Newcastle University

SCHOOL OF COMPUTING SCIENCE

Reasoning about Programs using Operational Semantics and the Role of a Proof Support Tool

Thesis by

John Robert Derek Hughes

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

October 2011

Abstract

A computer program is a text in a defined programming language; each such program can be thought of as defining state transitions — that is, the execution of a program in an initial state will result in a (or possibly one of a set of) final state(s). A program specification defines properties that relate initial and final states. For example, a specification might state that some property will hold in the final state after the program has been executed, as long as it was executed in an initial state where some other property held.

Defining the *semantics* of a programming language fixes the behaviour of the language and gives meaning to programs that are written in the language. One straightforward way of giving the semantics of a programming language is using *operational semantics*, which describes a language in terms of the effect execution has on the state: a program still defines state transitions, but for an abstract state.

This thesis investigates the possibility of using the operational semantic description of a programming language to reason about programs that are written in that language. Programs are shown to be correct with respect to a specification, which consists of a precondition and a postcondition. Natural deduction proofs about a program are written to show that if it is executed in a state that satisfies the precondition, then execution will result in a state that relates to the initial state such that the postcondition is true of the two states. The rules of an operational semantic description are used in the proof to show the steps a program will take to reach a final state, and the effect execution has on the state. This is contrasted with the use of *axiomatic semantics*, observing that using operational semantics allows us to handle a wider class of language features.

The acceptability of this approach will almost certainly depend on appropriate tool support. A prototype proof support tool is therefore developed as a 'proof of concept', to assist a user in creating these kinds of proof. The tool manages inference rules and semantic descriptions, and the layout of the proof, but does not automate the proof process.

Acknowledgements

I would first of all like to thanks my supervisor, Cliff Jones, for the opportunity to work on this thesis. I am grateful for his guidance, for always finding time for me and for all the things he has taught me.

I would also like to thank Joey Coleman for his support and advice, for always being willing help me and explain things, and for helpful feedback on drafts of this thesis.

I am also grateful to Peter Larsen for inviting me to work with him at IHA in Århus. Although brief, I was able to learn a lot during my time there.

I would also like to thank my colleagues and friends at Newcastle University for their support and advice and for contributing to a pleasant working environment.

Lastly, I would like to thank my parents and family for their support and encouragement.

I am grateful to EPSRC for funding this work.

Contents

Chapter 1

Introduction

Since the early days of electronic computers, people have been concerned about the correctness of computer programs. In [GvN47], the authors show that they are not only concerned about correctness of programs, but that they believed that it is possible to reason about their correctness. Their proposed approach was to introduce 'assertion boxes' at points in a program that are separate from the program, and describe its effect. Turing also wrote about the use of assertions to specify the effect of a computer program in [Tur49]. Turing's approach was to decorate 'flow diagrams' with assertions. He saw these assertions as a way to reason about arcs within the flow of a program so that the correctness of the program as a whole could be easily established.

The main advances in this area came after Floyd's paper, [Flo67], in which he annotates flow charts with general assertions. The flow charts represent the flow of control through a program, and the assertions, which relate the values of variables, are placed at each connection in the flow chart, and should hold whenever the connection is taken. The main contribution of Floyd's paper was not to present a way of proving programs correct, but to provide a "basis for formal definitions of the meanings of programs in appropriately defined programming languages, in such a way that a rigorous standard is established for proofs about computer programs" [Flo67].

Floyd's paper influenced the work of Hoare, who in [Hoa69] makes the move from flow diagrams to annotated program code. The assertions about a program are written in the form of a 'Hoare-triple', in which the program text itself is annotated with pre- and post-conditions. Hoare's paper also introduces an axiomatic approach to defining languages, in which a language is defined in terms of axioms and inference rules, which can then be used to reason about a program. The paper only covers basic language features and subsequent papers attempted to extend Hoare's work to cover additional features of programming languages, most notably, Hoare and Wirth's attempt to provide an axiomatic semantics for a portion of Pascal in [HW73].

One of the limitations of the axiomatic approach is that for most used programming languages a complete axiomatisation seems to be impractical without the explicit design constraint that they should have an axiomatic semantics[1] [Jon03b]. There are certain language features for which it is difficult to give an axiomatic semantics, such as concurrency (see Section 3.2.3). For some features of modern imperative languages, writing an axiomatisation becomes difficult as their axiom requires more 'machinery' from the language, resulting in an axiom that is more operational.

1.1 Thesis Hypotheses

This thesis examines the following hypotheses:

1. The operational semantics of a programming language can be used as a basis for reasoning about programs written in the language — this approach is applicable to a wider class of languages than that covered by axiomatic semantics.

2. Although some setup is required when basing program reasoning on operational semantics, this effort can be made acceptable with a suitable support system.

[1]Turing [HMRC88] is an example of such a language.

1.2 Thesis Contribution

The aim of this thesis is to investigate an alternative method of reasoning about programs, using an operational approach (see Section 3.4). More specifically, it aims to use an operational semantic description of a language to reason about programs that are written in that language.

Most features of modern imperative programming languages can be described in an operational semantics. If it is possible to reason about a program by using the operational semantics of the language it is written in, then we will be able to reason about programs that are written in more complex languages than we can using just axiomatic semantics.

The operational semantics of a language defines the execution of the language in terms of the transitions it can make from an initial state to a final state, and the effect execution has on the state. Using the operational semantic rules of a language, we can show that if a program is executed in an initial state, then it will reach a final state after execution in which a specific property will hold, the property being part of the program's specification.

As with Hoare logic, a program's specification is represented as a *precondition* and a *postcondition*. The precondition makes assertions that should hold before the execution of the program, and the postcondition makes assertions that should hold after execution is completed. Unlike Hoare logic, a postcondition is an assertion over two states (initial and final) so it can relate the final value of a variable to its original value before execution. To show that a program meets its specification, natural deduction proofs are written in which operational semantic rules are used as inference rules to justify steps in the proof (see Section 3.4.4).

This thesis gives several examples of this operational semantics approach to proving properties of programs, including an example that uses concurrency (see Chapter 4).

Since each step in the program will have a corresponding step in the proof, these kinds of proofs may get quite long and detailed. Therefore the acceptability of this approach will likely depend on a suitable support system (see Chapter 6). A suitable system should have an intuitive interface that allows the user to work with the proof. The system should also be accessible to novice users and so should not have a steep learning curve.

The type of system considered to be most suitable is a *proof support tool* (see Chapter 5). Part of this thesis is to develop a prototype proof support tool that will assist a user to reason about a program using operational semantics (see Chapter 7). The tool is intended to be a user guided proof system that allows the user to write natural deduction proofs that are displayed in as easy-to-read format. The user should not be restricted by the tool to write a proof in a particular way, but should be free to complete lines and justifications of the proof in any order. The tool will not try and automatically complete the proof, but will assist the user where possible. The kind of assistance provided by the tool will be to manage the proof layout including automatically numbering proof lines, and suggesting justifications for proof steps. The tool should also have a proof library and rule library that can be added to by the user, from which the tool will be able to make suggestions for justifications to the user.

1.3 Related Work

Burstall and Honsell in [BH91] extend a natural deduction framework so that it can be used to define a programming language in an operational semantics. Their work however only defines programming languages, but does not go as far as trying to verify programs written in the language.

There are various types of tools available that are intended to assist the user to reason about programs. The Java Modelling Language (JML) [jml10] allows the user to annotate program code with assertions such as invariants and Hoare-like pre- and post- conditions, which can then be checked during runtime.

Jinja is a comprehensive model of a Java-like programming language that includes many of the core features of the Java language and covers both the source language and the virtual machine [Nip06]. It is designed to be a realistic language while maintaining a comprehensible model. Jinja has a big and small step operational semantics which are shown to be equivalent. The small step semantics are need to be able to refer to intermediate states during execution, allowing parallelism to be modelled. The development of Jinja was carried out in Isabelle/HOL [PNW10].

Ott [SN10] is a tool for writing definitions of programming languages and calculi, by specifying the syntax and semantics of the language. This definition can then be typeset in LaTeX, or be used to generate formal definitions of the language for Coq, HOL and Isabelle.

The tool that most closely meets the requirements outline in Chapter 6 is the Mural tool [JJLM91], although it is no longer available. Mural was a support tool for writing and storing VDM specifications, but also included a proof assistant that allowed the user to write Jones style [Jon90] proofs.

1.4 Overview

Following this introduction chapter, Chapter 2 reviews logical frameworks, natural deduction proofs and inference rules, providing a logical basis for the thesis.

Chapter 3 covers three main approaches to defining semantics of programming languages – *Axiomatic*, *Denotational* and *Operational* – with a focus on operational semantics, and also describes the approach used throughout this thesis to defining the abstract syntax and context conditions of programming languages.

Section 3.4.4 introduces and explains the use of operational semantics to reason about programs, using a simple program as an example. Further examples are given in Chapter 4, including an example of a concurrent array search program (Section 4.4). Section 4.5 uses the same concurrent search program to give an example of a Hoare style proof for comparison.

A review of formal verification tools is given in Chapter 5, providing a context for the proof support tool developed as part of this thesis.

Chapter 6 outlines the requirements and features of a proof support tool for reasoning about programming languages using operational semantics, as well as explaining the intended interaction between the user and tool when writing a proof (see Section 6.4).

The development of the prototype proof support tool is described in Chapter 7. The chapter first describes the formal model of the proof system as written in the *Vienna Development Method*(VDM) [Jon90, Daw91] (see Section 7.1), then the Java implementation of the prototype (see Section 7.2), followed by how to use the prototype tool (see Section 7.3), and a discussion of challenges encountered during the development of the prototype (see Section 7.4).

The final chapter, Chapter 8, summarises the results of the thesis and suggests areas for further work.

Chapter 2

Logic and Proof

Logic is said to be the study of reasoning [vD94]. In a natural language we can write *declarative statements*, such as “the sky is blue”, that have a *truth value* — that is, they are either *true* or *false*. We can also define our own languages that are more formal than natural languages and use *connectives* (such as *and*, *or* and *not*) to make more complex propositions.

It can be determined if a proposition is *true* or *false* by using *truth tables.* The most basic parts of a proposition are simply *true* or *false*. The truth value of more complex propositions can be determined using *truth tables.* A truth table lists the possible values that a free variable might take in a proposition, along with the truth value of the whole proposition for those possible values. For example, the truth table for the logical sentance $a \vee b$ is given in Figure 2.1.

a	b	$a \vee b$
true	true	true
true	false	true
false	true	true
false	false	false

Figure 2.1: Truth table for $a \vee b$

Truth tables can also be used to determine the truth value of more complex propositions, but the more connectors there are, the larger the table becomes.

Instead of determining the truth value of a proposition using truth tables, we can also use a system of deriving conclusions from premises [vD94]. For each connective we can define derivation or inference rules, which are explained further in Section 2.3. This approach is known as *natural deduction* and was introduced by Gentzen [Gen69]. This thesis follows this style of deduction; the rest of this chapter explains how it works.

This chapter discusses *logic*, as well as axioms and inference rules and how they are used in deductions. To be able to reason about programs using the semantics of a programming language, we need to first have a *formal language* in which assertions (typically about the states of program executions) can be written, and a set of *axioms* and *inference rules* that show what inferences are valid in our language [BFL+94].

There are many system of logic, each with varying degrees of complexity and expressiveness. In this thesis we will be using first order predicate calculus.

Section 2.1 describes deductions in an axiomatisation of a logic, and explains the way in which proofs are presented in this thesis. Section 2.5 covers how a proof is built and the different reasoning techniques that might be used. Section 2.6 explains the use of quantifiers.

2.1 Proof

A proof is intended to show that, from certain known properties, another property must hold. We can think of a proof as a structured argument that is intended to increase confidence that some property holds. Such proofs may be formal or informal. We first look at formal proofs, in which each step is an instance of an axiom, or is derived by a rule of inference from earlier proved statements. Section 2.5 discusses how to construct a formal proof.

In [Fit52] a proof is defined as a “finite sequence of items such that each item of the sequence satisfies at least one of the following two conditions: (1) It is an axiom of the system. (2) It is a direct consequence of preceding items of the sequence".

This thesis borrows its style of presenting proofs from [Jon90], which is a form of natural deduction proof. As an example of this style of proof, Figure 2.2 shows a proof for the *and introduction* rule.

	from	a, b	
1		$\neg\neg a$	$\neg\neg$-I(h1)
2		$\neg\neg b$	$\neg\neg$-I(h2)
3		$\neg(\neg a \vee \neg b)$	$\neg\vee$-I(1, 2)
	infer	$a \wedge b$	folding(3)

Figure 2.2: Proof of $\wedge$-I.

The inference rules used in the proof in Figure 2.2, and in the rest of this chapter, can be found in Appendix A.

The keyword ***from*** is used to indicate that we are working from the specified set of hypotheses, which are written next to the **from**. For convenience in referring to hypotheses we label them *'hn'*, where n is the position of the hypothesis in the list. So in Figure 2.2, the hypothesis a has the label *h1*.

The lines of the proof are numbered so that they can be referred to easily. For convenience in reading the proof, the lines are numbered in ascending numerical order starting from the first line of the proof and working down to the last line.

The property that is to be proven to hold (the goal, or conclusion) is written at the end of the proof and labelled with the keyword ***infer***.

Looking at line 1 of the proof, the body of the line is the expression $\neg\neg a$. The justification of this line is made by referencing the inference rule $\neg\neg$-I, as given to the right of the line body. Notice also that the reference to the inference rule also has a parameter, which is may be a line number, but in this case it is a reference to a hypothesis.

Although there is no restriction on the ordering of lines in a proof, it should be remembered that layout of the proof is intended to make a proof easy to read. The proof will generally be read starting at the hypotheses and then line by line to the conclusion, and should therefore be arranged so that each line of the proof builds upon previous lines (or hypotheses), until the conclusion can be inferred.

An informal proof has a much looser structure and steps can be made with only an outline justification. Its aim is to show how a proof could be constructed [Jon90], and it appeals to the reader's intuition, rather than relying on formal justifications from axioms and inference rules.

In the simplest case, an informal proof may be written in sentences, rather than in a formal language. It may also be written in a more formal style, but using informal (or no) justifications, and having missing steps. A single step in an informal proof might represent several steps if the proof was made formal, but still convey the essential meaning to the reader. We can think of an informal proof as an outline for a formal proof, since it should be possible to fill in the gaps to make the proof formal.

Since informal proofs rely on a reader's intuition for interpretation, they do not lend themselves to being machine read and checked.

2.2 Sequents

A *sequent* is a statement about logical expressions, stating that some conclusion is derivable from some hypotheses. We write sequents using the $\vdash$ symbol (often called a *turnstile*) to separate hypotheses and conclusion [Jon90].

$$P \vdash R$$

The above sequent asserts that R can be deduced from P. So in all cases where the hypothesis P is **true**, the conclusion R is also **true**.

It also possible to define *sequent variables* that are variables that are bound throughout a sequent [BFL+94]. This is signified by a subscript on the turnstile, as in the below sequent.

$$a\colon A \vdash_a P(a)$$

Here, the variable a is bound throughout the sequent, meaning that any occurrence of a, on either side of the turnstile, refers to the same variable.

The validity of a sequent can be checked using truth tables, or by writing a natural deduction proof. The hypotheses and conclusion of a sequent relate to the hypotheses and conclusion of a proof. The proof in Figure 2.2 written as a sequent would give us $a, b \vdash a \wedge b$.

A sequent may be used in inference rules, as in Figure 2.3. Using such a rule as a justification in a proof would require a nested or *subordinate* proof [BFL+94], as explained in Section 2.5.1.

2.3 Inference Rules

As mentioned in Section 2.1, inference rules are used as justifications in formal proofs. They describe what can be inferred from a set of knowns. The form of an inference rule is to have a set of hypotheses (above the line) and a conclusion (below the line), as shown in Figure 2.3. In this case the hypotheses are separated by a semi-colon.

$$\boxed{\wedge\text{-subs-left}}\,\frac{e_1 \wedge e_2;\, e_1 \vdash e}{e \wedge e_2}$$

Figure 2.3: $\wedge$ substitution inference rule

The variables used in an inference rule are known as *metavariables*, and are used to represent arbitrary expressions [BFL+94]. When a rule is used to justify lines of a proof the metavariables are *instantiated*, that is they are replaced by a term or expression.

Valid Rules

An inference rule whose validity is taken to be self-evident is known as an *axiom* [BFL+94]. Axioms do not require to be proven and form the foundation of a system of inference. We can build on a set of axioms to create more expressive inference rules. In the system used in this thesis, axioms may have hypotheses.

