemma is fairly direct due to the use of rely/guarantee conditions to characterize interference, structural induction to "push" the interference further into the program structure, and the *Isolation-Par-L* and *Isolation-Par-R* lemmas which allow for the manipulation of the rely/guarantee conditions in the context of the augmented semantics.

The deduction proceeds by establishing that the four conjuncts of the left-hand side of the implication in the *Par-I* rule[3] hold. The first is discharged by the hypothesis; the second using *Isolation-Par-L* and structural induction; the third by an argument symmetric to the second; and the last by structural induction to obtain a behavioural result on the overall program.

6.3.1 While

Though the structure of the proof of the *While-Post* lemma for the *While-I* rule is essentially the same as that of all of the post condition lemma proofs, it is the use of well-founded induction that sets this lemma apart from the rest. The first few lines in Figure 6.7 import antecedents from the construct's development rule, then an existential elimination is set up on a set of configurations intermediate to all computation paths between the source and target of the hypothesis, and the final inference is just the application of the existential elimination rule.

The pinch set that is defined by the quantifying predicate of line 6 is precisely the set of reachable and fully-evaluated If constructs that follow from the source configuration of the hypothesis — the computation between the source configuration in the hypothesis and all configurations in the pinch set amounts to the set of computations prior to the first iteration of the body. This is accomplished by restricting the semantic relation in the last term of the quantifying predicate to use only specific rules. This takes some care, as rules

[3]Specifically, $\overleftarrow{P} \wedge Q_l \wedge Q_r \wedge (R \vee G_l \vee G_r)^* \;\Rightarrow\; Q$

$\boxed{\textit{While-Post}}$

	from $wh = \textit{mk-While}(b_s \wedge b_u, \textit{body});\ (P, R) \vdash wh\ \textbf{sat}\ (G, W^* \wedge P \wedge \neg(b_s \wedge b_u));$ $[\![P]\!](\sigma_0);\ C_0 = (wh, \sigma_0);\ C_f = (\textbf{nil}, \sigma_f);\ C_0 \xrightarrow[R]{r}{*}\ C_f;\ sw = \textit{mk-Seq}(\textit{body}, wh);$ $\textit{IH-S}(\textit{body})$	
1	$\textit{well-founded}(W)$	h, *While-I*
2	$R\ \Rightarrow\ W^* \wedge I_{\textit{Vars}(b_s)}$	h, *While-I*
3	$\textit{SingleSharedVar}(b_u, R)$	h, *While-I*
4	$\neg\overleftarrow{(b_s \wedge b_u)} \wedge R\ \Rightarrow\ \neg(b_s \wedge b_u)$	h, *While-I*
5	$(P \wedge b_s, R) \vdash \textit{body}\ \textbf{sat}\ (G, W \wedge P)$	h, *While-I*
6	$\exists C_1 \in \textit{Config}, v \in \mathbb{B}, \sigma_1 \in \Sigma \cdot$ $C_1 = (\textit{mk-If}(v, sw), \sigma_1) \wedge C_0 \xrightarrow[R]{r}{*}\ C_1 \xrightarrow[R]{r}{*}\ C_f$ $\wedge\ (C_0, C_1) \in (\text{A-R-Step} \cup \text{While} \cup \text{If-Eval})^*$	h, inspection of $\xrightarrow[R]{r}$
7	**from** $C_1 \in \textit{Config}, v \in \mathbb{B}, \sigma_1 \in \Sigma$ **st** $C_1 = (\textit{mk-If}(v, sw), \sigma_1) \wedge C_0 \xrightarrow[R]{r}{*}\ C_1 \xrightarrow[R]{r}{*}\ C_f$ $\wedge\ (C_0, C_1) \in (\text{A-R-Step} \cup \text{While} \cup \text{If-Eval})^*$	
7.1	$[\![R]\!](\sigma_0, \sigma_1)$	h7, *Rely-Trivial*
7.2	**from** $\neg v$	
7.2.1	$(C_1, C_f) \in (\text{A-R-Step} \cup \text{If-F-E})^*$	h, h7.2, inspection of $\xrightarrow[R]{r}$
7.2.2	$[\![R]\!](\sigma_1, \sigma_f)$	7.2.1, *Rely-Trivial*
7.2.3	$[\![R]\!](\sigma_0, \sigma_f)$	7.1, 7.2.2
7.2.4	$[\![W^*]\!](\sigma_0, \sigma_f)$	2, 7.2.3
7.2.5	$[\![P]\!](\sigma_f)$	h, 7.2.3, *PR-ident*
7.2.6	$[\![\neg(b_s \wedge b_u)]\!](\sigma_f)$	2, 3, 4, h7.2, 7.2.2
	infer $[\![W^* \wedge P \wedge \neg(b_s \wedge b_u)]\!](\sigma_0, \sigma_f)$	7.2.4–7.2.6
7.3	**from** v	
7.3.1	$\exists \sigma_2 \in \Sigma \cdot$ $C_1 \xrightarrow[R]{r}{*}\ (wh, \sigma_2) \xrightarrow[R]{r}{*}\ C_f$ $\wedge\ (C_1, (wh, \sigma_2)) \in (\text{A-R-Step} \cup \text{If-T-E} \cup \text{Seq-Step} \cup \text{Seq-E})^*$	h7, h7.3, inspection of $\xrightarrow[R]{r}$
7.3.2	**from** $\sigma_2 \in \Sigma$ **st** $C_1 \xrightarrow[R]{r}{*}\ (wh, \sigma_2) \xrightarrow[R]{r}{*}\ C_f$ $\wedge\ (C_1, (wh, \sigma_2)) \in (\text{A-R-Step} \cup \text{If-T-E} \cup \text{Seq-Step} \cup \text{Seq-E})^*$	
7.3.2.1	$(\textit{body}, \sigma_1) \xrightarrow[R]{r}{*}\ (\textbf{nil}, \sigma_2)$	h7.3.2, *Isolation-While*
7.3.2.2	$[\![P]\!](\sigma_1)$	h, 7.1, *PR-ident*
7.3.2.3	$[\![b_s]\!](\sigma_1)$	2, h7, h7.3, *Single-Eval-If*
7.3.2.4	$[\![W \wedge P]\!](\sigma_1, \sigma_2)$	h, 5, 7.3.2.1–7.3.2.3, IH-S(body)
7.3.2.5	$[\![W^*]\!](\sigma_0, \sigma_1)$	2, 7.1
7.3.2.6	$[\![W \wedge P]\!](\sigma_0, \sigma_2)$	7.3.2.4, 7.3.2.5
7.3.2.7	**from** $\forall \sigma' \in \{\sigma'' \mid [\![W \wedge P]\!](\sigma_0, \sigma'') \wedge h[\sigma''/\sigma_0]\} \cdot$ $[\![W^* \wedge P \wedge \neg(b_s \wedge b_u)]\!](\sigma', \sigma_f)$	
7.3.2.7.1	$\sigma_2 \in \{\sigma'' \mid [\![W \wedge P]\!](\sigma_0, \sigma'') \wedge h[\sigma''/\sigma_0]\}$	h, h7.3.2, 7.3.2.6
7.3.2.7.2	$[\![W^* \wedge P \wedge \neg(b_s \wedge b_u)]\!](\sigma_2, \sigma_f)$	$\forall$-E(h7.3.2.7,7.3.2.7.1)
	infer $[\![W^* \wedge P \wedge \neg(b_s \wedge b_u)]\!](\sigma_0, \sigma_f)$	7.3.2.6, 7.3.2.7.2
	infer $[\![W^* \wedge P \wedge \neg(b_s \wedge b_u)]\!](\sigma_0, \sigma_f)$	1, 7.3.2.7, *W-Indn*
	infer $[\![W^* \wedge P \wedge \neg(b_s \wedge b_u)]\!](\sigma_0, \sigma_f)$	$\exists$-E(7.3.1,7.3.2)
	infer $[\![W^* \wedge P \wedge \neg(b_s \wedge b_u)]\!](\sigma_0, \sigma_f)$	$\vee$-E(h7.5.2,7.3)
	infer $[\![W^* \wedge P \wedge \neg(b_s \wedge b_u)]\!](\sigma_0, \sigma_f)$	$\exists$-E(6,7)

Figure 6.7: Proof of the post condition lemma for the *While* construct.

cannot be excluded in such a way as to exclude possible computational paths; however, choosing rules such that only a portion of the computation has occurred –up to the end of test evaluation in this case– can be very useful.

The **from/infer** box starting on line 7 uses the pinch set quantified in line 6, and the fact that all configurations in that pinch set contain fully evaluated test expressions allows for an obvious case distinction on whether or not the loop has terminated.

Line 7.2 deals with the situation where the test expression has evaluated to **false**, and the loop has terminated. To show that the post condition holds, we use a combination of the *PR-ident* lemma; the constraint from the *While-I* development rule that interference falls within the reflexive closure of the well-founded relation; and the observation that if the test expression evaluated to **false** initially, then the test expression must still evaluate to **false** in the final state. That the post condition is satisfied then follows directly.

The situation in line 7.3 where the test expression has evaluated to true, however, is rather more complicated. To show that the overall computation satisfied the post condition we require the use of well-founded induction. The *While-I* development rule requires that the W relation in the post condition be well-founded for precisely this reason.

To perform the inductive step we will need a pinch set such that the configurations contained have the original While as the textual component, but further constrained so that they are immediate successors to the first iteration of the body — allowing multiple iterations of the body makes it difficult to use structural induction in this context. So, inside the **from/infer** of line 7.3 we set up an existential elimination to use the pinch set. Once inside line 7.3.2, it is straightforward to show that the computation from C_0 to C_2 –the first iteration of the loop body– satisfies the post condition of the body. This is done in lines 7.3.2.1–7.3.2.6.

The well-founded induction in this proof happens in line 7.3.2.7 and the inference of line 7.3.2. The latter is that application of the well-founded induction rule to lines 1 (establishing the well-foundedness of W) and 7.3.2.6. The inductive step that is line 7.3.2.6 turns out to be reasonably direct, but setting up the inductive assumption in the hypothesis of line 7.3.2.6 takes some care. The assumption itself is that all states that are closer to the minimal elements of W will satisfy the While construct's post condition when combined with the final state from the overall hypothesis. The trick is to specify the set of states that we make the assumption on: that set must be essentially isomorphic to the well-founded relation. Thus, we define the set from which σ' is selected from to be all those states related through the well-founded relation W, with the additional restriction that the states must satisfy the conditions of the overall hypothesis if the original σ_0 is replaced by a state from this set. The selected set can be thought of as a restriction of W to only those states which are actually reachable through the semantic relation. On a syntactic note, the condition of the hypothesis with the σ_0 state substituted by the states of the set is represented by the notation $h[\sigma''/\sigma_0]$ in line 7.3.2.7, as writing out the properties in full takes a lot of space and adds little clarity.

Once past the hypothesis of line 7.3.2.7, actually performing the inductive step is easy. Fist we note that σ_2 is in the set described in the hypothesis of line 7.2.3.7, and that allows the application of universal elimination to instantiate the inductive assumption on σ_2. That the post condition holds is a direct result of combining lines 7.3.2.6 and 7.3.2.7.2, and is recorded in the **infer** of line 7.3.2.7. From there the fact that the post condition holds cascades out of the nested **from/infer** boxes to the final inference.

6.3.2 Atomic/STM

The proof of the *Atomic-Post* lemma is similar to that for *Atomic-Within*. This should not be surprising as satisfaction of the post condition, here, rests upon the same transition as the behavioural proof does.

$\boxed{\textit{Atomic-Post}}$

	from $(P, R) \vdash \textit{mk-Atomic}(\textit{body})$ **psat** (G, Q); $[\![P]\!](\sigma_0)$; $C_0 = (\textit{mk-Atomic}(\textit{body}), \sigma_0)$;	
	$C_f = (\mathbf{nil}, \sigma_f)$; $C_0 \xrightarrow[R]{r}\!\!* \; C_f$; $\textit{IH-S}(\textit{body})$	
1	$(P, I) \vdash \textit{body}$ **psat** $(\mathbf{true}, Q')$	h, *Atomic-psat-I*
2	$\overleftarrow{P} \wedge R \diamond Q' \diamond R \;\Rightarrow\; Q$	h, *Atomic-psat-I*
3	$\exists C_1, C_2, C_3 \in \textit{Config}, \sigma_1, \sigma_2, \sigma_3, \sigma_s \in \Sigma \cdot$	h, inspection of $\xrightarrow[R]{r}$
	$C_1 = (\textit{mk-STM}(\textit{body}, \sigma_1, \textit{body}, \sigma_1), \sigma_1)$	
	$\wedge\; C_2 = (\textit{mk-STM}(\textit{body}, \sigma_1, \mathbf{nil}, \sigma_s), \sigma_2)$	
	$\wedge\; C_3 = (\mathbf{nil}, \sigma_3)$	
	$\wedge\; C_0 \xrightarrow[R]{r}\!\!* \; C_1 \xrightarrow[R]{r}\!\!* \; C_2 \xrightarrow[R]{r} C_3 \xrightarrow[R]{r}\!\!* \; C_f$	
	$\wedge\; (C_0, C_1) \in (\textsf{A-R-Step} \cup \textsf{STM-Atomic} \cup \textsf{STM-Step} \cup \textsf{STM-Retry})^*$	
	$\wedge\; (C_1, C_2) \in (\textsf{A-R-Step} \cup \textsf{STM-Step})^*$	
	$\wedge\; (C_2, C_3) \in \textsf{STM-E} \wedge (C_3, C_f) \in \textsf{A-R-Step}^*$	
4	**from** $C_1, C_2, C_3 \in \textit{Config}, \sigma_1, \sigma_2, \sigma_3, \sigma_s \in \Sigma$ **st**	
	$C_1 = (\textit{mk-STM}(\textit{body}, \sigma_1, \textit{body}, \sigma_1), \sigma_1)$	
	$\wedge\; C_2 = (\textit{mk-STM}(\textit{body}, \sigma_1, \mathbf{nil}, \sigma_s), \sigma_2)$	
	$\wedge\; C_3 = (\mathbf{nil}, \sigma_3)$	
	$\wedge\; C_0 \xrightarrow[R]{r}\!\!* \; C_1 \xrightarrow[R]{r}\!\!* \; C_2 \xrightarrow[R]{r} C_3 \xrightarrow[R]{r}\!\!* \; C_f$	
	$\wedge\; (C_0, C_1) \in (\textsf{A-R-Step} \cup \textsf{STM-Atomic} \cup \textsf{STM-Step} \cup \textsf{STM-Retry})^*$	
	$\wedge\; (C_1, C_2) \in (\textsf{A-R-Step} \cup \textsf{STM-Step})^*$	
	$\wedge\; (C_2, C_3) \in \textsf{STM-E} \wedge (C_3, C_f) \in \textsf{A-R-Step}^*$	
4.1	$[\![R]\!](\sigma_0, \sigma_1)$	h4, *Rely-Trivial*
4.2	$[\![R]\!](\sigma_1, \sigma_2)$	h4, *Rely-Trivial*
4.3	$[\![P]\!](\sigma_2)$	h, 4.1, 4.2, *PR-ident*
4.4	$(\textit{body}, \sigma_1) \xrightarrow[I]{r}\!\!* \; (\mathbf{nil}, \sigma_s)$	h4, *Isolation-STM*
4.5	$\sigma_s = \sigma_1 \dagger (\textit{Vars}(\textit{body}) \lhd \sigma_s)$	4.4, *Sequential-Effect*
4.6	$(\textit{body}, \sigma_1) \xrightarrow[I]{r}\!\!* \; (\mathbf{nil}, \sigma_1 \dagger (\textit{Vars}(\textit{body}) \lhd \sigma_s))$	4.4, 4.5
4.7	$[\![I_{\textit{Vars}(\textit{body})}]\!](\sigma_1, \sigma_2)$	h4, STM-Step
4.8	$(\textit{Vars}(\textit{body}) \lhd \sigma_1) = (\textit{Vars}(\textit{body}) \lhd \sigma_2)$	4.7
4.9	$(\textit{body}, \sigma_2) \xrightarrow[I]{r}\!\!* \; (\mathbf{nil}, \sigma_2 \dagger (\textit{Vars}(\textit{body}) \lhd \sigma_s))$	4.6, 4.8, *Frame-Rule*
4.10	$[\![Q']\!](\sigma_2, \sigma_2 \dagger (\textit{Vars}(\textit{body}) \lhd \sigma_s))$	h, 1, 4.3, 4.9, IH-S(body)
4.11	$\sigma_3 = \sigma_2 \dagger (\textit{Vars}(\textit{body}) \lhd \sigma_s)$	h4, STM-E
4.12	$[\![Q']\!](\sigma_2, \sigma_3)$	4.10, 4.11
4.13	$[\![R]\!](\sigma_3, \sigma_f)$	h4, *Rely-Trivial*
4.14	$[\![R \diamond Q' \diamond R]\!](\sigma_0, \sigma_f)$	4.1, 4.2, 4.12, 4.13
	infer $[\![Q]\!](\sigma_0, \sigma_f)$	h, 2, 4.14
	infer $[\![Q]\!](\sigma_0, \sigma_f)$	$\exists$-E(3,4)

Figure 6.8: Proof of the post condition lemma for the $\textit{Atomic/STM}$ construct.

As usual, lines 1 and 2 start by importing antecedents from the *Atomic-psat-I*[4] development rule. Lines 3 and 4 set up an existential elimination, and then the final inference applies that elimination rule to get the post condition.

The existential assertion that is in line 3 takes some thought to unpack. Its purpose is to identify three pinch sets, based on the computation in the over hypothesis, and in the process, name elements of those sets. The first pinch set is all possible configurations denoted as C_1 in line 3: these represent the start of the final (successful) attempt to execute the body. The second, denoted C_2, are those configurations immediately prior to the STM-E semantic rule. Finally, the third, denoted C_3, are those configurations immediately (ignoring interference transitions) following those of the previous pinch set: the configurations immediately after the STM-E semantic rule. Reasoning about the relationships between these three pinch sets is sufficient to show that the post condition holds for terminating computations. It is possible to split the quantifier in line 3 into three separate existential eliminations, but there is no benefit to doing so.

The deduction inside line 4 can be broken down into a few sub-goals. Line 4.14 directly implies the **infer** of line 4, and is, itself, just a composition of the body's post condition with the overall rely on either side. Gaining the rely condition on either side of the body's action –made visible between C_2 and C_3 in the computation– is trivial, and accomplished in lines 4.1, 4.2 and 4.13.

Lines 4.3 through 4.12 can be viewed as a sort of sub-block with the goal of showing that the body's post condition holds between σ_2 and σ_3. Gaining the body's post condition follows exactly the same chain of logic here in Figure 6.8 as was used in the proof of *Atomic-Within* in Figure 6.6, lines 4.4.3.4–4.4.3.13.

6.4 Convergence Lemmas

The remaining lemmas all deal with the convergence properties of the various constructs. As with the behavioural lemmas, the proofs here use universal introduction as their primary tactic as the $Converges$ property depends upon all possible computational paths converging, and each must be examined. The general strategy in these proofs is a combination of using structural induction on the components of a construct, and the use of the semantics to identify target sets that a given construct must reach regardless of the environment and the construct's components. As an example of the latter strategy, any *If* construct will eventually transition to another *If* construct with its test expression completely evaluated to a Boolean value: the semantics of the language of Chapter 2 leave no other possibility. Note that this is, of course, trivially true if the construct's test expression is a Boolean value to begin with.

It should also be noted that these proofs are done relative to the merging augmented semantics rather than the distinguishing augmented semantics. The primary reason for this is that it avoids the need to deal explicitly with infinite sequences of interference transitions.

As has been the pattern for the behavioural and post condition sections, the remainder of this section will touch on interesting details of the convergence lemma proofs contained

[4]We use the *Atomic-psat-I* rule rather than the *Atomic-I* rule as the former is more general than the latter.

in Appendix D but not examined in detail in this chapter.

We assume that all expressions in the language terminate cleanly. Though this is not proven here, the spare definition we give to expressions –and the lack of any remotely complex structures– makes the techniques given in Sites' thesis [Sit74] applicable to the task of showing that well-formed expressions always terminate.

The assignment construct in this language always terminates, regardless of the environment it is executed in, and this can be seen by inspection of the language semantics. As such, its convergence property is stated as a lemma in Figure 6.9 and a formal proof is omitted. The outline of the proof, however, relies on the fact that since expression evaluation converges on a constant, the Assign-Eval semantic rule converges on a configuration which it cannot be applied to. The only rule in the language definition that applies to an $Assign$ construct with a constant is the Assign-E semantic rule, and that reduces the assignment to a **nil**. This is, by necessity, true of any assignment and any interference constraint; if a more constrained convergence property is required, the conclusion of *Assign-Converges* can be weakened as necessary with *Conv-Weaken* and *Conv-Concrete*.

$$\boxed{\text{Assign-Converges}}\ \frac{}{Converges_s(mk\text{-}Assign(id, e), \textbf{true}, \textbf{true}, \{\textbf{nil}\})}$$

Figure 6.9: Convergence lemma for the $Assign$ construct.

For the *If* construct we use essentially the same argument as for the assignment construct that the *If* will reach a fully evaluated test expression. Then it is a matter of case analysis on the value of the test expression: a **false** value always reaches **nil** immediately as the If-F-E semantic rule is the only option; a **true** value immediately reduces to the body on its own through the If-T-E semantic rule. From convergence of the body we use structural induction to show that it converges on **nil**, then disjunction elimination to show that the construct as a whole must converge on **nil**.

The Seq construct relies almost completely on structural induction –with some logical glue– to show that it must reach **nil**. As the semantic rules for the sequential construct consist of one rule to execute the left-hand component, and one to reduce the sequential construct to the right-hand component, the simplicity in the proof appropriately matches the simplicity in the semantics.

The parallel construct is slightly more complicated as it relies on the definition of the $Converges$ property to manipulate the rely conditions correctly on the semantic relation. The argument, however, is essentially that since both branches will reach **nil** (by structural induction), then the whole construct will reach two **nil** statements in parallel. This will, through the Par-E rule, reduce to a single **nil** on its own.

6.4.1 While

The convergence proof of the $While$ construct –just like the *While-Post* proof– depends upon well-founded induction to deal with the construct's repetitive nature. The structure of this proof, however, requires that the inductive assumption (and, therefore, the inductive step) be introduced much earlier in the proof.

Lines 1–7 in Figure 6.10 introduce obvious elements of the proof: lines 1–6 are the antecedents from the *While-I* development rule, and line 7 is the result of structural induction on the body.

The **from/infer** box of line 8 comprises the inductive step: on the basis of the assumption of convergence to **nil** on every reachable configuration of a certain form, then the original configuration will converge to **nil**. Those reachable configurations must be such that the textual component is the original While construct and the state component is a state that is closer to the minimal elements of the well-founded relation, $W \wedge P$, than the original state, σ_0. The final **infer** of the overall proof is then just the application of the well-founded induction rule to line 8. One of the oddities of the structure of this proof is that, though we set up the induction very early, we do not actually use the inductive assumption until near the very end of the **from/infer** box, at line 8.2.4.13.

The intermediate deductions inside line 8 break down into two major parts: first, in lines 8.1 and 8.2 we show that the source configuration from the hypothesis, C_0, must converge on a fully evaluated *If* construct; and second, in lines 8.3–8.5, that all configurations reachable from C_0 with that fully evaluated *If* construct must converge on **nil**. This split is necessary to ensure that the initial configuration is not a part of the reasoning that follows much later, beginning on line 8.2.4.10.

Lines 8.1 and 8.2 are a trivial conclusion based on the While and If-Eval semantic rules. It serves to set up the case distinction inside line 8.4, however. Line 8.4 concludes that any configuration in the set defined on line 8.3 will converges on **nil**; line 8.3 defines the set of all of the fully evaluated *If* constructs that line 8.2 has asserted that C_0 must converge upon. The set C^v also has the interesting property the configurations it includes may come after an arbitrary number of iterations of the body of the While loop.

Inside line 8.4 is the case analysis based on whether the test expression evaluated to **true** or **false**. The **false** case corresponds to both the vacuous case of the induction –where σ_0 is a minimal element of the well-formed relation– and to the case where interference may have caused the While to terminate early. The **true** case, however, corresponds only to those cases for which σ_0 is not a minimal element.

The **false** case is dealt with in lines 8.4.1 and is discharged easily: an *If* with **false** test expression can only result in a **nil**. Note that the reason the test expression evaluated to **false** is not relevant: whether σ_0 is a minimal element or not, all paths that reach a **false** test expression will converge on **nil**. The **true** case is dealt with in line 8.4.2 and requires more effort as the proof requires examination of all the constructs which a While construct is transformed into due to the While rule.

Lines 8.4.2.1–8.4.2.2 allow us to deduce that the *If* construct in the configuration C_v will converge on its body. The body is a sequence containing the body of the initial While in the left-hand component, and the whole initial While in the right-hand component. Lines 8.4.2.3–8.4.2.5 take that sequence and deduce that its left-hand component will converge on **nil** by use of structural induction through line 4. This requires that the pre condition is established, however; that is done through the use of the *While-interstices-pre*

$\boxed{\textit{While-Converges}}$

from $wh = \textit{mk-While}(b_s \wedge b_u, \textit{body})$; $(P, R) \vdash wh$ **sat** $(G, W^* \wedge P \wedge \neg\,(b_s \wedge b_u))$;
$[\![P]\!](\sigma_0)$; $sw = \textit{mk-Seq}(\textit{body}, wh)$; $C_0 = (wh, \sigma_0)$; $\textit{IH-T}(\textit{body})$

1 $\textit{well-founded}(W)$ — h, *While-I*

2 $\textit{bottoms}(W) \subseteq \neg\,(b_s \wedge b_u)$ — h, *While-I*

3 $R \;\Rightarrow\; W^* \wedge I_{\textit{Vars}(b_s)}$ — h, *While-I*

4 $\textit{SingleSharedVar}(b_u, R)$ — h, *While-I*

5 $\neg\,\overleftarrow{(b_s \wedge b_u)} \wedge R \;\Rightarrow\; \neg\,(b_s \wedge b_u)$ — h, *While-I*

6 $(P, R) \vdash \textit{body}$ **sat** $(G, W \wedge P)$ — h, *While-I*

7 $\textit{Converges}_s(\textit{body}, P, R, \{\textbf{nil}\})$ — 1, IH-T(body)

8 **from** $\forall C' \in \{(wh, \sigma'') \mid C_0 \xrightarrow[R]{m}{}^{*} (wh, \sigma'') \wedge [\![W \wedge P]\!](\sigma_0, \sigma'')\} \cdot$
$\textit{Converges}_c(C', R, \{\textbf{nil}\})$

8.1 **from** $S \in \textit{While}$; $\sigma \in \Sigma$; $R' \in \textit{Rely}$; $sw' = \textit{mk-Seq}(S.\textit{body}, S)$
infer $\textit{Converges}_c((S, \sigma), R', \{\textit{mk-If}(v, sw') \mid v \in \mathbb{B}\})$ — h8.1, inspection of $\xrightarrow[R']{m}$

8.2 $\textit{Converges}_c(C_0, R, \{\textit{mk-If}(v, sw) \mid v \in \mathbb{B}\})$ — h, 8.1

8.3 $C^v = \{(\textit{mk-If}(v, sw), \sigma_v) \mid C_0 \xrightarrow[R]{m}{}^{*} (\textit{mk-If}(v, sw), \sigma_v) \wedge v \in \mathbb{B}\}$ — definition

8.4 **from** $C_v \in C^v$; $C_v = (\textit{mk-If}(v, sw), \sigma_v)$

8.4.1 **from** $\neg\, v$

8.4.1.1 **from** $S \in \textit{If} \wedge S.b =$ **false**; $\sigma \in \Sigma$; $R' \in \textit{Rely}$
infer $\textit{Converges}_c((S, \sigma), R', \{\textbf{nil}\})$ — h8.4.1.1, inspection of $\xrightarrow[R']{m}$

infer $\textit{Converges}_c(C_v, R, \{\textbf{nil}\})$ — h, h8.4, h8.4.1, 8.4.1.1

8.4.2 **from** v

8.4.2.1 **from** $S \in \textit{If} \wedge S.b =$ **true**; $\sigma \in \Sigma$; $R' \in \textit{Rely}$
infer $\textit{Converges}_c((S, \sigma), R', \{S.\textit{body}\})$ — h8.4.2.1, inspection of $\xrightarrow[R']{m}$

8.4.2.2 $\textit{Converges}_c(C_v, R, \{sw\})$ — h, h8.4, h8.4.2, 8.4.2.1

8.4.2.3 $C^b = \{(sw, \sigma_b) \mid C_v \xrightarrow[R]{m}{}^{*} (sw, \sigma_b)\}$ — definition

8.4.2.4 **from** $C_b \in C^b$; $C_b = (sw, \sigma_b)$

8.4.2.4.1 $[\![P \wedge b_s]\!](\sigma_b)$ — h, 6, h8.4, h8.4.2, h8.4.2.4, *While-interstices-pre*

8.4.2.4.2 $\textit{Converges}_c((\textit{body}, \sigma_b), R, \{\textbf{nil}\})$ — 7, 8.4.2.4.1, *Conv-Concrete*

infer $\textit{Converges}_c(C_b, R, \{\textit{mk-Seq}(\textbf{nil}, wh)\})$ — 8.4.2.4.2, *Conv-Wrap-Seq*

8.4.2.5 $\forall C_b \in C^b \cdot \textit{Converges}(C_b, R, \{\textit{mk-Seq}(\textbf{nil}, wh)\})$ — $\forall$-I(8.4.2.4)

8.4.2.6 $\textit{Converges}_c(C_v, R, \{\textit{mk-Seq}(\textbf{nil}, wh)\})$ — 8.4.2.2, 8.4.2.3, 8.4.2.5, *Conv-Concat*

8.4.2.7 **from** $S \in \textit{Seq} \wedge S.\textit{left} =$ **nil**; $\sigma \in \Sigma$; $R' \in \textit{Rely}$
infer $\textit{Converges}_c((S, \sigma), R', \{S.\textit{right}\})$ — h8.4.2.7, inspection of $\xrightarrow[R]{m}$

8.4.2.8 $\forall \sigma \in \Sigma \cdot \textit{Converges}((\textit{mk-Seq}(\textbf{nil}, wh), \sigma), R, \{wh\})$ — $\forall$-I(8.4.2.7)

8.4.2.9 $\textit{Converges}_c(C_v, R, \{wh\})$ — 8.4.2.6, 8.4.2.8, *Conv-Concat*

8.4.2.10 $C^w = \{(wh, \sigma_w) \mid C_0 \xrightarrow[R]{m}{}^{*} C_v \xrightarrow[R]{m}{}^{*} (wh, \sigma_w)\}$ — definition

8.4.2.11 $\forall (S_w, \sigma_w) \in C^w \cdot [\![W \wedge P]\!](\sigma_0, \sigma_w)$ — h, 3, 6, 8.4.2.10, *While-interstices-psat*

8.4.2.12 $C^w \subseteq \{(wh, \sigma'') \mid C_0 \xrightarrow[R]{m}{}^{*} (wh, \sigma'') \wedge [\![W \wedge P]\!](\sigma_0, \sigma'')\}$ — h8, 8.4.2.10, 8.4.2.11

8.4.2.13 $\forall C_w \in C^w \cdot \textit{Converges}(C_w, R, \{\textbf{nil}\})$ — h8, 8.4.2.12

infer $\textit{Converges}_c(C_v, R, \{\textbf{nil}\})$ — 8.4.2.9, 8.4.2.10, 8.4.2.13, *Conv-Concat*

infer $\textit{Converges}_c(C_v, R, \{\textbf{nil}\})$ — $\vee$-E(h8.4,8.4.1,8.4.2)

8.5 $\forall C_v \in C^v \cdot \textit{Converges}_c(C_v, R, \{\textbf{nil}\})$ — $\forall$-I(8.4)

infer $\textit{Converges}_c(C_0, R, \{\textbf{nil}, wh\})$ — 8.2, 8.3, 8.5, *Conv-Concat*

infer $\textit{Converges}_c(C_0, R, \{\textbf{nil}\})$ — 1, 2, 8, *W-Indn*

Figure 6.10: Proof of the convergence lemma for the *While* construct.

lemma inside line 8.4.2.4. With that done, we can conclude in line 8.4.2.6 that C_v will reach a sequence with a **nil** left-hand component and the original While in the right-hand component. From there it is trivial to conclude that C_v will converge on configurations containing the original While, and this is done in lines 8.4.2.7–8.4.2.9.