Inference rules that are not axioms need to be shown to be valid by constructing a proof showing that the conclusion can be reached from the hypotheses, using axioms or other proven rules as justifications. These inference rules are known as *derived rules*, and can be used in the proofs of other derived rules. A derived rule essentially acts as a shorthand for its proof, so that in a proof we can just refer to the derived rules as the justification of a step, rather than having to copy all of the steps from its proof [BFL+94].

Rule Application

Applying an inference rule as a justification to a line in a proof is essentially a pattern matching exercise. The line being justified must match the conclusion of the inference rule, and the hypotheses of the inference rule must be matched against other lines in the proof.

Using the ¬¬ introduction rule (below) as an example, we will go through how inference rules are instantiated and used in a proof.

$$\boxed{\neg\neg\text{-I}}\,\frac{e}{\neg\neg e}$$

This rule states that for any known, we can infer its double negation. It should be apparent that this rule is *true* or *self-evident* and therefore an axiom. The term e in this inference rule is what is known as a *free variable*, meaning that it is not bound to any particular value, and that it may be substituted for a value. A substitution for e does not necessarily have to be single term, it could substituted by an expression. For example, our substitution for e could be the expression $a \wedge b$, giving us $\neg\neg(a \wedge b)$.

Looking back to the example proof for the ∧-I inference rule, in Figure 2.2, we can see that the ¬¬-I rule was used as the justification for two lines in the proof. In line 1 of the proof, selecting the inference rule ¬¬-I as a justification means that the body of the line must match the conclusion of the inference rule. The justification also instantiates the rule so that e is now bound to a. Since e is the only free variable, and appears in the conclusion and the hypotheses, we are now required to match the hypothesis against a line of the proof that consists of a. The justification in the proof uses h1, which meets this requirement.

Introduction and Elimination

In a natural deduction system we define *introduction* and *elimination* inference rules for each of the connectives in our formal language [vD94, Pra06].

The *not not introduction* rule has just been introduced, which allows the double negation of a term to be inferred. There also exists an elimination rule that removes a term's double negation.

$$\boxed{\neg\neg\text{-E}}\ \frac{\neg\neg e}{e}$$

It is possible to combine these introduction and elimination rules into a single bi-directional rule, that allows inference to be made in either direction. To distinguish bi-directional rules from regular inference rules we use a double horizontal.

$$\boxed{\neg\neg\text{-I/E}}\ \begin{array}{c} e \\ \hline\hline \neg\neg e \end{array}$$

It is not always possible to combine introduction and elimination rules into a single bi-directional rule. For example, there are actually two *and elimination* rules (Figure 2.4), one for each of the terms to be eliminated.

2.4 Definitions

Simple propositions can be combined into a more complex logical expression using logical operators or *connectives* such as $\wedge$ (*and*), $\vee$ (*or*) and $\neg$ (*not*).

$$\boxed{\wedge\text{-E-Left}}\,\frac{a \wedge b}{b}$$

$$\boxed{\wedge\text{-E-Right}}\,\frac{a \wedge b}{a}$$

Figure 2.4: $\wedge$ Elimination rules

It is convenient to choose a minimal set of connectives and define others in terms of the minimal *'axiomatised'* set. For example, using *not* ($\neg$) and *or* ($\vee$) as our minimal set we can define the operators *and* ($\wedge$), *implication* ($\Rightarrow$) and *equivalence* ($\Leftrightarrow$), as shown in Figure 2.5.

$$a \wedge b \overset{def}{=} \neg(\neg a \vee \neg b)$$

$$a \Rightarrow b \overset{def}{=} \neg a \vee b$$

$$a \Leftrightarrow b \overset{def}{=} (a \Rightarrow b) \wedge (b \Rightarrow a)$$

Figure 2.5: Example Definitions

These are syntactic definitions[1] that allow us to introduce new symbols or constants. When we write a new definition we are stating that two expressions are semantically equivalent and can be interchanged.

Using a definition to rewrite an expression is known as *folding* or *unfolding*. An expression written using operators that are part of an axiomatised set can be *folded* to give an equivalent expression that contains defined operators. The expression $\neg(\neg a \vee \neg b)$ can be folded to $a \wedge b$, which in turn can be unfolded back to $\neg(\neg a \vee \neg b)$.

Definitions can be used in a proof to rewrite a line so that a particular rule can be used. When using a definition in a proof we write *folding* or *unfolding* as a justification, and a reference to the line that is being folded/unfolded.

[1]All definitions used can be found in Appendix A

Examples can be seen in the proofs in Section 2.5.

2.5 Building a Proof

The layout of a proof is intended to make it easy to read and understand. The obvious way to read a proof is to start from the hypotheses and work down to the *infer* line. Since each line builds upon previous ones it should be straightforward to follow the reasoning in the proof from the hypotheses down to the goal.

When writing a proof on paper it is usual to write the hypotheses at the top of the page, and the *infer* or conclusion line at the bottom of the page. It is then possible to work downwards from the hypotheses until you can infer the goal. This is known as *forward reasoning.* When working forward in a proof we attempt to match existing lines and hypotheses against the hypotheses of inference rules, allowing us to introduce a new line in the proof that matches against the conclusion of the inference rule.

As an example of building a proof using *forward reasoning* we will use the proof for the *and elimination left* inference rule. So, starting with the hypothesis and conclusion from the rule gives us:

$$\begin{array}{ll} \textbf{from} & a \wedge b \\ & \vdots \\ \textbf{infer} & b \end{array}$$

In another proof we might try to apply an *and elimination* rule, but that isn't an option here, since we are writing a proof of one of the *and elimination* rules. We do know, however, that *and* is defined in terms of *not* and *or*, so we can start by unfolding the hypothesis.

```
    from  $a \wedge b$
1         $\neg(\neg a \vee \neg b)$                unfolding(h1)
          $\vdots$
    infer $b$
```

The form of this new line can be matched against the hypothesis of the *not or elimination* inference rules. So, applying the left elimination rule will leave us with a the expression $\neg\neg b$.

```
    from  $a \wedge b$
1         $\neg(\neg a \vee \neg b)$                unfolding(h1)
2         $\neg\neg b$                              $\neg$-$\vee$-E-Left(1)
    infer $b$                                       $\neg\neg$-E(2)
```

The justification of the *infer* line uses the *not not elimination* rule to remove the double negation from line 2 and completes the proof.

Backwards Reasoning

Sometimes it is more convenient to build a proof backwards, starting from the goal, and working back to the hypotheses. When working backwards we try to match a line of the proof against the conclusion of an inference rule. Once a rule is selected it is added as a justification to the line, and additional lines that match the hypotheses of the inference rule are inserted into the proof.

As an example, we will use the proof for the *and introduction* inference rule. Starting with two terms, a and b, we want to be able to infer their conjunction.

$$\begin{array}{ll} \textbf{from} & a; b \\ & \vdots \\ \textbf{infer} & a \wedge b \end{array}$$

Looking at the proof outline it is difficult to know where to go from the hypotheses. But looking at the conclusions we know that *and* can be written in terms of *not* and *or*, by unfolding.

$$\begin{array}{lll} & \textbf{from} & a; b \\ & & \vdots \\ \alpha & & \neg(\neg a \vee \neg b) \\ & \textbf{infer} & a \wedge b & \qquad \text{folding}(\alpha) \end{array}$$

We add a new line to the proof that when folded will give us the infer line. This new line is labelled α, and will be numbered correctly once the proof has been filled out.

Line α can be matched against the conclusion of the *not or introduction* rule. This rule has two hypotheses, so we insert two new lines into the proof that will match against the hypotheses. We also add the *not or introduction* rule as a justification, along with references to the two new lines that have been added, to show which lines match the rule hypotheses.

After adding the two new lines we can see that the lines can be easily justified from the proof hypotheses using the *not not introduction* rule, completing the proof.

	from	$a; b$	
1		$\neg\neg a$	$\neg\neg$-I(h1)
2		$\neg\neg b$	$\neg\neg$-I(h2)
3		$\neg(\neg a \vee \neg b)$	$\neg$-$\vee$-I(1, 2)
	infer	$a \wedge b$	folding(3)

Notice that in the completed proof the lines of the proof have been renumbered so that they are numbered in descending order. This makes for a better presentation of the proof. Renumbering the lines of a proof also means that the line references in justifications should be adjusted to correspond to the appropriate line.

Middle-out Reasoning

When constructing a proof, the direction we work in is not important. We can work forwards from the hypotheses, backwards from the goal or a mixture of both. What is important is that the complete proof links the hypotheses and the goal through a series of justified steps. As long as this final property of the proof is met, the order in which these steps are added does not matter.

We might decide to start a proof by inserting a line "in the middle" of the proof; a line that is not linked to the hypotheses or the goal, and essentially becomes a sub-goal. We call this approach *middle-out reasoning* since we are starting in the "middle" of the proof. This method of inserting unlinked steps at any point of a proof was allowed in the *Mural* proof tool [JJLM91].

As with all of these approaches, the proof is only complete when all steps are correctly justified and link the hypotheses to the conclusion.

2.5.1 Subordinate Proofs

A *subordinate proof* (or *subproof*) is a proof that is nested within another proof. A subproof has the same form as a proof, with a *from* line listing hypotheses, an *infer* line stating the goal, and lines in between. Figure 2.6

shows the proof for the inference rule *and distributes over or*, which contains two subproofs on lines 3 and 4. Within the main proof a subproof is referenced as a single line. The hypotheses and conclusion of the *from* and *infer* lines of a proof are equivalent to a sequent. For example, line 3 in the proof can be written as the sequent $b \vdash (a \wedge b) \vee (a \wedge c)$.

```
      **from** $a \wedge (b \vee c)$
1          $a$                                          $\wedge$-E-Right(h1)
2          $b \vee c$                                   $\wedge$-E-Left(h1)
3          **from** $b$
3.1             $a \wedge b$                            $\wedge$-I(1, 3.h1)
           **infer** $(a \wedge b) \vee (a \wedge c)$   $\vee$-I-Right(3.1)
4          **from** $c$
4.1             $a \wedge c$                            $\wedge$-I(1, 4.h1)
           **infer** $(a \wedge b) \vee (a \wedge c)$   $\vee$-I-Left(4.1)
      **infer** $(a \wedge b) \vee (a \wedge c)$        $\vee$-E(2,3,4)
```

Figure 2.6: Proof for $\wedge$ distributes over $\vee$

The conclusion of the proof in Figure 2.6 is justified by the *or elimination* rule, which contains two sequents in its hypotheses (see Appendix A), requiring the proof to contain two subproofs.

As long as the subproof is not too long, the proof remains readable and it is convenient to have the subproof within the main proof. A subproof could be removed from the body of the main proof and made into a stand alone proof, and also made into an inference rule so that it can be referred to from another proof.

2.5.2 Valid Proof

A proof is *valid* when each line has a valid *justification*. A justification is valid if it refers only to preceding lines [BFL$^+$94], and uses only *proven* inference rules whose hypotheses match assertions on the line referred to, and whose conclusions match the body of the line being justified. An inference rule is *proven* if it is an axiom or it has a valid proof that uses only axioms on other proved inference rules.

For any justification in a proof it should be possible to trace dependance on any inference rule to a set of axioms, via the proof of the inference rule.

2.6 Quantifiers

A proposition can be formed by using truth valued functions [Jon90]. For example, the proposition $\textit{is-prime}(x)$ has the value *true* in those cases where x is a natural number that is a prime number, and *false* otherwise. So the proposition $\textit{is-prime}(3)$ is *true* since 3 is a prime number.

When dealing with multiple values, we may want to discuss properties that are common to *some* or *all* of the values. We could do this by combining propositions using conjunctions or disjunctions. If we wanted to say that at least one number in the set $\{2, 3, 4\}$ is a prime number, we could write the disjunction $\textit{is-prime}(2) \vee \textit{is-prime}(3) \vee \textit{is-prime}(4)$. Alternatively we could write

$$\exists x \in \{2, 3, 4\} \cdot \textit{is-prime}(x)$$

For larger sets of values it is obviously more convenient to use quantifiers, rather than writing out a disjunction. When dealing with infinite sets the use of quantifiers is essential. There are two types of quantifiers, *existential* and *universal.*

Existential Quantifiers

An *existential quantifier* is used to describe a property that applies to *some* of a set of values. This property might apply to only one of the values, or it might apply to all of them, or somewhere in between. An existentially quantified expression will evaluate to *true* as long as the property applies to at least one of the values.

The symbol for existential quantification ($\exists$) is usually read as *exists* or *there exists*. The below expression states that in the set $\{2, 3, 4\}$ there exists at least one value that satisfies the function *is-prime*.

$$\exists x \in \{2, 3, 4\} \cdot \textit{is-prime}(x)$$

The expression consists of an *existential quantifier symbol* ($\exists$), a *bound variable* (x), and a *predicate* (is-prime(x)). In the above example the variable x is bound to the values {2,3,4}, and any occurrences of x in the predicate will refer to the same x that is bound to these values.

It is possible for quantified expressions to bind more than one variable, as in the below example which states that in the set of values bound to x there exists a value that is also in the set of values bound to y. In other words, there is a value that is common to both x and y.

$$\exists x \in \{1, 2, 3\}, y \in \{3, 4, 5\} \cdot y = x$$

Universal Quantifiers

The form of both kinds of quantified expressions is the same, but with a different symbol. A quantified expression that uses the *universal quantifier* symbol states that in a set of values, some property applies to *all* of the values. The universal quantifier symbol is $\forall$, and is usually read as *for all.*

Taking the first example from the section on existential quantifiers and replacing the symbol gives us:

$$\forall x \in \{2, 3, 4\} \cdot \textit{is-prime}(x)$$

This expression states that for all the values in the set bound to x the function *is-prime* is satisfied. Although a valid expression, it is actually a *false* expression, since 4 is not a prime number.

The following quantified expression is a true expression that states that in the set of values {2,4,6} all values either satisfy the function *is-even*, or *is-odd*.

$$\forall x \in \{2,4,6\} \cdot \textit{is-even}(x) \lor \textit{is-odd}(x)$$

It is possible to combine both types of quantified expressions into a single expression as below.

$$\forall x \in \mathbb{N} \ \cdot \exists y \in \mathbb{N} \cdot y = x + 1$$

This expression states that for all natural numbers, there exists a natural number that is one greater. The order in which the quantifier and variable pairings are placed is important, as illustrated by the below expression.

$$\exists y \in \mathbb{N} \ \cdot \forall x \in \mathbb{N} \cdot y = x + 1$$

Writing the expression in this way actually makes the expression *false*, since it says that there is a natural number that is one greater than every other natural number.

Binding

A quantified expression binds a variable (or multiple variables) to a set of values. The scope of this binding is local to the expression, and a variable bound in a quantified expression cannot be referenced outside of the expression.

$$x > 5 \land \forall x \in \{1, \ldots, 10\} \cdot x > 0$$

For example, the above expression is a conjunction of a relational expression and a quantified expression. The x in the relational expression is a free variable and the x in the quantified expression is bound within the expression.

Confusion can be avoided by using a different name for the variable that is bound by the quantified expression.

$$x > 5 \land \forall y \in \{1, \ldots, 10\} \cdot y > 0$$

This expression is logically equivalent to the previous expression. Changing the name of the bound variable in the quantified expression does not change its meaning, but it does make it clear that the variables are not the same.

Substitution and Variable Capture

Syntactic substitution is the process of replacing an occurrence of a free variable in an expression with another variable or expression. For example, in the below expression we substitute all free occurrences of x with 3.

$$(x = y + z)(3/x) = (3 = y + z)$$

Bound variables are not affected by a substitution since substitution only replaces the free variables in an expression. So in the following expression the variable x bound by the quantified expression is unchanged by the substitution.

$$(x > 5 \land \forall x \in \mathbb{N} \cdot x > 0)(10/x) = (10 > 5 \land \forall x \in \mathbb{N} \cdot x > 0)$$

Care needs to be taken with the substitution of variables in expressions that contain bound variables to avoid *variable capture*. This occurs when a variable substitution causes confusion between bound and free variables in an expression. For example, substituting y for x in the below expression will lead to confusion between the bound and free variable in the quantified expression.

$$(y > 5 \land \forall x \in \{1, \ldots, 5\} \cdot x \neq y)(x/y)$$

This problem can be avoid by renaming the bound variable in the quantified expression, as explained earlier, before performing the substitution.