It is at this point in the proof that we are almost ready to use the inductive assumption placed in the hypothesis of line 5. Before actually doing so, however, we must show that it actually applies. To that end, line 8.4.2.10 defines and binds the set of configurations that line 8.4.2.9 asserts that C_v must converge upon, doing so relative to C_0. Line 8.4.2.11, then, uses the *While-interstices-psat* lemma to deduce that the body's post condition –and, thus, the well-founded relation– hold between σ_0 and all of the state components in the set of configurations in line 8.4.2.10. That allows line 8.4.2.12 to deduce that the set defined in line 8.4.2.10 is a subset of the set in the inductive assumption and, therefore, all of those configurations converge upon **nil**, as recorded in line 8.4.2.13. That last deduction allows us to conclude that C_v with a **true** test expression must converge upon **nil**.

The **infer** of line 8.4 uses disjunction elimination to conclude that any given C_v will converge upon **nil**, and line 8.5 uses universal introduction to place that into a universal quantifier. That is concatenated with line 8.2 to give us the inductive step, that is, C_0 will converge upon **nil** if all reachable following configurations with the original While converge upon **nil**. The *W-Indn* rule then allows us to conclude, simply, that C_0 must converge upon **nil** for the final **infer** of the proof.

6.4.2 Atomic/STM

Considering the $\mathit{Atomic/STM}$ construct, it is easy to see that there are only two things that would cause it never to reach **nil**: either the failure of the body to converge on **nil**, or interference triggering the STM-Retry semantic rule every time before the STM-E semantic rule can be applied. The approach taken by the *Atomic-I* development rule is to constrain the interference so that it cannot trigger the STM-Retry semantic rule at all. As noted in Section 3.2 where the *Atomic-I* is given, this constraint on interference strongly affects the potential uses of the construct in a rely/guarantee development; however, use of the *sat-I* inference rule with an alternate termination argument can mitigate this problem.

Inside the proof itself, we end up with the situation where it is necessary to show that all transitions that follow a given configuration cannot be due to the STM-Retry semantic rule. This is done through the use of the unusual mechanism of proof by contradiction. In the **from/infer** box of line 7.3 we show that the presence of a STM-Retry transition implies that the rely condition cannot have been true of the environment's behaviour. Since the rely condition –and that the environment's behaviour conforms to it– is a framing assumption of the entire rely/guarantee framework, this allows the introduction of an absurdity, and by the restricted principle of contradiction elimination, allows us to deduce that, in fact, no STM-Retry transition can be present.

Looking at the structure of the proof of *Atomic-Converges* in Figure 6.11, then, we see the usual initial step of importing the relevant antecedents in lines 1 and 2. Lines 3–5 show that the Atomic construct must become a STM construct, and line 6 defines the

Atomic-Converges

	from $(P, R) \vdash \textit{mk-Atomic}(\textit{body})$ **sat** (G, Q); $[\![P]\!](\sigma_0)$; $C_0 = (\textit{mk-Atomic}(\textit{body}), \sigma_0)$; $\textit{IH-T}(\textit{body})$	
1	$(P, I) \vdash \textit{body}$ **sat** $(\textbf{true}, Q')$	h, *Atomic-I*
2	$R \Rightarrow I_{\textit{Vars}(\textit{body})}$	h, *Atomic-I*
3	$\textit{Converges}_s(\textit{body}, P, I, \{\textbf{nil}\})$	1, IH-T(body)
4	**from** $S \in \textit{Atomic}; \sigma \in \Sigma, R' \in \textit{Rely}$	
	infer $\textit{Converges}_c((S, \sigma), R, \{\textit{mk-STM}(S.\textit{body}, \sigma', S.\textit{body}, \sigma') \mid [\![R]\!](\sigma, \sigma')\})$	h4, inspection of $\xrightarrow[R]{m}$, STM-Atomic
5	$\textit{Converges}_c(C_0, R, \{\textit{mk-STM}(\textit{body}, \sigma_b, \textit{body}, \sigma_b) \mid [\![R]\!](\sigma_0, \sigma_b)\})$	h, 4
6	$C^i = \{C_i \mid C_0 \xrightarrow[R]{m}{*}\ C_i \wedge C_i = (\textit{mk-STM}(\textit{body}, \sigma_b, \textit{body}, \sigma_b), \sigma_i) \wedge [\![R]\!](\sigma_0, \sigma_b)\}$	definition
7	**from** $C_i \in C^i$; $C_i = (\textit{mk-STM}(\textit{body}, \sigma_b, \textit{body}, \sigma_b), \sigma_i)$	
7.1	**from** $(C_j, C_k) \in \{(C_j, C_k) \mid C_i \xrightarrow[R]{m}{*}\ C_j \xrightarrow[R]{m} C_k\}$	
7.1.1	$(C_j, C_k) \in$ STM-Retry $\vee$ $(C_j, C_k) \notin$ STM-Retry	h7.1, inspection of $\xrightarrow[R]{m}$
7.1.2	**from** $(C_j, C_k) \in$ STM-Retry; $C_j = (S_j, \sigma_j)$; $\Sigma^j = \{\sigma \mid [\![R]\!](\sigma_j, \sigma)\}$	
7.1.2.1	$[\![R]\!](\sigma_b, \sigma_i)$	h7, inspection of $\xrightarrow[R]{m}$, STM-Atomic
7.1.2.2	$(C_j, C_k) \in (\text{STM-Atomic} \cup \text{STM-Step} \cup \text{STM-E} \cup \text{STM-Retry})^*$	h7.1.2, inspection of $\xrightarrow[R]{m}$
7.1.2.3	$[\![R]\!](\sigma_i, \sigma_j)$	7.1.2.2, *Rely-Trivial*
7.1.2.4	$\forall \sigma' \in \Sigma^j \cdot [\![I_{\textit{Vars}(\textit{body})}]\!](\sigma_j, \sigma)$	2, h7.1.2
7.1.2.5	$\forall \sigma' \in \Sigma^j \cdot [\![I_{\textit{Vars}(\textit{body})}]\!](\sigma_b, \sigma)$	7.1.2.1, 7.1.2.3, 7.1.2.4
7.1.2.6	$\forall \sigma' \in \Sigma^j \cdot (\textit{Vars}(\textit{body}) \lhd \sigma_b) = (\textit{Vars}(\textit{body}) \lhd \sigma')$	7.1.2.5
7.1.2.7	$\exists \sigma' \in \Sigma^j \cdot (\textit{Vars}(\textit{body}) \lhd \sigma_b) \neq (\textit{Vars}(\textit{body}) \lhd \sigma')$	h7.1.2, STM-Retry
	infer $\curlywedge$	7.1.2.6, 7.1.2.7, $\curlywedge$-I
	infer $(C_j, C_k) \notin$ STM-Retry	7.1.1, 7.1.2, $\curlywedge$-E
7.2	$\forall t \in \{(C_j, C_k) \mid C_i \xrightarrow[R]{m}{*}\ C_j \xrightarrow[R]{m} C_k\} \cdot t \notin$ STM-Retry	$\forall$-I(7.1)
7.3	$[\![P]\!](\sigma_b)$	h, h7, *PR-ident*
7.4	$\textit{Converges}_c((\textit{body}, \sigma_b), I, \{\textbf{nil}\})$	3, 7.3, *Conv-Concrete*
7.5	$\textit{Converges}_c(C_i, R, \{S \in \textit{STM} \mid S.\textit{body} = \textbf{nil}\})$	h7, 7.2, 7.4, *Conv-Wrap-STM*
7.6	$C^e = \{(S_e, \sigma_e) \mid C_i \xrightarrow[R]{m}{*}\ (S_e, \sigma_e) \wedge S_e \in \textit{STM} \wedge S_e.\textit{body} = \textbf{nil}\}$	definition
7.7	**from** $C_e \in C^e$	
7.7.1	**from** $S \in \textit{STM} \wedge S.\textit{body} = \textbf{nil}; \sigma \in \Sigma; R' \in \textit{Rely}; R' \Rightarrow I_{\textit{Vars}(S.\textit{body})}$	
	infer $\textit{Converges}_c((S, \sigma), R', \{\textbf{nil}\})$	
	infer $\textit{Converges}_c(C_e, R, \{\textbf{nil}\})$	2, h7.7, 7.7.1
7.8	$\forall C_e \in C^e \cdot \textit{Converges}_c(C_e, R, \{\textbf{nil}\})$	$\forall$-I(7.7)
	infer $\textit{Converges}_c(C_i, R, \{\textbf{nil}\})$	7.5, 7.6, 7.8, *Conv-Concat*
8	$\forall C_i \in C^i \cdot \textit{Converges}_c(C_i, R, \{\textbf{nil}\})$	$\forall$-I(7)
	infer $\textit{Converges}_c(C_0, R, \{\textbf{nil}\})$	5, 6, 8, *Conv-Concat*

Figure 6.11: Proof of the convergence lemma for the $\textit{Atomic}/\textit{STM}$ construct.

set of possible configurations that follow from line 5. Lines 7 and 8 introduce a universal quantification to the effect that all elements of the set defined in line 6 must converge to **nil**, and the proof ends by concatenating the $Converges$ properties in lines 5 and 8 to conclude that the initial configuration must converge to **nil**.

Line 7 is the **from/infer** box that deduces that any arbitrary element, C_i of the set defined in line 6 must converge to **nil**. The content of the **from/infer** box can be sectioned into three parts: lines 7.1 and 7.2 deduce that the STM-Retry transition cannot follow C_i; lines 7.3–7.5 build that into the more general assertion that the configuration C_i must reach a STM construct with a **nil** body; and last, lines 7.6–7.8 and the **infer** show that C_i must, indeed, reach **nil**.

Line 7.1 is the deduction that any arbitrary transition that follows C_i cannot be due to the STM-Retry semantic rule, and line 7.2 generalizes it by introducing a universal quantifier to that effect. The hypothesis of line 7.1 binds C_j and C_k to be the configurations related by a single transition that follows from C_i. Then, on the basis that C_j and C_k must be related by some rule of the semantics, line 7.1.1 notes that the pair of configurations either is or is not related by the STM-Retry semantic rule. This disjunction is always defined as every transition must be via some semantic rule, and thus means that we are able to use proof by contradiction to derive one of the terms of the disjunction. Line 7.1.2 makes the assumption that the pair of configurations is related through STM-Retry, and from that derives an absurdity. The final **infer** of line 7.1 can then use the absurdity elimination rule –the embodiment of the principle of restricted contradiction elimination– to conclude that the transition that relates the pair of configurations is not STM-Retry.

Inside line 7.1.2 we need to derive two assertions that contradict each other. The second of these is line 7.1.2.7, which asserts that there is some external state for which the variables of the body differ in value from those in the body's starting state. The external states that the existential quantifier selects from are those that are one interference step past σ_j — as defined in the hypothesis of line 7.1.2. The definition of the STM-Retry semantic rule allows line 7.1.2.7 as a direct consequence. Set against line 7.1.2.7 is line 7.1.2.6, asserting that the values of the variables of the body have the same values in the external state and the starting state. Line 7.1.2.6 is deduced by showing that the rely condition holds between the body's starting state and all external states. As the rely condition implies an identity relation restricted to the variables of the body, this proceeds directly.

Line 7.3 deduces that the pre condition holds on the body's starting state and is necessary for line 7.4 to deduce that the body will converge to **nil** given that particular state. The combination of lines 7.2, 7.4 and the hypothesis of line 7 allows the application of the *Conv-Wrap-STM* inference rule, giving us the result in line 7.5 that C_i will converge to a STM construct with a **nil** in its body. This step relies on the fact that the *Conv-Wrap-STM* rule leaves the external state free in its conclusion.

At this point all that remains is to show that any **nil**-bodied STM construct that C_i can reach will, by necessity, reach **nil**. This is done through line 7.7 (with lines 7.6 and 7.8 providing the usual structure for the introduction of a universal quantifier) as the contained **from/infer** constrains its hypothetical configuration enough that it is only in the domain of the STM-E rule of the semantic relation. The conclusion of line 7.7 is then gained by matching up known deductions with the hypothesis of line 7.7.1, giving us the result that any **nil**-bodied STM construct will reach **nil** in this context. The remainder of the proof is just alternating applications of universal quantifier introduction and the *Conv-Concat* rule.

7 — A Development Example

This chapter is concerned with the application of the rely/guarantee development rules given in Chapter 3. As such, we are not so much concerned with the reasons and motivation behind particular design decisions made in a development, but rather the effect of design decisions on the application of the development rules.

The example in this chapter –called FINDP— of a rely/guarantee development is an elaboration of the example used in [CJ07]. This example originates in Owicki's thesis [Owi75]; the first treatment of this example using rely/guarantee reasoning was in Jones' thesis [Jon81].

The task of FINDP is to find the least index into a vector such that the value of the vector at that index satisfies some predicate. The predicate here is left arbitrary — it could be a simple test to see if the value is a positive integer, or it could be arbitrarily complex. For the sake of motivating the use of parallelism in the development, we are taking the predicate to be both computationally expensive to evaluate and suitable for parallel execution.

The semantics of the language described in Chapter 2 does not include either function calls nor vectors, both of which are used in the development. This is not a serious issue in this case, however, as both can be reified down to constructs that are available in the language. For the sake of clarity and length, however, this reification is not performed here.

We start the development, then, with the specification of FINDP, as given in Figure 7.1. In the specification, v represents the vector we are interested in, r will hold the result of computation –the least index– and $pred$ is the predicate we are looking to find a satisfactory value for. The specification of FINDP is our top-level specification: it is used to start the development, and the eventual implementation must satisfy it.

$$
\begin{array}{rl}
\multicolumn{2}{l}{\textsc{FindP}} \\
\textbf{rd} & v : X^* \\
\textbf{wr} & r : \mathbb{N}_1 \\
\textbf{pre} & \forall i \in \{1..\textbf{len}\, v\} \cdot \delta(pred(v(i))) \\
\textbf{rely} & v = \overleftarrow{v} \wedge r = \overleftarrow{r} \\
\textbf{guar} & \textbf{true} \\
\textbf{post} & (r = 1 + \textbf{len}\, v \vee 1 \leq r \leq \textbf{len}\, v \wedge pred(v(r))) \\
 & \wedge \forall i \in \{1..r-1\} \cdot \neg\, pred(v(i))
\end{array}
$$

Figure 7.1: Specification of FINDP.

Breaking the specification of FINDP down, we first note that we are allowing read access to the vector and write access to the result variable. By a strict interpretation of this specification, any conformant implementation would be restricted from using any variable not named by either a **rd** or **wr** keyword. However, we will be adding new variables as we decompose and reify this specification and, as the language has no notion of local variables, we will have to alter the definition of FINDP to include these variables as we proceed. These changes are perfectly consistent with the rely/guarantee –and VDM– method of development. The **rd** and **wr** keywords help to implicitly define the guarantee and post conditions for the specification; in particular, a variable mentioned only in the **rd** keyword will have an implicit guarantee that the variable will not be altered by a conformant implementation.

The next two keywords in the specification –**pre** and **rely**– give the context in which a conformant implementation is expected to execute within. For FINDP, the pre condition asserts only that the predicate is defined for all values in the vector. The rely condition asserts that the environment will not modify either the vector or the result variable. Thus we are assuming that the environment will do nothing that can affect the execution of FINDP.

The final two keywords –**guar** and **post**– give the expected behaviour and end result of the program. The guarantee, here, is listed as being simply **true**, which would imply that FINDP is free to modify any variable in any fashion. This must be read taking the **rd** and **wr** keywords into consideration, however, and as such the guarantee of FINDP becomes $\overleftarrow{v} = v$. The overall guarantee condition is reached by starting with the identity relation, relaxing all restrictions on variables named in **wr** keywords, then adding the constraints given in the **guar** keyword.

The post condition of FINDP is a straightforward formalisation of the purpose stated earlier, with a little bit of elaboration to describe the potential case where no value in the vector satisfied the predicate. The first term in the conjunct restricts the resulting index to be either a value one more than the length of the vector or any index for which its value in the vector satisfies the predicate. That, alone, is not quite enough to satisfy the informal specification, so the second term of the conjunct restricts the result further so that the values at all lesser indices in the vector do not satisfy the predicate.

Having now described the conditions in the FINDP specification, we need to check that all of the conditions meet the constraints required by the rely/guarantee framework. This means ensuring that the rely and guarantee conditions are reflexive and transitive; that, if a state satisfies the pre condition, then states related to that state through the rely condition also satisfy the pre condition;[1] and if a pair of states satisfy the post condition, then the rely condition cannot result in a state that no longer satisfies the post condition.[2] We will not verify these constraint explicitly in this chapter, but rather leave this validation implicit for the specifications that will follow as we develop the example.

First we ensure that the rely condition of FINDP cannot produce a state which does not satisfy pre-FINDP. It is not hard to see that this is the case: pre-FINDP make an assertion solely about the vector, and $rely$-FINDP asserts that the environment never alters the vector. Second, we check that $rely$-FINDP cannot produce a state which would no longer satisfy $post$-FINDP (relative to an arbitrary, fixed, pre condition-satisfying initial state). As $post$-FINDP only references the vector and result variables, and $rely$-FINDP is effectively an identity for those variables, it is easy to see, once again, that is true. Finally, we ensure that both $rely$-FINDP and $guar$-FINDP are reflective and transitive; here this is self-evident.

[1] See the *PR-ident* lemma.
[2] See the *QR-ident* lemma.

With the top-level specification defined, we are now ready to perform the first decomposition. This development step is a little delicate as it introduces a new variable; a language with local variables would not have difficulty with this. The approach taken here is to amend the top-level specification to include the new variables, and we make the assumption that the new variables are fresh –unused and not otherwise present– at the top-level. This amendment to the top-level specification is as simple as adding $t \colon \mathbb{N}_1$ to the **wr** keyword, and adding $\overleftarrow{t} = t$ to the rely condition of FINDP.[3]

The decomposition itself –given the existence of t– has the purpose of separating the initialization and finalization steps of FINDP from the actual task of searching for the least index. Thus we have

$$\text{FINDP} \triangleq (\text{INIT} \ ; \ \text{SEARCHES}) \ ; \ \text{FINAL}$$

where INIT, FINAL, and SEARCHES are defined in Figures 7.2, 7.3 and 7.4, respectively.

INIT
rd $v \colon X^*$
wr $t \colon \mathbb{N}_1$
pre **true**
rely $v = \overleftarrow{v} \land t = \overleftarrow{t}$
guar **true**
post $t = 1 + \textbf{len}\, \overleftarrow{v}$

Figure 7.2: Specification of INIT.

The specification for INIT is pleasantly simple: given read access to the vector and write access to the temporary variable t, a conforming implementation will finish with t set to a value one greater than the length of the vector. The specification requires that interference leaves the vector and temporary variable (t) alone, but places no restriction on the initial state. The *guar*-INIT condition is left open, though there is the implicit constraint that a conformant implementation will not modify the vector as only read access is assumed.

FINAL
rd $t \colon \mathbb{N}_1$
wr $r \colon \mathbb{N}_1$
pre **true**
rely $t = \overleftarrow{t} \land r = \overleftarrow{r}$
guar **true**
post $r = \overleftarrow{t}$

Figure 7.3: Specification of FINAL.

The specification of FINAL is similar to that of INIT, but has the effect of making the temporary variable and the result variable equivalent at its conclusion. Note that *post*-FINAL equates the final value of the result variable with the initial value of the temporary variable: this ensures that an implementation that sets t to the value of r is non-conforming.

[3]Thus making *rely*-FINDP equivalent to $\overleftarrow{v} = v \land \overleftarrow{r} = r \land \overleftarrow{t} = t$.

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 $\left[t = \overleftarrow{t} \vee (t < \overleftarrow{t} \wedge pred(v(t)))\right]$
$\wedge\, \forall i \in \{1..t-1\} \cdot \neg\, pred(v(i))$

Figure 7.4: Specification of SEARCHES.

The SEARCHES specification is now where the bulk of the work necessary to satisfy the FINDP specification is done. This specification is intended to be more general than that of FINDP: indeed, one could take the view that FINDP is a specialization of SEARCHES that covers the entire vector. To see this, first consider *pre*-SEARCHES: this pre condition only requires that the predicate be defined on values in the vector up to (but not including) an index given by t. We know from FINDP and INIT that t will be just past the end of the array, but SEARCHES does not require that in and of itself. Given that the rely and guarantee conditions of FINDP and SEARCHES are essentially the same (modulo t and r), let us consider what the post condition tells us about the result of executing an implementation of SEARCHES. Examining *post*-SEARCHES in comparison to *post*-FINDP, we find that t is substituted consistently for r and that the default case for t is different than that for r in *post*-FINDP. Specifically, we assert in *post*-SEARCHES that t will be unchanged in the default case — considering this in the context of the decomposition, this makes perfect sense: t is initially set to one greater than the length of the vector.

These three specifications –INIT, SEARCHES, and FINAL– are shown to satisfy FINDP by the use of the *Seq-I* development rule and judicious application of the *Weaken* rule. The *Seq* language construct is a binary structure, and we have three sub-specifications, so we are required to first compose two of the specifications into an intermediate specification, then compose that with the remaining specification to show that this decomposition satisfies the specification of FINDP. Alternately, we could posit a development rule for the sequential composition of three, or arbitrarily many, specifications and prove it sound relative to the language semantics. If compositions of that form were done frequently enough in the development then creating such a rule would be to our advantage; in this case we are not doing enough compositions to justify it. The argument presented here first composes INIT and SEARCHES, then that with FINAL.

Before we get to using the rules, however, we need to check that the **rd** & **wr** keywords are compatible. The vector is only ever read in the sub-specifications, which matches FINDP perfectly, and the remaining variables named in the sub-specifications are present in FINDP's **wr** keyword.[4]

The pre condition of INIT can easily be weakened to match that of FINDP, and the rely condition of INIT is less restrictive than that of FINDP, so this portion is fine. Once the *pre*-INIT condition is weakened to become *pre*-FINDP, the pre condition of SEARCHES

[4]Note that we will have added t to FINDP's specification, though the addition is not shown in Figure 7.1.

is implied by the conjunction of pre-FINDP, and the rely, guarantee, and post conditions of INIT. Now, $post$-SEARCHES is very nearly the same form as $post$-FINDP but on t instead of r and using $\overleftarrow{t}$ instead of $1 + \textbf{len}\, v$. In the composition of INIT and SEARCHES, the $\overleftarrow{t}$ of $post$-SEARCHES becomes $1 + \textbf{len}\, v$ for the post condition of the composition, and composing that with FINAL allows the substitution of r for t, resulting in a final post condition that is the same as $post$-FINDP.

We will now concentrate on the further development of SEARCHES; INIT & FINAL can be directly implemented as assignments as shown in Figure 7.12. The decomposition of SEARCHES into a pair of SEARCH specifications is motivated by the desire to demonstrate how a parallel decomposition works rather than any specific opinion regarding the best implementation of the task.

The proposed decomposition is

$$\text{SEARCHES} \triangleq \text{SEARCH}(odds) \| \text{SEARCH}(evens)$$

where $odds$ is defined as

$$odds \triangleq \{i \in \mathbb{N}_1 \mid i < t \wedge is\text{-}odd(i)\}$$

and $evens$ is defined in a similar manner; the specification for search is shown in Figure 7.5. The t in the definition of $odds$ is the same t as in the **wr** keywords of both SEARCHES and SEARCH. This parallel split is, of course, to allocate a split on the odd and even indices of the vector. The pre condition of SEARCH is nearly equivalent to pre-SEARCHES except that it is restricted to only the indices in the is set that is given as a parameter. It is not difficult to see that pre-SEARCH can be weakened to become pre-SEARCHES. The rely condition of SEARCH allows for more interference than $rely$-SEARCHES; allowing t to decrease monotonically means that other threads running in parallel may decrease t and the current instantiation of SEARCH will act accordingly. The guarantee condition for SEARCH effectively states that each execution step will either leave t unaltered, or it will change t to a smaller index whose value in the vector satisfies the predicate. Last, the post condition for SEARCH is much simpler than that for SEARCHES: it only requires that all values in the vector at indices less than t (and in $odds$) do not satisfy the predicate.

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

Figure 7.5: Specification of SEARCH.

The *Par-I* development rule is the key to using the weaker $post$-SEARCH together with $guar$-SEARCH to show that $post$-SEARCHES must hold. The last antecedent of *Par-I* – $\overleftarrow{P} \wedge Q_l \wedge Q_r \wedge (R \vee G_l \vee G_r)^* \Rightarrow Q$– is the specific requirement: Q_l & Q_r correspond to $post$-SEARCH for the odd and even cases; G_l & G_r correspond to $guar$-SEARCH on the odd and even cases; and Q corresponds to $post$-SEARCHES. The $\overleftarrow{P}$ term corresponds to pre-SEARCHES — as mentioned, $post$-SEARCHES is a weakening of $post$-SEARCH. We must also check that $rely$-SEARCH$(odds)$ can be weakened to $rely$-SEARCH$(odds) \vee$ $guar$-SEARCH$(evens)$ so as to make sure the *Par-I* rule can be applied. It is not hard to see that it can: $guar$-SEARCH only references t and is effectively $t \leq \overleftarrow{t}$ (the restriction if $t < \overleftarrow{t}$ is not relevant in this consideration), so it is accurate to say that the actions in $guar$-SEARCH$(evens)$ are included in $rely$-SEARCH$(odds)$. The same argument applies in the symmetrical case for SEARCH$(evens)$, of course.