2.6.1 Quantifiers and Proofs

We can form axioms and inference rules for quantifiers, allowing us to reason about quantified expressions. As with other inference rules, we have *introduction* and *elimination* rules for each of the quantifiers. Figure 2.7 shows the introduction and elimination axioms for existential quantification.

$$\boxed{\exists\text{-E}}\ \frac{\exists x \in A \cdot P(x) \qquad y \in A, P(y) \vdash e}{e}$$

$$\boxed{\exists\text{-I}}\ \frac{a \in A \qquad P(a)}{\exists x \in A \cdot P(x)}$$

Figure 2.7: Exists introduction and elimination axioms

We can write a universally quantified expression using the existential quantifier, giving us the following definition

$$\forall x \in A \cdot P(x) \overset{def}{=} \neg\exists x \in A \cdot \neg P(x)$$

Using this definition and other axioms, we can derive inference rules for universal quantification. Figure 2.8 shows the proof of the *forall elimination* inference rule.

```
    from  $a \in A; \forall x \in A \cdot P(x)$
1         $\neg\exists x \in A \cdot \neg P(x)$                 unfolding(h2)
2         $\neg\neg P(a)$                                  ¬-∃-E(h1, 1)
    infer $P(a)$                                         ¬¬-E(2)
```

Figure 2.8: Proof of *forall elimination* rule

The example proof of a program in Section 4.3 uses inference rules for quantified expressions.

2.7 Consistency and Completeness

A *proposition* is a 'sentence' of expression that has a *true* or *false* value. A formal system is considered to be *consistent* if every proposition is either *true* or *false*, but not both [Heh84]. Consistency is essential since it makes no sense for a proposition to be both *true* and *false*.

A formal system is *complete* if every proposition is either *true* or *false*. This is not always essential since the system might not be descriptive enough to evaluate the truth value of every proposition. An incomplete system may still be useful if it can assign truth values to all of the propositions that we are interested in.

2.8 Summary

This chapter discusses logical frameworks in general, focusing on first order predicate calculus. It also covers inference rules and their use as justifications in natural deductions proofs.

This provides a logical foundation for the types of proofs that are written in this thesis using operational semantic rules as justifications. The next chapter will introduce such a proof, and then Chapter 4 will give further examples.

Chapter 3

Language Semantics

The semantics of a programming language describes the behaviour of the language and gives meaning to programs written in the language. There are three main approaches to defining a language semantics: *operational*, *denotational* and *axiomatic*.

The operational approach focuses on how program statements are executed and shows the effect execution has on an abstract state. The operational semantics of a program statement defines execution in terms of a transition of the statement and an initial state to a final state. This approach is covered in more detail in Section 3.4.

The denotational semantics of a language maps syntactic constructs to mathematical objects such as functions over states, and describes the effect that execution will have on a state. Denotational semantics are discussed in Section 3.3.

The original focus of the operational and denotational approaches was to provide a language definition that would be used as a basis for compiler development. The axiomatic approach introduced by Hoare in [Hoa69] was intended as a way of defining a language that would also allow programmers to prove properties about programs, and to “discharge their responsibility to write correct and efficient programs” [Hoa09]. The axiomatic approach describes the semantics of a language by defining axioms and inference rules for each of the statements in the language [Sto77]. These can then be used to prove properties of a program, specifically that a program is correct with respect to a specification. A program's specification is written as assertions about the values that variables will have after the program has finished executing, provided that certain stated conditions were met before execution.

This chapter, first of all, explains how programming languages are described by defining an *abstract syntax*, *context conditions* and *semantics* for the language. The definition of a simple sequential language, which we will call *Base*, is used throughout the chapter to illustrate the approach taken in this thesis to describe a programming language. The full language definition can be found in Appendix B. The main focus of the chapter is on the axiomatic, denotational and operational approaches to defining semantics for programming languages, with a specific focus on how the semantic descriptions can be used to reason about computer programs.

3.1 Describing Programming Languages

When specifying a programming language there are three main parts that are of interest: *abstract syntax*, *context conditions*, and *semantics*. The abstract syntax describes the structure of the various components of the language, such as statements and expressions. The context conditions restrict the set of programs allowed by an abstract syntax definition to only those programs to which we will assign a meaning. The semantics of a language describes its 'meaning', or what it means to execute a program that is written in that language.

In this section we will look at the abstract syntax and context conditions of the *Base* language (see Appendix B). This is a simple sequential language that consists of *If*, *While* and *Assignment* statements. Multiple statements are added to a program using the *Sequential* statement, which has a left and right statement which are executed sequentially.

3.1.1 Abstract Syntax

An *abstract syntax* defines the structure of a language, but not how to write statements in the language. For example we define an *If* statement as having a *condition* part, a *then* part and an *else* part, but we don't define the key words needed to write an unambiguous *If* statement. The key words used to write a statement are part of the *concrete syntax* of a language, and are used by the parser so that it can parse the text of a program. We are not concerned with the concrete syntax definitions of the languages used in this thesis.

When writing an abstract syntax for a language, the notation is borrowed from the Vienna Development Method (VDM) [Jon90, Daw91]. Language constructs can be specified using a VDM record, where the record fields represent the parts of the construct. Below is the abstract syntax definition for an *If* statement in the *Base* language.

$$\begin{array}{lcrcl} \mathit{If} & :: & \mathit{test} & : & \mathit{Expr} \\ & & \mathit{th} & : & \mathit{Stmt} \\ & & \mathit{el} & : & \mathit{Stmt} \end{array}$$

The *union type* is used to group language constructs together. The example below states that *If*, *Assign*, *While* and *Seq* are subsets of *Stmt*. So in the definition of the *If* statement above, the *th* and *el* parts of the *If* statement can be instantiated with any of the types in this union.

$$\mathit{Stmt} = \mathit{If} \mid \mathit{Assign} \mid \mathit{While} \mid \mathit{Seq}$$

To write context conditions and semantics we need to be able to instantiate these VDM records. This can be done using the record constructor, which we can think of as a function that takes the parts that make up a record, and returns an instance of that record. For example, the constructor for the *If* statement would have the following signature

$$\textit{mk-If}\colon \textit{Expr} \times \textit{Stmt} \times \textit{Stmt} \to \textit{If}$$

and we write the construtor as below.

$$\textit{mk-If}(\textit{test}, \textit{th}, \textit{el})$$

The abstract syntax definitions for the other statements in the language are given in Figure 3.1. The full language definition is given in Appendix B.

$$\begin{array}{lrcl} \textit{Assign} :: & \textit{lhs} & : & \textit{Id} \\ & \textit{rhs} & : & \textit{Expr} \end{array}$$

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

$$\begin{array}{lrcl} \textit{If} :: & \textit{test} & : & \textit{Expr} \\ & \textit{th} & : & \textit{Stmt} \\ & \textit{el} & : & \textit{Stmt} \end{array}$$

$$\begin{array}{lrcl} \textit{While} :: & \textit{test} & : & \textit{Expr} \\ & \textit{body} & : & \textit{Stmt} \end{array}$$

$$\begin{array}{lrcl} \textit{Program} :: & \textit{vars} & : & \textit{Id} \xrightarrow{m} \textit{Type} \\ & \textit{body} & : & \textit{Stmt} \end{array}$$

Figure 3.1: Abstract Syntax.

The body of a *Program* consist of a single *Stmt*, which may be a *Seq* statement which would allow the program to execute two statements sequentially. Nesting the *Seq* statements will allow a program to contain more *Stmt*s. The *vars* field contains variable declarations, with variable names being mapped to a type.

3.1.2 Context Conditions

An abstract syntax defines a context free grammar that does not prevent programs that are considered to be invalid from being written. For example, the abstract syntax of our language allows us to write

$$mk\text{-}If(3, mk\text{-}Assign(i, 4), mk\text{-}Assign(i, 5))$$

which we don't want to consider as a valid program[1]. This is an error of types, since we only want the condition on an *If* statement to be a *boolean* expression. A program that refers to an undeclared variable has no meaning and we will also consider it to be invalid.

Context conditions are used to restrict the set of possible programs to only those that we will assign meaning to, which we consider to be *well-formed*.

The usual way to write context conditions is as functions[2] that determine if a program is *well-formed*. Figure 3.2 shows the context conditions for a program. The second line of the function contains the parameter list, in which names are given to the parameters of the function using patterns [FL98]. A pattern may simply be a name, or in this case a *record pattern* is used which allows names to be given to the fields of the record.

A program consists of a single statement and a mapping of variable names to types which specifies the variables that are declared in the program. This mapping is referred to in this chapter as a 'type map'.

The *wf-Program* function takes a program as its only parameter and checks if the body of the program is a well-formed statement using the *wf-Stmt* function, providing it with the program's type map. A program is therefore *well-formed* if its body is *well-formed*, with respect to its declared variables.

Figure 3.3 shows part of the *wf-Stmt* function, along with its signature. This function takes a *Stmt* and a type map, and returns *true* if the statement is well-formed, or *false* otherwise.

[1]Some languages might just treat an integer as a boolean value, which could lead unexpected behaviour if the programmer wasn't aware they had provided a integer expression rather a boolean expression. To keep the definition of the *Base* language simple, it is strict on variable and expression types, and won't allow integer expressions where boolean expressions are expected.

[2]The functions are written in a VDM syntax.

$$
\begin{array}{l}
\textit{wf-Program}\colon \textit{Program} \to \mathbb{B} \\
\textit{wf-Program}(\textit{mk-Program}(\textit{vars}, \textit{body})) \triangleq \\
\qquad \textit{wf-Stmt}(\textit{body}, \textit{vars})
\end{array}
$$

Figure 3.2: Context Conditions for *Program*.

$$
\begin{array}{l}
\textit{wf-Stmt}\colon (\textit{Stmt} \times \textit{Id} \xrightarrow{m} \textit{Type}) \to \mathbb{B} \\
\textit{wf-Stmt}(\textit{mk-If}(\textit{test}, \textit{th}, \textit{el}), \textit{tpm}) \triangleq \\
\qquad \textit{typeof}(\textit{test}, \textit{tpm}) = \textsc{Bool} \wedge \\
\qquad \textit{wf-Stmt}(\textit{th}, \textit{tpm}) \wedge \\
\qquad \textit{wf-Stmt}(\textit{el}, \textit{tpm})
\end{array}
$$

Figure 3.3: Context Conditions for *If* statement.

The *wf-Stmt* function has multiple cases, one for each type of statement in the language. The cases are distinguished by the record pattern used in the parameter list of the function. The example in Figure 3.3 shows the case for an *If* statement. An *If* statement is well-formed if its condition expression will give a boolean result, and if the *then* and *else* parts of the statement are both well-formed statements.

The context conditions for the other statements in the language are given in Figure 3.4. The full language definition is given in Appendix B.

$$
\begin{array}{l}
\textit{wf-Stmt}\colon (\textit{Stmt} \times \textit{Id} \xrightarrow{m} \textit{Type}) \to \mathbb{B} \\
\\
\textit{wf-Stmt}(\textit{mk-Assign}(\textit{lhs}, \textit{rhs}), \textit{tpm}) \triangleq \\
\qquad \textit{lhs} \in \textit{tpm} \wedge \\
\qquad \textit{typeof}(\textit{rhs}, \textit{tpm}) = \textsc{Int} \\
\\
\textit{wf-Stmt}(\textit{mk-Seq}(\textit{sl}, \textit{sr}), \textit{tpm}) \triangleq \\
\qquad \textit{wf-Stmt}(\textit{sl}, \textit{tpm}) \wedge \\
\qquad \textit{wf-Stmt}(\textit{sr}, \textit{tpm}) \\
\\
\textit{wf-Stmt}(\textit{mk-While}(b, s), \textit{tpm}) \triangleq \\
\qquad \textit{typeof}(b, \textit{tpm}) = \textsc{Bool} \wedge \\
\qquad \textit{wf-Stmt}(s, \textit{tpm})
\end{array}
$$

Figure 3.4: Context Conditions for Statements.

The *typeof* function is an auxiliary function used by *wf-Stmt* to find the type of an expression. This is determined by looking at the relations and variables used in the expression. The definition for *typeof* can be found in Appendix B.

3.2 Axiomatic Semantics

The axiomatic approach aims to describe the semantics of a programming language in a way that allows proofs about programs to be easily constructed. A proof of a program shows that the program (or part of a program) is correct in the sense that it satisfies its specification.

Rather than describe the execution of a statement, axiomatic semantics focuses on what effect the execution of a statement has on the values of state variables, and what assertions can be made about them.

The specification of a program is usually written as a pair of assertions about the values of certain variables before and after execution. These assertions don't generally state what a variable's value will be, but will describe the general properties of the variable's value, and also how the variable relates to other variables.

For simple languages it has become normal to present the axiomatic semantics in terms of concrete syntax, and we follow this convention in this chapter.

Hoare's work was an extension of work by Floyd [Flo67] who annotated flowcharts representing programs with assertions, and developed rules for reasoning about the flowcharts with respect to the assertions [Win94]. Turing had also used annotated flowcharts to reason about programs in [Tur49].

3.2.1 Hoare Logic

Hoare's paper [Hoa69] introduced an axiomatic approach to giving a language definition, and also Hoare logic for reasoning about programs using the language definition. The approach was originally introduced for a simple sequential language. The use of axiomatic semantics for sequential languages is discussed in Section 3.2.3, and Section 3.2.4 discusses work to apply the approach to more complex languages.

Hoare's paper introduced the notation known as the Hoare triple, which states the relationship between a *precondition* P, a *program* S and a *postcondition* Q.

$$\{P\}\ \mathrm{S}\ \{Q\}$$

The Hoare triple states that in the cases where P holds before execution, after S is executed, Q will hold. If S is executed in a state where P does not hold, then we cannot be certain that Q will hold after execution.

Hoare logic originally only dealt with *partial correctness*, which does not include termination, so we can only state that the property Q will hold if the program terminates [NN07]. Work to include termination is discussed in Section 3.2.3.

An axiomatic semantics of a language consists of axioms and inference rules, usually with one axiom per primitive statement and one inference rule per composite statement [BJ82]. The inference rules have the same form as those introduced in Section 2.3, when hypotheses are above the line, and the conclusion is below the line.

This section presents a standard set of inference rules for a simple sequential language, and explains their meaning.

Assignment

$$\frac{}{\{P[\mathrm{E}/x]\}\ x := \mathrm{E}\ \{P\}}$$

Figure 3.5: Assignment Axiom.

The most basic axiom in Hoare logic is the axiom for the assignment statement, shown in Figure 3.5. This simply states that if we want to know that the property P holds after execution, then the property (with all free occurrences of x replace by E) should hold before execution. Or in other words, the pre-condition comes from the post-condition by substituting E for all free occurrences of x in P.

In some languages it may be difficult to write an axiom for assignment that shows that other state variables are left unchanged by an assignment to x. This is known as the *frame problem.* Difficulties in solving the frame problem for Hoare logic inspired work on Separation logic [Rey02], which restricts the state considered when reasoning about a program statement to only those variables that are accessed by the program.

Sequential Composition

$$\frac{\begin{array}{l}\{P\}\ \mathrm{S}_1\ \{Q\}\\ \{Q\}\ \mathrm{S}_2\ \{R\}\end{array}}{\{P\}\ \mathrm{S}_1\ ;\mathrm{S}_2\ \{R\}}$$

Figure 3.6: Sequential composition rule.

This rule states that if the precondition P holds before execution, then R will hold after execution, as long as there is a predicate Q that will hold after the execution of S_1, and if S_2 is executed in a state where Q holds, provided S_2 terminates, the predicate R will hold in the final state.

Conditional

$$\frac{\begin{array}{l}\{P\ \wedge\ C\}\ \mathrm{S}_1\ \{R\}\\ \{P\ \wedge\ \neg C\}\ \mathrm{S}_2\ \{R\}\end{array}}{\{P\}\ \textbf{if}\ C\ \textbf{then}\ \mathrm{S}_1\ \textbf{else}\ \mathrm{S}_2\ \{R\}}$$

Figure 3.7: Conditional rule.

The conditional rule states that if the statement '**if** C **then** S_1 **else** S_2' is executed in a state where the precondition P holds and if the statement terminates, then in the final state after execution, R will hold if two conditions are met. First, if the statement $S1$ is executed in a state where P holds, and also where the condition C holds, then after execution the postcondition R will hold. Second, if the statement S_2 is executed in a state where P holds, but C does not, then if S_2 terminates, R will hold in the final state.

Consequence

$$\frac{\begin{array}{l} P' \Rightarrow P \\ \{P\}\ S\ \{Q\} \\ Q \Rightarrow Q' \end{array}}{\{P'\}\ S\ \{Q'\}}$$

Figure 3.8: Consequence rule.