The only remaining antecedent in *Par-I* that must be checked is that $G_l \vee G_r \;\Rightarrow\; G$. The G_l & G_r terms represent the guarantees from the odd and even cases of SEARCH; G is the guarantee from SEARCHES. That this antecedent is satisfied is trivially true: anything implies true, which is the definition of $guar$-SEARCHES as written; the objection that we must take the **rd** & **wr** keywords into account is satisfied by noting that they are precisely the same between SEARCHES and SEARCH.

7.1 Atomicity Via Data Reification

This section details the development of the FINDP example from the SEARCH specification without the use of software transactional memory. The next step in the development –from SEARCH$(odds)$ to SEARCH-ODD, shown in Figure 7.6– is primarily reliant on the *Weaken* development rule, but also uses data reification to “split” the variable t into separate variables, ot and et, for the odd and even SEARCH specifications. An alternative approach is taken in Section 7.2 that uses the $Atomic$ construct instead of data reification.

SEARCH-ODD

rd	$v: X^*, et: \mathbb{N}_1$
wr	$oc, ot: \mathbb{N}_1$
pre	$\forall i \in \{j \in \mathbb{N}_1 \mid j < min(ot, et) \wedge is\text{-}odd(j)\} \cdot \delta(pred(v(i)))$
rely	$v = \overleftarrow{v} \wedge oc = \overleftarrow{oc} \wedge ot = \overleftarrow{ot} \wedge et \leq \overleftarrow{et}$
guar	$ot = \overleftarrow{ot} \vee (ot < \overleftarrow{ot} \wedge pred(v(ot)))$
post	$\forall i \in \{j \in \mathbb{N}_1 \mid j < min(ot, et) \wedge is\text{-}odd(j)\} \cdot \neg\, pred(v(i))$

Figure 7.6: Specification of SEARCH-ODD.

The data reification is done to control access to the temporary variable in the parallel copies of SEARCH. We split t into two new variables –ot and et, for the odd and even branches– and define the retrieve function such that $t = min(ot, et)$. As with the original introduction of t, we need to add ot & et to the more abstract specification as local variables.

The purpose and intent of the reification is to create what is essentially a local variable for each instantiation of SEARCH; because of this, we perform the reification as we specialize SEARCH$(odds)$ into SEARCH-ODD.[5] The specialization of SEARCH$(odds)$ into

[5]And SEARCH$(evens)$ into SEARCH-EVEN, though we will not show that development.

SEARCH-ODD also introduces another pseudo-local variable, oc, that is added to the specifications in the same manner as t was.

The variables named in SEARCH-ODD, then, should be no surprise: oc and ot –the pseudo-local variables– are named for write access; and v and et are named for read access. Strictly speaking, et is only named as an optimization to allow SEARCH-ODD to finish early in the case where SEARCH-EVEN finds a satisfactory index that is lower than the current odd index. The current odd index is held in oc and tracks how far along the vector SEARCH-ODD's implementation has reached. The ot variable acts both as an upper bound to indicate when to stop looking as well as a result variable: if ot is changed it is because a satisfactory index has been found by SEARCH-ODD. As is obvious from the preceding description, a number of assumptions regarding the implementation have now entered into the design of the specification. In particular it is assumed that an implementation of SEARCH-ODD will start at the lowest possible index and increase the index until it finds one that satisfies the predicate.

The pre condition for SEARCH-ODD is a specialized version of pre-SEARCH: the is parameter has been replaced with the set comprehension for the odd indices, and t has been replaced with $min(ot, et)$. The rely condition has been changed a bit to reflect the reification: ot must now be interference-free, and et may monotonically decrease.

The guarantee of SEARCH-ODD is that of $guar$-SEARCH but specialized to ot; et is implicitly unaltered as SEARCH-ODD does not have write access to it. Last, the post condition of SEARCH-ODD asserts that all odd indices less than the minimum of ot and et will not satisfy the predicate. Though the set comprehension in $post$-SEARCH-ODD is textually the same as that in pre-SEARCH-ODD, it should be noted that they may not necessarily generate the same set of indices: the set in the post condition will be a (non-strict) subset of that in the pre condition.

Comparing the rely and guarantee condition in the specialized SEARCH-ODD and SEARCH-EVEN, it is not hard to see that the only place that they can actually affect each other is in the post condition. And, even there, the practical effect is essentially nil.

Justifying the assertion that an implementation that conforms to SEARCH-ODD will also conform to SEARCH$(odds)$ is done via the *Weaken* development rule with the reification taken into account.

The SEARCH-ODD specification does not assume that oc has any particular initial value, however, it will certainly need to be initialized before any further work is done in the search. Thus we posit the following for the decomposition of SEARCH-ODD:

$$\text{SEARCH-ODD} \triangleq \text{SEARCH-ODD-INIT} \ \mathbf{;} \ \text{SEARCH-ODD-SCAN}$$

where SEARCH-ODD-INIT is given in Figure 7.7 and SEARCH-ODD-SCAN is given in Figure 7.8.

The specification of SEARCH-ODD-INIT is uninteresting save for the fact that its post condition equates oc to a constant value. At the abstract level we know that there are no odd values in the set of indices that are less than 1, and since finding a least index involves knowing something about all lesser indices, this is the obvious place to start. This constant

SEARCH-ODD-INIT
wr $oc\colon \mathbb{N}_1$
pre **true**
rely $oc = \overleftarrow{oc}$
guar **true**
post $oc = 1$

Figure 7.7: Specification of SEARCH-ODD-INIT.

value is later used to justify the satisfaction of *post*-SEARCH-ODD from the composition of *post*-SEARCH-ODD-INIT and *post*-SEARCH-ODD-SCAN.

SEARCH-ODD-SCAN
rd $v\colon X^*, et\colon \mathbb{N}_1$
wr $oc, ot\colon \mathbb{N}_1$
pre $\forall i \in \{j \in \mathbb{N}_1 \mid oc \leq j < min(ot, et) \wedge \textit{is-odd}(j)\} \cdot \delta(pred(v(i)))$
rely $v = \overleftarrow{v} \wedge oc = \overleftarrow{oc} \wedge ot = \overleftarrow{ot} \wedge et \leq \overleftarrow{et}$
guar $ot = \overleftarrow{ot} \vee (ot < \overleftarrow{ot} \wedge pred(v(ot)))$
post $(\overleftarrow{oc} \leq oc) \wedge \neg\,(oc < min(ot, et))$
$\wedge\, \forall i \in \{j \in \mathbb{N}_1 \mid \overleftarrow{oc} \leq j < min(ot, et) \wedge \textit{is-odd}(j)\} \cdot \neg\, pred(v(i))$

Figure 7.8: Specification of SEARCH-ODD-SCAN.

The SEARCH-ODD-SCAN specification is where the actual work of scanning through the indices is done. There is an implicit assumption that the start and end points of the scan –oc and ot respectively– are already set; or, seen from another prospective, this specification only scans the vector between the bounds set by oc at the (inclusive) bottom and ot at the (exclusive) top.

The pre condition of SEARCH-ODD-SCAN is the natural restriction of previous pre conditions to the definedness of only those indices between oc and the minimum of ot and et. As has been the pattern in the previous development steps, this is a further relaxation of the pre condition from the more abstract specification. Both of the rely and guarantee conditions of SEARCH-ODD-SCAN are precisely the same as those in SEARCH-ODD.

The post condition of SEARCH-ODD-SCAN implies that of SEARCH-ODD when composed sequentially with that of SEARCH-ODD-INIT. This sequential composition uses the *Seq-I* development rule and proceeds in a similar manner as does the (triple) composition of INIT, SEARCHES, and FINAL into FINDP. The structure of the post condition will be detailed after SEARCH-ODD-SCAN-BODY is covered as it makes more sense in the context of how the specification is decomposed.

This brings us to the net development step, which decomposes SEARCH-ODD-SCAN into a *While* loop.

$$\textit{test-odd} \triangleq oc < ot \wedge oc < et$$
$$\text{SEARCH-ODD-SCAN} \triangleq \textit{mk-While}(\textit{test-odd}, \text{SEARCH-ODD-SCAN-BODY})$$

The specification of SEARCH-ODD-SCAN-BODY is given in Figure 7.9. The justification of this decomposition is made through the use of the *While-I* development rule.

SEARCH-ODD-SCAN-BODY

rd $v: X^*$

wr $oc, ot: \mathbb{N}_1$

pre $oc < ot$

$\wedge\, \forall i \in \{j \in \mathbb{N}_1 \mid oc \leq j < ot \wedge \textit{is-odd}(j)\} \cdot \delta(\textit{pred}(v(i)))$

rely $v = \overleftarrow{v} \wedge oc = \overleftarrow{oc} \wedge ot = \overleftarrow{ot}$

guar $ot = \overleftarrow{ot} \vee (ot < \overleftarrow{ot} \wedge \textit{pred}(v(ot)))$

post $(\overleftarrow{oc} < oc)$

$\wedge\, \forall i \in \{j \in \mathbb{N}_1 \mid \overleftarrow{oc} \leq j < \textit{min}(oc, ot) \wedge \textit{is-odd}(j)\} \cdot \neg\, \textit{pred}(v(i))$

Figure 7.9: Specification of SEARCH-ODD-SCAN-BODY.

As noted much earlier –in Chapter 3– the *While-I* development rule depends on the test expression of the *While* loop having some interesting properties. The expression consists of two parts: the first part must be independent of all interference and the second part must be such that at most one variable is shared and that variable may be used only once. For the $\textit{test-odd}$ expression in this decomposition, it is not hard to see that these requirements are met. The given form of *While-I* denotes the independent, stable portion of the test as b_s and that corresponds to the $oc < ot$ term in $\textit{test-odd}$. This term only references the variables oc and ot, and so it is obvious from the $\textit{rely}$-SEARCH-ODD-SCAN that this term is free of interference in evaluation. The portion of the test with a shared variable is denoted as b_u in the *While-I* rule, and this corresponds to $oc < et$ in $\textit{test-odd}$. The only shared variable in this term is et and it is clear from the form of the term and the language semantics that the value of et is only read once during expression evaluation. The important part about this test is that we can depend on the value of the b_s element of the test to remain stable during execution of the loop body, and that we can treat the evaluation of the b_u component as though it were evaluated in a single state. This last property is required to ensure that the last antecedent from *While-I*[6] is relevant — the form of the antecedent assumes single-state evaluation.[7]

Looking at SEARCH-ODD-SCAN-BODY, we first note that it no longer has any access to the et variable; this specification is no longer dependent on any variables which have interesting interference. The practical effect of this is that further development of this specification can proceed almost as though we were developing a non-concurrent program.

The pre condition of SEARCH-ODD-SCAN-BODY includes the definedness condition that the previous specifications use, but it also includes the stable portion of $\textit{test-odd}$ from the *While* loop. This has the effect of ensuring that the initial value of oc is an index into the vector. The guarantee condition is an elaboration of that from SEARCH-ODD-SCAN, including the specific detail that if ot decreases it explicitly becomes the value of oc.

The post condition of SEARCH-ODD-SCAN-BODY has a similar structure to that of SEARCH-ODD-SCAN. The differences are connected and relate to the fact that this post condition must be a well-founded relation (where $\textit{post}$-SEARCH-ODD-SCAN need not be), and to the way this particular example deals with separating the search threads. First, the selection set in the universal quantifier uses $\textit{min}(oc, ot)$ instead of $\textit{min}(ot, et)$ as an upper

[6] $\overleftarrow{\neg\,(b_s \wedge b_u)} \wedge R \;\Rightarrow\; \neg\,(b_s \wedge b_u)$

[7] See the discussion on multiple-state evaluation in Chapter 3.

bound: inside the body of the $While$ loop we must ensure that there are no references to shared variables. That the overall $While$ loop can use $min(ot, et)$ as its upper bound is due to the requirement that the oc variable be greater than one of the top markers, ot or et. On this basis, then, it is not hard to see that $post$-SEARCH-ODD-SCAN is a weakened reflexive closure of $post$-SEARCH-ODD-SCAN-BODY.

That $post$-SEARCH-ODD-SCAN-BODY is a well-founded relation rests both on the strict inequality in the $\overleftarrow{oc} < oc$ term and on the implicit constraint that if oc is greater than ot the pre condition of the specification is not satisfied.

SEARCH-ODD-SCAN-STEP
wr $oc: \mathbb{N}_1$
pre **true**
rely $oc = \overleftarrow{oc}$
guar **true**
post $oc = \overleftarrow{oc} + 2$

Figure 7.10: Specification of SEARCH-ODD-SCAN-STEP.

The decomposition of SEARCH-ODD-SCAN-BODY is done as a sequential split, thus:

SEARCH-ODD-SCAN-BODY $\triangleq$
SEARCH-ODD-SCAN-CHECK **;** SEARCH-ODD-SCAN-STEP

The specification of SEARCH-ODD-SCAN-STEP is in Figure 7.10, and is responsible for incrementing the odd loop counter; we will not examine it in detail. The specification of SEARCH-ODD-SCAN-CHECK in Figure 7.11 is more interesting, however.

SEARCH-ODD-SCAN-CHECK
rd $v: X^*, oc: \mathbb{N}_1$
wr $ot: \mathbb{N}_1$
pre $\delta(pred(v(oc)))$
rely $v = \overleftarrow{v} \land oc = \overleftarrow{oc} \land ot = \overleftarrow{ot}$
guar $ot = \overleftarrow{ot} \lor (ot = \overleftarrow{oc} \land pred(v(\overleftarrow{oc})))$
post $\left(ot = \overleftarrow{oc} \land pred(v(\overleftarrow{oc}))\right) \lor \left(ot = \overleftarrow{ot} \land \neg\, pred(v(\overleftarrow{oc}))\right)$

Figure 7.11: Specification of SEARCH-ODD-SCAN-CHECK.

The further decomposition of SEARCH-ODD-SCAN-CHECK is intended to use the *If-b-I* development rule rather than the simpler *If-I* development rule. The implementation of this specification (as shown in Figure 7.12) is:

if $pred(v(oc))$ **then** $ot \leftarrow oc$ **fi;**

and it can be seen that interference from the rely condition cannot affect the evaluation of the test expression in the *If* construct. This allows for a pre condition on the body of the *If* construct that includes the fact that if the body is executed then the index held in oc satisfies the predicate. In turn we can conclude that, when execution of the implementation has completed, either the ot variable will be modified if oc is an index that satisfies the predicate, or ot will be left unmodified.

The SEARCH-ODD-SCAN-CHECK specification no longer depends upon et in the variable list — part of the decomposition to this step is the decision to have this implementation of SEARCH-ODD-SCAN-BODY only check the potential satisfaction of one index. The use of et in $post$-SEARCH-ODD-SCAN-BODY is a weakening of a post condition that could be derived from the composition of SEARCH-ODD-SCAN-CHECK and SEARCH-ODD-SCAN-STEP; using et in the specification of SEARCH-ODD-SCAN-BODY allows us to justify the optimization of terminating the loop when oc passes any satisfactory index.

The pre condition of SEARCH-ODD-SCAN-CHECK only requires that the predicate is defined for the value in the vector of the starting value of oc. This simplification is a direct consequence of the design decision to have the body only check one index. The rely condition is simple: it requires what is essentially the identity on the vector and the two odd-branch variables, oc and ot. The guarantee condition of SEARCH-ODD-SCAN-CHECK is the same as that for SEARCH-ODD-SCAN-BODY, as the expected behaviour is precisely the same: it will only modify ot if oc is less than the initial value of ot and it satisfies the predicate.

The post condition –fittingly for a specification that we expect to decompose using an If-based development rule– is a disjunction on two cases. The first case handles instances where the initial value of oc does satisfy the predicate; here ot will end up equivalent to the initial value of oc. The second case is where the initial value of oc does not satisfy the predicate, and asserts that ot will be unchanged in that case.

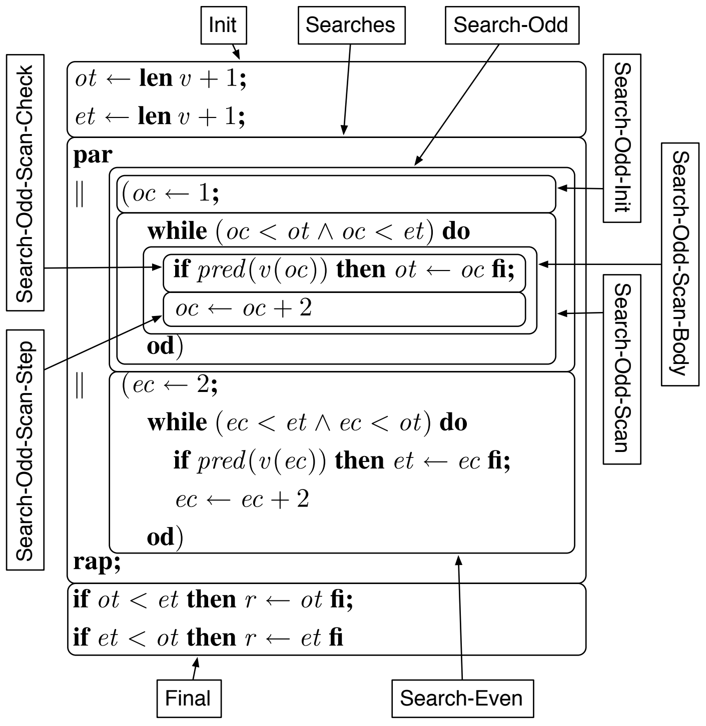
The SEARCH-ODD-SCAN-CHECK specification, as noted, can be decomposed into an *If* construct, and, as is visible in its implementation, the body of that *If* construct is an assignment. That assignment can be justified by the *Assign-I* rule, and the step between the richer post condition of the SEARCH-ODD-SCAN-CHECK specification and the assignment can be made using the *Weaken* rule.

Finishing this development, we present Figure 7.12 which contains a pseudo-code implementation of FINDP according to the development in this section. Portions of the pseudo-code have been boxed indicating their corresponding specification.

7.2 Atomicity Via Software Transactional Memory

This section presents an alternative development of the FINDP example, this time using software transactional memory to stay with the idea of a single variable t. We start from the specification of SEARCH in Figure 7.5 as the initial steps are common to both developments. Given that we will be using software transactional memory in this development we do not need to reify t as we did in the previous section. Instead we will use the language's $Atomic$ construct to manage access to t. The description and explanation of each development step in this section will be terse until we reach the actual use of the $Atomic$ construct as most of the development follows a parallel path to that in the previous section.

The first development step is from SEARCH on the odd indices to SEARCH-STM-ODD, and the resulting specification is shown in Figure 7.13. Justification of this development step is done using the *Weaken* rule, as all of the conditions are specializations from a



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

Figure 7.12: An annotated pseudo-code implementation of FINDP without atomic blocks.

SEARCH-STM-ODD
rd $v \colon X^*$
wr $oc, t \colon \mathbb{N}_1$
pre $\forall i \in \{j \in \mathbb{N}_1 \mid j < t \wedge \mathit{is\text{-}odd}(j)\} \cdot \delta(pred(v(i)))$
rely $v = \overleftarrow{v} \wedge oc = \overleftarrow{oc} \wedge t \leq \overleftarrow{t}$
guar $t = \overleftarrow{t} \vee (t < \overleftarrow{t} \wedge pred(v(t)))$
post $\forall i \in \{j \in \mathbb{N}_1 \mid j < t \wedge \mathit{is\text{-}odd}(j)\} \cdot \neg\, pred(v(i))$

Figure 7.13: Specification of SEARCH-STM-ODD.

general set of indices to the particular case of the odd indices.

Search-STM-Odd-Scan
rd $v \colon X^*$
wr $oc, t \colon \mathbb{N}_1$
pre $\forall i \in \{j \in \mathbb{N}_1 \mid oc \le j < t \land \textit{is-odd}(j)\} \cdot \delta(\textit{pred}(v(i)))$
rely $v = \overleftarrow{v} \land oc = \overleftarrow{oc} \land t \le \overleftarrow{t}$
guar $t = \overleftarrow{t} \lor (t < \overleftarrow{t} \land \textit{pred}(v(t)))$
post $\overleftarrow{oc} \le oc \land \neg\,(oc < t)$
$\land \forall i \in \{j \in \mathbb{N}_1 \mid \overleftarrow{oc} \le j < t \land \textit{is-odd}(j)\} \cdot \neg\,\textit{pred}(v(i))$

Figure 7.14: Specification of Search-STM-Odd-Scan.

The next development step decomposes Search-STM-Odd into two parts, thus:

$$\begin{array}{l}\text{Search-STM-Odd} \triangleq \\ \quad \text{Search-Odd-Init}\ ;\ \text{Search-STM-Odd-Scan}\end{array}$$

where Search-Odd-Init is the same as in Figure 7.7, and Search-STM-Odd-Scan is given in Figure 7.14.

The justification of this development step is done using the *Seq-I* development rule, naturally. The first significant difference in the specifications relative to the previous section appears here: the need to constantly reference the minimum of ot and et –thus reading a variable which is only written to in another thread– is eliminated.

Search-STM-Odd-Scan-Body
rd $v \colon X^*$
wr $oc, t \colon \mathbb{N}_1$
pre $oc < t \land \forall i \in \{j \in \mathbb{N}_1 \mid oc \le j < t \land \textit{is-odd}(j)\} \cdot \delta(\textit{pred}(v(i)))$
rely $v = \overleftarrow{v} \land oc = \overleftarrow{oc} \land t \le \overleftarrow{t}$
guar $t = \overleftarrow{t} \lor (t < \overleftarrow{t} \land \textit{pred}(v(t)))$
post $\overleftarrow{oc} < oc \land \forall i \in \{j \in \mathbb{N}_1 \mid \overleftarrow{oc} \le j < t \land \textit{is-odd}(j)\} \cdot \neg\,\textit{pred}(v(i))$

Figure 7.15: Specification of Search-STM-Odd-Scan-Body.

The next development step gives us a *While* loop with $oc < t$ as the test, thus:

$$\begin{array}{l}\text{Search-STM-Odd-Scan} \triangleq \\ \quad \textit{mk-While}(oc < t, \text{Search-STM-Odd-Scan-Body})\end{array}$$

where Search-STM-Odd-Scan-Body is given in Figure 7.15. This development step uses the *While-I* rule, though it only uses the unstable term, b_u, of the test defined in the rule. The stable term, b_s, is taken to be **true** for the purposes of this development.

This specification of the loop's body decomposes into two parts, thus:

$$\begin{array}{l}\text{Search-STM-Odd-Scan-Body} \triangleq \\ \quad \text{Search-STM-Odd-Scan-Check}\ ;\ \text{Search-Odd-Scan-Step}\end{array}$$

where Search-Odd-Scan-Step is given previously in Figure 7.10 and Search-STM-Odd-Scan-Check is given in Figure 7.16. As in the previous section, the "step" speci-

SEARCH-STM-ODD-SCAN-CHECK
rd $v\colon X^*,\ oc\colon \mathbb{N}_1$
wr $t\colon \mathbb{N}_1$
pre $oc < t \wedge \delta(pred(v(oc)))$
rely $v = \overleftarrow{v} \wedge oc = \overleftarrow{oc} \wedge t \leq \overleftarrow{t}$
guar $t = \overleftarrow{t} \vee (t < \overleftarrow{t} \wedge t = \overleftarrow{oc} \wedge pred(v(\overleftarrow{oc})))$
post $(t = \overleftarrow{oc} \wedge pred(v(\overleftarrow{oc}))) \vee (t = \overleftarrow{t} \wedge \neg\, pred(v(\overleftarrow{oc})))$

Figure 7.16: Specification of SEARCH-STM-ODD-SCAN-CHECK.

fication satisfies the irreflexive portion of the loop body's post condition, and the "check" specification deals with the potential need to alter the value of t.

SEARCH-STM-ODD-SCAN-SET
rd $oc\colon \mathbb{N}_1$
wr $t\colon \mathbb{N}_1$
pre **true**
rely $oc = \overleftarrow{oc} \wedge t \leq \overleftarrow{t}$
guar $t = \overleftarrow{t} \vee (\overleftarrow{oc} < \overleftarrow{t} \wedge t = \overleftarrow{oc})$
post $t \leq \overleftarrow{oc}$

Figure 7.17: Specification of SEARCH-STM-ODD-SCAN-SET.

The decomposition of SEARCH-STM-ODD-SCAN-CHECK pins down where the current index in oc is actually checked for satisfaction of the predicate. This decomposition uses the predicate directly as the test, thus:

$$\begin{aligned}&\text{SEARCH-STM-ODD-SCAN-CHECK} \triangleq\\ &\quad \textit{mk-If}\,(pred(v(oc)), \text{SEARCH-STM-ODD-SCAN-SET})\end{aligned}$$

where SEARCH-STM-ODD-SCAN-SET is in Figure 7.17. It is at this point where the development becomes interesting from the perspective of the $Atomic$ construct, as the guarantee condition of SEARCH-STM-ODD-SCAN-SET creates the need for an atomic test-and-set operation due to the shared nature of t and the fine-grained expression evaluation semantics of the language.

STM-TEST-AND-SET
rd $oc\colon \mathbb{N}_1$
wr $t\colon \mathbb{N}_1$
pre **true**
rely I
guar **true**
post $(\overleftarrow{oc} < \overleftarrow{t} \;\Rightarrow\; t = \overleftarrow{oc}) \wedge (\overleftarrow{t} \geq \overleftarrow{oc} \;\Rightarrow\; t = \overleftarrow{t}\,)$

Figure 7.18: Specification of STM-TEST-AND-SET.

Because of the need for a test-and-set operation, the development of SEARCH-STM-ODD-SCAN-SET is done using the *Atomic-psat-I* rule, thus:

$$\text{SEARCH-STM-ODD-SCAN-SET} \triangleq \textit{mk-Atomic}(\text{STM-TEST-AND-SET})$$

where STM-TEST-AND-SET is given in Figure 7.18.

At this point there is a problem if we try to prove that SEARCH-STM-ODD-SCAN-SET terminates: as we are unable to show that all of the variables on which STM-TEST-AND-SET depends are left untouched by the environment, the full satisfaction *Atomic-I* rule cannot be used. Instead we must use the partial satisfaction rule *Atomic-psat-I* for the decomposition, then establish a termination argument that allows the use of the *sat-I* rule.

For this example, the argument that SEARCH-STM-ODD-SCAN-SET must terminate comes from inspecting the even thread and the knowledge that a) the even thread is the only source of alterations to the shared variable, t; and b) that the even thread is designed in a manner that is essentially symmetric to the odd thread. We note that the even thread may only modify t a finite number of times, based on the fact that t will be modified at most once per even index, and there are a finite number of even indices in a finite vector. Combine that with the knowledge –from the language semantics– that it is not possible for two $Atomic$ constructs to cause mutual retry attempts –as one of the constructs must commit before the other– and we can then conclude that SEARCH-STM-ODD-SCAN-SET must terminate. This comes on the basis that its dependent variables can only be modified a finite number of times before the environment will act as an identity thenceforth on those variables.

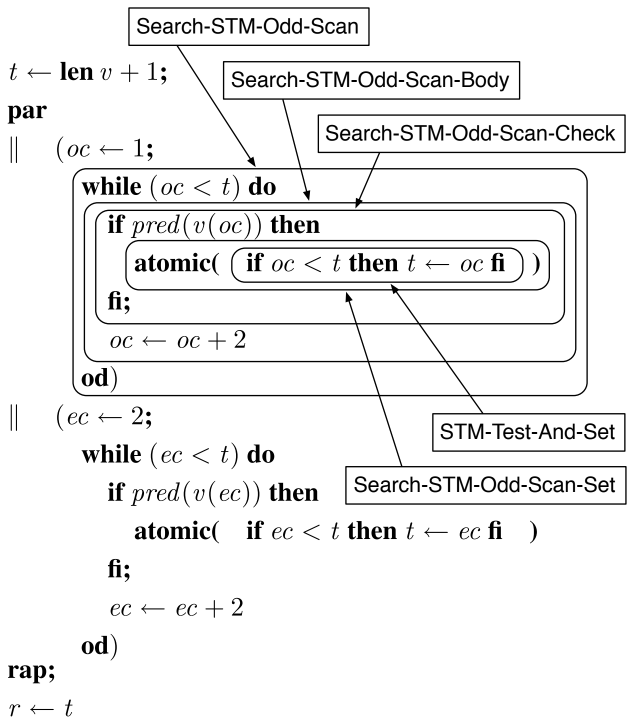
In fact, we know that the situation for SEARCH-STM-ODD-SCAN-SET is even more tightly constrained than we describe above: we know that the overall SEARCH-ODD and SEARCH-EVEN threads will only modify t at most once each for the lowest odd and even indices. The penalty of a STM retry that troubles the convergence proofs is something that may only happen once, at most, for any execution of this program. A STM retry can only be triggered by external writes to either the local counter variable (which will not happen) or to the variable t. In the latter situation, we know that the write comes from the parallel thread, and we know that it cannot happen again.