The consequence rule[3] states that we can replace the precondition, a long as the new precondition P' implies the old precondition P. We can also replace the postcondition Q, as long as the new postcondition Q' can be implied from Q. This means that we can strengthen the precondition P, and weaken the postcondition Q [NN07].

Iteration

$$\frac{\{P \ \wedge \ C\}\ \mathrm{S}\ \{P\}}{\{P\}\ \textbf{while}\ C\ \textbf{do}\ \mathrm{S}\ \{P \wedge \neg C\}}$$

Figure 3.9: Iteration rule.

The iteration rule deals with *while* loop statements. This rule makes use of a invariant P, which represents a property that must hold before and after each execution of the body of the *while* statement, S. Provided that the invariant P and the loop condition C both hold, and we can show that after each execution of S the invariant P will hold, then we know that once the

[3]Also known as the *Weakening rule*

loop terminates the invariant P will still hold, and the loop condition C will no longer hold.

3.2.2 Example

This section presents an example of using Hoare logic to construct a proof of a simple program. The example program, consisting of a single *if* statement, simply takes the absolute value of x and stores it in the variable r. The program works on the assumption that both x and r have been declared as integer variables.

$$\{\textbf{true}\}\ \textbf{if}\ x < 0\ \textbf{then}\ r{:=}-x\ \textbf{else}\ r{:=}\ x\ \{\mathrm{r} \geq 0\}$$

The pre-condition is just **true**, since there are no requirements on the initial state for the execution of the program. The post-condition states that in the final state, after execution, the value of r should be greater than or equal to zero (since it should be the absolute value of x).

Figure 3.10 goes through a proof of the example program, using Hoare axioms and rules. The goal of the proof is the final line. Steps in the proof are justified using the axioms given in Section 3.2.1, and may also refer to other lines in the proof.

Extra space between lines 4 and 5 of the proof was added to show that there are essentially two parts to the proof. Lines 1 to 4 deal with the statement that is executed in the *then* part of the *if* statement (S1 in the conditional rule, Figure 3.7). Lines 5 to 8 deal with the statement that is executed in the *else* part of the *if* statement (S2 in the conditional rule).

The proof was written by working backwards from the *if* statement using the conditional rule, to give lines 4 and 8. The program statement in line 4 is an *assignment* statement so the assignment axiom is used which gives us line 1. Applying the assignment axiom gives us a different precondition than the one we are looking for, so we strengthen the precondition in line 2 so that we can use the consequence rule on line 4. Line 3 is needed for the third hypothesis of the consequence rule, which is the weakening of the

1.	$\{-x \geq 0\}\ r{:}= -x\ \{r \geq 0\}$	Assignment Rule
2.	$\mathbf{true} \wedge x < 0 \ \Rightarrow\ -x \geq 0$	
3.	$r \geq 0 \ \Rightarrow\ r \geq 0$	
4.	$\{\mathbf{true} \wedge x < 0\}\ r{:}= -x\ \{r \geq 0\}$	Consequence(2,1,3)
5.	$\{x \geq 0\}\ r{:}= x\ \{r \geq 0\}$	Assignment Rule
6.	$\mathbf{true} \wedge \neg\,(x < 0) \ \Rightarrow\ \{x \geq 0\}$	
7.	$r \geq 0 \ \Rightarrow\ r \geq 0$	
8.	$\{\mathbf{true} \wedge \neg\,(x < 0)\}\ r{:}= x\ \{r \geq 0\}$	Consequence(6,5,7)
	$\mathbf{infer}\ \{\mathbf{true}\}\ \mathbf{if}\ x < 0\ \mathbf{then}\ r{:}= -x\ \mathbf{else}\ r{:}= x\ \{r \geq 0\}$	Conditional Rule(4,8)

Figure 3.10: Proof of program in Hoare logic.

postcondition, although in this case we just keep it the same. Lines 5 to 8 follow the same pattern as line 1 to 4, but for the *else* case of the *if* statement.

3.2.3 Assessment

There are a number of well known problems associated with reasoning about programs using an axiomatic semantics. This section covers some of these issues, along with work to overcome the problems.

A known problem with Hoare logic is when a location in memory is pointed to by two variables. If the value of one of the variables is modified by an assign statement, then the value of the other variable will also be updated since they point to the same location, which may be unexpected. This is known as *aliasing*.

The example program below is used to illustrate this, and consists of a single *assignment* statement, assigning a value to an array location.

$$\{a[j] = 2\}\ a[i]{:}= 5\ \{a[i] = 5 \wedge a[j] = 2\}$$

Where the assignment axiom does not work for this program is when the values of i and j are equal, meaning the expressions $a[i]$ and $a[j]$ point to the same array location. The postcondition of the program will not hold after execution since the value pointed to by $a[j]$ had been changed.

Since the postcondition will only hold in the cases where i and j are not equal, this needs to be added as an extra condition to the precondition.

In Hoare logic, preconditions and postconditions are predicates over only one state (initial or final state). The precondition makes assertions about the values of program variables in the initial state before execution. The postcondition makes assertions about the values of program variables in the final state after execution, and usually cannot refer to the original value (before execution) of a variable. For example, the following program swaps the values of the variables x and y.

$$r := x \ ; x := y \ ; y := r$$

For the postcondition of this program, we want to be able to say that values of x and y have been swapped. This would give a post condition that looks something like this

$$(\text{old})x = (\text{new})y \wedge (\text{old})y = (\text{new})x$$

This is not possible in traditional Hoare logic, since in the postcondition we only have access to the final value of the variables. One solution for this problem is the use of *logical variables* that are only used in the assertions about a program, and not in the program itself. Their only purpose is to remember the initial values of variables, and their values will not change during execution since they are not used in the program [NN07]. For this example, using logical variables would give

$$\{x = n \wedge y = m\} \ r := x \ ; x := y \ ; y := r \ \{x = m \wedge y = n\}$$

In VDM, postconditions are assertions over two states (initial and final), so in the postcondition we are able to access the values of variables in the initial state. For the above program, this gives

$$\{\mathbf{true}\}\ r := x\ ; x := y\ ; y := r\ \{x = \overleftharpoondown{y} \wedge y = \overleftharpoondown{x}\}$$

where $\overleftharpoondown{x}$ and $\overleftharpoondown{y}$ refer to the values of x and y in the initial state.

Since a precondition is a predicate of one state and a postcondition is a predicate of two states, a postcondition of one statement cannot serve as a precondition to the next statement, making a rule for sequential composition more complicated. In fact, using postconditions of two states made a set of proof rules which "were clumsy in comparison to those in [Hoa69]" [Jon03b].

We are unable to use Hoare logic to reason about concurrent programs, since the logic would no longer be compositional [dRdBH+01]. The difficulty with concurrent programs comes when parallel threads interfere with each other, as it is difficult to know how the interference will affect an executing thread. In the below program consisting of two assignment statements executed in parallel, we cannot determine what the value of x will be after execution.

$$\{x = 1\}\ x{:}= x + 1\ ||\ x{:}= 5\ \{x = ?\}$$

We can tell by inspection that the value of x after execution will either be 2, 5 or 6, depending on the granularity of the concurrency, and on how parallel threads interact. For the above example we could write a postcondition such as $x \in \{2, 5, 6\}$, but for other programs it might not be possible or desirable to work out how the parallel threads interact. It is difficult to formalise this in Hoare logic since we only reason about the initial and final states of a program. To be able to reason about the interactions of concurrent threads we also need to reason about intermediate states during execution.

In [Owi75], the *interference freedom test* is introduced which uses assertions at intermediate points to reason about the interference of parallel threads. If the execution of each thread preserves the assertions, then we can reason about the program as a whole.

Another approach is the use of rely/guarantee conditions to make assertions about interference [Jon81, Jon83]. As well as pre and postconditions, a process is annotated by *rely* and *guarantee* conditions, as shown below.

$$\{P, R\}\ C\ \{G, Q\}$$

The rely condition R records what assumptions can be made about interference from other processes. The guarantee condition G states what interference will be generated by the process [Jon03a]. So, in a state in which the precondition holds, and if the interference from other processes meets the rely condition R, then the execution of C will result in a final state in which the postcondition Q holds, and any interference caused by C will respect the guarantee condition G.

As mentioned earlier, Hoare logic traditionally deals only with partial correctness, meaning the assertions we make about a program will only hold if the program terminates. We refer to such assertions as *partial correctness properties* [NN07]. A *total correctness property* is an assertion that states that a program will terminate and some property will hold after termination.

It is possible to extend the Hoare axioms and rule to include total correctness, as shown in [NN07]. The Hoare triple is modified to give

$$\{P\}\ S\ \{\Downarrow Q\}$$

which states that if S is executed in a state where the precondition P holds, then S is guaranteed to terminate, and Q will hold in the final state after execution.

The *while* rule is modified to include a *variant function* which states that the loop is well-founded and that it will terminate after a finite number of iterations.

3.2.4 Programming Languages and Axiomatic Semantics

There have been several attempts to give an axiomatic semantics for an imperative programming language. Most attempts required certain features to be restricted or removed from a language, and so were only able to give an axiomatic semantic definition for a subset of the language. One early attempt was to give an axiomatic definition for Pascal [HW73], which defined axioms and inference rules for most of the language except for features such as real arithmetic and *goto* statements, and also did not allow side effects in the evaluation of functions and the execution of statements.

The only complete language with an axiomatic semantics is the Turing language [HMRC88]. Certain syntactic restrictions are imposed on the language such as preventing aliasing of variables and eliminating side effects in expressions. Also, the language definition does not include concurrency and error handling.

3.3 Denotational Semantics

Denotational semantics is an approach to defining the meaning of a programming language by giving it mathematical meaning and describing the effect of execution, in terms of an association between initial and final states. The denotational semantics of a language is defined by associating an appropriate mathematical object with each syntactic construct of a language [NN07]. The mathematical object is called the denotation of the syntactic construct. A “semantic valuation function” maps syntactic constructs in a language to the abstract values which they denote. Semantic valuation functions are usually defined compositionally, so that composite terms are defined in terms of the denotations of their sub-terms [Sto77].

Since execution of a program is described in terms of the effect it has, rather than how it is executed, a language description will include a state definition that relates variables names to values [BJ82]. For example,

$$\sigma\colon Id \rightarrow Value$$

We can now define the denotations of program statements as a partial function from state to state.

$$M\colon st \rightarrow (\Sigma \rightarrow \Sigma)$$

One of the main difficulties of denotational semantics is finding suitable mathematical objects as denotations. For example, a looping statement requires a fixed point that always exists and is unique [BJ82].

For simple language descriptions, a denotational and operational semantics would be very similar, and proving properties about programs in a simple language should be quite straightforward using a denotational semantics. The reason for not wanting to use denotational semantics to reason about programs is the more complex mathematics that is required for certain language features. For example, defining the passing of procedures requires domain theory [Sto77], and concurrency in a language requires power domains [Plo76]. While this additional mathematics might be required to prove properties of a language itself, it is not necessary for most proofs that programs written in the language satisfy their specifications.

We therefore choose to use operational semantics, which we describe below.

3.4 Operational Semantics

Operational semantics was one of the earliest approaches to defining the semantics of programming languages [Jon03c]. It is focused on the behaviour of a program and defines the transitions that can be made by a program, from an initial state to a final state. McCarthy introduced the idea of an "abstract interpreter" which used a recursive function that takes a program and starting state and returns (if possible) a final state [McC66]. One of the difficulties with this approach is that the definitions tended to have too much information in the state, resulting in a "grand state" which causes difficulties in deducing properties about a language definition [BJ82, Jon03c].

Plotkin's Århus notes [Plo81] introduce the inference rule style of presenting the operational semantics of a language. Structural operational semantics still describes the semantics of a programming language in terms of 'execution' on an abstract machine, but avoids complexity in the configurations by the use of rules [Plo04]. The choice of which rule to apply to a program abstracted away to the meta-level [Jon03c], making the choice implicit rather than explicit in the semantic definition. This approach to defining semantics also allows non-determinacy to be conveniently added to a language description, as discussed in Section 3.4.7.

This section covers the operational approach to defining language semantics, and introduces operational semantics for the *Base* language. Section 3.4.4 shows how the operational semantics of a language can be used to reason about a program, with the use of a simple example. Further examples are found in Chapter 4.

3.4.1 Semantic Objects

Semantic objects are additional structures that are needed to describe the dynamic behaviour of a language, and are used to model parts of the system such as the memory store.

The essence of an imperative programming language is that a program written in the language will change some 'state'. The representation of the memory store, or state, will vary between languages, and it is an important part of the description of the language, since operational semantics describes execution in terms of the effect a statement has on the state.

For most languages the state is normally some (possibly indirect) association of variable names and values. The simple sequential language under discussion in this chapter has a state definition that is a direct mapping of variable names to integer values.

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

A more complex state definition could be made by introducing a secondary mapping which we will call an *environment*. The environment maps variable identifiers to *pointers*, or *references*, then the state maps references to values.

$$\begin{aligned} &Env = Id \xrightarrow{m} Ref \\ &\Sigma = Ref \xrightarrow{m} Value \end{aligned}$$

This definition allows for the possibility of having two variable identifiers mapping to the same *Ref*, meaning both variables point to the exact same *Value*, and if the value of one of the variables was changed, then so would the value of the other.

3.4.2 Operational Semantic Rules

Plotkin-style rules [Plo81] are used to define operational semantics. Each rule describes the execution of language statements in terms of abstract states; describing how an initial state is changed to a final state. Figure 3.11 shows an example of a semantic rule that describes how to execute an *If* statement when the test evaluates to *true*.

$$\boxed{\text{If-T}}\ \frac{\begin{array}{l}(test, \sigma) \xrightarrow{e} \mathbf{true}, \\ (th, \sigma) \xrightarrow{s} \sigma'\end{array}}{(mk\text{-}If(test, th, el), \sigma) \xrightarrow{s} \sigma'}$$

Figure 3.11: If-True semantic rule.

The arrows in the rule are *semantic transition relations*, which are annotated with letters to distinguish them, with the letters corresponding to the labels in a labelled transition system. The $\xrightarrow{s}$ relation defines statement transitions, relating a program statement and a state to another state. The $\xrightarrow{e}$ relation defines the evaluation of expressions, relating expressions and a state to a value (see Section 3.4.3).

We can think of each semantic rule as having certain conditions that must be met before the rule can be applied, with the conclusion line stating the result of execution. It is useful to read semantic rules, starting from the bottom left, in a clockwise direction [Plo04]. So looking at Figure 3.11, starting from the bottom left we can see that this rule applies to *If* statements. Then looking above the line, the rule has two conditions that need to be met before the rule can be applied. Firstly the condition *test* must evaluate to *true*, then the execution of the *th* part will result in some state σ' (which might actually be the same as σ). If these conditions are met, then the result of execution in this case is a (possibly) modified state σ'. Exactly how the state might be modified will depend on what is executed in the *th* part of the statement.

If we define one or more semantic rule that can be applied to the same statement under the same conditions, but have different effects on the state, then we cannot say for certain which rule will be applied in the execution of the statement, since the choice of which rule to apply is implicit in the semantic definition and is abstracted away to the meta-level [Jon03c]. This makes the execution of the statement non-deterministic since we can not know before execution which rule will be used in the execution of the statement. For example, the *If* statement has two semantic rules, one for each of the possible values of the statement's condition. There will never be a state in which both rules may be applied, since the condition will evaluate to either *true* or *false*. We could introduce a new semantic rule for the *If* statement that ignored the value of the *If* statement and always executed the *th* part. If the condition of the *If* statement evaluates to *false*, then we cannot say for certain which part of the *If* statement will be executed, since we have two rules that may apply.

Introducing non-determinism into a language description can be useful for statements such as the *Guarded Do* (see Section 4.3), or for modelling concurrency (see Section 3.4.7).

The transition arrows can be viewed as infix relations, with the type of the *s* arrow being defined as below.

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

The $\xrightarrow{s}$ transition arrow defines a relation between a *configuration* and a state, where a configuration is a statement and a state. This definition for statement execution allows us to relate a program statement and all of its possible starting states, to all of its possible final states. This allows us to introduce non-determinism into our semantic definitions which is required for describing concurrency in a language. It is however just as useful to use such relations when defining sequential languages.

Figure 3.12 gives the full semantics for the statements of the *Base* language. This language will be used in examples later in this chapter. A full language description of *Base* is given in Appendix B.

3.4.3 Operational Semantics for Expressions

Discussion so far has been restricted to semantics for *statements*, but the semantics of a language must also include a description of expression evaluation. As with the semantics for statements, the semantics for expression evaluation is defined using Plotkin rules.