At this point the decomposition implementation of STM-TEST-AND-SET is straightforward; this is shown in the overall implementation in Figure 7.19. As with Figure 7.12 we have boxed portions of the pseudo-code to indicate the corresponding specification. Unlike the previous implementation, however, only a small portion of the pseudo-code has been boxed: only the portions specifically developed to use the $Atomic$ construct are annotated.

7.3 Comparison of the Developments

At a superficial level –just comparing the pseudo-code– the reification-style development has the simpler structure: it uses fewer conditional tests and, represented as a tree, its program structure is shallower. Furthermore, as it avoids the use of the $Atomic$ construct, its properties with respect to termination are easier to determine. The particular details of the two developments –at the superficial level and in the reasoning which lead to the decompositions– are less relevant than the trade-offs that arise due to the constraints around reification and the $Atomic$ construct.

Data reification is a powerful tool when used in the decomposition of specifications. It



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

Figure 7.19: An annotated pseudo-code implementation of FINDP with atomic blocks.

can be used –as is done in the present example– to take a data structure which is modified by parallel processes and replace it with a new data structure which avoids the contention altogether. Doing this in the present example is straightforward but, in general, finding a suitable reification can be very difficult.

The difficulty of finding a suitable reification –and, in some cases, the simple lack of any suitable reification– suggests that an alternate approach to reason about shared variables in necessary. Software transactional memory, as embodied in a construct which is similar to the $Atomic$ construct presented in this work, provides this alternative.

Use of the $Atomic$ construct allows for the decompositions of a program into components that have strict limits on the interference which they can tolerate. This ability comes at a cost, however, which depends on the development rule which is used to introduce the $Atomic$ construct. With the *Atomic-I* rule we may only use the $Atomic$ construct when we are trying to prevent other processes from reading intermediate states. This particular rule is useful for operations such as an atomic variable exchange. And with the *Atomic-psat-I* rule we have more flexibility in the use of the $Atomic$, but the termination properties become much harder to determine — to gain any termination property we need to provide an alternate argument that ensures that the construct will always converge to the **nil** statement.

In situations where both reification and the $Atomic$ construct could be used, the choice between them depends on the requirements of the particular situation. Reification requires the developer to expend more effort on the data abstractions used in the development, but can result in a program which avoids the need for any direct concurrency control. It can also result in the interaction between elements of the reified data structure being more fragile than the simpler, unreified data structures. Use of the $Atomic$ construct allows for simpler data structures, but at the cost of potential overhead due to the need to monitor changes made to the state outside of the $Atomic$, as well as some ambiguity in the termination properties of the program. On the other had, use of a direct concurrency control mechanism highlights the location of operations which are sensitive to interference.

Finally, we note that our use of the development rules in this example follows the same general style as operation decomposition in VDM and that VDM has a pragmatic philosophy with respect to hitting a “dead-end” in a development: that is, backtrack and attempt another approach. The use of reification and the $Atomic$ construct fit neatly in this philosophy: we would suggest using the $Atomic$ construct early in development if a suitable reification is not obvious, and only reify away the need for the $Atomic$ construct if it becomes necessary to do so.

$\boxed{8}$ — Conclusions

8.1 Recapitulation

This thesis starts, in Chapter 2, by presenting a semantic model for a language which includes an Atomic construct implemented as a form of software transactional memory. We continue, in Chapter 3, by introducing the rely/guarantee development rules and, in particular, we give two rules for the Atomic construct and a refined rule for the While construct.

Chapter 4 introduces the methods and tools used to connect the concepts in Chapters 2 and 3. We describe natural deductive proofs and the style of name binding we use in their application. The two forms of augmented semantics –distinguishing and merging– are introduced to aid reasoning about the behaviour of programs executed in an environment with interference. We also give definitions for the Within and $\mathit{Converges}$ properties so that we have a mechanism to reason about whether or not a program's behaviour satisfies its specification. Key to using these properties in the context of the semantic model is our notion of pinch sets, and these are used heavily in the soundness proofs of Chapter 6.

The subsidiary lemmas used in the soundness proofs are described in Chapter 5, and of particular interest are *Isolation-Par-L*, *Isolation-Par-R*, *Comp-Par*, and *Frame-Rule*. The first three of these are essentially formalizations of what we consider to be one of the great strengths of rely/guarantee reasoning: that separate components of a program may be decomposed (or composed) and reasoned about in isolation without knowledge of the internal structure of the other components. The last lemma *–Frame-Rule–* is inspired by work in separation logic,[1] and tries to emulate some of the ease with which separation logic focuses only on the relevant portions of the memory store.

The soundness proofs of the development rules with respect to the semantic model are described in Chapter 6, giving a guide to reading the detailed proofs in Appendix D. Finally, in Chapter 7 we work through an example in two ways to show the application of the rules in the development of a specification into pseudo-code.

8.2 Reflections and Future Work

8.2.1 Assessing the Initial Aims

This thesis has two major aims: to provide a rely/guarantee framework which is able to deal with interference in concurrent program development in a fine-grained manner; and to provide that framework in such a way as to minimize the semantic gap between the framework and the semantic model of the development language. Part of the aim to provide a framework which can deal with fine-grained interference includes the need for a mechanism to control the level of granularity for portions of the development. And above these

[1] See [Rey02] and [PBO07].

aims is the intent that we end up with a tractable framework for software development.

Addressing the smaller goals first, we have produced a rely/guarantee framework which deals with fine-grained concurrency. The development rules given in Chapter 3 do an effective job of characterizing how the expected interference and the required behaviour of a program interact. Multiple-state evaluation of the expression in a construct can be handled in a manner with more finesse than simply requiring that there be no effective interference during the evaluation; this is particularly apparent in the rule given for the While construct. Concurrency control is provided via the $\mathit{Atomic}/\mathit{STM}$ construct pair, though there is a need for development rules for this pair of constructs which have more flexible constraints than those of the *Atomic-I* and *Atomic-psat-I* rules.

The soundness proofs provided in Chapter 6 tie the provided development rules to the semantic model of Chapter 2. As the rules are shown to be sound in terms of the semantic model, it is clear that the semantic gap between the development rules and the semantic model has been eliminated, especially in light of the fact that the semantic rules and development rules may be used at the same logical level. Furthermore, because the soundness proofs use the semantic model directly instead of a secondary model Aczel trace semantics, we have minimized the number of possible steps between the development rules and the semantic model.

The broader goal of a tractable reasoning framework has been achieved, with some caveats. The provided framework is usable for languages similar to that described by our semantic model. However, before the model can be considered analogous to languages commonly used in software development it requires the addition of features such as aliasing, free variable introduction, scoping mechanisms, and procedure calls. These features pose their own challenges, and their addition to the framework is not expected to be simple.

The generation of the soundness proofs require a greater technical proficiency than using the development rules in a software development. We expect that the task of generating the soundness proofs will be done by an expert, and that the proofs, once done, need not be generated again. As such we do not consider the greater difficulty of these proofs to be a factor in the relative tractability of this method.

8.2.2 Extracting the Meta-Method

One clear avenue of future work would be to extract the meta-method which is implicit in this work and firmly characterize its requirements and applicability. At first glance the primary elements of such a method are the basic logical framework, a formal model of the development language, and a set of inference rules to reason about programs in the language. This thesis uses LPF, the SOS model in Chapter 2, and the rely/guarantee rules in Chapter 3, respectively, as its primary elements, but there is nothing which explicitly constrains those choices. The key feature of such a meta-method –on top of the basic elements– is to ensure that the development rules are an extension of the framework established by the basic logical framework and the language model.

Once the meta-method is characterized we would then be free to explore other choices for the basic elements. It would certainly be possible to replace the semantic model used here with models of other languages, and even the particular formalism used to model the language could be changed. We suggest the use of a modelling formalism which is easily accessible by the software developers who are expected to use the resulting framework.

A more interesting proposition, however, is the potential for the use of basic logical frameworks other than LPF; it is not hard to imagine that a modal logic could be used to good effect. The replacement of the basic logical framework must satisfy a practical constraint, however: as we hope to be able to develop software with the resulting development framework, the basic logical framework chosen must be able to address the potential for undefined terms. And, if the resulting framework is to be tractable, it must address the question of undefinedness in a manner which does not impede the software developer.

8.2.3 Software Transactional Memory and Termination

Software transactional memory is a fascinating concept: it promises a mechanism that eliminates most of the problems associated with locking protocols. It is not a panacea, of course: STM is only applicable to situations where every contained operation is fully reversible and where the effects of every operation can be hidden until the STM transaction is ready to commit. Constructs based on STM also have to deal with the costs associated with a transactional retry –actually reversing every operation that has happened– though careful implementation of the STM construct in the language can mitigate this.

It is possible to abuse STM transactions in many of the same ways that explicit locks may be abused. In particular, placing a large program inside a STM transaction is just as bad an idea as locking a variable and then running a large program before unlocking the variable. The difference between these two situations is that in the STM-based example there is the risk and cost of having that large program be undone and restarted whereas in the lock-based example any other program which needs access to the locked variable will be halted. Poor program designs will bring out the worst features of any construct.

On the topic of the implementation of the STM construct, many of the works cited in Section 2.5 have designed their implementations of STM as code libraries to be used in development rather than as features within the target language itself. From the perspective of rigorous development this is a mistake: it puts a burden on the user to ensure that nothing violates the pragmatic intent of the STM implementation. It is better to have the STM implementation done as a integral part of the language; the cost of ensuring that the constructs that implement STM do so correctly becomes a part of the cost of verifying that the language compiler is a valid implementation of its semantic model.

The termination arguments in the convergence proofs of Chapter 6 follow from one of three things: the inherent nature of the construct (in that it cannot fail to terminate), from well-founded induction (in the case of the While construct), or from restricting interference (in the case of the Atomic construct). There are other possible arguments that can be made and their use would allow for development rules which apply to situations not covered by those presented.

One possible argument involves an extension of the well-founded induction used in the convergence proof of the While construct to the convergence proof for the Atomic

construct. If it could be shown that interference would always reach –in a finite number of steps– a state such that interference never caused any further alteration to the value of certain values, then the rely condition representing this interference would correspond to a relation suitable for use with a modified form of well-founded induction. The proof outline would rest on two cases: the first is where there were enough consecutive interference transitions which did not alter the value of any variable the $Atomic$ depended on; and the second is in the case where interference did cause the $Atomic$ to retry, thus reducing the number of further possible retries. In this case we might say that interference eventually becomes quiescent with respect to the variable the $Atomic$ depends upon.

Another possibility is based on a notion of "periodic" quiescence in interference, essentially a formalization of the notion that an $Atomic$ block will always terminate because there are sufficiently long periods where interference does not alter the values of variables on which the $Atomic$ block depends. This notion is in conflict with the inherent expressive weakness of the rely/guarantee framework, however, and there is currently no mechanism to specify durations in the logic. Current work by Hayes, Jackson, and Jones [HJJ03, JHJ06] may lead to a useful means of dealing with this.

A combination of this eventual quiescence of interference with the wait conditions used by Stølen may allow us to reason about parallel compositions in which one branch depends upon the other branch performing some action. This combination would require that this entire work be extended to the overall rely/guarantee framework used by Stølen, however, and would be no small undertaking.

8.2.4 Constructs and Rules

Extension of the semantic model in Chapter 2 to include more language features –such as dynamic blocks– would have the effect of bringing the model closer to that of commonly used programming languages. This work has focused on basic interference behaviour; adding features such as dynamic blocks to the model would allow for development rules that are more precise than those presented in Chapter 3. In particular, the addition of dynamic blocks would give the semantic model the basis for investigating mechanisms in the development rules that can "hide" variables in the rely and guarantee conditions.

Another construct which might be suggested as an immediate candidate for inclusion into the semantic model is an explicit STM retry operation, analogous to the usual "break" command for loops. This would involve modelling non-local exits in the semantic model, and would result in profound effects on both the model and the resulting development rules. From this construct the next step could be to add a full fault-handling mechanism to the semantic model.

The most ambitious goal would be to provide a set of development rules that are sound with respect to a complete semantics for a language commonly used in industry. This would be a challenge as most commonly used languages have run-time systems which may disagree on various details of the language model. Although such a semantic model for the Java language has been developed in [KNvO$^+$02], much work remains if a set of rely/guarantee development rules are to be proven sound with respect to their model.

The soundness proofs for the corresponding development rules would be straightforward for most of the normal language constructs as they are just repetitions of the ideas in this work; however, many of the constructs in their model are not dealt with here.

A smaller goal, at the level of the rely/guarantee development rules, relates to the split-predicate structure used in the *While-I* development rule given in Chapter 3. This rule is structured so that the conditional expression in the *While* construct is a conjunction of two terms. The two terms are constrained in different ways, and the overall rule can handle development situations where only one of the parts is needed. This suggests that other rules which are affected by expression evaluation may benefit –or at least be generalizable– from being written in this split-predicate style.

8.2.5 Automated Reasoning

Automated reasoning in support of the development process has two components: support for reasoning with the semantic model; and support for the actual development steps using the development rules. Automated support for the generation of the soundness proofs – though not part of the development process per se– is integral to this, but should be less visible to a software developer.

Mechanization of the soundness proofs is certainly possible: Prensa Nieto does this in [PN03] for a set of rely/guarantee rules relative to a semantic model with flat,[2] coarse-grained concurrency. Prensa Nieto's work would make a good starting point to mechanize the work in this thesis if Isabelle/HOL were chosen as the inference engine. At a minimum, however, that work would have to be extended with relational post conditions, multiple-state expression evaluation, and nested concurrency. Generating a model of multiple-state expression evaluation is straightforward, but ensuring that the automated reasoner can handle the non-equivalence between the logical evaluation of an expression and the semantic evaluation in a useful way requires some finesse. After providing a means of handling the basic features of the work in this thesis, some time would have to be spent increasing the level of detail in the proofs of Chapter 6 and Appendix D, as well as providing proofs for the lemmas in Chapter 5.

The use of pinch sets will be a challenge for to automated reasoning: pinch sets require that we reason about the set of all reachable configurations from a given starting point. As it is not, in general, feasible to calculate this set of configurations the proofs contained in this thesis use knowledge of the semantic rules and the structure of the constructs to ensure that we have valid pinch sets. An automated tool would require a deep understanding of the semantic model to verify a proposed pinch set, much less instantiate one without user intervention.

Taken from the perspective of the actual reasoning effort, automated reasoning support for software development using the development rules would require nothing beyond that required for the soundness proofs. Creating a usable interface to such a tool, however, is another problem entirely. We envisage a workbench-style tool that could assist the development process, allowing the user to not only verify that a particular development step is

[2]In the sense that parallel construct cannot be nested.

valid, but also verify that a particular program fragment satisfies a specification. Work on such things as Mural [JJLM91] and the RODIN platform [Rod07] suggest that this should be possible.

9 — Bibliography

[Acz82] P. Aczel. A note on program verification. Letter to Cliff Jones, Manchester, January 1982.

[BFL^{+}94] Juan C. Bicarregui, John S. Fitzgerald, Peter A. Lindsay, Richard Moore, and Brian Ritchie. *Proof in VDM: A Practitioner's Guide*. Springer Series On Formal Approaches To Computing And Information Technology. Springer-Verlag, 1994.

[BS01] Manfred Broy and Ketil Stølen. *Specification and Development of Interactive Systems*. Springer-Verlag, 2001.

[Bue00] Martin Buechi. *Safe Language Mechanisms for Modularization and Concurrency*. PhD thesis, Turku, 2000.

[CH05] Christopher Cole and Maurice P. Herlihy. Snapshots and software transactional memory. *Science of Computer Programming*, 58(3):310–324, December 2005.

[CJ00] Pierre Collette and Cliff B. Jones. Enhancing the tractability of rely/guarantee specifications in the development of interfering operations. Foundations Of Computing, pages 277–307. MIT Press, Cambridge, MA, USA, 2000.

[CJ07] Joey W. Coleman and Cliff B. Jones. A structural proof of the soundness of rely/guarantee rules. *Journal of Logic and Computation*, 17(4):807–841, August 2007.

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

[Col94] Pierre Collette. *Design of Compositional Proof Systems Based on Assumption-Commitment Specifications — Application to UNITY*. PhD thesis, Louvain-la-Neuve, June 1994.

[Daw91] John Dawes. *The VDM-SL Reference Guide*. Pitman Publishing, 1991.

[dR01] Willem-Paul de Roever. *Concurrency Verification: Introduction to Compositional and Noncompositional Methods*. Number 54 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, New York, NY, USA, January 2001.

[Fit52] Frederic Benton Fitch. *Symbolic Logic: an Introduction*. Ronald Press Company, New York, 1952.

[HF03] Tim Harris and Keir Fraser. Language support for lightweight transactions. In *OOPSLA '03: Proceedings of the 18th annual ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications*, pages 388–402, New York, NY, USA, 2003. ACM Press.

[HJJ03] Ian J. Hayes, Michael A. Jackson, and Cliff B. Jones. Determining the specification of a control system from that of its environment. In *FM 2003:*

Formal Methods: International Symposium of Formal Methods Europe, volume 2805 of *Lecture Notes in Computer Science*, pages 154–169. Springer-Verlag, September 2003.

[HLMS03] Maurice Herlihy, Victor Luchangco, Mark Moir, and William N. Scherer, III. Software transactional memory for dynamic-sized data structures. In *PODC '03: Proceedings of the twenty-second annual symposium on Principles of distributed computing*, pages 92–101, New York, NY, USA, 2003. ACM Press.

[HM93] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: architectural support for lock-free data structures. In *ICCA '93: Proceedings of the 20th annual international symposium on Computer architecture*, pages 289–300, New York, NY, USA, May 1993. ACM Press.

[HMPJH05] Tim Harris, Simon Marlow, Simon Peyton Jones, and Maurice Herlihy. Composable memory transactions. In *PPoPP '05: Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 48–60, New York, NY, USA, 2005. ACM Press.

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

[Hoa72] C. A. R. Hoare. *Towards a theory of parallel programming*, pages 61–71. In Hoare and Perrott [HP72], 1972.

[HP72] C. A. R. Hoare and R. H. Perrott. *Operating System Techniques*. Academic Press, 1972.

[JHJ06] Cliff B. Jones, Ian J. Hayes, and Michael A. Jackson. Specifying systems that connect to the physical world. Technical Report Series CS-TR-964, Newcastle University, May 2006.

[JJLM91] Cliff B. Jones, K. D. Jones, P. A. Lindsay, and R. Moore. *mural: A Formal Development Support System*. Springer-Verlag, 1991.

[Jon81] Cliff B. Jones. *Development Methods for Computer Programs including a Notion of Interference*. PhD thesis, Oxford University, June 1981. Printed as: Programming Research Group, Technical Monograph 25.

[Jon83a] C. B. Jones. Tentative steps toward a development method for interfering programs. *ACM Transactions on Programming Languages and Systems*, 5(4):596–619, October 1983.

[Jon83b] Cliff B. Jones. Specification and design of (parallel) programs. In R. E. A. Mason, editor, *Information Processing 83: Proceedings of the IFIP 9th World Congress*, pages 321–332. IFIP, 1983.

[Jon90] Cliff B. Jones. *Systematic Software Development Using VDM*. Prentice-Hall, Inc., 2nd edition edition, 1990.

[Jon96] Cliff B. Jones. Accommodating interference in the formal design of concurrent object-based programs. *Formal Methods in System Design*, 8(2):105–122, 1996.

[Jon01] C. B. Jones. The transition from VDL to VDM. *Journal of Univeral Computer Science*, 7(8):631–640, August 2001.

[Jon03a] Cliff B Jones. The early search for tractable ways of reasonning about programs. *IEEE, Annals of the History of Computing*, 25(2):26–49, April–June 2003.

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

[Jon07] Cliff B. Jones. Annotated bibliography on rely/guarantee conditions. Available on the WWW as `http://homepages.cs.ncl.ac.uk/cliff.jones/ftp-stuff/rg-hist.pdf`, 2007.

[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.

[LN79] H. C. Lauer and R. M. Needham. On the duality of operating system structures. *Operating Systems Review*, 13(2):3–19, April 1979.

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

[McC63] John McCarthy. Towards a mathematical science of computation. In Cicely M. Popplewell, editor, *Proceedings of the IFIP Congress, Munich, 1962*, pages 21–28. North-Holland Publishing Company, Amsterdam, 1963.

[McC66] John McCarthy. A formal description of a subset of ALGOL. In Steel [Ste66].

[Mor88] Carroll Morgan. The specification statement. *ACM Transactions on Programming Languages and Systems*, 10(3):403–419, 1988.

[MSH$^+$06] Virendra J. Marathe, Michael F. Spear, Christopher Heriot, Athul Acharya, David Eisenstat, William N. Scherer, III, and Michael L. Scott. Lowering the overhead of nonblocking software transactional memory. Presented at TRANSACT: First ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing, PLDI, Ottawa, June 2006, June 2006.

[MSS05] Virendra J. Marathe, William N. Scherer, III, and Michael L. Scott. Adaptive software transactional memory. In *Distributed Computing*, volume 3724 of *Lecture Notes in Computer Science*, pages 354–368. Springer, 2005.

[OG76] Susan Owicki and David Gries. Verifying properties of parallel programs: An axiomatic approach. *Communications of the ACM*, 19(5):279–285, May 1976.

[Owi75] Susan S. Owicki. *Axiomatic Proof Techniques for Parallel Programs*. PhD thesis, Department of Computer Science, Cornell University, Ithaca, NY, USA, 1975. 75-251.

[PBO07] Matthew Parkinson, Richard Bornat, and Peter O'Hearn. Modular verification of a non-blocking stack. In *Proceedings of POPL'07*, January 2007.

[Plo81] Gordon D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, 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.

[PN03] Leonor Prensa Nieto. The Rely-Guarantee method in Isabelle/HOL. In P. Degano, editor, *European Symposium on Programming (ESOP'03)*, volume 2618 of *LNCS*, pages 348–362. Springer, 2003.

[Pra65] Dag Prawitz. *Natural Deduction: A Proof-Theoretical Study*. Dover Publications, Inc., 1965.

[Rey02] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS '02: Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science*, pages 55–74, Washington, DC, USA, 2002. IEEE Computer Society.

[RNN92] Hanne Riis Nielson and Flemming Nielson. *Semantics with Applications: A Formal Introduction*. Wiley, 1992.

[RNN07] Hanne Riis Nielson and Flemming Nielson. *Semantics with Applications: An Appetizer*. Undergraduate Topics in Computer Science. Springer, 2007.

[Rod07] The RODIN project. `http://rodin.cs.ncl.ac.uk/`, December 2007.

[Sch97] Fred B. Schneider. *On Concurrent Programming*. Graduate Texts in Computer Science. Springer-Verlag New York, Inc., 1997.

[Sit74] Richard L. Sites. *Proving That Computer Programs Terminate Cleanly*. PhD thesis, Stanford University, May 1974.

[ST95] Nir Shavit and Dan Touitou. Software transactional memory. In *PODC '95: Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing*, pages 204–213, New York, NY, USA, 1995. ACM Press.

[Ste66] Thomas B. Steel, Jr., editor. *Formal Language Description Languages for Computer Programming*. IFIP Working Conference on Formal Language Description Languages, North-Holland Publishing Company, Amsterdam, 1966.

[Sti86] Colin Stirling. A compositional reformulation of Owicki-Gries's partial correctness logic for a concurrent while language. In L. Kott, editor, *Automata, Languages and Programming: ICALP'86*, volume 226 of *Lecture Notes in Computer Science*, pages 407–415. Springer-Verlag, 1986.

[Sto77] Joseph Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*. MIT Press, 1977.

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

[Stø91a] Ketil Stølen. An attempt to reason about shared-state concurrency in the style of VDM. In *VDM '91: Proceedings of the 4th International Symposium of VDM Europe on Formal Software Development-Volume I*, volume 551 of *Lecture Notes in Computer Science*, pages 324–342, London, UK, 1991. Springer-Verlag.

[Stø91b] Ketil Stølen. A method for the development of totally correct shared-state parallel programs. In *CONCUR '91: Proceedings of the 2nd International Conference on Concurrency Theory*, pages 510–525, London, UK, 1991. Springer-Verlag.

[Sza69] M. E. Szabo, editor. *The Collected Papers of Gerhard Gentzen*. North-Holland Publishing Company, Amsterdam, The Netherlands, 1969.

[Win93] Glynn Winskel. *The Formal Semantics of Programming Languages: An Introduction*. MIT Press, 1993.

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

Appendices

$\boxed{\text{A}}$ — Language Definition

A.1 Semantic Model

A.1.1 Abstract Syntax

$$Stmt = Assign \mid Atomic \mid If \mid Par \mid Seq \mid STM \mid While \mid \mathbf{nil}$$

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

$$\begin{aligned} Assign &:: \; id : Id \\ &\quad\;\; e : Expr \end{aligned}$$

$$Atomic :: \; body : Stmt$$

$$\begin{aligned} If &:: \; b : Expr \\ &\quad\; body : Stmt \end{aligned}$$

$$\begin{aligned} Par &:: \; left : Stmt \\ &\quad\; right : Stmt \end{aligned}$$

$$\begin{aligned} Seq &:: \; left : Stmt \\ &\quad\; right : Stmt \end{aligned}$$

$$\begin{aligned} STM &:: \; orig : Stmt \\ &\quad\; \sigma_0 : \Sigma \\ &\quad\; body : Stmt \\ &\quad\; \sigma : \Sigma \end{aligned}$$

$$\begin{aligned} While &:: \; b : Expr \\ &\quad\; body : Stmt \end{aligned}$$

$$\begin{aligned} Dyad &:: \; op : + \mid - \mid < \mid = \mid > \mid \\ &\qquad\quad \wedge \mid \vee \\ &\quad\; left : Expr \\ &\quad\; right : Expr \end{aligned}$$

A.1.2 Context Conditions

```
$wf\text{-}Expr\colon (Expr \times Id\text{-}\mathbf{set}) \to \{\textsc{Bool}, \textsc{Int}, \textsc{Error}\}$
$wf\text{-}Expr(e, ids) \triangleq$
    **cases** $e$ **of**
                        $e \in \mathbb{B} \to \textsc{Bool}$
                        $e \in \mathbb{Z} \to \textsc{Int}$
              $e \in Id \wedge e \in ids \to \textsc{Int}$
        $mk\text{-}Dyad(op, left, right) \to$ **let** $ltype = wf\text{-}Expr(left, ids)$ **in**
                                   **if** $ltype = wf\text{-}Expr(right, ids) \wedge$
                                      $ltype \neq \textsc{Error}$
                                   **then cases** $(op, ltype)$ **of**
                                          $(+, \textsc{Int}) \to \textsc{Int}$
                                          $(-, \textsc{Int}) \to \textsc{Int}$
                                          $(<, \textsc{Int}) \to \textsc{Bool}$
                                          $(=, \textsc{Int}) \to \textsc{Bool}$
                                          $(>, \textsc{Int}) \to \textsc{Bool}$
                                        $(=, \textsc{Bool}) \to \textsc{Bool}$
                                        $(\wedge, \textsc{Bool}) \to \textsc{Bool}$
                                        $(\vee, \textsc{Bool}) \to \textsc{Bool}$
                                        **others** $\textsc{Error}$
                                        **end**
                                   **else** $\textsc{Error}$
    **others** $\textsc{Error}$
    **end**
$wf\text{-}Stmt\colon (Stmt \times Id\text{-}\mathbf{set}) \to \mathbb{B}$
```

$$
\begin{array}{l}
\mathit{wf\text{-}Stmt}(\mathit{stmt}, \mathit{ids}) \triangleq \\
\quad \textbf{cases } \mathit{stmt} \textbf{ of} \\
\qquad \begin{array}{rl}
\textbf{nil} \to & \textbf{true} \\
\mathit{mk\text{-}Assign}(\mathit{id}, e) \to & \mathit{id} \in \mathit{ids} \wedge \mathit{wf\text{-}Expr}(e, \mathit{ids}) = \textsc{Int} \\
\mathit{mk\text{-}Atomic}(\mathit{body}) \to & \mathit{wf\text{-}Stmt}(\mathit{body}, \mathit{ids}) \\
\mathit{mk\text{-}If}(b, \mathit{body}) \to & \mathit{wf\text{-}Expr}(b, \mathit{ids}) = \textsc{Bool} \wedge \mathit{wf\text{-}Stmt}(\mathit{body}, \mathit{ids}) \\
\mathit{mk\text{-}Par}(\mathit{left}, \mathit{right}) \to & \mathit{wf\text{-}Stmt}(\mathit{left}, \mathit{ids}) \wedge \mathit{wf\text{-}Stmt}(\mathit{right}, \mathit{ids}) \\
\mathit{mk\text{-}Seq}(\mathit{left}, \mathit{right}) \to & \mathit{wf\text{-}Stmt}(\mathit{left}, \mathit{ids}) \wedge \mathit{wf\text{-}Stmt}(\mathit{right}, \mathit{ids}) \\
\mathit{mk\text{-}STM}(\mathit{orig}, \sigma_0, \mathit{body}, \sigma) \to & \mathit{wf\text{-}Stmt}(\mathit{orig}, \mathit{ids}) \wedge \mathit{wf\text{-}Stmt}(\mathit{body}, \mathit{ids}) \wedge \\
& \textbf{dom}\, \sigma_0 = \textbf{dom}\, \sigma = \mathit{ids} \\
\mathit{mk\text{-}While}(b, \mathit{body}) \to & \mathit{wf\text{-}Expr}(b, \mathit{ids}) = \textsc{Bool} \wedge \mathit{wf\text{-}Stmt}(\mathit{body}, \mathit{ids})
\end{array} \\
\quad \textbf{others false} \\
\quad \textbf{end}
\end{array}
$$

A.1.3 Semantic Objects

$$\Sigma = \mathit{Id} \xrightarrow{m} \mathbb{Z}$$

$$\mathit{Config} = \mathit{Stmt} \times \Sigma$$

A.1.4 Semantic Rules

Expressions

$$\xrightarrow{e} : (\mathit{Expr} \times \Sigma) \times \mathit{Expr}$$

Identifiers

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

Dyads

$$\boxed{\text{Dyad-L}}\ \frac{(\mathit{left}, \sigma) \xrightarrow{e} \mathit{left}'}{(\mathit{mk\text{-}Dyad}(\mathit{op}, \mathit{left}, \mathit{right}), \sigma) \xrightarrow{e} \mathit{mk\text{-}Dyad}(\mathit{op}, \mathit{left}', \mathit{right})}$$

$$\boxed{\text{Dyad-R}}\ \frac{(\mathit{right}, \sigma) \xrightarrow{e} \mathit{right}'}{(\mathit{mk\text{-}Dyad}(\mathit{op}, \mathit{left}, \mathit{right}), \sigma) \xrightarrow{e} \mathit{mk\text{-}Dyad}(\mathit{op}, \mathit{left}, \mathit{right}')}$$

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

Statements

$$\xrightarrow{s} : \mathit{Config} \times \mathit{Config}$$

Assign

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

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

Atomic/STM

$$\boxed{\text{STM-Atomic}}\ \frac{}{(mk\text{-}Atomic(body), \sigma) \xrightarrow{s} (mk\text{-}STM(body, \sigma, body, \sigma), \sigma)}$$

$$\boxed{\text{STM-Step}}\ \frac{\begin{array}{c}(Vars(orig) \lhd \sigma_0) = (Vars(orig) \lhd \sigma)\\ (body, \sigma_s) \xrightarrow{s} (body', \sigma'_s)\end{array}}{(mk\text{-}STM(orig, \sigma_0, body, \sigma_s), \sigma) \xrightarrow{s} (mk\text{-}STM(orig, \sigma_0, body', \sigma'_s), \sigma)}$$

$$\boxed{\text{STM-E}}\ \frac{(Vars(orig) \lhd \sigma_0) = (Vars(orig) \lhd \sigma)}{(mk\text{-}STM(orig, \sigma_0, \mathbf{nil}, \sigma_s), \sigma) \xrightarrow{s} (\mathbf{nil}, \sigma \dagger (Vars(orig) \lhd \sigma_s))}$$

$$\boxed{\text{STM-Retry}}\ \frac{(Vars(orig) \lhd \sigma_0) \neq (Vars(orig) \lhd \sigma)}{(mk\text{-}STM(orig, \sigma_0, body, \sigma_s), \sigma) \xrightarrow{s} (mk\text{-}Atomic(orig), \sigma)}$$

If

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

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

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

Parallel

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

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

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

Sequence

$$\boxed{\text{Seq-Step}}\ \frac{(\mathit{left}, \sigma) \xrightarrow{s} (\mathit{left}', \sigma')}{(\mathit{mk\text{-}Seq}(\mathit{left}, \mathit{right}), \sigma) \xrightarrow{s} (\mathit{mk\text{-}Seq}(\mathit{left}', \mathit{right}), \sigma')}$$

$$\boxed{\text{Seq-E}}\ \frac{}{(\mathit{mk\text{-}Seq}(\mathbf{nil}, \mathit{right}), \sigma) \xrightarrow{s} (\mathit{right}, \sigma)}$$

While

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

A.2 Augmented Semantic Model

$$\mathit{Rely} = \Sigma \times \Sigma$$
$$R \in \mathit{Rely}$$
$$R^{\mathit{Config}} \triangleq \{((S, \sigma), (S, \sigma')) \mid S \in \mathit{Stmt} \wedge (\sigma, \sigma') \in R\}$$

A.2.1 Distinguishing Semantics

$$\xrightarrow[\_]{r} : \mathit{Config} \times \mathit{Rely} \times \mathit{Config}$$

$$\xrightarrow[R]{r} \equiv \xrightarrow{s} \cup R^{\mathit{Config}}$$

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

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

A.2.2 Merging Semantics

$$\xrightarrow[\_]{m} : \mathit{Config} \times \mathit{Rely} \times \mathit{Config}$$

M-Step $\equiv$ A-R-Step $\diamond$ A-S-Step $\diamond$ A-R-Step

$$\xrightarrow[R]{m} \equiv R^{\mathit{Config}} \diamond \xrightarrow{s} \diamond\, R^{\mathit{Config}}$$

$$\boxed{\text{M-Step}}\ \frac{\begin{array}{c}[\![R]\!](\sigma_0, \sigma_1) \\ (S, \sigma_1) \xrightarrow{s} (S', \sigma_2) \\ [\![R]\!](\sigma_1, \sigma_f)\end{array}}{(S, \sigma_0) \xrightarrow[R]{m} (S', \sigma_f)}$$

B — Rely/Guarantee Rules

B.1 Framework Assumptions

$$\boxed{\text{PR-ident}}\ \frac{}{\overleftarrow{P} \land R \ \Rightarrow\ P}$$

$$\boxed{\text{QR-ident}}\ \frac{}{Q \diamond R \ \Rightarrow\ Q}$$

B.2 Meta Rules

$$\boxed{\text{Weaken}}\ \frac{\begin{array}{c}(P, R) \vdash S \ \mathbf{sat}\ (G, Q)\\ P' \ \Rightarrow\ P\\ R' \ \Rightarrow\ R\\ G \ \Rightarrow\ G'\\ Q \ \Rightarrow\ Q'\end{array}}{(P', R') \vdash S \ \mathbf{sat}\ (G', Q')}$$

$$\boxed{\text{psat-I}}\ \frac{\begin{array}{c}\mathit{Within}_s(P, R, S, G)\\ \forall \sigma, \sigma' \in \Sigma \cdot \Big(\llbracket P \rrbracket(\sigma) \land (S, \sigma) \xrightarrow[R]{r}\!\!* \ (\mathbf{nil}, \sigma')\Big) \ \Rightarrow\ \llbracket Q \rrbracket(\sigma, \sigma')\end{array}}{(P, R) \vdash S \ \mathbf{psat}\ (G, Q)}$$

$$\boxed{\text{sat-I}}\ \frac{\begin{array}{c}(P, R) \vdash S \ \mathbf{psat}\ (G, Q)\\ \mathit{Converges}_s(S, P, R, \{\mathbf{nil}\})\end{array}}{(P, R) \vdash S \ \mathbf{sat}\ (G, Q)}$$

B.3 Development Rules

Assignment

$$\boxed{\text{Assign-I}}\ \frac{\begin{array}{c}R \ \Rightarrow\ I_{\mathit{Vars}(e) \cup \{id\}}\\ G = \{(\sigma, \sigma \dagger \{id \mapsto \llbracket e \rrbracket(\sigma)\}) \mid \sigma \in \Sigma\} \cup I\\ Q = \{(\sigma, \sigma') \mid \sigma, \sigma' \in \Sigma \land \sigma'(id) = \llbracket e \rrbracket(\sigma)\}\end{array}}{(P, R) \vdash \mathit{mk\text{-}Assign}(id, e) \ \mathbf{sat}\ (G, Q)}$$

Atomic blocks

$$\boxed{\text{Atomic-psat-I}}\ \frac{\begin{array}{c}(P, I) \vdash \mathit{body} \ \mathbf{psat}\ (\mathbf{true}, Q')\\ Q' \ \Rightarrow\ G\\ \overleftarrow{P} \land R \diamond Q' \diamond R \ \Rightarrow\ Q\end{array}}{(P, R) \vdash \mathit{mk\text{-}Atomic}(\mathit{body}) \ \mathbf{psat}\ (G, Q)}$$

$$\boxed{\text{Atomic-I}}\ \frac{\begin{array}{c}(P, I) \vdash \mathit{body}\ \mathbf{sat}\ (\mathbf{true}, Q') \\ Q' \ \Rightarrow\ G \\ R \ \Rightarrow\ I_{\mathit{Vars}(\mathit{body})} \\ \overleftarrow{P} \wedge R \diamond Q' \diamond R \ \Rightarrow\ Q\end{array}}{(P, R) \vdash \mathit{mk\text{-}Atomic}(\mathit{body})\ \mathbf{sat}\ (G, Q)}$$

Conditional Execution

$$\boxed{\text{If-b-I}}\ \frac{\begin{array}{c}(P \wedge b, R) \vdash \mathit{body}\ \mathbf{sat}\ (G, Q) \\ R \ \Rightarrow\ I_{\mathit{Vars}(b)} \\ \overleftarrow{P} \wedge \neg\, \overleftarrow{b} \wedge R \ \Rightarrow\ Q\end{array}}{(P, R) \vdash \mathit{mk\text{-}If}(b, \mathit{body})\ \mathbf{sat}\ (G, Q)}$$

$$\boxed{\text{If-I}}\ \frac{\begin{array}{c}(P, R) \vdash \mathit{body}\ \mathbf{sat}\ (G, Q) \\ \overleftarrow{P} \wedge R \ \Rightarrow\ Q\end{array}}{(P, R) \vdash \mathit{mk\text{-}If}(b, \mathit{body})\ \mathbf{sat}\ (G, Q)}$$

Sequential Composition

$$\boxed{\text{Seq-I}}\ \frac{\begin{array}{c}(P, R) \vdash \mathit{left}\ \mathbf{sat}\ (G, Q_l \wedge P_r) \\ (P_r, R) \vdash \mathit{right}\ \mathbf{sat}\ (G, Q_r) \\ Q_l \diamond Q_r \ \Rightarrow\ Q\end{array}}{(P, R) \vdash \mathit{mk\text{-}Seq}(\mathit{left}, \mathit{right})\ \mathbf{sat}\ (G, Q)}$$

$$\boxed{\text{Seq-raw-I}}\ \frac{\begin{array}{c}(P_l, R_l) \vdash \mathit{left}\ \mathbf{sat}\ (G_l, Q_l) \\ (P_r, R_r) \vdash \mathit{right}\ \mathbf{sat}\ (G_r, Q_r) \\ Q_l \ \Rightarrow\ P_r\end{array}}{(P_l, R_l \wedge R_r) \vdash \mathit{mk\text{-}Seq}(\mathit{left}, \mathit{right})\ \mathbf{sat}\ (G_l \vee G_r, Q_l \diamond Q_R)}$$

Parallel Composition

$$\boxed{\text{Par-I}}\ \frac{\begin{array}{c}(P, R \vee G_r) \vdash \mathit{left}\ \mathbf{sat}\ (G_l, Q_l) \\ (P, R \vee G_l) \vdash \mathit{right}\ \mathbf{sat}\ (G_r, Q_r) \\ G_l \vee G_r \ \Rightarrow\ G \\ \overleftarrow{P} \wedge Q_l \wedge Q_r \wedge (R \vee G_l \vee G_r)^* \ \Rightarrow\ Q\end{array}}{(P, R) \vdash \mathit{mk\text{-}Par}(\mathit{left}, \mathit{right})\ \mathbf{sat}\ (G, Q)}$$

Iteration

$$\boxed{\text{While-I}}\ \frac{\begin{array}{c}\mathit{well\text{-}founded}(W) \\ \mathit{bottoms}(W) \subseteq [\![\neg\,(b_s \wedge b_u)]\!] \\ R \ \Rightarrow\ W^* \wedge I_{\mathit{Vars}(b_s)} \\ \mathit{SingleSharedVar}(b_u, R) \\ \overleftarrow{\neg\,(b_s \wedge b_u)} \wedge R \ \Rightarrow\ \neg\,(b_s \wedge b_u) \\ (P \wedge b_s, R) \vdash \mathit{body}\ \mathbf{sat}\ (G, W \wedge P)\end{array}}{(P, R) \vdash \mathit{mk\text{-}While}(b_s \wedge b_u, \mathit{body})\ \mathbf{sat}\ (G, W^* \wedge P \wedge \neg\,(b_s \wedge b_u))}$$

$\boxed{\text{C}}$ — Collected Lemmas

C.1 Behavioural

$$\boxed{\text{Within-Rely}}\ \frac{(C, C') \in \text{A-R-Step}}{Within_1(C, C', G)}$$

$$\boxed{\text{Within-Prog}}\ \frac{\begin{array}{c}((S, \sigma), (S', \sigma')) \in \text{A-S-Step} \\ [\![G]\!](\sigma, \sigma')\end{array}}{Within_1((S, \sigma), (S', \sigma'), G)}$$

$$\boxed{\text{Within-Weaken}}\ \frac{\begin{array}{c}G \ \Rightarrow \ G' \\ Within_1(C, C', G)\end{array}}{Within_1(C, C', G')}$$

$$\boxed{\text{Within-Multi}}\ \frac{\begin{array}{c}C^{ij} = \{(C_i, C_j) \mid C_0 \xrightarrow[R]{r}* \ C_i \xrightarrow[R]{r} C_j \xrightarrow[R]{r}* \ C_f\} \\ \forall (C_i, C_j) \in C^{ij} \ \cdot \ Within_1(C_i, C_j, G)\end{array}}{Within_m(R, C_0, C_f, G)}$$

$$\boxed{\text{Within-Weaken-Multi}}\ \frac{\begin{array}{c}R' \ \Rightarrow \ R \\ G \ \Rightarrow \ G' \\ Within_m(R, C, C', G)\end{array}}{Within_m(R', C, C', G')}$$

$$\boxed{\text{Within-Concat}}\ \frac{\begin{array}{c}C^i = \{C_i \mid C_0 \xrightarrow[R]{r}* \ C_i \xrightarrow[R]{r}* \ C_f\} \\ C^p \subseteq C^i \\ \forall C_i \in C^i \cdot \left(\exists C_p \in C^p \cdot C_i \xrightarrow[R]{r}* \ C_p \vee C_p \xrightarrow[R]{r}* \ C_i\right) \\ \forall C_p \in C^p \cdot Within_m(R, C_0, C_p, G) \wedge Within_m(R, C_p, C_f, G)\end{array}}{Within_m(R, C_0, C_f, G)}$$

$$\boxed{\text{Within-Relation}}\ \frac{Within_m(R, (S, \sigma), (S', \sigma'), G)}{[\![(R \vee G)^*]\!](\sigma, \sigma')}$$

$$\boxed{\text{Within-Equiv}}\ \frac{\begin{array}{c}(S_0, \sigma_0) \xrightarrow[R]{r} (S_1, \sigma_1) \\ (S_0', \sigma_0) \xrightarrow[R]{r} (S_1', \sigma_1) \\ (S_0 = S_1) \ \Leftrightarrow \ (S_0' = S_1') \\ Within_1((S_0, \sigma_0), (S_1, \sigma_1), G)\end{array}}{Within_1((S_0', \sigma_0), (S_1', \sigma_1), G)}$$

$$\boxed{\text{Within-Prog-Cor}}\ \dfrac{(C, C') \in \left(\begin{array}{l}\text{A-R-Step} \cup \text{Assign-Eval} \cup \text{STM-Atomic} \cup \\ \text{If-Eval} \cup \text{If-T-E} \cup \text{If-F-E} \cup \text{Seq-E} \cup \\ \text{STM-Step} \cup \text{STM-Retry} \cup \text{Par-E} \cup \text{While}\end{array}\right)}{\mathit{Within}_1(C, C', G)}$$

$$\boxed{\text{Within-Rely-Trivial}}\ \dfrac{(\mathbf{nil}, \sigma) \xrightarrow[R]{r}* (\mathbf{nil}, \sigma')}{\mathit{Within}_m(R, (\mathbf{nil}, \sigma), (\mathbf{nil}, \sigma'), G)}$$

$$\boxed{\text{Within-Abstract}}\ \dfrac{\forall \sigma, \sigma' \in \Sigma \cdot \left(\begin{array}{l}[\![P]\!]\sigma \wedge (S, \sigma) \xrightarrow[R]{r}* (\mathbf{nil}, \sigma') \\ \quad \Rightarrow \ \mathit{Within}_m(R, (S, \sigma), (\mathbf{nil}, \sigma'), G)\end{array}\right)}{\mathit{Within}_s(P, R, S, G)}$$

$$\boxed{\text{Within-Concrete}}\ \dfrac{\begin{array}{c}\mathit{Within}_s(P, R, S, G) \\ [\![P]\!](\sigma) \\ (S, \sigma) \xrightarrow[R]{r}* (\mathbf{nil}, \sigma')\end{array}}{\mathit{Within}_m(R, (S, \sigma), (\mathbf{nil}, \sigma'), G)}$$

C.2 Convergence

$$\boxed{\text{Conv-I}}\ \dfrac{\begin{array}{c}C^i = \{C_i \mid C_0 \xrightarrow[R]{m}* C_i\} \\ C^f = \{(S_f, \sigma_f) \in \mathit{Config} \mid S_f \in \mathit{Set}_f\} \\ \forall C_i \in C^i \cdot \left(\exists C_f \in C^f \cdot (C_i \xrightarrow[R]{m}* C_f) \vee (C_f \xrightarrow[R]{m}* C_i)\right)\end{array}}{\mathit{Converges}_c(C_0, R, \mathit{Set}_f)}$$

$$\boxed{\text{Conv-Weaken}}\ \dfrac{\begin{array}{c}R' \ \Rightarrow \ R \\ \mathit{Set}_f \subseteq \mathit{Set}'_f \\ \mathit{Converges}_c(C, R, \mathit{Set}_f)\end{array}}{\mathit{Converges}_c(C, R', \mathit{Set}'_f)}$$

$$\boxed{\text{Conv-Concat}}\ \dfrac{\begin{array}{c}\mathit{Converges}_c(C_0, R, \mathit{Set}_0) \\ C^i = \{(S_i, \sigma_i) \mid C_0 \xrightarrow[R]{m}* (S_i, \sigma_i) \wedge S_i \in \mathit{Set}_0\} \\ \forall C_i \in C^i \cdot \mathit{Converges}_c(C_i, R, \mathit{Set}_f)\end{array}}{\mathit{Converges}_c(C_0, R, \mathit{Set}_f)}$$

$$\boxed{\text{Conv-Abstract}}\ \dfrac{\forall \sigma \in \Sigma \cdot [\![P]\!](\sigma) \ \Rightarrow \ \mathit{Converges}_c((S, \sigma), R, \mathit{Set}_f)}{\mathit{Converges}_s(S, P, R, \mathit{Set}_f)}$$

$$\boxed{\text{Conv-Concrete}}\ \dfrac{\begin{array}{c}[\![P]\!](\sigma) \\ \mathit{Converges}_s(S, P, R, \mathit{Set}_f)\end{array}}{\mathit{Converges}_c((S, \sigma), R, \mathit{Set}_f)}$$

$$\boxed{\text{Conv-Wrap-Seq}}\ \frac{\begin{array}{c} Converges_c((left, \sigma), R, Set_f) \\ Set'_f = \{mk\text{-}Seq(S, right) \mid S \in Set_f\} \end{array}}{Converges_c((mk\text{-}Seq(left, right), \sigma), R, Set'_f)}$$

$$\boxed{\text{Conv-Wrap-Par}}\ \frac{\begin{array}{c} (P, R \vee G_r) \vdash left\ \mathbf{psat}\ (G_l, Q_l) \\ (P, R \vee G_l) \vdash right\ \mathbf{psat}\ (G_r, Q_r) \\ Converges_c((left, \sigma), R \vee G_r, Set_l) \\ Converges_c((right, \sigma), R \vee G_l, Set_r) \\ Set_f = \{mk\text{-}Par(S_l, S_r) \mid S_l \in Set_l \wedge S_r \in Set_r\} \end{array}}{Converges_c((mk\text{-}Par(left, right), \sigma), R, Set_f)}$$

$$\boxed{\text{Conv-Wrap-STM}}\ \frac{\begin{array}{c} C = (mk\text{-}STM(body, \sigma_0, body, \sigma_0), \sigma) \\ Converges_c((body, \sigma_0), I, Set_f) \\ \forall \tau \in \{(C', C'') \mid C \xrightarrow[R]{m}{*}\ C' \xrightarrow[R]{m} C''\} \cdot \tau \notin \text{STM-Retry} \end{array}}{Converges_c(C, R, \{S \in STM \mid S.body \in Set_f\})}$$

C.3 Isolation & Composition

$$\boxed{\text{Seq-Equiv}}\ \frac{(S, \sigma) \xrightarrow[R]{m}{*}\ (S', \sigma')}{(mk\text{-}Seq(S, right), \sigma) \xrightarrow[R]{m}{*}\ (mk\text{-}Seq(S', right), \sigma')}$$

$$\boxed{\text{Isolation-If}}\ \frac{\begin{array}{c} S' \in If \ \Rightarrow\ S'.body \neq S \\ (mk\text{-}If(b, S), \sigma_0) \xrightarrow[R]{r}{*}\ (mk\text{-}If(\mathbf{true}, S), \sigma_i) \xrightarrow[R]{r}{*}\ (S', \sigma_j) \end{array}}{(S, \sigma_0) \xrightarrow[R]{r}{*}\ (S', \sigma_j)}$$

$$\boxed{\text{Comp-Par}}\ \frac{\begin{array}{c} [\![P]\!](\sigma) \\ Within_s(P, R \vee G_r, left, G_l) \\ Within_s(P, R \vee G_l, right, G_r) \\ (left, \sigma) \xrightarrow[R \vee G_r]{m}{*}\ (left', \sigma') \\ (right, \sigma) \xrightarrow[R \vee G_l]{m}{*}\ (right', \sigma') \end{array}}{(mk\text{-}Par(left, right), \sigma) \xrightarrow[R \vee G_l \vee G_r]{m}{*}\ (mk\text{-}Par(left', right'), \sigma')}$$

$$\boxed{\text{Isolation-Par-L}}\ \frac{\begin{array}{c} [\![P]\!](\sigma) \\ (P, R \vee G_r) \vdash left\ \mathbf{sat}\ (G_l, Q_l) \\ (P, R \vee G_l) \vdash right\ \mathbf{sat}\ (G_r, Q_l) \\ (mk\text{-}Par(left, right), \sigma) \xrightarrow[R]{r}{*}\ (mk\text{-}Par(left', right'), \sigma') \end{array}}{(left, \sigma) \xrightarrow[R \vee G_r]{r}{*}\ (left', \sigma')}$$

$$\boxed{\text{Isolation-Par-R}}\ \frac{\begin{array}{c}[\![P]\!](\sigma)\\ (P, R \vee G_r) \vdash \mathit{left}\ \mathbf{sat}\ (G_l, Q_l)\\ (P, R \vee G_l) \vdash \mathit{right}\ \mathbf{sat}\ (G_r, Q_l)\\ (\mathit{mk}\text{-}\mathit{Par}(\mathit{left}, \mathit{right}), \sigma) \xrightarrow[R]{r}\!\!*\ (\mathit{mk}\text{-}\mathit{Par}(\mathit{left}', \mathit{right}'), \sigma')\end{array}}{(\mathit{right}, \sigma) \xrightarrow[R \vee G_l]{r}\!\!*\ (\mathit{right}', \sigma')}$$

$$\boxed{\text{Isolation-Seq-R}}\ \frac{\begin{array}{c}S' \in \mathit{Seq} \ \Rightarrow\ S'.\mathit{right} \neq S\\ (\mathit{mk}\text{-}\mathit{Seq}(\mathbf{nil}, S), \sigma) \xrightarrow[R]{r}\!\!*\ (S', \sigma')\end{array}}{(S, \sigma) \xrightarrow[R]{r}\!\!*\ (S', \sigma')}$$

$$\boxed{\text{Isolation-STM}}\ \frac{(\mathit{mk}\text{-}\mathit{STM}(\mathit{orig}, \sigma_0, S, \sigma_s), \sigma) \xrightarrow[R]{r}\!\!*\ (\mathit{mk}\text{-}\mathit{STM}(\mathit{orig}, \sigma_0, S', \sigma_s'), \sigma')}{(S, \sigma_s) \xrightarrow[I]{r}\!\!*\ (S', \sigma_s')}$$

$$\boxed{\text{Isolation-While}}\ \frac{\begin{array}{c}\mathit{wh} = \mathit{mk}\text{-}\mathit{While}(b, \mathit{body})\\ \tau = ((\mathit{mk}\text{-}\mathit{If}(\mathbf{true}, \mathit{mk}\text{-}\mathit{Seq}(\mathit{body}, \mathit{wh})), \sigma), (\mathit{wh}, \sigma'))\\ \tau \in \big(\text{A-R-Step} \cup \text{If-T-E} \cup \text{Seq-Step} \cup \text{Seq-E}\big)^*\end{array}}{(\mathit{body}, \sigma) \xrightarrow[R]{r}\!\!*\ (\mathbf{nil}, \sigma')}$$

C.4 Miscellaneous

$$\boxed{\text{Frame-Rule}}\ \frac{\begin{array}{c}(S_0, \sigma_0) \xrightarrow[I]{r}\!\!*\ (S_1, \sigma_1)\\ (\mathit{Vars}(S_0) \lhd \sigma_0) = (\mathit{Vars}(S_0) \lhd \sigma_0')\\ (\mathit{Vars}(S_0) \lhd \sigma_1) = (\mathit{Vars}(S_0) \lhd \sigma_1')\\ (\mathit{Vars}(S_0) \ntriangleleft \sigma_0) = (\mathit{Vars}(S_0) \ntriangleleft \sigma_1)\\ (\mathit{Vars}(S_0) \ntriangleleft \sigma_0') = (\mathit{Vars}(S_0) \ntriangleleft \sigma_1')\end{array}}{(S_0, \sigma_0') \xrightarrow[I]{r}\!\!*\ (S_1, \sigma_1')}$$

$$\boxed{\text{Rely-Trivial}}\ \frac{\begin{array}{c}(S, \sigma) \xrightarrow[R]{r}\!\!*\ (S', \sigma')\\ ((S, \sigma), (S', \sigma')) \in \left(\begin{array}{l}\text{A-R-Step} \cup \text{Assign-Eval} \cup \text{STM-Atomic} \cup\\ \text{If-Eval} \cup \text{If-T-E} \cup \text{If-F-E} \cup \text{Seq-E} \cup\\ \text{STM-Step} \cup \text{STM-Retry} \cup \text{Par-E} \cup \text{While}\end{array}\right)^*\end{array}}{[\![R]\!](\sigma, \sigma')}$$

$$\boxed{\text{Sequential-Effect}}\ \frac{(S, \sigma) \xrightarrow[I]{r}\!\!*\ (S', \sigma')}{\sigma' = \sigma \dagger (\mathit{Vars}(S) \lhd \sigma')}$$

$$\boxed{\text{Single-Eval-Assign}}\;\frac{\begin{array}{c} R \;\Rightarrow\; I_{Vars(e)} \\ (mk\text{-}Assign(id, e), \sigma) \xrightarrow[R]{r}* (mk\text{-}Assign(id, v), \sigma') \\ v \in \mathbb{Z} \end{array}}{v = [\![e]\!](\sigma) = [\![e]\!](\sigma')}$$

$$\boxed{\text{Single-Eval-If}}\;\frac{\begin{array}{c} R \;\Rightarrow\; I_{Vars(e)} \\ (mk\text{-}If(b, body), \sigma) \xrightarrow[R]{r}* (mk\text{-}If(v, body), \sigma') \\ v \in \mathbb{B} \end{array}}{v = [\![b]\!](\sigma) = [\![b]\!](\sigma')}$$

$$\boxed{\text{While-interstices-pre}}\;\frac{\begin{array}{c} [\![P]\!](\sigma) \\ (P \wedge b_s, R) \vdash body \ \mathbf{psat}\ (G, W \wedge P) \\ R \;\Rightarrow\; W^* \wedge I_{Vars(b_s)} \\ wh = mk\text{-}While(b_s \wedge b_u, body) \\ (wh, \sigma) \xrightarrow[R]{r}* (mk\text{-}Seq(body, wh), \sigma') \end{array}}{[\![P \wedge b_s]\!](\sigma')}$$

$$\boxed{\text{While-interstices-psat}}\;\frac{\begin{array}{c} [\![P]\!](\sigma) \\ (P \wedge b_s, R) \vdash body \ \mathbf{psat}\ (G, W \wedge P) \\ R \;\Rightarrow\; W^* \wedge I_{Vars(b_s)} \\ wh = mk\text{-}While(b_s \wedge b_u, body) \\ C^w = \{(wh, \sigma') \mid (wh, \sigma) \xrightarrow[R]{m}* (wh, \sigma') \wedge \sigma \neq \sigma'\} \end{array}}{\forall (wh, \sigma') \in C^w \cdot [\![W \wedge P]\!](\sigma, \sigma')}$$