The abstract syntax for the expressions of the simple sequential language is given in Figure 3.13.

As mentioned above, an expression and a state is related to a value by a semantic transition relation. The definition for the relation used in the example sequential language is given below.

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

Variable names are of type *Id*, which is a subtype of *Expr*, so that variable names can be used in expressions. A simple example of an expression semantic rule is a rule for looking up the value of a variable, shown in Figure 3.14. The rule states that if we have an expression that is of type *Id*, then it is evaluated by looking up its value in the state, so the value returned is whatever value *id* maps to in the state σ. We know that *id* exists in the state because the context conditions make sure that variables used in expressions are declared (and so will be in the state).

In the language we have a *RelExpr* type, which is a relational expression,

$$\boxed{\textsf{If-T}}\ \frac{(test, \sigma) \xrightarrow{e} true \quad (th, \sigma) \xrightarrow{s} \sigma'}{(mk\text{-}If(test, th, el), \sigma) \xrightarrow{s} \sigma'}$$

$$\boxed{\textsf{If-F}}\ \frac{(test, \sigma) \xrightarrow{e} false \quad (el, \sigma) \xrightarrow{s} \sigma'}{(mk\text{-}If(test, th, el), \sigma) \xrightarrow{s} \sigma'}$$

$$\boxed{\textsf{Assign}}\ \frac{(rhs, \sigma) \xrightarrow{e} v}{(mk\text{-}Assign(lhs, rhs), \sigma) \xrightarrow{s} \sigma \dagger \{lhs \mapsto v\}}$$

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

$$\boxed{\textsf{While-T}}\ \frac{(test, \sigma) \xrightarrow{e} true \quad (body, \sigma) \xrightarrow{s} \sigma' \quad (mk\text{-}While(test, body), \sigma') \xrightarrow{s} \sigma''}{(mk\text{-}While(test, body), \sigma) \xrightarrow{s} \sigma''}$$

$$\boxed{\textsf{While-F}}\ \frac{(test, \sigma) \xrightarrow{e} false}{(mk\text{-}While(test, body), \sigma) \xrightarrow{s} \sigma}$$

Figure 3.12: Semantic rules for the *Base* language.

that is used to compare two expressions using some operator such as $=, >, <$ etc. Figure 3.15 shows the semantic rule for a *RelExpr* that uses the EQUALS relation, which checks two expressions for equality.

The rule, first of all, evaluates the two expressions in the state σ to obtain two values, $v1$ and $v2$. The result of evaluating the *RelExpr* then becomes the result of comparing the two values $v1$ and $v2$ for equality.

Notice that in these example expression semantics the state is not modi-

$$Expr = Id \mid Value \mid ArithExpr \mid RelExpr \mid UnaryExpr$$

$$Value = \mathbb{N}$$

$$\begin{aligned} ArithExpr \; :: \; opd1 &: Expr \\ op &: \textsc{Plus} \mid \textsc{Minus} \\ opd2 &: Expr \end{aligned}$$

$$\begin{aligned} RelExpr \; :: \; opd1 &: Expr \\ op &: \textsc{Equals} \mid \textsc{NotEquals} \mid \text{LT} \\ opd2 &: Expr \end{aligned}$$

$$\begin{aligned} UnaryExpr \; :: \; opd &: Expr \\ op &: \textsc{Minus} \end{aligned}$$

Figure 3.13: Abstract syntax for Expressions.

$$\boxed{\mathsf{Id}}\ \frac{id \in Id}{(id, \sigma) \xrightarrow{e} \sigma(id)}$$

Figure 3.14: Semantic rule for *Id*.

$$\boxed{\mathsf{Rel\text{-}Equals}}\ \frac{\begin{array}{l}(e1, \sigma) \xrightarrow{e} v1 \\ (e2, \sigma) \xrightarrow{e} v2\end{array}}{(mk\text{-}RelExpr(e1, \textsc{Equals}, e2), \sigma) \xrightarrow{e} v1 = v2}$$

Figure 3.15: Semantic rule for an Equality relation.

fied, so there are no side effects to expression evaluation. There is no possible way to modify the state with the above definition of $\xrightarrow{e}$. The expression semantic rules can only access the state to look up variables.

We could of course define an expression semantics that allows side effects by writing a different definition of the expression transition relation (which we will label e') so that it can return a (possibly modified) state. An example of a definition that would allow for side effects is as follows.

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

Adding side effects to expression evaluation is necessary if we want to be able to add procedure calls to an expression. The language definitions used in this thesis only use simple expression semantics that don't have side effects.

3.4.4 Operational Semantics and Proofs

When writing a proof about a program, the purpose is to show that the program meets its specification. As explained at the beginning of this chapter, the operational semantic definition of a language defines the behaviour of the language by defining the transitions that can be made by a program, and its effect on an abstract state. Given a program, we can use the semantics of the language to show that execution of the program in an initial state will result in a final state in which the required property holds.

We see the semantic rules as being used in a proof in the same way inference rules are used (see Section 2.3), to justify lines of the proof. By applying the semantic rules to a program in a proof, we are stepping through execution and showing how the initial state is changed by execution to give us a final state. It should then be possible to show that the required property holds in the final state.

As an example of using semantic rules in a proof we will use the same program used in Section 3.2.2. This program consists of a single *If* statement and simply takes the absolute value of x and stores it in the variable r. The program is also surrounded by a precondition and post condition.

$$\{\textbf{true}\}\ \textbf{if}\ x < 0\ \textbf{then}\ r := -x\ \textbf{else}\ r := x\ \{\mathrm{r} \geq 0\}$$

We continue to use the *Base* language for this example and for the proof we will re-write the program in an abstract syntax.

$$\{\textbf{true}\}\ \textit{mk-If}(x < 0, \textit{mk-Assign}(r, -x), \textit{mk-Assign}(r, x))\ \{\mathrm{r} \geq 0\}$$

The proof of the program is given in Figure 3.16 to show that the execution of the program in a state that satisfies the precondition will result in a state that satisfies the postcondition.

In this example program and in the proof, expressions are written using a concrete syntax rather than the abstract syntax, to make them easier to read. For example, an expression written as $x < 0$ should actually be written as $\textit{mk-RelExpr}(x, <, 0)$.

Since the value in r after execution should be the absolute value of x, then it should be greater than or equal to zero. The proof should show that after the program has finished executing, the value of r in the final state will be greater than or equal to zero.

From our definition of the language we know that the condition on the *If* statement is a boolean expression that will evaluate to either *true* or *false*. The result of the evaluated condition determines which branch of the *If* statement is executed, which is determined by the value of x. We need to show that whichever branch of the *If* statement is executed, our property will still hold. Lines 3 and 4 of the proof are subproofs that show that the proof holds for the cases where the condition on the *If* statement evaluates to *true* and *false* respectively.

Semantic rules are used in the proof in the same way we would use an inference rule, to justify lines in the proof. Line 3.3 is justified by application of the *Assign* semantic rule. To use a semantic rule as a justification the line of the proof must match the conclusion line of the semantic rule. The line number given as a parameter to the justification refers to the line that matches against the 'hypothesis' of the *Assign* rule. The semantic rules used in this proof can be found in Figure 3.12 and in Appendix B, which gives the whole language definition.

The proof doesn't just use semantic rules for the justifications, it also uses inference rules. Many of the inference rules used in this proof are given in Appendix B, except for the *or elimination* and other rules of propositional logic, which are found in Appendix A.

	from $(\mathit{mk\text{-}If}(x < 0, \mathit{mk\text{-}Assign}(r, -x), \mathit{mk\text{-}Assign}(r, x)), \sigma_0) \xrightarrow{s} \sigma_f$, $x \colon \mathit{Id}, r \colon \mathit{Id}$	
1.	$x < 0 \in \mathbb{B}$	
2.	$(x < 0, \sigma_0) \xrightarrow{e} \mathbf{true} \vee (x < 0, \sigma_0) \xrightarrow{e} \mathbf{false}$	
3.	**from** $(x < 0, \sigma_0) \xrightarrow{e} \mathbf{true}$	
3.1	$(x, \sigma_0) \xrightarrow{e} \sigma_0(x)$	Id(h2)
3.2	$(-x, \sigma_0) \xrightarrow{e} -\sigma_0(x)$	UnaryExpr(3.1)
3.3	$(\mathit{mk\text{-}Assign}(r, -x), \sigma_0) \xrightarrow{s}$ $\sigma_0 \dagger \{r \mapsto -\sigma_0(x)\}$	Assign(3.2)
3.4	$(\mathit{mk\text{-}If}(x < 0, \mathit{mk\text{-}Assign}(r, -x), \mathit{mk\text{-}Assign}(r, x)), \sigma_0)$ $\xrightarrow{s} \sigma_0 \dagger \{r \mapsto -\sigma_0(x)\}$	if-T(3.h1, 3.3)
3.5	$\sigma_f = \sigma_0 \dagger \{r \mapsto -\sigma_0(x)\}$	σ-Equiv(h1, 3.4)
3.6	$\sigma_f(x) = -\sigma_0(x)$	†-Equiv(3.5)
3.7	$-\sigma_f(x) = \sigma_0(x)$	
3.8	$\sigma_0(x) < 0$	LT-T(3.h1)
3.9	$-\sigma_f(x) < 0$	
	infer $\sigma_f(r) \geq 0$	
4.	**from** $(x < 0, \sigma_0) \xrightarrow{e} \mathbf{false}$	
4.1	$(x, \sigma_0) \xrightarrow{e} \sigma_0(x)$	Id
4.2	$(\mathit{mk\text{-}Assign}(r, x), \sigma_0) \xrightarrow{s} \sigma_0 \dagger \{r \mapsto \sigma_0(x)\}$	Assign(4.1)
4.3	$(\mathit{mk\text{-}If}(x < 0, \mathit{mk\text{-}Assign}(r, -x), \mathit{mk\text{-}Assign}(r, x)), \sigma_0)$ $\xrightarrow{s} \sigma_0 \dagger \{r \mapsto \sigma_0(x)\}$	if-F(4.h1, 4.2)
4.4	$\sigma_f = \sigma_0 \dagger \{r \mapsto \sigma_0(x)\}$	σ-Equiv(h1, 4.3)
4.5	$\sigma_f(r) = \sigma_0(x)$	†-Equiv(4.5)
4.6	$\sigma_0(x) \geq 0$	LT-F(4.h1)
	infer $\sigma_f(r) \geq 0$	=-sub-left(4.5, 4.6)
	infer $\sigma_f(r) \geq 0$	$\vee$-E(2, 3, 4)

Figure 3.16: Proof of Absolute Value program.

3.4.5 Big Step, Small Step

When specifying an operational semantics it is possible to define the size of the steps [Plo81] made by the semantic rules. We generally refer to an operational semantic definition as being a *big step* or a *small step* semantics.

Plotkin's work on Structural Operational semantics uses a small step semantics [Plo04]. Kahn uses a big step semantics in [Kah87], which he calls 'Natural Semantics'.

A big step means that the execution of a program statement will be defined in a single semantic rule. For example, the semantic rule in Figure 3.11

is an example of a semantic rule in a big step semantics and shows the execution of an *If* statement (in the case where the condition evaluates to *true*) in a single step, taking the program text and an initial state to a final state.

With a small step semantics, the size of the steps made by the semantic rules are smaller. A single semantic rule will generally only define a partial execution of a program statement, so it won't define the transition from an initial state to a final state, but will define the transition to some intermediate state. The transition relation for a small step semantics can be defined so that it takes a program and an initial state and takes that to a modified program text and a (possibly) modified state, as shown below. This transition relation is labelled s', to distinguish it from the statement transition relation used for the big step semantics earlier in the chapter.

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

The *Base* language used in this chapter has a big step semantics. To illustrate the differences between big and small step semantics, we will look at the semantics of the *Assign* and *If* statements in the *Base* language and compare them to small step semantics of equivalent statements.

Figure 3.17 shows the semantic rule for *Assign* statement in the *Base* language. The *assignment* statement is executed in a single step, giving a final state. The execution of an *assignment* statement basically consists of two tasks; 1) evaluate the expression, 2) assign value to variable. To define assignment in a small step semantics these two tasks could be used as the basis of the rules.

$$\boxed{\textsf{Assign}}\ \frac{(e, \sigma) \xrightarrow{e} v}{(mk\text{-}Assign(id, e), \sigma) \xrightarrow{s} \sigma \dagger \{id \mapsto v\}}$$

Figure 3.17: Assign Statement - big step rule.

Figure 3.18 shows the how the semantics for assignment statement would be defined in a language with a small step semantics. In this definition the semantics of *Assign* is separated into two rules; one rule for each of the tasks described above.

The Assign-E rule in Figure 3.18 uses a **nil** statement to show that execution of the assignment statement is complete. The **nil** statement is also used in the *Parallel* language (Appendix D) which has a small step semantics.

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

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

Figure 3.18: Assign Statement - Small step rules.

The first rule, Assign-Eval, evaluates the expression part of the assign statement and returns a modified assignment statement where the expression part is replaced by the result of evaluation. Notice that if the expression evaluation rules are also small step then the expression might only be partially evaluated, meaning the Assign-Eval rule might be applied several times before the expression is evaluated resulting in an actual value. Once the expression has been evaluated to an integer[4] then the Assign-E rule can be applied to the assignment statement. This rule modifies the state, updating the value of the variable *id*. Since this is the last step in the execution of the assignment statement, the rule results in a **nil** statement, to show that there is nothing left to execute.

Looking again at Figure 3.11 we can see that the execution of an *If* statement in the *Base* language consists of two parts; the evaluation of the condition, and the execution of the *th* or *el* parts. We could write our semantics in such a way that a rule only does one thing at a time, giving us a small step semantics.

[4]This language only allows integer variables

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

$$\boxed{\textsf{If-T}}\ \frac{}{(mk\text{-}If(\mathbf{true}, th, el), \sigma) \xrightarrow{s'} (th, \sigma)}$$

$$\boxed{\textsf{If-F}}\ \frac{}{(mk\text{-}If(\mathbf{false}, th, el), \sigma) \xrightarrow{s'} (el, \sigma)}$$

Figure 3.19: Semantic rules for *If* in a small step.

Figure 3.19 shows the semantics for an *If* statement in a small step. The evaluation of the condition may take a few applications of the *If-Eval* rule (depending on how the expression semantics are defined) so we say that the *If* statement will have a potentially modified *test*. Once the condition has been replaced with a boolean value one of the other rules can then be applied. Unlike the big step semantic rules, these rules do not actually execute the *then* or *else* parts of the *If* statement, but only decide which part should be executed. Both the If-T and If-F rules remove the *If* statement and leave the *th* or *el* part to be executed.

The examples in Chapter 4 use languages that have large step semantics as well as languages that have small step semantics, showing semantic descriptions of either size step can be used to reason about programs. Section 3.4.7 explains that a small step semantics is necessary to describe concurrent languages, so being able to reason about programs using a small step semantics is necessary for reasoning about concurrent programs.

3.4.6 Termination

We can see from the definition of the *Base* language that there is the potential for writing non-terminating programs. The program **while** *true* **do** $x := x+1$, is a valid program in the simple sequential language used in this chapter, but it is not a desirable program to write since it will never reach a state in which it can terminate.

From the semantics of the language in Figure 3.12, we can see that the only place in the language where we have the potential for non-terminating programs is in the *While* statement, due to the recursive call in the hypotheses of the While-T semantic rule.

Potentially the context condition for the *While* statement could be modified to try to eliminate non-terminating programs. For example, we could state that the condition on the *While* statement cannot be *true*. A more complex condition could be added that states that any variables in the condition of the *While* statement must appear on the left-hand side of an *Assign* statement in the body of the *While* statement. This condition however, would not be able to remove all non-terminating programs, and might also disallow valid terminating *While* statements.

If we are going to reason about a program containing a *While* statement in this language, then we need to be able to show that it will terminate. Once we can show that the program terminates, we can then talk about the state in which it will terminate.

3.4.7 Concurrency and Non-Determinism

In Section 3.4.2 we mentioned that using *transition relations* when defining the semantics of a language allowed us to introduce non-determinism into the language. This is useful because we need to use non-determinism in a semantic definition to model concurrency. If a program has multiple threads that are being executed concurrently, then the execution of these threads will be interleaved in such a way that cannot be determined prior to execution, and may be different each time they are executed.

Non-determinism is introduced into a language semantics by writing rules such that two or more rules may be applied to a given configuration. The choice of which rule to apply is not defined by the semantics, but is abstracted away to the meta-level [Jon03c].