$\boxed{\text{D}}$ — Full Proofs

D.1 Theorem PSAT

from $(P, R) \vdash S$ **sat** $(G, Q);\ \Pi = \{(\sigma, \sigma') \in \Sigma \times \Sigma \mid [\![P]\!](\sigma) \wedge (S, \sigma) \xrightarrow[R]{r}{}^{*} (\textbf{nil}, \sigma')\}$

1 **from** $(\sigma_0, \sigma_f) \in \Pi$

1.1 $S \in \mathit{Stmt}$ — h

1.2 **from** $S \in \mathit{Assign}$
infer $\mathit{Within}_m(R, (S, \sigma_0), (\textbf{nil}, \sigma_f), G) \wedge [\![Q]\!](\sigma_0, \sigma_f)$
h, h1, h1.2, *Assign-Within*, *Assign-Post*

1.3 **from** $S \in \mathit{Atomic}$
infer $\mathit{Within}_m(R, (S, \sigma_0), (\textbf{nil}, \sigma_f), G) \wedge [\![Q]\!](\sigma_0, \sigma_f)$
h, h1, h1.3, *Atomic-Within*, *Atomic-Post*

1.4 **from** $S \in \mathit{If}$
infer $\mathit{Within}_m(R, (S, \sigma_0), (\textbf{nil}, \sigma_f), G) \wedge [\![Q]\!](\sigma_0, \sigma_f)$
h, h1, h1.4, *If-Within*, *If-Post*

1.5 **from** $S \in \mathit{Par}$
infer $\mathit{Within}_m(R, (S, \sigma_0), (\textbf{nil}, \sigma_f), G) \wedge [\![Q]\!](\sigma_0, \sigma_f)$
h, h1, h1.5, *Par-Within*, *Par-Post*

1.6 **from** $S \in \mathit{Seq}$
infer $\mathit{Within}_m(R, (S, \sigma_0), (\textbf{nil}, \sigma_f), G) \wedge [\![Q]\!](\sigma_0, \sigma_f)$
h, h1, h1.6, *Seq-Within*, *Seq-Post*

1.7 **from** $S \in \mathit{While}$
infer $\mathit{Within}_m(R, (S, \sigma_0), (\textbf{nil}, \sigma_f), G) \wedge [\![Q]\!](\sigma_0, \sigma_f)$
h, h1, h1.7, *While-Within*, *While-Post*

infer $\mathit{Within}_m(R, (S, \sigma_0), (\textbf{nil}, \sigma_f), G) \wedge [\![Q]\!](\sigma_0, \sigma_f)$ — $\vee$-E(1.1–1.7)

2 $\forall (\sigma_0, \sigma_f) \in \Pi \cdot \mathit{Within}_m(R, (S, \sigma_0), (\textbf{nil}, \sigma_f), G)$ — $\forall$-I(1)

3 $\mathit{Within}_s(P, R, S, G)$ — h, 2, *Within-Abstract*

4 $\forall (\sigma_0, \sigma_f) \in \Pi \cdot [\![Q]\!](\sigma_0, \sigma_f)$ — $\forall$-I(1)

infer $\mathit{Within}_s(P, R, S, G) \wedge \forall (\sigma_0, \sigma_f) \in \Pi \cdot [\![Q]\!](\sigma_0, \sigma_f)$ — $\wedge$-I(3,4)

D.2 Guarantee Condition

D.2.1 Assignment

$\boxed{\textit{Assign-Within}}$

	from $(P, R) \vdash mk\text{-}Assign(id, e)$ **sat** (G, Q); $[\![P]\!](\sigma_0)$; $(mk\text{-}Assign(id, e), \sigma_0) \xrightarrow[R]{r}{*} (\textbf{nil}, \sigma_f)$	
1	$R \Rightarrow I_{Vars(e)\cup\{id\}}$	h, *Assign-I*
2	$G = \{(\sigma, \sigma \dagger \{id \mapsto [\![e]\!](\sigma)\}) \mid \sigma \in \Sigma\} \cup I$	h, *Assign-I*
3	$T^i = \{(C_1, C_2) \mid (mk\text{-}Assign(id, e), \sigma_0) \xrightarrow[R]{r}{*}\ C_1 \xrightarrow[R]{r} C_2 \xrightarrow[R]{r}{*} (\textbf{nil}, \sigma_f)\}$	definition
4	**from** $(C_1, C_2) \in T^i$	
4.1	$(C_1, C_2) \in (\text{A-R-Step} \cup \text{Assign-Eval} \cup \text{Assign-E})$	h4, inspection of $\xrightarrow[R]{r}$
4.2	**from** $(C_1, C_2) \in \text{A-R-Step}$	
	infer $Within_1(C_1, C_2, G)$	h4.2, *Within-Rely*
4.3	**from** $(C_1, C_2) \in \text{Assign-Eval}$	
	infer $Within_1(C_1, C_2, G)$	h4.3, *Within-Prog-Cor*
4.4	**from** $(C_1, C_2) \in \text{Assign-E}$; $C_1 = (S_1, \sigma_1)$; $C_2 = (S_2, \sigma_2)$	
4.4.1	$S_1 \in Assign$	h4.4, Assign-E
4.4.2	$S_1.id = id$	h4, 4.4.1, inspection of $\xrightarrow[R]{r}$
4.4.3	$S_1.e \in \mathbb{Z}$	h4.4, 4.4.1, Assign-E
4.4.4	$\sigma_2 = \sigma_1 \dagger \{id \mapsto S_1.e\}$	h4.4, Assign-E
4.4.5	$S_1.e = [\![e]\!](\sigma_0) = [\![e]\!](\sigma_1)$	1, h4, 4.4.1, 4.4.2, 4.4.3, *Single-Eval-Assign*
4.4.6	$[\![G]\!](\sigma_1, \sigma_2)$	2, 4.4.4, 4.4.5
	infer $Within_1(C_1, C_2, G)$	h4.4, 4.4.6, *Within-Prog*
	infer $Within_1(C_1, C_2, G)$	$\vee$-E(4.1–4.4)
5	$\forall (C_1, C_2) \in T^i \cdot Within_1(C_1, C_2, G)$	$\forall$-I(4)
	infer $Within_m(R, (mk\text{-}Assign(id, e), \sigma_0), (\textbf{nil}, \sigma_f), G)$	3, 5, *Within-Multi*

D.2.2 Atomic/STM

$\boxed{\textit{Atomic-Within}}$

from $(P, R) \vdash \textit{mk-Atomic}(\textit{body})$ **psat** (G, Q); $[\![P]\!](\sigma_0)$; $C_0 = (\textit{mk-Atomic}(\textit{body}), \sigma_0)$;
$C_f = (\mathbf{nil}, \sigma_f)$; $C_0 \xrightarrow[R]{r}\!\!* \; C_f$; $\textit{IH-S}(\textit{body})$

1 $(P, I) \vdash \textit{body}$ **sat** $(\mathbf{true}, Q')$ — h, *Atomic-psat-I*
2 $Q' \Rightarrow G$ — h, *Atomic-psat-I*
3 $T^i = \{(C_1, C_2) \mid C_0 \xrightarrow[R]{r}\!\!* \; C_1 \xrightarrow[R]{r} C_2 \xrightarrow[R]{r}\!\!* \; C_f\}$ — definition
4 **from** $(C_1, C_2) \in T^i$
4.1 $(C_1, C_2) \in (\text{A-R-Step} \cup \text{STM-Atomic} \cup \text{STM-Step} \cup \text{STM-Retry} \cup \text{STM-E})$ — h4, inspection of $\xrightarrow[R]{r}$
4.2 **from** $(C_1, C_2) \in$ A-R-Step
infer $\textit{Within}_1(C_1, C_2, G)$ — h4.2, *Within-Rely*
4.3 **from** $(C_1, C_2) \in (\text{STM-Atomic} \cup \text{STM-Step} \cup \text{STM-Retry})$
infer $\textit{Within}_1(C_1, C_2, G)$ — h4.3, *Within-Prog-Cor*
4.4 **from** $(C_1, C_2) \in$ STM-E; $C_1 = (S_1, \sigma_1)$; $C_2 = (S_2, \sigma_2)$
4.4.1 $S_1 \in \textit{STM} \land S_1.\textit{body} = \mathbf{nil} \land S_2 = \mathbf{nil}$ — h4.4, STM-E
4.4.2 $\exists C_a \in \textit{Config}, \sigma_a, \sigma_s \in \Sigma \cdot$ — h4, h4.4, inspection of $\xrightarrow[R]{r}$
$C_0 \xrightarrow[R]{r}\!\!* \; C_a \xrightarrow[R]{r}\!\!* \; C_1 \xrightarrow[R]{r} C_2$
$\land\ C_a = (\textit{mk-STM}(\textit{body}, \sigma_a, \textit{body}, \sigma_a), \sigma_a)$
$\land\ S_1 = (\textit{mk-STM}(\textit{body}, \sigma_a, \mathbf{nil}, \sigma_s), \sigma_1)$
$\land\ (C_a, C_1) \in (\text{A-R-Step} \cup \text{STM-Step})^*$
4.4.3 **from** $C_a \in \textit{Config}, \sigma_a, \sigma_s \in \Sigma$ **st**
$C_0 \xrightarrow[R]{r}\!\!* \; C_a \xrightarrow[R]{r}\!\!* \; C_1 \xrightarrow[R]{r} C_2$
$\land\ C_a = (\textit{mk-STM}(\textit{body}, \sigma_a, \textit{body}, \sigma_a), \sigma_a)$
$\land\ S_1 = (\textit{mk-STM}(\textit{body}, \sigma_a, \mathbf{nil}, \sigma_s), \sigma_1)$
$\land\ (C_a, C_1) \in (\text{A-R-Step} \cup \text{STM-Step})^*$
4.4.3.1 $(C_0, C_a) \in (\text{A-R-Step} \cup \text{STM-Atomic} \cup \text{STM-Step} \cup \text{STM-Retry})$ — h4.4.3, inspection of $\xrightarrow[R]{r}$
4.4.3.2 $[\![R]\!](\sigma_0, \sigma_a)$ — 4.4.3.1, *Rely-Trivial*
4.4.3.3 $[\![R]\!](\sigma_a, \sigma_1)$ — h4.4.3, *Rely-Trivial*
4.4.3.4 $[\![P]\!](\sigma_1)$ — h, 4.4.3.2, 4.4.3.3, *PR-ident*
4.4.3.5 $(\textit{body}, \sigma_a) \xrightarrow[I]{r} (\mathbf{nil}, \sigma_s)$ — h4.4.3, *Isolation-STM*
4.4.3.6 $\sigma_s = \sigma_a \dagger (\textit{Vars}(\textit{body}) \lhd \sigma_s)$ — 4.4.3.5, *Sequential-Effect*
4.4.3.7 $(\textit{body}, \sigma_a) \xrightarrow[I]{r} (\mathbf{nil}, \sigma_a \dagger (\textit{Vars}(\textit{body}) \lhd \sigma_s))$ — 4.4.3.5, 4.4.3.6
4.4.3.8 $[\![I_{\textit{Vars}(\textit{body})}]\!](\sigma_a, \sigma_1)$ — h4.4.3, STM-E
4.4.3.9 $(\textit{Vars}(\textit{body}) \lhd \sigma_a) = (\textit{Vars}(\textit{body}) \lhd \sigma_1)$ — 4.4.3.8
4.4.3.10 $(\textit{body}, \sigma_1) \xrightarrow[I]{r} (\mathbf{nil}, \sigma_1 \dagger (\textit{Vars}(\textit{body}) \lhd \sigma_s))$ — 4.4.3.7, 4.4.3.9, *Frame-Rule*
4.4.3.11 $[\![Q']\!](\sigma_1, \sigma_1 \dagger (\textit{Vars}(\textit{body}) \lhd \sigma_s))$ — h, 1, 4.4.3.4, 4.4.3.10, IH-S(body)
4.4.3.12 $\sigma_2 = \sigma_1 \dagger (\textit{Vars}(\textit{body}) \lhd \sigma_s)$ — h4.4, h4.4.3, STM-E
4.4.3.13 $[\![Q']\!](\sigma_1, \sigma_2)$ — 4.4.3.11, 4.4.3.12
4.4.3.14 $[\![G]\!](\sigma_1, \sigma_2)$ — 2, 4.4.3.13
infer $\textit{Within}_1(C_1, C_2, G)$ — h4.4, h4.4.3, 4.4.3.14, *Within-Prog*
infer $\textit{Within}_1(C_1, C_2, G)$ — $\exists$-E(4.4.2,4.4.3)
infer $\textit{Within}_1(C_1, C_2, G)$ — $\vee$-E(4.1–4.4)
5 $\forall (C_1, C_2) \in T^i \cdot \textit{Within}_1(C_0, C_f, G)$ — $\forall$-I(4)
infer $\textit{Within}_m(R, C_0, C_f, G)$ — 3, 5, *Within-Multi*

D.2.3 If

$\boxed{\mathit{If\text{-}Within}}$

	from $(P, R) \vdash \mathit{mk\text{-}If}(b, \mathit{body})$ **sat** (G, Q); $[\![P]\!](\sigma_0)$; $C_0 = (\mathit{mk\text{-}If}(b, \mathit{body}), \sigma_0)$; $C_1 = (\mathbf{nil}, \sigma_f)$; $C_0 \xrightarrow[R]{r}{*}\, C_f$; $\mathit{IH\text{-}S}(\mathit{body})$	
1	$(P, R) \vdash \mathit{body}$ **sat** (G, Q)	h, *If-I*
2	$C^b = \{(S, \sigma_b) \mid S \in \{\mathit{body}, \mathbf{nil}\} \land C_0 \xrightarrow[R]{r}{*}\, (S, \sigma_b) \xrightarrow[R]{r}{*}\, C_f\}$	definition
3	**from** $C_b \in C^b$; $C_b = (S_b, \sigma_b)$	
3.1	$\quad T^i = \{(C_1, C_2) \mid C_0 \xrightarrow[R]{r}{*}\, C_1 \xrightarrow[R]{r} C_2 \xrightarrow[R]{r}{*}\, C_b\}$	definition
3.2	$\quad$ **from** $(C_1, C_2) \in T^i$	
3.2.1	$\qquad (C_1, C_2) \in (\mathsf{A\text{-}R\text{-}Step} \cup \mathsf{If\text{-}Eval} \cup \mathsf{If\text{-}T\text{-}E} \cup \mathsf{If\text{-}F\text{-}E})$	h3, inspection of $\xrightarrow[R]{r}$
3.2.2	$\qquad$ **from** $(C_1, C_2) \in \mathsf{A\text{-}R\text{-}Step}$	
	$\qquad$ **infer** $\mathit{Within}_1(C_1, C_2, G)$	h3.2.2, *Within-Rely*
3.2.3	$\qquad$ **from** $(C_1, C_2) \in (\mathsf{If\text{-}Eval} \cup \mathsf{If\text{-}T\text{-}E} \cup \mathsf{If\text{-}F\text{-}E})$	
	$\qquad$ **infer** $\mathit{Within}_1(C_1, C_2, G)$	h3.2.3, *Within-Prog-Cor*
	$\quad$ **infer** $\mathit{Within}_1(C_1, C_2, G)$	$\lor$-E(3.2.1–3.2.3)
3.3	$\quad \forall (C_1, C_2) \in T^i \cdot \mathit{Within}_1(C_1, C_2, G)$	$\forall$-I(3.2)
3.4	$\quad \mathit{Within}_m(R, C_0, C_b, G)$	3.1, 3.3, *Within-Multi*
3.5	$\quad$ **from** $S_b = \mathit{body}$	
3.5.1	$\qquad (C_0, C_b) \in (\mathsf{A\text{-}R\text{-}Step} \cup \mathsf{If\text{-}Eval} \cup \mathsf{If\text{-}T\text{-}E})^*$	h3, inspection of $\xrightarrow[R]{r}$
3.5.2	$\qquad [\![R]\!](\sigma_0, \sigma_b)$	h3, h3.5, 3.5.1, *Rely-Trivial*
3.5.3	$\qquad [\![P]\!](\sigma_b)$	h, 3.5.2, *PR-ident*
	$\quad$ **infer** $\mathit{Within}_m(R, C_b, C_f, G)$	h, 1, h3, h3.5, 3.5.3, IH-S(body)
3.6	$\quad$ **from** $S_b = \mathbf{nil}$	
	$\quad$ **infer** $\mathit{Within}_m(R, C_b, C_f, G)$	h3, h3.6 *Within-Rely-Trivial*
3.7	$\quad \mathit{Within}_m(R, C_b, C_f, G)$	$\lor$-E(h3,3.5,3.6)
	infer $\mathit{Within}_m(R, C_0, C_b, G) \land \mathit{Within}_m(R, C_b, C_f, G)$	$\land$-I(3.4,3.7)
4	$\forall C_b \in C^b \cdot \mathit{Within}_m(R, C_0, C_b, G) \land \mathit{Within}_m(R, C_b, C_f, G)$	$\forall$-I(3)
	infer $\mathit{Within}_m(R, C_0, C_f, G)$	2, 4, *Within-Concat*

D.2.4 Sequence

$\boxed{\textit{Seq-Within}}$

	from $(P, R) \vdash \textit{mk-Seq}(\textit{left}, \textit{right})$ **sat** (G, Q); $[\![P]\!](\sigma_0)$; $C_0 = (\textit{mk-Seq}(\textit{left}, \textit{right}), \sigma_0)$; $C_f = (\textbf{nil}, \sigma_f)$; $C_0 \xrightarrow[R]{r}* C_f$; $\textit{IH-S}(\textit{left})$; $\textit{IH-S}(\textit{right})$	
1	$(P, R) \vdash \textit{left}$ **sat** $(G, Q_l \wedge P_r)$	h, *Seq-I*
2	$(P_r, R) \vdash \textit{right}$ **sat** (G, Q_r)	h, *Seq-I*
3	$C^r = \{(\textit{right}, \sigma_r) \mid C_0 \xrightarrow[R]{r}* (\textit{right}, \sigma_r) \xrightarrow[R]{r}* C_f\}$	definition
4	**from** $C_r \in C^r$; $C_r = (\textit{right}, \sigma_r)$	
4.1	$T^i = \{(C_1, C_2) \mid C_0 \xrightarrow[R]{r}* C_1 \xrightarrow[R]{r} C_2 \xrightarrow[R]{r}* C_r\}$	definition
4.2	**from** $(C_1, C_2) \in T^i$	
4.2.1	$(C_1, C_2) \in (\text{A-R-Step} \cup \text{Seq-E} \cup \text{Seq-Step})$	h4.2, inspection of $\xrightarrow[R]{r}*$
4.2.2	**from** $(C_1, C_2) \in \text{A-R-Step}$	
	infer $\textit{Within}_1(C_1, C_2, G)$	h4.2.2, *Within-Rely*
4.2.3	**from** $(C_1, C_2) \in \text{Seq-E}$	
	infer $\textit{Within}_1(C_1, C_2, G)$	h4.2.3, *Within-Prog-Cor*
4.2.4	**from** $(C_1, C_2) \in \text{Seq-Step}$; $C_1 = (S_1, \sigma_1)$; $C_2 = (S_2, \sigma_2)$	
4.2.4.1	$(\textit{left}, \sigma_0) \xrightarrow[R]{r}* (S_1.\textit{left}, \sigma_1) \xrightarrow[R]{r} (S_2.\textit{left}, \sigma_2)$	h4.2, h4.2.4, *Seq-Equiv*
4.2.4.2	$\textit{Within}_1((S_1.\textit{left}, \sigma_1), (S_2.\textit{left}, \sigma_2), G)$	h, 1, 4.2.4.1, IH-S(left)
	infer $\textit{Within}_1(C_1, C_2, G)$	h4.2.4, 4.2.4.2, *Within-Equiv*
	infer $\textit{Within}_1(C_1, C_2, G)$	$\vee$-E(4.2.1–4.2.4)
4.3	$\forall (C_1, C_2) \in T^i \cdot \textit{Within}_1(C_1, C_2, G)$	$\forall$-I(4.2)
4.4	$\textit{Within}_m(R, C_0, C_r, G)$	4.1, 4.3, *Within-Multi*
4.5	$\exists \sigma_a \in \Sigma \cdot C_0 \xrightarrow[R]{r}* (\textit{mk-Seq}(\textbf{nil}, \textit{right}), \sigma_a) \xrightarrow[R]{r}* C_r \xrightarrow[R]{r}* C_f$	h4, inspection of $\xrightarrow[R]{r}$
4.6	**from** $\sigma_a \in \Sigma$ **st** $C_0 \xrightarrow[R]{r}* (\textit{mk-Seq}(\textbf{nil}, \textit{right}), \sigma_a) \xrightarrow[R]{r}* C_r \xrightarrow[R]{r}* C_f$	
4.6.1	$(\textit{left}, \sigma_0) \xrightarrow[R]{r}* (\textbf{nil}, \sigma_a)$	h4.6, *Seq-Equiv*
4.6.2	$[\![Q_l \wedge P_r]\!](\sigma_0, \sigma_a)$	h, 1, 4.6.1, IH-S(left)
4.6.3	$((\textit{mk-Seq}(\textbf{nil}, \textit{right}), \sigma_a), C_r) \in (\text{A-R-Step} \cup \text{Seq-E})^*$	h4.6, inspection of $\xrightarrow[R]{r}$
4.6.4	$[\![R]\!](\sigma_a, \sigma_r)$	h4.6, 4.6.3, *Rely-Trivial*
	infer $[\![P_r]\!](\sigma_r)$	4.6.2, 4.6.4, *PR-ident*
4.7	$[\![P_r]\!](\sigma_r)$	$\exists$-E(4.5, 4.6)
4.8	$\textit{Within}_m(R, C_r, C_f, G)$	2, h4, 4.7, IH-S(right)
	infer $\textit{Within}_m(R, C_0, C_r, G) \wedge \textit{Within}_m(R, C_r, C_f, G)$	$\wedge$-I(4.4,4.8)
5	$\forall C_r \in C^r \cdot \textit{Within}_m(R, C_0, C_r, G) \wedge \textit{Within}_m(R, C_r, C_f, G)$	$\forall$-I(4)
	infer $\textit{Within}_m(R, C_0, C_f, G)$	3, 5, *Within-Concat*

D.2.5 Parallel

$\boxed{\textit{Par-Within}}$

from $(P, R) \vdash \textit{mk-Par}(\textit{left}, \textit{right})\ \mathbf{sat}\ (G, Q)$; $[\![P]\!](\sigma_0)$; $C_0 = (\textit{mk-Par}(\textit{left}, \textit{right}), \sigma_0)$;
$C_f = (\mathbf{nil}, \sigma_f)$; $C_0 \xrightarrow[R]{r}{*}\ C_f$; $\textit{IH-S}(\textit{left})$; $\textit{IH-S}(\textit{right})$

1 $(P, R \vee G_r) \vdash \textit{left}\ \mathbf{sat}\ (G_l, Q_l)$ — h, *Par-I*

2 $(P, R \vee G_l) \vdash \textit{right}\ \mathbf{sat}\ (G_r, Q_r)$ — h, *Par-I*

3 $G_l \vee G_r \ \Rightarrow\ G$ — h, *Par-I*

4 $T^i = \{(C_1, C_2) \mid C_0 \xrightarrow[R]{r}{*}\ C_1 \xrightarrow[R]{r} C_2 \xrightarrow[R]{r}{*}\ C_f\}$ — definition

5 **from** $(C_1, C_2) \in T^i$

5.1 $(C_1, C_2) \in (\text{A-R-Step} \cup \text{Par-L} \cup \text{Par-R} \cup \text{Par-E})$ — h5, inspection of $\xrightarrow[R]{r}{*}$

5.2 **from** $(C_1, C_2) \in \text{A-R-Step}$
infer $\textit{Within}_1(C_1, C_2, G)$ — h5.2, *Within-Rely*

5.3 **from** $(C_1, C_2) \in \text{Par-E}$
infer $\textit{Within}_1(C_1, C_2, G)$ — h5.3, *Within-Prog-Cor*

5.4 **from** $(C_1, C_2) \in \text{Par-L}$; $C_1 = (S_1, \sigma_1)$; $C_2 = (S_2, \sigma_2)$

5.4.1 $(\textit{left}, \sigma_0) \xrightarrow[R \vee G_r]{r}{*}\ (S_1.\textit{left}, \sigma_1) \xrightarrow[R \vee G_r]{r} (S_2.\textit{left}, \sigma_2)$ — h, 1, 2, h5, *Isolation-Par-L*

5.4.2 $\textit{Within}_1((S_1.\textit{left}, \sigma_1), (S_2.\textit{left}, \sigma_2), G_l)$ — h, 1, 5.4.1, IH-S(left)

5.4.3 $\textit{Within}_1((S_1.\textit{left}, \sigma_1), (S_2.\textit{left}, \sigma_2), G)$ — 3, 5.4.2, *Within-Weaken*

infer $\textit{Within}_1(C_1, C_2, G)$ — h5.4, 5.4.3, *Within-Equiv*

5.5 **from** $(C_1, C_2) \in \text{Par-R}$
infer $\textit{Within}_1(C_1, C_2, G)$ — symmetric to 5.4

infer $\textit{Within}_1(C_1, C_2, G)$ — $\vee$-E(5.1–5.5)

6 $\forall (C_1, C_2) \in T^i \cdot \textit{Within}_1(C_1, C_2, G)$ — $\forall$-I(5)

infer $\textit{Within}_m(R, C_0, C_f, G)$ — 4, 6, *Within-Multi*

D.2.6 While

$\boxed{\textit{While-Within}}$

from $(P, R) \vdash \textit{mk-While}(b, \textit{body})$ **sat** (G, Q); $[\![P]\!](\sigma_0)$; $C_0 = (\textit{mk-While}(b, \textit{body}), \sigma_0)$;
  $C_f = (\textbf{nil}, \sigma_f)$; $C_0 \xrightarrow[R]{r}{}^{*} C_f$; $\textit{IH-S}(\textit{body})$

1 $(P \wedge b_s, R) \vdash \textit{body}$ **sat** $(G, W \wedge P)$ — h, *While-I*

2 $T^i = \{(C_1, C_2) \mid C_0 \xrightarrow[R]{r}{}^{*} C_1 \xrightarrow[R]{r} C_2 \xrightarrow[R]{r}{}^{*} C_f\}$ — definition

3 **from** $(C_1, C_2) \in T^i$

3.1 $(C_1, C_2) \in (\text{A-R-Step} \cup \text{While} \cup \text{If-Eval} \cup \text{If-T-E} \cup \text{If-F-E} \cup \text{Seq-E} \cup \text{Seq-Step})$ — h3, inspection of $\xrightarrow[R]{r}$

3.2 **from** $(C_1, C_2) \in \text{A-R-Step}$
  infer $\textit{Within}_1(C_1, C_2, G)$ — h3.2, *Within-Rely*

3.3 **from** $(C_1, C_2) \in (\text{While} \cup \text{If-Eval} \cup \text{If-T-E} \cup \text{If-F-E} \cup \text{Seq-E})$
  infer $\textit{Within}_1(C_1, C_2, G)$ — h3.3, *Within-Prog-Cor*

3.4 **from** $(C_1, C_2) \in \text{Seq-Step}$; $C_1 = (S_1, \sigma_1)$; $C_2 = (S_2, \sigma_2)$

3.4.1 $\exists C_a \in \textit{Config}, \sigma_a \in \Sigma \cdot$ — h3, h3.4, inspection of $\xrightarrow[R]{r}$
  $C_a = (\textit{mk-Seq}(\textit{body}, \textit{mk-While}(b, \textit{body})), \sigma_a)$
  $\wedge\ C_0 \xrightarrow[R]{r}{}^{*} C_a \xrightarrow[R]{r}{}^{*} C_1 \xrightarrow[R]{r} C_2$
  $\wedge\ (C_a, C_1) \in (\text{A-R-Step} \cup \text{Seq-Step})^*$

3.4.2 **from** $C_a \in \textit{Config}, \sigma_a \in \Sigma$ **st**
  $C_a = (\textit{mk-Seq}(\textit{body}, \textit{mk-While}(b, \textit{body})), \sigma_a)$
  $\wedge\ C_0 \xrightarrow[R]{r}{}^{*} C_a \xrightarrow[R]{r}{}^{*} C_1 \xrightarrow[R]{r} C_2$
  $\wedge\ (C_a, C_1) \in (\text{A-R-Step} \cup \text{Seq-Step})^*$

3.4.2.1 $[\![P \wedge b_s]\!](\sigma_a)$ — h, 1, h3.4.2, *While-interstices-pre*

3.4.2.2 $(\textit{body}, \sigma_a) \xrightarrow[R]{r}{}^{*} (S_1.\textit{left}, \sigma_1) \xrightarrow[R]{r} (S_2.\textit{left}, \sigma_2)$ — h3.4, h3.4.2, *Seq-Equiv*

3.4.2.3 $\textit{Within}_1((S_1.\textit{left}, \sigma_1), (S_2.\textit{left}, \sigma_2), G)$ — h, 1, 3.4.2.1, 3.4.2.2, IH-S(body)

  infer $\textit{Within}_1(C_1, C_2, G)$ — 3.4.2.3, *Within-Equiv*

  infer $\textit{Within}_1(C_1, C_2, G)$ — $\exists$-E(3.4.1,3.4.2)

 infer $\textit{Within}_1(C_1, C_2, G)$ — $\vee$-E(3.1–3.4)

4 $\forall (C_1, C_2) \in T^i \cdot \textit{Within}_1(C_1, C_2, G)$ — $\forall$-I(3)

infer $\textit{Within}_m(R, C_0, C_f, G)$ — 2, 4, *Within-Multi*

D.3 Post Condition

D.3.1 Assignment

$\boxed{\textit{Assign-Post}}$

from $(P, R) \vdash \textit{mk-Assign}(id, e)$ **sat** (G, Q); $[\![P]\!](\sigma_0)$; $C_0 = (\textit{mk-Assign}(id, e), \sigma_0)$;
$C_f = (\mathbf{nil}, \sigma_f)$; $C_0 \xrightarrow[R]{r}\!\!* \ C_f$

1 $R \Rightarrow I_{Vars(e) \cup \{id\}}$ — h, *Assign-I*

2 $Q = \{(\sigma, \sigma') \mid \sigma, \sigma' \in \Sigma \wedge \sigma'(id) = [\![e]\!](\sigma)\}$ — h, *Assign-I*

3 $\exists v \in \mathbb{Z}, \sigma_1, \sigma_2 \in \Sigma \cdot$ — h, inspection of $\xrightarrow[R]{r}$
$C_0 \xrightarrow[R]{r}\!\!* \ (\textit{mk-Assign}(id, v), \sigma_1) \xrightarrow[R]{r} (\mathbf{nil}, \sigma_2) \xrightarrow[R]{r}\!\!* \ C_f$
$\wedge ((\textit{mk-Assign}(id, v), \sigma_1), (\mathbf{nil}, \sigma_2)) \in$ Assign-E

4 **from** $v \in \mathbb{Z}, \sigma_1, \sigma_2 \in \Sigma$ **st**
$C_0 \xrightarrow[R]{r}\!\!* \ (\textit{mk-Assign}(id, v), \sigma_1) \xrightarrow[R]{r} (\mathbf{nil}, \sigma_2) \xrightarrow[R]{r}\!\!* \ C_f$
$\wedge ((\textit{mk-Assign}(id, v), \sigma_1), (\mathbf{nil}, \sigma_2)) \in$ Assign-E

4.1 $v = [\![e]\!](\sigma_0) = [\![e]\!](\sigma_1)$ — 1, h4, *Single-Eval-Assign*

4.2 $\sigma_2 = \sigma_1 \dagger \{id \mapsto v\}$ — h4, Assign-E

4.3 $\sigma_2(id) = [\![e]\!](\sigma_0)$ — 4.1, 4.2

4.4 $[\![Q]\!](\sigma_0, \sigma_2)$ — 2, 4.3

4.5 $((\mathbf{nil}, \sigma_2), C_f) \in$ A-R-Step — h4, inspection of $\xrightarrow[R]{r}$

4.6 $[\![R]\!](\sigma_2, \sigma_f)$ — 4.5, *Rely-Trivial*

infer $[\![Q]\!](\sigma_0, \sigma_f)$ — 4.4, 4.6, *QR-ident*

infer $[\![Q]\!](\sigma_0, \sigma_f)$ — $\exists$-E(3, 4)

D.3.2 Atomic/STM

$\boxed{\textit{Atomic-Post}}$

	from $(P, R) \vdash \textit{mk-Atomic}(\textit{body})$ **psat** (G, Q); $\llbracket P \rrbracket(\sigma_0)$; $C_0 = (\textit{mk-Atomic}(\textit{body}), \sigma_0)$;	
	$C_f = (\mathbf{nil}, \sigma_f)$; $C_0 \xrightarrow[R]{r}\!* \; C_f$; $\textit{IH-S}(\textit{body})$	
1	$(P, I) \vdash \textit{body}$ **psat** $(\mathbf{true}, Q')$	h, *Atomic-psat-I*
2	$\overleftharpoon{P} \wedge R \diamond Q' \diamond R \;\Rightarrow\; Q$	h, *Atomic-psat-I*
3	$\exists C_1, C_2, C_3 \in \textit{Config}, \sigma_1, \sigma_2, \sigma_3, \sigma_s \in \Sigma \cdot$	h, inspection of $\xrightarrow[R]{r}$
	$C_1 = (\textit{mk-STM}(\textit{body}, \sigma_1, \textit{body}, \sigma_1), \sigma_1)$	
	$\wedge\; C_2 = (\textit{mk-STM}(\textit{body}, \sigma_1, \mathbf{nil}, \sigma_s), \sigma_2)$	
	$\wedge\; C_3 = (\mathbf{nil}, \sigma_3)$	
	$\wedge\; C_0 \xrightarrow[R]{r}\!* \; C_1 \xrightarrow[R]{r}\!* \; C_2 \xrightarrow[R]{r} C_3 \xrightarrow[R]{r}\!* \; C_f$	
	$\wedge\; (C_0, C_1) \in (\textsf{A-R-Step} \cup \textsf{STM-Atomic} \cup \textsf{STM-Step} \cup \textsf{STM-Retry})^*$	
	$\wedge\; (C_1, C_2) \in (\textsf{A-R-Step} \cup \textsf{STM-Step})^*$	
	$\wedge\; (C_2, C_3) \in \textsf{STM-E}$	
	$\wedge\; (C_3, C_f) \in \textsf{A-R-Step}^*$	
4	**from** $C_1, C_2, C_3 \in \textit{Config}, \sigma_1, \sigma_2, \sigma_3, \sigma_s \in \Sigma$ **st**	
	$C_1 = (\textit{mk-STM}(\textit{body}, \sigma_1, \textit{body}, \sigma_1), \sigma_1)$	
	$\wedge\; C_2 = (\textit{mk-STM}(\textit{body}, \sigma_1, \mathbf{nil}, \sigma_s), \sigma_2)$	
	$\wedge\; C_3 = (\mathbf{nil}, \sigma_3)$	
	$\wedge\; C_0 \xrightarrow[R]{r}\!* \; C_1 \xrightarrow[R]{r}\!* \; C_2 \xrightarrow[R]{r} C_3 \xrightarrow[R]{r}\!* \; C_f$	
	$\wedge\; (C_0, C_1) \in (\textsf{A-R-Step} \cup \textsf{STM-Atomic} \cup \textsf{STM-Step} \cup \textsf{STM-Retry})^*$	
	$\wedge\; (C_1, C_2) \in (\textsf{A-R-Step} \cup \textsf{STM-Step})^*$	
	$\wedge\; (C_2, C_3) \in \textsf{STM-E}$	
	$\wedge\; (C_3, C_f) \in \textsf{A-R-Step}^*$	
4.1	$\llbracket R \rrbracket(\sigma_0, \sigma_1)$	h4, *Rely-Trivial*
4.2	$\llbracket R \rrbracket(\sigma_1, \sigma_2)$	h4, *Rely-Trivial*
4.3	$\llbracket P \rrbracket(\sigma_2)$	h, 4.1, 4.2, *PR-ident*
4.4	$(\textit{body}, \sigma_1) \xrightarrow[I]{r}\!* \; (\mathbf{nil}, \sigma_s)$	h4, *Isolation-STM*
4.5	$\sigma_s = \sigma_1 \dagger (\textit{Vars}(\textit{body}) \lhd \sigma_s)$	4.4, *Sequential-Effect*
4.6	$(\textit{body}, \sigma_1) \xrightarrow[I]{r}\!* \; (\mathbf{nil}, \sigma_1 \dagger (\textit{Vars}(\textit{body}) \lhd \sigma_s))$	4.4, 4.5
4.7	$\llbracket I_{\textit{Vars}(\textit{body})} \rrbracket(\sigma_1, \sigma_2)$	h4, STM-Step
4.8	$(\textit{Vars}(\textit{body}) \lhd \sigma_1) = (\textit{Vars}(\textit{body}) \lhd \sigma_2)$	4.7
4.9	$(\textit{body}, \sigma_2) \xrightarrow[I]{r}\!* \; (\mathbf{nil}, \sigma_2 \dagger (\textit{Vars}(\textit{body}) \lhd \sigma_s))$	4.6, 4.8, *Frame-Rule*
4.10	$\llbracket Q' \rrbracket(\sigma_2, \sigma_2 \dagger (\textit{Vars}(\textit{body}) \lhd \sigma_s))$	h, 1, 4.3, 4.9, IH-S(body)
4.11	$\sigma_3 = \sigma_2 \dagger (\textit{Vars}(\textit{body}) \lhd \sigma_s)$	h4, STM-E
4.12	$\llbracket Q' \rrbracket(\sigma_2, \sigma_3)$	4.10, 4.11
4.13	$\llbracket R \rrbracket(\sigma_3, \sigma_f)$	h4, *Rely-Trivial*
4.14	$\llbracket R \diamond Q' \diamond R \rrbracket(\sigma_0, \sigma_f)$	4.1, 4.2, 4.12, 4.13
	infer $\llbracket Q \rrbracket(\sigma_0, \sigma_f)$	h, 2, 4.14
	infer $\llbracket Q \rrbracket(\sigma_0, \sigma_f)$	$\exists$-E(3,4)

D.3.3 If

Unconstrained body — *If-I*

$\boxed{\textit{If-Post}}$

from	$(P, R) \vdash \textit{mk-If}(b, \textit{body})\ \textbf{sat}\ (G, Q);\ [\![P]\!](\sigma_0);\ C_0 = (\textit{mk-If}(b, \textit{body}), \sigma_0);$ $C_f = (\textbf{nil}, \sigma_f);\ C_0 \xrightarrow[R]{r}\!\!* \ C_f;\ \textit{IH-S}(\textit{body})$	
1	$(P, R) \vdash \textit{body}\ \textbf{sat}\ (G, Q)$	h, *If-I*
2	$\overleftarrow{P} \wedge R \Rightarrow Q$	h, *If-I*
3	$\exists v \in \mathbb{B}, \sigma_1 \in \Sigma \cdot C_0 \xrightarrow[R]{r}\!\!* \ (\textit{mk-If}(v, \textit{body}), \sigma_1) \xrightarrow[R]{r}\!\!* \ C_f$	h, inspection of $\xrightarrow[R]{r}$
4	**from** $v \in \mathbb{B}, \sigma_1 \in \Sigma$ **st** $C_0 \xrightarrow[R]{r}\!\!* \ (\textit{mk-If}(v, \textit{body}), \sigma_1) \xrightarrow[R]{r}\!\!* \ C_f$	
4.1	$(C_0, (\textit{mk-If}(v, \textit{body}), \sigma_1)) \in (\textsf{A-R-Step} \cup \textsf{If-Eval})^*$	h4, inspection of $\xrightarrow[R]{r}$
4.2	$[\![R]\!](\sigma_0, \sigma_1)$	h4, 4.1, *Rely-Trivial*
4.3	**from** $\neg\, v$	
4.3.1	$((\textit{mk-If}(v, \textit{body}), \sigma_1), C_f) \in (\textsf{A-R-Step} \cup \textsf{If-F-E})^*$	h4, h4.3, inspection of $\xrightarrow[R]{r}$
4.3.2	$[\![R]\!](\sigma_1, \sigma_f)$	h4, h4.3, 4.3.1, *Rely-Trivial*
4.3.3	$[\![R]\!](\sigma_0, \sigma_f)$	4.2, 4.3.2
4.3.4	$[\![\overleftarrow{P} \wedge R]\!](\sigma_0, \sigma_f)$	h, 4.3.3
	infer $[\![Q]\!](\sigma_0, \sigma_f)$	2, 4.3.4
4.4	**from** v	
4.4.1	$(\textit{body}, \sigma_0) \xrightarrow[R]{r}\!\!* \ (\textbf{nil}, \sigma_f)$	h4, h4.4, *Isolation-If*
	infer $[\![Q]\!](\sigma_0, \sigma_f)$	h, 1, 4.4.1, IH-S(body)
	infer $[\![Q]\!](\sigma_0, \sigma_f)$	$\vee$-E(h4,4.3,4.4)
	infer $[\![Q]\!](\sigma_0, \sigma_f)$	$\exists$-E(3,4)

Constrained body — *If-b-I*

$\boxed{\textit{If-Post}}$

	from $(P, R) \vdash \textit{mk-If}(b, \textit{body})$ **sat** (G, Q); $[\![P]\!](\sigma_0)$; $C_0 = (\textit{mk-If}(b, \textit{body}), \sigma_0)$; $C_f = (\textbf{nil}, \sigma_f)$; $C_0 \xrightarrow[R]{r}{}^* C_f$; $\textit{IH-S}(\textit{body})$	
1	$(P \wedge b, R) \vdash \textit{body}$ **sat** (G, Q)	h, *If-b-I*
2	$R \Rightarrow I_{\textit{Vars}(b)}$	h, *If-b-I*
3	$\overleftarrow{P} \wedge \neg \overleftarrow{b} \wedge R \Rightarrow Q$	h, *If-b-I*
4	$\exists v \in \mathbb{B}, \sigma_1 \in \Sigma \cdot C_0 \xrightarrow[R]{r}{}^* (\textit{mk-If}(v, \textit{body}), \sigma_1) \xrightarrow[R]{r}{}^* C_f$	h, inspection of $\xrightarrow[R]{r}$
5	**from** $v \in \mathbb{B}, \sigma_1 \in \Sigma$ **st** $C_0 \xrightarrow[R]{r}{}^* (\textit{mk-If}(v, \textit{body}), \sigma_1) \xrightarrow[R]{r}{}^* C_f$	
5.1	$(C_0, (\textit{mk-If}(v, \textit{body}), \sigma_1)) \in (\textsf{A-R-Step} \cup \textsf{If-Eval})^*$	h4, inspection of $\xrightarrow[R]{r}$
5.2	$[\![R]\!](\sigma_0, \sigma_1)$	h4, 4.1, *Rely-Trivial*
5.3	$v = [\![b]\!](\sigma_0) = [\![b]\!](\sigma_1)$	2, h5, *Single-Eval-If*
5.4	**from** $\neg v$	
5.4.1	$((\textit{mk-If}(v, \textit{body}), \sigma_1), C_f) \in (\textsf{A-R-Step} \cup \textsf{If-F-E})^*$	h5, h5.4, inspection of $\xrightarrow[R]{r}$
5.4.2	$[\![R]\!](\sigma_1, \sigma_f)$	h5, h5.4, 5.4.1, *Rely-Trivial*
5.4.3	$[\![R]\!](\sigma_0, \sigma_f)$	5.2, 5.4.3
5.4.4	$[\![\neg b]\!](\sigma_0)$	5.3, h5.4
5.4.5	$[\![\overleftarrow{P} \wedge \neg \overleftarrow{b} \wedge R]\!](\sigma_0, \sigma_f)$	h, 5.4.3, 5.4.4
	infer $[\![Q]\!](\sigma_0, \sigma_f)$	3, 5.4.5
5.5	**from** v	
5.5.1	$[\![b]\!](\sigma_0)$	5.3, h5.5
5.5.2	$[\![P \wedge b]\!](\sigma_0)$	h, 5.5.1
5.5.3	$(\textit{body}, \sigma_0) \xrightarrow[R]{r}{}^* (\textbf{nil}, \sigma_f)$	h5, h5.5, *Isolation-If*
	infer $[\![Q]\!](\sigma_0, \sigma_f)$	h, 1, 5.5.2, 5.5.3, IH-S(body)
	infer $[\![Q]\!](\sigma_0, \sigma_f)$	$\vee$-E(h5,5.4,5.5)
	infer $[\![Q]\!](\sigma_0, \sigma_f)$	$\exists$-E(4,5)

D.3.4 Sequence

$\boxed{\textit{Seq-Post}}$

	from $(P, R) \vdash \textit{mk-Seq}(\textit{left}, \textit{right})$ **sat** (G, Q); $[\![P]\!](\sigma_0)$; $C_0 = (\textit{mk-Seq}(\textit{left}, \textit{right}), \sigma_0)$; $C_f = (\textbf{nil}, \sigma_f)$; $C_0 \xrightarrow[R]{r} * \; C_f$; $\textit{IH-S}(\textit{left})$; $\textit{IH-S}(\textit{right})$	
1	$(P, R) \vdash \textit{left}$ **sat** $(G, Q_l \wedge P_r)$	h, *Seq-I*
2	$(P_r, R) \vdash \textit{right}$ **sat** (G, Q_r)	h, *Seq-I*
3	$Q_l \diamond Q_r \;\Rightarrow\; Q$	h, *Seq-I*
4	$\exists \sigma_1 \in \Sigma \cdot C_0 \xrightarrow[R]{r} * \; (\textit{mk-Seq}(\textbf{nil}, \textit{right}), \sigma_1) \xrightarrow[R]{r} * \; C_f$	h, inspection of $\xrightarrow[R]{r}$
5	**from** $\sigma_1 \in \Sigma$ **st** $C_0 \xrightarrow[R]{r} * \; (\textit{mk-Seq}(\textbf{nil}, \textit{right}), \sigma_1) \xrightarrow[R]{r} * \; C_f$	
5.1	$(\textit{left}, \sigma_0) \xrightarrow[R]{r} * \; (\textbf{nil}, \sigma_1)$	h5, *Seq-Equiv*
5.2	$[\![Q_l \wedge P_r]\!](\sigma_0, \sigma_1)$	h, 1, 5.1, IH-S(left)
5.3	$(\textit{right}, \sigma_1) \xrightarrow[R]{r} * \; (\textbf{nil}, \sigma_f)$	h5, *Isolation-Seq-R*
5.4	$[\![Q_r]\!](\sigma_1, \sigma_f)$	h, 2, 5.2, 5.3, IH-S(right)
5.5	$[\![Q_l \diamond Q_r]\!](\sigma_0, \sigma_f)$	5.2, 5.4
	infer $[\![Q]\!](\sigma_0, \sigma_f)$	3, 5.5
	infer $[\![Q]\!](\sigma_0, \sigma_f)$	$\exists$-E(4,5)

D.3.5 Parallel

$\boxed{\textit{Par-Post}}$

	from $(P, R) \vdash \textit{mk-Par}(\textit{left}, \textit{right})$ **sat** (G, Q); $[\![P]\!](\sigma_0)$; $C_0 = (\textit{mk-Par}(\textit{left}, \textit{right}), \sigma_0)$; $C_f = (\mathbf{nil}, \sigma_f)$; $C_0 \xrightarrow[R]{r}* \ C_f$; $\textit{IH-S}(\textit{left})$; $\textit{IH-S}(\textit{right})$	
1	$(P, R \vee G_r) \vdash \textit{left}$ **sat** (G_l, Q_l)	h, *Par-I*
2	$(P, R \vee G_l) \vdash \textit{right}$ **sat** (G_r, Q_r)	h, *Par-I*
3	$\overleftarrow{P} \wedge Q_l \wedge Q_r \wedge (R \vee G_l \vee G_r)^* \ \Rightarrow \ Q$	h, *Par-I*
4	$\exists \sigma_1 \in \Sigma \cdot C_0 \xrightarrow[R]{r}* \ (\textit{mk-Par}(\mathbf{nil}, \mathbf{nil}), \sigma_1) \xrightarrow[R]{r}* \ C_f$	h, inspection of $\xrightarrow[R]{r}$
5	**from** $\sigma_1 \in \Sigma$ **st** $C_0 \xrightarrow[R]{r}* \ (\textit{mk-Par}(\mathbf{nil}, \mathbf{nil}), \sigma_1) \xrightarrow[R]{r}* \ C_f$	
5.1	$(\textit{left}, \sigma_0) \xrightarrow[R \vee G_r]{r}* \ (\mathbf{nil}, \sigma_1)$	h5, *Isolation-Par-L*
5.2	$[\![Q_l]\!](\sigma_0, \sigma_1)$	h, 1, h5, 5.1, IH-S(left)
5.3	$((\textit{mk-Par}(\mathbf{nil}, \mathbf{nil}), \sigma_1), C_f) \in (\textsf{A-R-Step} \cup \textsf{Par-E})^*$	h5, inspection of $\xrightarrow[R]{r}$
5.4	$[\![R]\!](\sigma_1, \sigma_f)$	h5, 5.3, *Rely-Trivial*
	infer $[\![Q_l]\!](\sigma_0, \sigma_f)$	5.2, 5.4, *QR-ident*
6	$[\![Q_l]\!](\sigma_0, \sigma_f)$	$\exists$-E(4,5)
7	$[\![Q_r]\!](\sigma_0, \sigma_f)$	symmetrical to 4–6
8	**from** $\sigma_1 \in \Sigma$ **st** $C_0 \xrightarrow[R]{r}* \ (\textit{mk-Par}(\mathbf{nil}, \mathbf{nil}), \sigma_1) \xrightarrow[R]{r}* \ C_f$	
8.1	$\textit{Within}_s(P, R \vee G_r, \textit{left}, G_l)$	h, 1, IH-S(left)
8.2	$(\textit{left}, \sigma_0) \xrightarrow[R \vee G_r]{r}* \ (\mathbf{nil}, \sigma_1)$	h8, *Isolation-Par-L*
8.3	$\textit{Within}_m(R \vee G_r, (\textit{left}, \sigma_0), (\mathbf{nil}, \sigma_1), G_l)$	h, 8.1, 8.2, *Within-Concrete*
8.4	$[\![(R \vee G_r \vee G_l)^*]\!](\sigma_0, \sigma_1)$	8.3, *Within-Relation*
8.5	$((\textit{mk-Par}(\mathbf{nil}, \mathbf{nil}), \sigma_1), C_f) \in (\textsf{A-R-Step} \cup \textsf{Par-E})^*$	h8, inspection of $\xrightarrow[R]{r}$
8.6	$[\![R]\!](\sigma_1, \sigma_f)$	h8, 8.5, *Rely-Trivial*
	infer $[\![(R \vee G_l \vee G_r)^*]\!](\sigma_0, \sigma_f)$	8.4, 8.6
9	$[\![(R \vee G_l \vee G_r)^*]\!](\sigma_0, \sigma_f)$	$\exists$-E(4,8)
	infer $[\![Q]\!](\sigma_0, \sigma_f)$	h, 3, 6, 7, 9

D.3.6 While

$\boxed{\textit{While-Post}}$

	from $wh = \textit{mk-While}(b_s \wedge b_u, \textit{body});\ (P, R) \vdash wh\ \mathbf{sat}\ (G, W^* \wedge P \wedge \neg(b_s \wedge b_u));$ $[\![P]\!](\sigma_0);\ C_0 = (wh, \sigma_0);\ C_f = (\mathbf{nil}, \sigma_f);\ C_0 \xrightarrow[R]{r}* C_f;\ sw = \textit{mk-Seq}(\textit{body}, wh);$ $\textit{IH-S}(\textit{body})$	
1	$\textit{well-founded}(W)$	h, *While-I*
2	$R \Rightarrow W^* \wedge I_{\textit{Vars}(b_s)}$	h, *While-I*
3	$\textit{SingleSharedVar}(b_u, R)$	h, *While-I*
4	$\neg\overleftarrow{(b_s \wedge b_u)} \wedge R \Rightarrow \neg(b_s \wedge b_u)$	h, *While-I*
5	$(P \wedge b_s, R) \vdash \textit{body}\ \mathbf{sat}\ (G, W \wedge P)$	h, *While-I*
6	$\exists C_1 \in \textit{Config}, v \in \mathbb{B}, \sigma_1 \in \Sigma \cdot$ $C_1 = (\textit{mk-If}(v, sw), \sigma_1) \wedge C_0 \xrightarrow[R]{r}* C_1 \xrightarrow[R]{r}* C_f$ $\wedge\ (C_0, C_1) \in (\text{A-R-Step} \cup \text{While} \cup \text{If-Eval})^*$	h, inspection of $\xrightarrow[R]{r}$
7	**from** $C_1 \in \textit{Config}, v \in \mathbb{B}, \sigma_1 \in \Sigma$ **st** $C_1 = (\textit{mk-If}(v, sw), \sigma_1) \wedge C_0 \xrightarrow[R]{r}* C_1 \xrightarrow[R]{r}* C_f$ $\wedge\ (C_0, C_1) \in (\text{A-R-Step} \cup \text{While} \cup \text{If-Eval})^*$	
7.1	$[\![R]\!](\sigma_0, \sigma_1)$	h7, *Rely-Trivial*
7.2	**from** $\neg v$	
7.2.1	$(C_1, C_f) \in (\text{A-R-Step} \cup \text{If-F-E})^*$	h, h7.2, inspection of $\xrightarrow[R]{r}$
7.2.2	$[\![R]\!](\sigma_1, \sigma_f)$	7.2.1, *Rely-Trivial*
7.2.3	$[\![R]\!](\sigma_0, \sigma_f)$	7.1, 7.2.2
7.2.4	$[\![W^*]\!](\sigma_0, \sigma_f)$	2, 7.2.3
7.2.5	$[\![P]\!](\sigma_f)$	h, 7.2.3, *PR-ident*
7.2.6	$[\![\neg(b_s \wedge b_u)]\!](\sigma_f)$	2, 3, 4, h7.2, 7.2.2
	infer $[\![W^* \wedge P \wedge \neg(b_s \wedge b_u)]\!](\sigma_0, \sigma_f)$	7.2.4–7.2.6
7.3	**from** v	
7.3.1	$\exists \sigma_2 \in \Sigma \cdot$ $C_1 \xrightarrow[R]{r}* (wh, \sigma_2) \xrightarrow[R]{r}* C_f$ $\wedge\ (C_1, (wh, \sigma_2)) \in (\text{A-R-Step} \cup \text{If-T-E} \cup \text{Seq-Step} \cup \text{Seq-E})^*$	h7, h7.3, inspection of $\xrightarrow[R]{r}$
7.3.2	**from** $\sigma_2 \in \Sigma$ **st** $C_1 \xrightarrow[R]{r}* (wh, \sigma_2) \xrightarrow[R]{r}* C_f$ $\wedge\ (C_1, (wh, \sigma_2)) \in (\text{A-R-Step} \cup \text{If-T-E} \cup \text{Seq-Step} \cup \text{Seq-E})^*$	
7.3.2.1	$(\textit{body}, \sigma_1) \xrightarrow[R]{r}* (\mathbf{nil}, \sigma_2)$	h7.3.2, *Isolation-While*
7.3.2.2	$[\![P]\!](\sigma_1)$	h, 7.1, *PR-ident*
7.3.2.3	$[\![b_s]\!](\sigma_1)$	2, h7, h7.3, *Single-Eval-If*
7.3.2.4	$[\![W \wedge P]\!](\sigma_1, \sigma_2)$	h, 5, 7.3.2.1–7.3.2.3, IH-S(body)
7.3.2.5	$[\![W^*]\!](\sigma_0, \sigma_1)$	2, 7.1
7.3.2.6	$[\![W \wedge P]\!](\sigma_0, \sigma_2)$	7.3.2.4, 7.3.2.5
7.3.2.7	**from** $\forall \sigma' \in \{\sigma'' \mid [\![W \wedge P]\!](\sigma_0, \sigma'') \wedge h[\sigma''/\sigma_0]\} \cdot$ $[\![W^* \wedge P \wedge \neg(b_s \wedge b_u)]\!](\sigma', \sigma_f)$	
7.3.2.7.1	$\sigma_2 \in \{\sigma'' \mid [\![W \wedge P]\!](\sigma_0, \sigma'') \wedge h[\sigma''/\sigma_0]\}$	h, h7.3.2, 7.3.2.6
7.3.2.7.2	$[\![W^* \wedge P \wedge \neg(b_s \wedge b_u)]\!](\sigma_2, \sigma_f)$	$\forall$-E(h7.3.2.7,7.3.2.7.1)
	infer $[\![W^* \wedge P \wedge \neg(b_s \wedge b_u)]\!](\sigma_0, \sigma_f)$	7.3.2.6, 7.3.2.7.2
	infer $[\![W^* \wedge P \wedge \neg(b_s \wedge b_u)]\!](\sigma_0, \sigma_f)$	1, 7.3.2.7, *W-Indn*
	infer $[\![W^* \wedge P \wedge \neg(b_s \wedge b_u)]\!](\sigma_0, \sigma_f)$	$\exists$-E(7.3.1,7.3.2)
	infer $[\![W^* \wedge P \wedge \neg(b_s \wedge b_u)]\!](\sigma_0, \sigma_f)$	$\vee$-E(h7.5.2,7.3)
	infer $[\![W^* \wedge P \wedge \neg(b_s \wedge b_u)]\!](\sigma_0, \sigma_f)$	$\exists$-E(6,7)