A simple way to add concurrency to a language definition is to add a *parallel* statement. The *parallel* statement used here as an example is taken from the *Parallel* language which is a simple concurrent language, the definition of which is given in full in Appendix D. Section 4.4 gives an example of reasoning about a parallel program using this language definition.

The *parallel* statement has a *left* and a *right* statement which are to be executed concurrently. The definition of the *parallel* statement is given below.

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

The semantics of the *Par* statement is given in Figure 3.20. The non-determinism comes in the definitions of the *Par-L* and *Par-R* semantic rules which can both be applied to a *parallel* statement. The *Par-L* rule executes the *left* statement, and the *Par-R* rule executes the *right* statement. The rule *Par-E* is known as the parallel elimination rule, and is executed when both the *left* and *right* statements have finished executing. It simply returns a *nil* statement to indicate that the *Par* statement has completed execution.

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

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

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

Figure 3.20: Semantic rules for Parallel Statement.

Although the abstract syntax definition for the *parallel* statement only has a single left and right *Stmt*, we can add multiple *Stmt*s by using the *Seq* statement, which executes statements sequentially. This will allow us to have two threads of execution, each sequentially executing multiple statements.

If we look at the semantic rules for the *parallel* statement in more detail, we see that the *left* and the *right* side are not executed truly concurrently, but each statement is executed one 'piece' at a time, with the execution of theses pieces of both statements being interleaved in a non-deterministic way. We refer to these 'pieces' as *atomic actions*, and the size of these atomic actions is determined by the size of the steps in the semantic definition.

When we are dealing with a big step semantics we only have access to initial and final states. Using a small step semantics gives access to intermediate states during execution, allowing us to model interference from concurrent threads. For example, during the execution of an *assign* statement the value of a variable may have been changed by another thread between the expression being evaluated and the result being stored, causing unexpected results. Being able to access these intermediate states allows us to reason about such interference of concurrent threads of execution.

Section 4.4 works through a proof of a parallel program that consists of two parallel threads that search through an array, one working from the start, and the other from the end of the array, and both working towards the centre.

In [Col08, CJ07] a small step semantics is used to define a language that allows for fine-grained concurrency, which is used to reason about interference in concurrent programs.

3.5 Summary

This chapter started by explaining the use of abstract syntax and context conditions in describing programming languages, and then moved on to the main topic of the chapter, semantics of programming languages. As a background to the work of this thesis, the chapter has covered the three main approaches of defining semantics for programming languages – axiomatic, denotational and operational semantics – with a focus on operational semantics.

The section on operational semantics introduced an approach to reasoning about a computer program using the operational semantics of the programming language that it is written in. Chapter 4 gives more examples of using operational semantic rules in proofs of a program, and subsequent chapters argue that this method becomes more useful with appropriate tool support.

Chapter 4

Examples

This chapter presents examples of using operational semantics to reason about a program, starting with a simple example and then moving on to more complicated examples. Each example specifies a program with a precondition and a postcondition, and gives a proof of the program using the semantics of the language that the program is written in.

The examples in this chapter make use of two languages – *Simple* and *Parallel*– each with different features to allow a good range of examples, and both languages kept relatively simple so that the examples are not affected by complex language details. It is worth explaining the features of the language here before moving into the examples. The definitions of these languages are not given in this chapter, but full language definitions are given in the appendices.

The first language used in this chapter is the *Simple* language which is a sequential languages based on the *Base* language introduced in Chapter 3, with the addition of the *Swap* and *Guarded Do* statements and one-dimensional arrays. The *Swap* statement swaps the values of two variables, without needing to use an intermediate variable. The *Guarded Do* statement is a looping statement that can have multiple guards, each with an attached statement. For each execution of the loop a statement whose guard evaluates to *true* will be non-deterministically selected and executed. The loop terminates when all guards evaluate to *false*. The language also contains the standard statements such as *Assign*, *If*, and *While*. Statements are sequentially composed using the *Seq* statement, which consists of a left and a right statement. *Seq* statements can also be nested inside other *Seq* statements.

The assignment statement can be used to assign values to array locations as well as to variables, and array locations can be used in expressions. The state definition maps variable identifiers to an integer value or an array value. The full language definition can be found in Appendix C.

Parallel is a simple concurrent language that provides concurrency though a *parallel* statement that consists of two statements that are executed in parallel. To allow concurrency and interference of concurrent threads to be modelled, the language semantics are defined using a small-step, so that a statement is not executed in a single step but in a series of small steps[1]. The language also includes *Assign*, *If*, *While*, *Swap* and *Seq* statements as well as one-dimensional arrays. To prevent interference from concurrent threads during the evaluation of an expression and the storing of the result, the *Assign* statement is defined using a big-step semantics so assignment is atomic.

The *Parallel* language also contains a **nil** statement which shows that a statement has completed execution. The execution of a statement in a small-step semantics is done over a number of steps, each step resulting in a slightly modified program text and a potentially modified state. When the statement has finished executing it ends in a **nil** statement and a state. Statements that contain nested statements, such as the *If* and *Seq* statements do not result in a **nil** statement, but instead modify the program text so that it consists of one of the nested statements. In the case of a *Seq* statement, once the left statement results in a **nil** statement, the Seq-E rule removes the *Seq* statement resulting in the program text consisting of just the right statement.

[1]See Section 3.4.5 for more detail on big-step and small-step semantics.

4.1 Variable Swap

For the first example the *Simple* language is used, as defined in Appendix C. The example is of a program that swaps the values of the variables x and y, using the variable r as a temporary variable, working on the assumption that all three variables have been declared previously. For this example it is also important that the three variables are distinct variables, and not aliases of the same variable. We know that the variables are distinct since the definition of the language does not allow variable aliasing, since the state is modelled as a mapping of *Id* to *Value*.

We would like to show that after execution of the program has completed, the value of x is equal to the old value (before execution) of y, and the value of y is equal to the old value of x. The precondition of the program is simply *true*, since there are no restrictions on the values of the variables before execution.

$$\{\mathbf{true}\}\ r := x\ ; x := y\ ; y := r\ \{x = \overleftarrow{y} \wedge y = \overleftarrow{x}\}$$

In the postcondition we have access to the original values of variables from the precondition since our postconditions are predicates over two states.

The program is written here in a concrete syntax to make it easier to read, but the proof will only use abstract syntax.

Figure 4.1 shows the proof of the program. The first hypothesis of the proof states that the program executed in an initial state, σ_0, will result in a final state, σ_f. The goal of the proof is to show that the postcondition of the program will hold in this final state. The other hypotheses state that x, y and z are all of type *Id*, and are needed to use the Id semantic rule as a justification. The type *Id* is a subtype of *Expr*. Expressions are evaluated for a given state using the $\xrightarrow{e}$ relation. Evaluating an expression that is an *id* consists of looking up the *id* in the state, to get its value. The Id semantic rule has the condition that an *Expr* is of type *Id* so that the rule cannot be applied to other *Expr* subtypes.

We know that the variable will have been declared and therefore exist in the state because of the context condition of the *Assign* statement.

Blank lines have been added to the proof to make it more readable, separating the proof into distinct parts. The most convenient way to read the proof is to start at line 8, the line containing the whole program. This line states that the execution of the program will result in a final state that is the initial state, with updated values for certain variables. The lines above (1-7) show how this updated state is reached, the lines below show how this updated state is related to the final state and the proof goal.

from $(\mathit{mk\text{-}Seq}(\mathit{mk\text{-}Assign}(r, x),$
$\quad \mathit{mk\text{-}Seq}(\mathit{mk\text{-}Assign}(x, y), \mathit{mk\text{-}Assign}(y, r))), \sigma_0) \xrightarrow{s} \sigma_f,$
$x\colon Id, y\colon Id, r\colon Id$

1 $(x, \sigma_0) \xrightarrow{e} \sigma_0(x)$ Id(h2)
2 $(\mathit{mk\text{-}Assign}(r, x), \sigma_0) \xrightarrow{s} \sigma_0 \dagger \{r \mapsto \sigma_0(x)\}$ Assign(1)

3 $(y, \sigma_0 \dagger \{r \mapsto \sigma_0(x)\}) \xrightarrow{e} \sigma_0(y)$ Id(h3)
4 $(\mathit{mk\text{-}Assign}(x, y), \sigma_0 \dagger \{r \mapsto \sigma_0(x)\}) \xrightarrow{s}$
$\quad \sigma_0 \dagger \{r \mapsto \sigma_0(x), x \mapsto \sigma_0(y)\}$ Assign(3)

5 $(r, \sigma_0 \dagger \{r \mapsto \sigma_0(x), x \mapsto \sigma_0(y)\}) \xrightarrow{e} \sigma_0(x)$ Id(h4)
6 $(\mathit{mk\text{-}Assign}(y, r), \sigma_0 \dagger \{r \mapsto \sigma_0(x), x \mapsto \sigma_0(y)\}) \xrightarrow{s}$
$\quad \sigma_0 \dagger \{r \mapsto \sigma_0(x), x \mapsto \sigma_0(y), y \mapsto \sigma_0(x)\}$ Assign(5)

7 $(\mathit{mk\text{-}Seq}(\mathit{mk\text{-}Assign}(x, y), \mathit{mk\text{-}Assign}(y, r)),$
$\quad \sigma_0 \dagger \{r \mapsto \sigma_0(x)\} \xrightarrow{s}$
$\quad \sigma_0 \dagger \{r \mapsto \sigma_0(x), x \mapsto \sigma_0(y), y \mapsto \sigma_0(x)\}$ Seq(4,6)

8 $(\mathit{mk\text{-}Seq}(\mathit{mk\text{-}Assign}(r, x),$
$\quad \mathit{mk\text{-}Seq}(\mathit{mk\text{-}Assign}(x, y), \mathit{mk\text{-}Assign}(y, r))), \sigma_0) \xrightarrow{s}$
$\quad \sigma_0 \dagger \{r \mapsto \sigma_0(x), x \mapsto \sigma_0(y), y \mapsto \sigma_0(x)\}$ Seq(2,7)

9 $\sigma_f = \sigma_0 \dagger \{r \mapsto \sigma_0(x), x \mapsto \sigma_0(y), y \mapsto \sigma_0(x)\}$ σ-Equiv(h1, 8)
10 $\sigma_f(x) = \sigma_0(y)$ †-Equiv(9)
11 $\sigma_f(y) = \sigma_0(x)$ †-Equiv(9)
infer $\sigma_f(x) = \sigma_0(y) \wedge \sigma_f(y) = \sigma_0(x)$ $\wedge$-I(10,11)

Figure 4.1: Proof of Variable Swap program.

To write the proof we start with line 8, which states that the execution of the whole program in σ_0 will result in a new state that we called σ_1 to start with. We than apply semantic rules to the program and work backwards to where the state is updated so that we can see how the initial state is changed to give us our initial state. The state σ_1 is then replaced by updated state in line 8.

We start with the outermost rule, since that will match against the conclusion of the semantic rule. So we apply the Seq semantic rule as a justification to line 8. From the semantic rule we can see that there are two parts to the *Seq* statement that need evaluating, a left and a right statement, so we add these statements as lines 2 and 7. Line 2 consists of an *Assign* statement so we apply the Assign semantic rule as a justification. Line 2 gives us an updated initial state which becomes the initial state for the execution of the statement on line 7.

Once we have filled out the proof backwards from line 8, we know exactly how the initial state is updated to reach a final state after the execution of the program. Line 9 shows that this final state is the same as the final state we called σ_f in the hypotheses. The rest of the proof shows that our post condition holds in the final state.

$$\boxed{\sigma\text{-Equiv}}\ \frac{(s,\sigma) \xrightarrow{s} \sigma' \quad (s,\sigma) \xrightarrow{s} \sigma''}{\sigma' = \sigma''}$$

$$\boxed{\dagger\text{-Equiv}}\ \frac{\sigma' = \sigma \dagger \{x_1 \mapsto v_1, \ldots, x_n \mapsto v_n\}}{\sigma'(x_i) = v_i}$$

Figure 4.2: Additional inference rule.

Figure 4.2 show two inference rules that we used in the above proof (lines 9-11), and that are also used in other proofs in this chapter.

4.1.1 Variable Swap (Version 2)

This example is a variation on the first example, but instead on using a temporary variable and multiple assignments, the *Swap* statement, that is part of the *Simple* language, is used. The *Swap* statement switches the values of two variables without the need for a temporary variable.

$$\{\mathbf{true}\}\ \textit{mk-Swap}(x, y)\ \{x = \overleftarrow{y} \wedge y = \overleftarrow{x}\}$$

The abstract syntax for *Swap*, shown below, consists of the *Id*s of the two variables whose values are to be swapped.

$$\begin{array}{lllll} \textit{Swap} & :: & \textit{lhs} & : & \textit{Id} \\ & & \textit{rhs} & : & \textit{Id} \end{array}$$

The context condition for the *Swap* statement states that both of the *Id*s should be declared variables. The variable *tpm* is a mapping of all declared variables to their declared types. Context conditions are explained in detail in Section 3.1.2.

$$\begin{array}{l} \textit{wf-Stmt}(\textit{mk-Swap}(\textit{lhs}, \textit{rhs}), \textit{tpm}) \triangleq \\ \quad \textit{lhs} \in \mathbf{dom}\ \textit{tpm} \wedge \\ \quad \textit{rhs} \in \mathbf{dom}\ \textit{tpm} \end{array}$$

The semantic rule for the *Swap* statement is shown in Figure 4.3. When executing a *Swap* statement, first the values of both variables are obtained from the state, then both variables are updated with the swapped values at the same time.

$$\boxed{\mathsf{Swap}}\ \frac{\begin{array}{l}(\textit{lhs}, \sigma) \xrightarrow{e} v_1 \\ (\textit{rhs}, \sigma) \xrightarrow{e} v_2\end{array}}{(\textit{mk-Swap}(\textit{lhs}, \textit{rhs}), \sigma) \xrightarrow{s} \sigma \dagger \{\textit{lhs} \mapsto v_2, \textit{rhs} \mapsto v_1\}}$$

Figure 4.3: Semantic rule for *Swap*.

Figure 4.4 shows the proof of the program. The first hypothesis of the proof is that the program will execute in an initial state, σ_0, and result in a final state σ_f. The goal of the proof is to show that in this final state, the postcondition will hold, as stated by the conclusion line of the proof.

	from $(mk\text{-}Swap(x, y), \sigma_0) \xrightarrow{s} \sigma_f, x\colon Id, y\colon Id$	
1	$(x, \sigma_0) \xrightarrow{e} \sigma_0(x)$	Id(h2)
2	$(y, \sigma_0) \xrightarrow{e} \sigma_0(y)$	Id(h3)
3	$(mk\text{-}Swap(x, y), \sigma_0) \xrightarrow{s} \sigma_0 \dagger \{x \mapsto \sigma_0(y), y \mapsto \sigma_0(x)\}$	Swap(1,2)
4	$\sigma_f = \sigma_0 \dagger \{x \mapsto \sigma_0(y), y \mapsto \sigma_0(x)\}$	σ-Equiv(h1, 3)
5	$\sigma_f(x) = \sigma_0(y)$	†-Equiv(4)
6	$\sigma_f(y) = \sigma_0(x)$	†-Equiv(4)
	infer $\sigma_f(x) = \sigma_0(y) \wedge \sigma_f(y) = \sigma_0(x)$	$\wedge$-I(5, 6)

Figure 4.4: Proof of Variable Swap program.

The introduction of the *Swap* statement makes the variable swap program simpler than it was in the first version, and also makes the proof of the program simpler.

As with the proof in the first example, this proof consists of two parts. First, we use the semantic rules to show how the initial state is updated to give the final state after execution (lines 1-3). Second, we relate the final state to the goal of the proof (lines 4 to conclusion). The addition to the language semantics resulted in the first part of the proof being simpler and shorter, but the second part (lines 4 to conclusion) are mostly the same as the second part of the first proof (lines 9 to conclusion), since this part does not use semantic rules as justifications.

4.1.2 Variable Swap (Version 3)

For this example we use the same program as in the second version, but this time we modify the *Swap* semantic rule so that the hypotheses are combined with the conclusion line (see Figure 4.5). We are able to do this since the *Swap* statement takes two *Id*s, and the type of the domain of the state is

Id, so we can lookup values in the state using the *Id*s specified by the *Swap* statement. The context condition of the *Swap* statement require the *Id*s to be declared variables, so we will not have a problem of an *Id* not existing in the domain of the state.

$$\boxed{\text{Swap}}\ \overline{\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}$$
$$(\textit{mk-Swap}(\textit{lhs}, \textit{rhs}), \sigma) \xrightarrow{s} \sigma \dagger \{\textit{lhs} \mapsto \sigma(\textit{rhs}), \textit{rhs} \mapsto \sigma(\textit{lhs})\}$$

Figure 4.5: Alternative semantic rule for *Swap*.

The proof of the program is shown in Figure 4.6 and is now shorter, since less proof lines are required to use the *Swap* semantic rule as a justification.

	from $(\textit{mk-Swap}(x, y), \sigma_0) \xrightarrow{s} \sigma_f$	
1	$(\textit{mk-Swap}(x, y), \sigma_0) \xrightarrow{s} \sigma_0 \dagger \{x \mapsto \sigma_0(y), y \mapsto \sigma_0(x)\}$	Swap
2	$\sigma_f = \sigma_0 \dagger \{x \mapsto \sigma_0(y), y \mapsto \sigma_0(x)\}$	σ-Equiv(h1, 1)
3	$\sigma_f(x) = \sigma_0(y)$	†-Equiv(2)
4	$\sigma_f(y) = \sigma_0(x)$	†-Equiv(2)
	infer $\sigma_f(x) = \sigma_0(y) \wedge \sigma_f(y) = \sigma_0(x)$	$\wedge$-I(3, 4)

Figure 4.6: Proof of Variable Swap program.

As with the previous versions, the proof consists of two parts, showing how the program will reach a final state from the program and the initial state, and then showing that the postcondition holds in the final state. Again, the second part of the proof (lines 2 to conclusion) is the same as in the previous proof examples.

4.2 Aliasing Example

This example makes use of the *Simple* language, as defined in Appendix C. The example program is a single statement that assigns the value 5 to the *i*-th element in the array. The precondition states that the *j*-th element of the array equals 2 before execution. The post condition states that the value of the j-th element should still have the value 2 (i.e. should not have changed) after execution, and that the value of the i-th element should be 5.

We assume that the variables i and j have been declared and have been assigned a value before execution of this statement.

$$\{a[j] = 2\}\ a[i] := 5\ \{a[i] = 5 \wedge a[j] = 2\}$$

The difficulty with this program comes when the values of i and j are equal, so that when looking up elements in the array, i and j point to the same element, which means that after execution, the value of the j-th element will have been changed, so that the postcondition does not hold.

To overcome this difficulty we need to add an extra condition to the precondition that states that i and j cannot be equal, giving us a precondition of $\{i \neq j \wedge a[j] = 2\}$.

The proof of the program is shown in Figure 4.7. The proof first shows that execution will update the i-th value of the array, then shows that the j-th element of the array remains unchanged.

from	$i \neq j, \sigma_0(a)(j) = 2, \sigma_0(i) \in \mathbf{dom}\, \sigma_0(a), \sigma_0(j) \in \mathbf{dom}\, \sigma_0(a), i\colon \mathit{Id}, j\colon \mathit{Id},$ $(\mathit{mk\text{-}Assign}(\mathit{mk\text{-}ArrayRef}(a, i), 5), \sigma_0) \xrightarrow{s} \sigma_f$	
1	$(i, \sigma_0) \xrightarrow{e} \sigma_0(i)$	Id(h3)
2	$(5, \sigma_0) \xrightarrow{e} 5$	
3	$(\mathit{mk\text{-}Assign}(\mathit{mk\text{-}ArrayRef}(a, i), 5), \sigma_0) \xrightarrow{s}$ $\sigma_0 \dagger \{a \mapsto \sigma_0(a) \dagger \{\sigma_0(i) \mapsto 5\}\}$	Assign-Array(1,h3, 2)
4	$\sigma_f = \sigma_0 \dagger \{a \mapsto \sigma_0(a) \dagger \{\sigma_0(i) \mapsto 5\}\}$	σ-Equiv(h7, 3)
5	$\sigma_f(a) = \sigma_0(a) \dagger \{\sigma_0(i) \mapsto 5\}$	$\dagger$-Equiv(4)
6	$\sigma_f(a)(i) = 5$	5
7	$\sigma_f(a)(j) = 2$	h1, h2, 5
infer	$\sigma_f(a)(j) = 2 \wedge \sigma_f(a)(i) = 5$	$\wedge$-I(7, 6)

Figure 4.7: Proof of array assign program.

The third and fourth hypotheses state that the values of the variables i and j are in the bounds of the array indices. Hypotheses five and six tell us the types of the variables i and j.

4.2.1 Aliasing Example - Extended

The simple definition of an array in the *Simple* language allows for the possibility of variable aliasing when dealing with the elements of the array. This problem was avoided by adding the precondition that the variables i and j should not contain the same values.

If the language used a more complicated representation for the state and array, then this extra condition would not be enough to avoid the problem of variable aliasing.

The following changes to the definition of the *array* language means that variable names now point to an abstract location, and that location points to a value. This means that more than one variable can point to the same location, and therefore the same value, and any change to one of the variables will also change the value of the other variables that point to the same location.

$$\begin{aligned}&\mathit{Env} = \mathit{Id} \xrightarrow{m} \mathit{Loc} \mid \mathit{ArrayLoc}\\ &\Sigma = \mathit{Loc} \xrightarrow{m} \mathit{Value}\\ &\mathit{ArrayLoc} = \mathbb{N} \xrightarrow{m} \mathit{Loc}\end{aligned}$$

A variable can either point to a location (Loc) or an array location ($\mathit{ArrayLoc}$). Locations can be mapped to values, and array location can be mapped to arrays. An array is represented by a mapping of natural numbers to locations, and the locations will point to values.

With this representation for the state and arrays, it is possible for an array element and a variable to have a shared location, and for two array elements to have shared locations. The problem with this is that an update to one of the elements of the array may also alter the value of another element. Below is an array that has two elements pointing to the same location.

$$\{1 \mapsto \alpha_1, 2 \mapsto \alpha_2, 3 \mapsto \alpha_1, 4 \mapsto \alpha_4\}$$

Here the symbol α represents a location (*Loc*), and a subscript distinguishes different locations. If we execute our program and i has the value of 1, and j has the value of 3, then after execution the value of the j-th element will have changed meaning the postcondition does not hold, even though i and j had different values before execution.

To overcome this potential problem we need to modify our definition of an array so that multiple elements cannot point to the same locations.

This example again shows how changes to a language definition can have an affect on how we reason about a program.

4.3 Guarded Do

This example program is written in the *Simple* language as defined in Appendix C, and makes use of the *Guarded Do* statement. The *Guarded Do* statement is a looping statement whose body consists of a number of guarded statements. The execution of a *Guarded Do* consists of evaluating all of the guards, and then non-deterministically executing one of the statements whose guard evaluated to *true*, and then looping back round, evaluating all of the guards again and then again executing one of the statements whose guard evaluated to *true*. The statement terminates when all of the guards evaluate to *false*.

The abstract syntax and semantic for the *Guarded Do* statement is displayed in Figure 4.8.

The example program using the *Guarded Do* statement, shown below, swaps the values of the variables x, y and z so that their values are ordered in ascending numerical order, with the smallest value in x and the largest value in z.

```
do
  $x > y$   →   $x, y := y, x$
  $y > z$   →   $y, z := z, y$
od
```

$$\begin{array}{lllll} \mathit{GDo} & :: & \mathit{body} & : & \mathit{Clause}^* \\[1ex] \mathit{Clause} & :: & b & : & \mathit{Expr} \\ & & s & : & \mathit{Statement} \end{array}$$

$$\boxed{\textsf{GDo-F}}\ \frac{\forall i \in \mathbf{inds}\ cs \cdot (cs(i).b, \sigma) \xrightarrow{e} \mathbf{false}}{(\mathit{mk}\text{-}\mathit{GDo}(cs), \sigma) \xrightarrow{s} \sigma}$$

$$\boxed{\textsf{GDo-T}}\ \frac{\begin{array}{l} i \in \mathbf{inds}\ cs \\ (cs(i).b, \sigma) \xrightarrow{e} \mathbf{true} \\ (cs(i).s, \sigma) \xrightarrow{s} \sigma' \\ (\mathit{mk}\text{-}\mathit{GDo}(cs), \sigma') \xrightarrow{s} \sigma'' \end{array}}{(\mathit{mk}\text{-}\mathit{GDo}(cs), \sigma) \xrightarrow{s} \sigma''}$$

Figure 4.8: Semantic rules for *Guarded Do* statement.

The postcondition of the program is that the values of the variables x, y and z are arranged in ascending order.

$$\{x \leq y \wedge y \leq z\}$$

The proof of the program is shown in Figure 4.9. There are basically two parts to this proof (lines 2 and 3). First, the subproof on line 2 deals with the case where all of the guards evaluate to false, so we use the GDo-F semantic rule to show that the state execution results in no change to the state, so the final state is the same as the initial state. We then show that the post condition holds in the final state.

The second part of the proof is the subproof on line 3. This shows that in the case where at least one of the guards evaluates to *true*, we will eventually reach a state where the guards all evaluate to false and the program will terminate.

	from $(\textit{mk-GDo}([\text{“}x > y \rightarrow x, y{:=}\ y, x\text{”}, \text{“}y > z \rightarrow y, z{:=}\ z, y\text{”}], \sigma_0) \xrightarrow{s} \sigma_f$	
	$x{:}\ \mathit{Id}, y\mathit{Id}, z{:}\ \mathit{Id}$	
0	$cs = [\text{“}x > y \rightarrow x, y{:=}\ y, x\text{”}, \text{“}y > z \rightarrow y, z{:=}\ z, y\text{”}]$	def’n
1	$(\exists i \in \textbf{inds}\ cs \cdot (cs(i).b, \sigma_0) \xrightarrow{e} \textbf{true}) \vee \neg(\exists i \in \textbf{inds}\ cs \cdot (cs(i).b, \sigma_0) \xrightarrow{e} \textbf{true})$	
2	**from** $\neg(\exists i \in \textbf{inds}\ cs \cdot (cs(i).b, \sigma_0) \xrightarrow{e} \textbf{true})$	
2.1	$\forall i \in \textbf{inds}\ cs \cdot (cs(i).b, \sigma_0) \xrightarrow{e} \textbf{false})$	deM(h2)
2.2	$(\textit{mk-GDo}(cs), \sigma_0) \xrightarrow{s} \sigma_0$	GDo-F(2.1)
2.3	$\sigma_f = \sigma_0$	σ-Equiv(h1, 2.2)
2.4	$(\textit{mk-RelExpr}(x, GT, y), \sigma_0) \xrightarrow{e} \textbf{false}$	$\forall$-E-set(0, 2.1)
2.5	$\sigma_0(x) \le \sigma_0(y)$	GT-F(2.4)
2.6	$\sigma_f(x) \le \sigma_f(y)$	=-sub-left(2.3, 2.5)
2.7	$(\textit{mk-RelExpr}(y, GT, z), \sigma_0) \xrightarrow{e} \textbf{false}$	$\forall$-E-set(0, 2.1)
2.8	$\sigma_0(y) \le \sigma_0(z)$	GT-F(2.7)
2.9	$\sigma_f(y) \le \sigma_f(z)$	=-sub-left(2.3, 2.8)
	infer $\sigma_f(x) \le \sigma_f(y) \wedge \sigma_f(y) \le \sigma_f(z)$	$\wedge$-I(2.6, 2.9)
3	**from** $\exists i \in \textbf{inds}\ cs \cdot (cs(i).b, \sigma_0) \xrightarrow{e} \textbf{true}$	
3.1	**from** $i \in \textbf{inds}\ cs \cdot (cs(i).b, \sigma_0) \xrightarrow{e} \textbf{true}$	
3.2	$(cs(i).s, \sigma_0) \xrightarrow{s} \sigma_1$	
3.3	**from** $[\![W]\!](\sigma_0, \sigma_1)$	
3.3.2	**from** $(\textit{mk-GDo}(cs), \sigma_1) \xrightarrow{s} \sigma_2$	
	infer $(\textit{mk-GDo}(cs), \sigma_0) \xrightarrow{s} \sigma_2$	GDo-T(3.h1, 3.3.h1, 3.3.2.h1)
3.3.3	$\sigma_f = \sigma_2$	σ-Equiv(h1, 3.3.2)
	infer $(\sigma_f(x) \le \sigma_f(y)) \wedge (\sigma_f(y) \le \sigma_f(z))$	W-Indn(3.3.2)
	infer $(\sigma_f(x) \le \sigma_f(y)) \wedge (\sigma_f(y) \le \sigma_f(z))$	
	infer $(\sigma_f(x) \le \sigma_f(y)) \wedge (\sigma_f(y) \le \sigma_f(z))$	$\exists$-E(h3, 3.1)
	infer $(\sigma_f(x) \le \sigma_f(y)) \wedge (\sigma_f(y) \le \sigma_f(z))$	$\vee$-E(1,2,3)

Figure 4.9: Proof of Guarded Do program.

4.4 Parallel Array Search

The final example program is a concurrent program written in the *Parallel* language as defined in Appendix D. The example is a search program that looks through an array for an element that has some specific property. The search is made parallel by using two threads that start at each end of the array and work towards the centre. Once one of the threads has found an appropriate element in the array, it is able to communicate with the other thread by setting the value of the variable r to the index of the element. Once a thread sets r away from zero, the other thread will stop searching, and the program will terminate.

The program has a precondition that states that at least one of the elements of the array must have the value 5, allowing us to simplify the program. We do not need extra conditions in the program to prevent it from searching out of the bounds of the array, since the threads start at the bounds and work towards the middle of the array. Even if the execution of the threads occurs in such a way that one thread searches the whole array by itself, it will always find an element with the value 5, and terminate before it has the chance to search out of bounds.

The example program is given below in a concrete syntax, along with the pre and post conditions.

```
{$\exists k \in \{1, \ldots, a.length\} \cdot a(k) = 5$}
r := 0 ;
i := 1 ;
j := a.length ;
(
  while $a(i) \neq 5 \ \wedge r = 0$
      $i++$ ;
      if $a(i) = 5$ then
          $r := i$
)
||
(
  while $a(j) \neq 5 \ \wedge r = 0$
      $j--$ ;
      if $a(j) = 5$ then
          $r := j$
)
{$a(r) = 5$}
```

Concurrency in the language is provided by the *Par* statement that contains a left and a right *Stmt*, which are executed in parallel. To allow for a finer interleaving in the execution of the left and right statements, the semantics of the language are defined using a small step.

Section 3.4.7 explains how concurrency could be added to a language by using a parallel statement, using the *Par* statement from this language as an example. It is worth here explaining a bit more about how this statement works, before going through the proof of the program. The semantic rules for the *Par* statement are shown in Figure 4.10.

We can see from the semantic rules of *Par* that we have separate rules for executing the left and right statements, both of which can be applied to a *Par* statement, and the choice of which rule to apply is non-deterministic. Since we are using a small step semantics, execution of a step is done in a number of small steps, rather than all at once. So in the case of the *Par* statement, the left and right statements are executed a bit at a time, in an undetermined order. It is entirely possible for the left statement to be

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

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

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

Figure 4.10: Semantic rules for *Par* statement.

executed until it is finished, and then for the right statement to be executed, without any interleaving or statements.

When a statement has finished executing, we end up with a **nil** statement. A parallel statement has finished executing when both the left and the right statements consist of **nil** statements.

In this language, assignment is atomic, meaning the evaluation and the assignment of the result are done in the same step, so that there can be no interference from another process between the steps, causing unexpected values to be stored.

In the proof of this program, we use a function M that takes an expression and a state and returns the value of that expression in the given state. By using this function we are stating that all of the free variables used in the expression are part of the state. This allows us to write $M(x, \sigma)$ instead of $\sigma(x)$, which might not improve the clarity of a small expression such as this, but makes the precondition expression easier to read than

$$\exists k \in \{\sigma_m(1), \ldots, \sigma_m(a.length)\} \cdot \sigma_m(a)(k) = 5$$

The proof of this program makes use of several lemmas, which are worth discussing before looking at the proof of the whole program. For convenience when discussing the program, the left and right statements of the parallel statement are referred to as g and d respectively. We would have used l and r for left and right statements, but we wanted to avoid confusion between r meaning the right statement, and r meaning the variable identifier.

The precondition of the program states that there exists at least one element in the array that has the value of 5. The program first of all initialises the variables i, j and r, and we assume that the array a has been initialised before this. Looking at the values of i and j we can see that after they have been initialised they can be substituted into the precondition so we know that

$$M(\exists k \in \{i, \ldots, j\} \cdot a(k) = 5, \sigma_n)$$

will hold after the variables are initialised, and before the parallel statement is executed.