D.4 Theorem SAT

	from $(P, R) \vdash S$ **sat** $(G, Q);\ \Pi = \{\sigma \in \Sigma \mid [\![P]\!](\sigma)\};\ \Pi \neq \{\,\}$	
1	**from** $\sigma_0 \in \Pi$	
1.1	$S \in \mathit{Stmt}$	h
1.2	**from** $S \in \mathit{Assign}$	
	infer $\mathit{Converges}_c((S, \sigma_0), R, \{\mathbf{nil}\})$	h, h1, h1.2, *Assign-Converges*
1.3	**from** $S \in \mathit{Atomic}$	
	infer $\mathit{Converges}_c((S, \sigma_0), R, \{\mathbf{nil}\})$	h, h1, h1.3, *Atomic-Converges*
1.4	**from** $S \in \mathit{If}$	
	infer $\mathit{Converges}_c((S, \sigma_0), R, \{\mathbf{nil}\})$	h, h1, h1.4, *If-Converges*
1.5	**from** $S \in \mathit{Par}$	
	infer $\mathit{Converges}_c((S, \sigma_0), R, \{\mathbf{nil}\})$	h, h1, h1.5, *Par-Converges*
1.6	**from** $S \in \mathit{Seq}$	
	infer $\mathit{Converges}_c((S, \sigma_0), R, \{\mathbf{nil}\})$	h, h1, h1.6, *Seq-Converges*
1.7	**from** $S \in \mathit{While}$	
	infer $\mathit{Converges}_c((S, \sigma_0), R, \{\mathbf{nil}\})$	h, h1, h1.7, *While-Converges*
	infer $\mathit{Converges}_c((S, \sigma_0), R, \{\mathbf{nil}\})$	$\vee$-E(1.1–1.7)
2	$\forall \sigma \in \Pi \cdot \mathit{Converges}_c((S, \sigma), R, \{\mathbf{nil}\})$	$\forall$-I(1)
	infer $\mathit{Converges}_s(S, P, R, \{\mathbf{nil}\})$	h, 2, *Conv-Abstract*

D.4.1 Assign

$$\boxed{\text{Assign-Converges}}\ \frac{}{\mathit{Converges}_s(\mathit{mk\text{-}Assign}(id, e), \mathbf{true}, \mathbf{true}, \{\mathbf{nil}\})}$$

D.4.2 Atomic/STM

$\boxed{\textit{Atomic-Converges}}$

	from $(P, R) \vdash \textit{mk-Atomic}(\textit{body})$ **sat** (G, Q); $[\![P]\!](\sigma_0)$; $C_0 = (\textit{mk-Atomic}(\textit{body}), \sigma_0)$; $\textit{IH-T}(\textit{body})$	
1	$(P, I) \vdash \textit{body}$ **sat** $(\textbf{true}, Q')$	h, *Atomic-I*
2	$R \Rightarrow I_{\textit{Vars}(\textit{body})}$	h, *Atomic-I*
3	$\textit{Converges}_s(\textit{body}, P, I, \{\textbf{nil}\})$	1, IH-T(body)
4	**from** $S \in \textit{Atomic}; \sigma \in \Sigma, R' \in \textit{Rely}$	
	infer $\textit{Converges}_c((S, \sigma), R, \{\textit{mk-STM}(S.\textit{body}, \sigma', S.\textit{body}, \sigma') \mid [\![R]\!](\sigma, \sigma')\})$	h4, inspection of $\xrightarrow[R]{m}$, STM-Atomic
5	$\textit{Converges}_c(C_0, R, \{\textit{mk-STM}(\textit{body}, \sigma_b, \textit{body}, \sigma_b) \mid [\![R]\!](\sigma_0, \sigma_b)\})$	h, 4
6	$C^i = \{C_i \mid C_0 \xrightarrow[R]{m}{*}\ C_i \wedge C_i = (\textit{mk-STM}(\textit{body}, \sigma_b, \textit{body}, \sigma_b), \sigma_i) \wedge [\![R]\!](\sigma_0, \sigma_b)\}$	definition
7	**from** $C_i \in C^i$; $C_i = (\textit{mk-STM}(\textit{body}, \sigma_b, \textit{body}, \sigma_b), \sigma_i)$	
7.1	**from** $(C_j, C_k) \in \{(C_j, C_k) \mid C_i \xrightarrow[R]{m}{*}\ C_j \xrightarrow[R]{m} C_k\}$	
7.1.1	$(C_j, C_k) \in$ STM-Retry $\vee$ $(C_j, C_k) \notin$ STM-Retry	h7.1, inspection of $\xrightarrow[R]{m}$
7.1.2	**from** $(C_j, C_k) \in$ STM-Retry; $C_j = (S_j, \sigma_j)$; $\Sigma^j = \{\sigma \mid [\![R]\!](\sigma_j, \sigma)\}$	
7.1.2.1	$[\![R]\!](\sigma_b, \sigma_i)$	h7, inspection of $\xrightarrow[R]{m}$, STM-Atomic
7.1.2.2	$(C_j, C_k) \in$ (STM-Atomic $\cup$ STM-Step $\cup$ STM-E $\cup$ STM-Retry)*	h7.1.2, inspection of $\xrightarrow[R]{m}$
7.1.2.3	$[\![R]\!](\sigma_i, \sigma_j)$	7.1.2.2, *Rely-Trivial*
7.1.2.4	$\forall \sigma' \in \Sigma^j \cdot [\![I_{\textit{Vars}(\textit{body})}]\!](\sigma_j, \sigma)$	2, h7.1.2
7.1.2.5	$\forall \sigma' \in \Sigma^j \cdot [\![I_{\textit{Vars}(\textit{body})}]\!](\sigma_b, \sigma)$	7.1.2.1, 7.1.2.3, 7.1.2.4
7.1.2.6	$\forall \sigma' \in \Sigma^j \cdot (\textit{Vars}(\textit{body}) \lhd \sigma_b) = (\textit{Vars}(\textit{body}) \lhd \sigma')$	7.1.2.5
7.1.2.7	$\exists \sigma' \in \Sigma^j \cdot (\textit{Vars}(\textit{body}) \lhd \sigma_b) \neq (\textit{Vars}(\textit{body}) \lhd \sigma')$	h7.1.2, STM-Retry
	infer $\curlywedge$	7.1.2.6, 7.1.2.7, $\curlywedge$-I
	infer $(C_j, C_k) \notin$ STM-Retry	7.1.1, 7.1.2, $\curlywedge$-E
7.2	$\forall t \in \{(C_j, C_k) \mid C_i \xrightarrow[R]{m}{*}\ C_j \xrightarrow[R]{m} C_k\} \cdot t \notin$ STM-Retry	$\forall$-I(7.1)
7.3	$[\![P]\!](\sigma_b)$	h, h7, *PR-ident*
7.4	$\textit{Converges}_c((\textit{body}, \sigma_b), I, \{\textbf{nil}\})$	3, 7.3, *Conv-Concrete*
7.5	$\textit{Converges}_c(C_i, R, \{S \in \textit{STM} \mid S.\textit{body} = \textbf{nil}\})$	h7, 7.2, 7.4, *Conv-Wrap-STM*
7.6	$C^e = \{(S_e, \sigma_e) \mid C_i \xrightarrow[R]{m}{*}\ (S_e, \sigma_e) \wedge S_e \in \textit{STM} \wedge S_e.\textit{body} = \textbf{nil}\}$	definition
7.7	**from** $C_e \in C^e$	
7.7.1	**from** $S \in \textit{STM} \wedge S.\textit{body} = \textbf{nil}; \sigma \in \Sigma; R' \in \textit{Rely}; R' \Rightarrow I_{\textit{Vars}(S.\textit{body})}$	
	infer $\textit{Converges}_c((S, \sigma), R', \{\textbf{nil}\})$	
	infer $\textit{Converges}_c(C_e, R, \{\textbf{nil}\})$	2, h7.7, 7.7.1
7.8	$\forall C_e \in C^e \cdot \textit{Converges}_c(C_e, R, \{\textbf{nil}\})$	$\forall$-I(7.7)
	infer $\textit{Converges}_c(C_i, R, \{\textbf{nil}\})$	7.5, 7.6, 7.8, *Conv-Concat*
8	$\forall C_i \in C^i \cdot \textit{Converges}_c(C_i, R, \{\textbf{nil}\})$	$\forall$-I(7)
	infer $\textit{Converges}_c(C_0, R, \{\textbf{nil}\})$	5, 6, 8, *Conv-Concat*

D.4.3 If

$\boxed{\textit{If-Converges}}$

	from $(P, R) \vdash \textit{mk-If}(b, \textit{body})$ **sat** (G, Q); $[\![P]\!](\sigma_0)$; $C_0 = (\textit{mk-If}(b, \textit{body}), \sigma_0)$; $\textit{IH-T}(\textit{body})$	
1	$(P, R) \vdash \textit{body}$ **sat** (G, Q)	h, *If-I*
2	$\textit{Converges}_s(\textit{body}, P, R, \{$**nil**$\})$	1, IH-T(body)
3	**from** $S \in \textit{If}$; $\sigma \in \Sigma$; $R' \in \textit{Rely}$	
	infer $\textit{Converges}_c((S, \sigma), R', \{\textit{mk-If}(v, S.\textit{body}) \mid v \in \mathbb{B}\})$	h3, inspection of $\xrightarrow[R]{m}$, If-Eval
4	$\textit{Converges}_c((\textit{mk-If}(b, \textit{body}), \sigma_0), R, \{\textit{mk-If}(v, \textit{body}) \mid v \in \mathbb{B}\})$	h, 3
5	$C^v = \{C_v \mid C_0 \xrightarrow[R]{m}{*}\, C_v \wedge C_v = (\textit{mk-If}(v, \textit{body}), \sigma_v) \wedge v \in \mathbb{B}\}$	definition
6	**from** $C_v \in C^v$; $C_v = (\textit{mk-If}(v, \textit{body}), \sigma_v)$	
6.1	$v \in \mathbb{B}$	h6
6.2	**from** $\neg\, v$	
6.2.1	**from** $S \in \textit{If} \wedge S.b =$ **false**; $\sigma \in \Sigma$; $R' \in \textit{Rely}$	
	infer $\textit{Converges}_c((S, \sigma), R', \{$**nil**$\})$	h6.2.1, inspection of $\xrightarrow[R]{m}$, If-F-E
	infer $\textit{Converges}_c(C_v, R, \{$**nil**$\})$	h6, h6.2, 6.2.1
6.3	**from** v	
6.3.1	**from** $S \in \textit{If} \wedge S.b =$ **true**; $\sigma \in \Sigma$; $R' \in \textit{Rely}$	
	infer $\textit{Converges}_c((S, \sigma), R', \{S.\textit{body}\})$	h6.3.1, inspection of $\xrightarrow[R]{m}$, If-T-E
6.3.4	$\textit{Converges}_c(C_v, R, \{\textit{body}\})$	h6, h6.3, 6.3.1
6.3.5	$C^b = \{(\textit{body}, \sigma_b) \mid C_v \xrightarrow[R]{m}{*}\, (\textit{body}, \sigma_b)\}$	definition
6.3.6	**from** $C_b \in C^b$; $C_b = (\textit{body}, \sigma_b)$	
6.3.6.1	$(C_0, C_b) \in (\text{If-Eval} \cup \text{If-T-E})^*$	h6, h6.3, h6.3.6, inspection of $\xrightarrow[R]{m}$
6.3.6.1	$[\![R]\!](\sigma_0, \sigma_b)$	6.3.6.1, *Rely-Trivial*
6.3.6.2	$[\![P]\!](\sigma_b)$	h, 6.3.6.2, *PR-ident*
	infer $\textit{Converges}_c(C_b, R, \{$**nil**$\})$	2, 6.3.6.3, *Conv-Concrete*
6.3.7	$\forall C_b \in C^b \cdot \textit{Converges}_c(C_b, R, \{$**nil**$\})$	$\forall$-I(6.3.6)
	infer $\textit{Converges}_c(C_v, R, \{$**nil**$\})$	6.3.4, 6.3.5, 6.3.7, *Conv-Concat*
	infer $\textit{Converges}_c(C_v, R, \{$**nil**$\})$	$\vee$-E(6.1,6.2,6.3)
7	$\forall C_v \in C^v \cdot \textit{Converges}_c(C_v, R, \{$**nil**$\})$	$\forall$-I(6)
	infer $\textit{Converges}_c(C_0, R, \{$**nil**$\})$	4, 5, 7, *Conv-Concat*

D.4.4 Sequence

$\boxed{\textit{Seq-Converges}}$

	from $(P, R) \vdash \textit{mk-Seq}(\textit{left}, \textit{right})$ **sat** (G, Q); $[\![P]\!](\sigma_0)$; $C_0 = (\textit{mk-Seq}(\textit{left}, \textit{right}), \sigma_0)$; $\textit{IH-T}(\textit{left})$; $\textit{IH-T}(\textit{right})$	
1	$(P, R) \vdash \textit{left}$ **sat** $(G, Q_l \wedge P_r)$	h, *Seq-I*
2	$\textit{Converges}_s(\textit{left}, P, R, \{\textbf{nil}\})$	h, 1, IH-T(left)
3	$\textit{Converges}_c((\textit{left}, \sigma_0), R, \{\textbf{nil}\})$	h, 2, *Conv-Concrete*
4	$\textit{Converges}_c(C_0, R, \{\textit{mk-Seq}(\textbf{nil}, \textit{right})\})$	3, *Conv-Wrap-Seq*
5	$C^i = \{C_i \mid C_0 \xrightarrow[R]{m}{*}\, C_i \wedge C_i = (\textit{mk-Seq}(\textbf{nil}, \textit{right}), \sigma_i)\}$	definition
6	**from** $C_i \in C^i$; $C_i = (\textit{mk-Seq}(\textbf{nil}, \textit{right}), \sigma_i)$	
6.1	**from** $S \in \textit{Seq} \wedge S.\textit{left} = \textbf{nil}$; $\sigma \in \Sigma$; $R' \in \textit{Rely}$	
	infer $\textit{Converges}_c((S, \sigma), R', \{S.\textit{right}\})$	h, h6.1, inspection of $\xrightarrow[R]{m}$, Seq-E
	infer $\textit{Converges}_c(C_i, R, \{\textit{right}\})$	h, h6, 6.1
7	$\forall C_i \in C^i \cdot \textit{Converges}(C_i, R, \{\textit{right}\})$	$\forall$-I(6)
8	$\textit{Converges}_c(C_0, R, \{\textit{right}\})$	4, 5, 7, *Conv-Concat*
9	$(P_r, R) \vdash \textit{right}$ **sat** (G, Q_r)	h, *Seq-I*
10	$\textit{Converges}_c(\textit{right}, P_r, R, \{\textbf{nil}\})$	h, 9, IH-T(right)
11	$C^r = \{(\textit{right}, \sigma_r) \mid C_0 \xrightarrow[R]{m}{*}\, (\textit{right}, \sigma_r)\}$	definition
12	**from** $(\textit{right}, \sigma_r) \in C^r$	
12.1	$[\![P_r]\!](\sigma_r)$	1, h12, *Theorem PSAT*
	infer $\textit{Converges}_c((\textit{right}, \sigma_r), R, \{\textbf{nil}\})$	10, 12.1, *Conv-Concrete*
13	$\forall C_r \in C^r \cdot \textit{Converges}_c((\textit{right}, \sigma_r), R, \{\textbf{nil}\})$	$\forall$-I(12)
	infer $\textit{Converges}_c(C_0, R, \{\textbf{nil}\})$	8, 11, 13, *Conv-Concat*

D.4.5 Parallel

$\boxed{\textit{Par-Converges}}$

from	$(P, R) \vdash \textit{mk-Par}(\textit{left}, \textit{right})$ **sat** (G, Q); $[\![P]\!](\sigma_0)$; $C_0 = (\textit{mk-Par}(\textit{left}, \textit{right}), \sigma_0)$; $\textit{IH-T}(\textit{left})$; $\textit{IH-T}(\textit{right})$	
1	$(P, R \vee G_r) \vdash \textit{left}$ **sat** (G_l, Q_l)	h, *Par-I*
2	$(P, R \vee G_l) \vdash \textit{right}$ **sat** (G_r, Q_r)	h, *Par-I*
3	$\textit{Converges}_s(\textit{left}, P, R \vee G_r, \{\textbf{nil}\})$	1, IH-T(left)
4	$\textit{Converges}_s((\textit{right}, \sigma_0), P, R \vee G_l, \{\textbf{nil}\})$	2, IH-T(right)
5	$\textit{Converges}_c((\textit{left}, \sigma_0), P, R \vee G_r, \{\textbf{nil}\})$	h, 3, *Conv-Concrete*
6	$\textit{Converges}_c((\textit{right}, \sigma_0), P, R \vee G_l, \{\textbf{nil}\})$	h, 4, *Conv-Concrete*
7	$\textit{Converges}_c(C_0, R, \{\textit{mk-Par}(\textbf{nil}, \textbf{nil})\})$	1, 2, 5, 6, *Conv-Wrap-Par*
8	$C^e = \{C_e \mid C_0 \xrightarrow[R]{m}\!\!* \; C_e \wedge C_e = (\textit{mk-Par}(\textbf{nil}, \textbf{nil}), \sigma_e)\}$	definition
9	**from** $C_e \in C^e$; $C_i = (\textit{mk-Par}(\textbf{nil}, \textbf{nil}), \sigma_e)$	
9.1	**from** $S = \textit{mk-Par}(\textbf{nil}, \textbf{nil})$; $\sigma \in \Sigma$; $R' \in \textit{Rely}$	
	infer $\textit{Converges}_c((S, \sigma), R', \{\textbf{nil}\})$	h9.1, inspection of $\xrightarrow[R]{m}$, Par-E
	infer $\textit{Converges}_c(C_e, R, \{\textbf{nil}\})$	h9, h9.1
10	$\forall C_e \in C^e \cdot \textit{Converges}_c(C_e, R, \{\textbf{nil}\})$	$\forall$-I(9)
infer	$\textit{Converges}_c(C_0, R, \{\textbf{nil}\})$	7, 8, 10, *Conv-Concat*

D.4.6 While

$\boxed{\textit{While-Converges}}$

from $wh = \textit{mk-While}(b_s \wedge b_u, \textit{body})$; $(P, R) \vdash wh$ **sat** $(G, W^* \wedge P \wedge \neg(b_s \wedge b_u))$;
$[\![P]\!](\sigma_0)$; $sw = \textit{mk-Seq}(\textit{body}, wh)$; $C_0 = (wh, \sigma_0)$; $\textit{IH-T}(\textit{body})$

1 $\textit{well-founded}(W)$ — h, *While-I*

2 $\textit{bottoms}(W) \subseteq [\![\neg(b_s \wedge b_u)]\!]$ — h, *While-I*

3 $R \Rightarrow W^* \wedge I_{\textit{Vars}(b_s)}$ — h, *While-I*

4 $\textit{SingleSharedVar}(b_u, R)$ — h, *While-I*

5 $\overleftarrow{\neg(b_s \wedge b_u)} \wedge R \Rightarrow \neg(b_s \wedge b_u)$ — h, *While-I*

6 $(P, R) \vdash \textit{body}$ **sat** $(G, W \wedge P)$ — h, *While-I*

7 $\textit{Converges}_s(\textit{body}, P, R, \{\textbf{nil}\})$ — 1, IH-T(body)

8 **from** $\forall C' \in \{(wh, \sigma'') \mid C_0 \xrightarrow[R]{m}{}^{*} (wh, \sigma'') \wedge [\![W \wedge P]\!](\sigma_0, \sigma'')\} \cdot$
$\textit{Converges}_c(C', R, \{\textbf{nil}\})$

8.1 **from** $S \in \textit{While}$; $\sigma \in \Sigma$; $R' \in \textit{Rely}$; $sw' = \textit{mk-Seq}(S.\textit{body}, S)$
infer $\textit{Converges}_c((S, \sigma), R', \{\textit{mk-If}(v, sw') \mid v \in \mathbb{B}\})$ — h8.1, inspection of $\xrightarrow[R']{m}$

8.2 $\textit{Converges}_c(C_0, R, \{\textit{mk-If}(v, sw) \mid v \in \mathbb{B}\})$ — h, 8.1

8.3 $C^v = \{(\textit{mk-If}(v, sw), \sigma_v) \mid C_0 \xrightarrow[R]{m}{}^{*} (\textit{mk-If}(v, sw), \sigma_v) \wedge v \in \mathbb{B}\}$ — definition

8.4 **from** $C_v \in C^v$; $C_v = (\textit{mk-If}(v, sw), \sigma_v)$

8.4.1 **from** $\neg v$

8.4.1.1 **from** $S \in \textit{If} \wedge S.b =$ **false**; $\sigma \in \Sigma$; $R' \in \textit{Rely}$
infer $\textit{Converges}_c((S, \sigma), R', \{\textbf{nil}\})$ — h8.4.1.1, inspection of $\xrightarrow[R']{m}$

infer $\textit{Converges}_c(C_v, R, \{\textbf{nil}\})$ — h, h8.4, h8.4.1, 8.4.1.1

8.4.2 **from** v

8.4.2.1 **from** $S \in \textit{If} \wedge S.b =$ **true**; $\sigma \in \Sigma$; $R' \in \textit{Rely}$
infer $\textit{Converges}_c((S, \sigma), R', \{S.\textit{body}\})$ — h8.4.2.1, inspection of $\xrightarrow[R']{m}$

8.4.2.2 $\textit{Converges}_c(C_v, R, \{sw\})$ — h, h8.4, h8.4.2, 8.4.2.1

8.4.2.3 $C^b = \{(sw, \sigma_b) \mid C_v \xrightarrow[R]{m}{}^{*} (sw, \sigma_b)\}$ — definition

8.4.2.4 **from** $C_b \in C^b$; $C_b = (sw, \sigma_b)$

8.4.2.4.1 $[\![P \wedge b_s]\!](\sigma_b)$ — h, 6, h8.4, h8.4.2, h8.4.2.4, *While-interstices-pre*

8.4.2.4.2 $\textit{Converges}_c((\textit{body}, \sigma_b), R, \{\textbf{nil}\})$ — 7, 8.4.2.4.1, *Conv-Concrete*

infer $\textit{Converges}_c(C_b, R, \{\textit{mk-Seq}(\textbf{nil}, wh)\})$ — 8.4.2.4.2, *Conv-Wrap-Seq*

8.4.2.5 $\forall C_b \in C^b \cdot \textit{Converges}(C_b, R, \{\textit{mk-Seq}(\textbf{nil}, wh)\})$ — $\forall$-I(8.4.2.4)

8.4.2.6 $\textit{Converges}_c(C_v, R, \{\textit{mk-Seq}(\textbf{nil}, wh)\})$
— 8.4.2.2, 8.4.2.3, 8.4.2.5, *Conv-Concat*

8.4.2.7 **from** $S \in \textit{Seq} \wedge S.\textit{left} =$ **nil**; $\sigma \in \Sigma$; $R' \in \textit{Rely}$
infer $\textit{Converges}_c((S, \sigma), R', \{S.\textit{right}\})$ — h8.4.2.7, inspection of $\xrightarrow[R]{m}$

8.4.2.8 $\forall \sigma \in \Sigma \cdot \textit{Converges}((\textit{mk-Seq}(\textbf{nil}, wh), \sigma), R, \{wh\})$ — $\forall$-I(8.4.2.7)

8.4.2.9 $\textit{Converges}_c(C_v, R, \{wh\})$ — 8.4.2.6, 8.4.2.8, *Conv-Concat*

8.4.2.10 $C^w = \{(wh, \sigma_w) \mid C_0 \xrightarrow[R]{m}{}^{*} C_v \xrightarrow[R]{m}{}^{*} (wh, \sigma_w)\}$ — definition

8.4.2.11 $\forall (S_w, \sigma_w) \in C^w \cdot [\![W \wedge P]\!](\sigma_0, \sigma_w)$
— h, 3, 6, 8.4.2.10, *While-interstices-psat*

8.4.2.12 $C^w \subseteq \{(wh, \sigma'') \mid C_0 \xrightarrow[R]{m}{}^{*} (wh, \sigma'') \wedge [\![W \wedge P]\!](\sigma_0, \sigma'')\}$
— h8, 8.4.2.10, 8.4.2.11

8.4.2.13 $\forall C_w \in C^w \cdot \textit{Converges}(C_w, R, \{\textbf{nil}\})$ — h8, 8.4.2.12

infer $\textit{Converges}_c(C_v, R, \{\textbf{nil}\})$ — 8.4.2.9, 8.4.2.10, 8.4.2.13, *Conv-Concat*

infer $\textit{Converges}_c(C_v, R, \{\textbf{nil}\})$ — $\vee$-E(h8.4,8.4.1,8.4.2)

8.5 $\forall C_v \in C^v \cdot \textit{Converges}_c(C_v, R, \{\textbf{nil}\})$ — $\forall$-I(8.4)

infer $\textit{Converges}_c(C_0, R, \{\textbf{nil}, wh\})$ — 8.2, 8.3, 8.5, *Conv-Concat*

infer $\textit{Converges}_c(C_0, R, \{\textbf{nil}\})$ — 1, 2, 8, *W-Indn*

$\boxed{\text{E}}$ — Selected VDM Syntax Definitions

A note on types

- M are mappings
- R are relations
- S are sets
- x, y, z are elements in a set or a domain/range of a mapping or relation

Mappings

dom	$\mathbf{dom}\, M \equiv$ the domain of M
rng	$\mathbf{rng}\, M \equiv$ the range of M
fld	$\mathbf{fld}\, M \equiv \mathbf{dom}\, M \cup \mathbf{rng}\, M$
$\lhd$	$S \lhd M \equiv \{x \mapsto M(x) \mid x \in (S \cap \mathbf{dom}\, M)\}$
$\rhd$	$M \rhd S \equiv \{x \mapsto M(x) \mid x \in \mathbf{dom}\, M \wedge M(x) \in S\}$
$⩤$	$S ⩤ M \equiv \{x \mapsto M(x) \mid x \in (\mathbf{dom}\, M - S)\}$
$⩥$	$M ⩥ S \equiv \{x \mapsto M(x) \mid x \in \mathbf{dom}\, M \wedge M(x) \notin S\}$
$\dagger$	$M_0 \dagger M_1 \equiv \left\{ x \mapsto y \;\middle|\; \begin{array}{l} (x \in (\mathbf{dom}\, M_0 - \mathbf{dom}\, M_1) \wedge y = M_0(x)) \\ \vee\ (x \in \mathbf{dom}\, M_1 \wedge y = M_1(x)) \end{array} \right\}$

Relations

dom	$\mathbf{dom}\, R \equiv$ the domain of R
rng	$\mathbf{rng}\, R \equiv$ the range of R
fld	$\mathbf{fld}\, R \equiv \mathbf{dom}\, R \cup \mathbf{rng}\, R$
$\lhd$	$S \lhd R \equiv \{(x, y) \mid x \in S \wedge (x, y) \in R\}$
$\rhd$	$R \rhd S \equiv \{(x, y) \mid y \in S \wedge (x, y) \in R\}$
$⩤$	$S ⩤ R \equiv \{(x, y) \mid x \notin S \wedge (x, y) \in R\}$
$⩥$	$R ⩥ S \equiv \{(x, y) \mid y \notin S \wedge (x, y) \in R\}$
$\diamond$	$R_0 \diamond R_1 \equiv \{(x, z) \mid \exists y \cdot ((x, y) \in R_0 \wedge (y, z) \in R_1)\}$

F — Index of Rules, Lemmas, and Proofs