This could be proved formally using the semantics of assignment, but that would not illustrate anything beyond Section 4.1 above. Our interest in this example lies in the parallel search part of the program.

Lemma 1

The first lemma states that if the property $\exists k \in \{i, \ldots, j\} \cdot a(k) = 5$ holds in σ_m, then any step in g that changes the value of the variable i, will preserve the property, i.e $\exists k \in \{i, \ldots, j\} \cdot a(k) = 5$ holds in σ_n, where σ_n is the new state, such that $\sigma_m(i) \neq \sigma_n(i)$.

The proof of the lemma is shown in Figure 4.11. To save space a mix of concrete and abstract syntax is used in the proof, with the outer statements being written in an abstract syntax to show that they can be matched with a semantic rule for a justification. Also, to aid readability, w_g is used in the proof to mean $\textit{mk-While}(a(i) \neq 5 \wedge r = 0, i++)$.

The hypotheses of the proof state that if our property holds in an initial state σ_m, and we execute the *While* statement in σ_m, we will reach one of two possible configurations. We either reach a configuration where we have the whole *While* to execute again (meaning we have completed one iteration of the loop) and a new state σ_n (which is different from σ_m), or we will reach a configuration consisting of a **nil** statement and a state σ_n (meaning the condition on the *While* statement evaluated to false, and the loop was not executed). The conclusion of the proof states that our initial property will still hold in the new state σ_n.

In proofs that involve small step semantics we use $\xrightarrow{s} *$ to mean one or more applications of the $\xrightarrow{s}$ transition relation. When using a big step semantics, a statement or a whole program is executed in a single step, so we can say that the execution of a statement in a state will give us a final state. A small step semantics executes a statement or program a little bit at a time, so we need to say that the execution of a statement in a state will reach some configuration that we are interested in.

The difference in the size of the steps also results in differences in proof layout. When using a small step semantics, we tend to work forward when writing the proof, whereas the direction is more mixed when working with big step semantics (See Section 6.4).

The proof in Figure 4.11 shows that the condition will hold for the two possible values of the condition on the *While* loop. Either the condition will evaluate to *true* and the loop will be executed once (line 2), or the condition will evaluate to false and the loop will not be executed (line 3).

The simple case is when the condition on the loop evaluates to false and the loop is not executed. This results in a **nil** statement and an unchanged state, meaning the property will still hold.

Lemma 2

Lemma 1 only deals with the thread g, so we introduce Lemma 2 to show that the same property holds for d as for g. Lemma 2 states that if the property $\exists k \in \{i, \ldots, j\} \cdot a(k) = 5$ holds in σ_m, then any step in d that changes the value of the variable j, will preserve the property, i.e $\exists k \in \{i, \ldots, j\} \cdot a(k) = 5$ holds in σ_n, where σ_n is the new state, such that $\sigma_m(j) \neq \sigma_n(j)$.

The proof of lemma 2 is not given here, but instead it is enough to say

	from $(\mathit{mk\text{-}While}(a(i) \neq 5 \wedge r = 0, i++), \sigma_m) \xrightarrow{s}{}^{*}$ $(\mathit{mk\text{-}While}(a(i) \neq 5 \wedge r = 0, i++), \sigma_n) \vee$ $(\mathit{mk\text{-}While}(a(i) \neq 5 \wedge r = 0, i++), \sigma_m) \xrightarrow{s}{}^{*} (\mathbf{nil}, \sigma_n)$, $M(\exists k \in \{i, \ldots, j\} \cdot a(k) = 5, \sigma_m)$	
1	$(a(i) \neq 5 \wedge r = 0, \sigma_m) \xrightarrow{e} \mathbf{true} \vee (a(i) \neq 5 \wedge r = 0, \sigma_m) \xrightarrow{e} \mathbf{false}$	
2	**from** $(a(i) \neq 5 \wedge r = 0, \sigma_m) \xrightarrow{e} \mathbf{true}$	
2.1	$(w_g, \sigma_m) \xrightarrow{s} (\mathit{mk\text{-}If}(a(i) \neq 5 \wedge r = 0,$ $\mathit{mk\text{-}Seq}(i++, w_g), \mathbf{nil}), \sigma_m)$	While-E
2.2	$(\mathit{mk\text{-}If}(a(i) \neq 5 \wedge r = 0, \mathit{mk\text{-}Seq}(i++, w_g), \mathbf{nil}), \sigma_m) \xrightarrow{s}$ $(\mathit{mk\text{-}If}(\mathbf{true}, \mathit{mk\text{-}Seq}(i++, w_g), \mathbf{nil}), \sigma_m)$	If-Eval(2.h1)
2.3	$(\mathit{mk\text{-}If}(\mathbf{true}, \mathit{mk\text{-}Seq}(i++, w_g), \mathbf{nil}), \sigma_m) \xrightarrow{s}$ $(\mathit{mk\text{-}Seq}(i++, w_g), \sigma_m)$	If-T
2.4	$(i++, \sigma_m) \xrightarrow{e} [\![+]\!](\sigma_m(i), 1)$	
2.5	$(\mathit{mk\text{-}Assign}(i, i+1), \sigma_n) \xrightarrow{s}$ $(\mathbf{nil}, \sigma_m \dagger \{i \mapsto \sigma_m(i) + 1\})$	Assign(2.4)
2.6	$(\mathit{mk\text{-}Seq}(i++, w_g), \sigma_m) \xrightarrow{s}$ $(\mathit{mk\text{-}Seq}(\mathbf{nil}, w_g), \sigma_m \dagger \{i \mapsto \sigma_m(i) + 1\})$	Seq-Step(2.5)
2.7	$(\mathit{mk\text{-}Seq}(\mathbf{nil}, w_g), \sigma_m \dagger \{i \mapsto \sigma_m(i) + 1\}) \xrightarrow{s}$ $(w_g, \sigma_m \dagger \{i \mapsto \sigma_m(i) + 1\})$	Seq-E
2.8	$\sigma_n = \sigma_m \dagger \{i \mapsto \sigma_m(i) + 1\}$	(h1, 2.7)
2.9	$\sigma_n(i) = \sigma_m(i) + 1$	2.8
2.10	$M(a(i) \neq 5, \sigma_m)$	2.h1
2.11	$M(\exists k \in \{i+1, \ldots, j\} \cdot a(k) = 5, \sigma_m)$	(h2, 2.10)
	infer $M(\exists k \in \{i, \ldots, j\} \cdot a(k) = 5, \sigma_n)$	=-sub-left(2.9, 2.11)
3	**from** $(a(i) \neq 5 \wedge r = 0, \sigma_m) \xrightarrow{e} \mathbf{false}$	
3.1	$(w_g, \sigma_m) \xrightarrow{s} (\mathit{mk\text{-}If}(a(i) \neq 5 \wedge r = 0,$ $\mathit{mk\text{-}Seq}(i++, w_g), \mathbf{nil}), \sigma_m)$	While-E
3.2	$(\mathit{mk\text{-}If}(a(i) \neq 5 \wedge r = 0, \mathit{mk\text{-}Seq}(i++, w_g), \mathbf{nil}), \sigma_m) \xrightarrow{s}$ $(\mathit{mk\text{-}If}(\mathbf{false}, \mathit{mk\text{-}Seq}(i++, w_g), \mathbf{nil}), \sigma_m)$	If-Eval(3.h1)
3.3	$(\mathit{mk\text{-}If}(\mathbf{false}, \mathit{mk\text{-}Seq}(i++, w_g), \mathbf{nil}), \sigma_m) \xrightarrow{s} (\mathbf{nil}, \sigma_m)$	If-F
3.4	$\sigma_n = \sigma_m$	
	infer $M(\exists k \in \{i, \ldots, j\} \cdot a(k) = 5, \sigma_n)$	=-sub-left(3.4, h2)
	infer $M(\exists k \in \{i, \ldots, j\} \cdot a(k) = 5, \sigma_n)$	$\vee$-E(1,2,3)

Figure 4.11: Proof of Lemma 1.

that since both threads g and d are similar, and lemma 2 is stating the same property on d that lemma 1 does on g, then the proof for lemma 2 will be very similar to the proof in for lemma 1 in Figure 4.11, the difference being that d is decrementing the value of j with each execution of the *While* loop. This would make line 2.10 of the proof look like

$$M(\exists k \in \{j, \ldots, j-1\} \cdot a(k) = 5, \sigma_n)$$

From this we can infer the conclusion of the subproof, since line 2.9 would state that $M(a(i) \neq 5, \sigma_m)$.

Lemma 3

The postcondition of the program states that, after execution, the value of r will be the index of an element in the array a that has the value of 5. So we want to be sure that if the value of r is changed by either thread, then the postcondition will hold for the new value of r. We again state this as two lemmas, one for each of the threads g and d.

Lemma 3 states that any step of g that changes the value of $\sigma_0(r)$, ensures that σ_f is such that $M(a(r), \sigma_f) = 5$.

The only part of g that can modify the value of r is the *Assign* statement that is part of the *If* statement. The proof of lemma 3 is shown in Figure 4.12 and states that the execution of the *If* statement will result in a state where, if the value of r has been changed, it will have been changed to a value such that $a(r) = 5$ holds.

This deals with the two cases that can arrive in the execution of the *If* statement, that is when the condition evaluates to *true* or *false*. If the condition evaluates to *true*, then the value of r is changed, so we show that the change in r is such that our property holds. If the condition evaluates to *false*, then we show that the value of r does not changes, and our property still holds.

Lemma 4

Lemma 4 is similar to lemma 3, but for the other parallel thread, d. It states that any step of d that changes the value of $\sigma_0(r)$, ensures that σ_f is such that $M(a(r), \sigma_f) = 5$. We don't give the proof of it here, but it is enough for us to say that a proof of lemma 4 would be almost identical to the proof of lemma 3, the only difference being that the *If* statement in d uses the variable j instead of i.

	from $(\textit{mk-If}(a(i) = 5, \textit{mk-Assign}(r, i), \mathbf{nil}), \sigma_0) \xrightarrow{s}{*} (\mathbf{nil}, \sigma_f), i \colon \mathit{Id}$	
1	$(a(i) = 5, \sigma_0) \xrightarrow{e} \mathbf{true} \vee (a(i) = 5, \sigma_0) \xrightarrow{e} \mathbf{false}$	
2	**from** $(a(i) = 5, \sigma_0) \xrightarrow{e} \mathbf{true}$	
2.1	$(\textit{mk-If}(a(i) = 5, \textit{mk-Assign}(r, i), \mathbf{nil}), \sigma_0) \xrightarrow{s}$ $(\textit{mk-If}(\mathbf{true}, \textit{mk-Assign}(r, i), \mathbf{nil}), \sigma_0)$	If-Eval(2.h1)
2.2	$(\textit{mk-If}(\mathbf{true}, \textit{mk-Assign}(r, i), \mathbf{nil}), \sigma_0) \xrightarrow{s}$ $(\textit{mk-Assign}(r, i), \sigma_0)$	If-T
2.3	$(i, \sigma_0) \xrightarrow{e} \sigma_0(i)$	Id(h2)
2.4	$(\textit{mk-Assign}(r, i), \sigma_0) \xrightarrow{s} (\mathbf{nil}, \sigma_0 \dagger \{r \mapsto \sigma_0(i)\})$	Assign(2.3)
2.5	$\sigma_f = \sigma_0 \dagger \{r \mapsto \sigma_0(i)\}$	
2.6	$\sigma_0(a)(\sigma_0(i)) = 5$	EQ-T(2.h1)
2.7	$\sigma_f(r) = \sigma_0(i)$	†-Equiv(2.5)
2.8	$\sigma_f(a) = \sigma_0(a)$	2.5
2.9	$\sigma_0(a)(\sigma_f(r)) = 5$	=-sub-left(2.7, 2.6)
2.10	$\sigma_f(a)(\sigma_f(r)) = 5$	=-sub-left(2.8, 2.6)
	infer $\sigma_0(r) \neq \sigma_f(r) \Rightarrow \sigma_f(a)(\sigma_f(r)) = 5$	⇒-I-L-vac(2.10)
3	**from** $(a(i) = 5, \sigma_0) \xrightarrow{e} \mathbf{false}$	
3.1	$(\textit{mk-If}(a(i) = 5, \textit{mk-Assign}(r, i), \mathbf{nil}), \sigma_0) \xrightarrow{s}$ $(\textit{mk-If}(\mathbf{false}, \textit{mk-Assign}(r, i), \mathbf{nil}), \sigma_0)$	If-Eval(3.h1)
3.2	$(\textit{mk-If}(\mathbf{false}, \textit{mk-Assign}(r, i), \mathbf{nil})\sigma_0) \xrightarrow{s} (\mathbf{nil}, \sigma_0)$	If-F
3.3	$\sigma_f = \sigma_0$	3.2, h1
3.4	$\sigma_f(r) = \sigma_0(r)$	3.3
3.5	$\neg(\sigma_f(r) \neq \sigma_0(r))$	3.4
	infer $\sigma_0(r) \neq \sigma_f(r) \Rightarrow \sigma_f(a)(\sigma_f(r)) = 5$	⇒-I-R-vac(3.5)
	infer $\sigma_0(r) \neq \sigma_f(r) \Rightarrow \sigma_f(a)(\sigma_f(r)) = 5$	∨-E(1, 2, 3)

Figure 4.12: Proof of Lemma 3.

Lemmas 5 and 6

Lemmas 3 and 4 state that if r is changed, it will be changed to a value that respects the postcondition of the program. We also need to show that the value of r will be changed by one of the threads. Lemma 5 states that once g has finished executing, the value of r in the final state will not be equal to zero, or in other words, the value of r has changed.

If the thread g does all of the work and d does not do anything, then because of the precondition on the program that states that there is an element in the array with the value 5, g will find an element of the array that meets the requirement, causing the *While* loop to exit and the *If* statement to write the index of the found element to r. The only way for the loop in g to exit without finding a 5 in the array is if d has found a 5 in the array and has changed the value of r. In which case the loop will exit, and d will finish without modifying the value of r.

The proof of lemma 5 is given in Figure 4.13. To save space, w_g is written in the proof to mean $mk\text{-}While(a(i) \neq 5 \wedge r = 0, i++)$, and if_g to mean $mk\text{-}If(a(i) = 5, mk\text{-}Assign(r, i), \mathbf{nil})$.

It is possible for both g and d to find elements in the array that have the value 5[2], which means that both threads will change the value of r. It doesn't matter which one gets to write to r first, either value will satisfy the postcondition. This possibility is dealt with in the subproof on line 6.

From the hypothesis of line 6, we know that the value of r has changed, so d must have found a 5 in the array, but we still need to show that g will not change it back to zero. It is possible that g has also found a 5 in the array, as well as d. Line 6.2 shows that if g hasn't found a 5 in the array, then it will not change the value of r, so it will still be not equal to zero. Line 6.3 shows that if g has found a 5, then it will change r to a value that is not equal to zero.

Lemma 6 states that once d has finished executing, the value of r in the final state will not be equal to zero. The proof for lemma 6 will be similar to the proof for lemma 5.

Lemma 7

The program will terminate when both of the while loops in the parallel threads, g and d, have exited. For each loop, there are two conditions under which they will exit. If one of the threads finds an element in the array that contains the value 5, then it will exit the loop and set the value of r to the index of the element that contains the value 5. Changing the value of r will cause the while loop in the other thread to exit.

[2]These could be the same element or different ones, since the array may have multiple elements with the value 5.

```
        from $(mk\text{-}Seq(w_g, if_g), \sigma_m) \xrightarrow{s} * (\mathbf{nil}, \sigma_n)$,
             $\exists k \in \{i, \ldots, j\} \cdot a(k) = 5, i\colon Id$
1            $(mk\text{-}Seq(w_g, if_g), \sigma_m) \xrightarrow{s} * (mk\text{-}Seq(\mathbf{nil}, if_g), \sigma_t)$
2            $(a(i) \neq 5 \wedge r = 0, \sigma_t) \xrightarrow{e} \mathbf{false}$                                1
3            $(a(i) \neq 5, \sigma_t) \xrightarrow{e} \mathbf{false} \vee (r = 0, \sigma_t) \xrightarrow{e} \mathbf{false}$        2
4            $(mk\text{-}Seq(\mathbf{nil}, if_g), \sigma_t) \xrightarrow{s} (if_g, \sigma_t)$                          Seq-E
5            from $(a(i) \neq 5, \sigma_t) \xrightarrow{e} \mathbf{false}$
5.1               $(a(i) = 5, \sigma_t) \xrightarrow{e} \mathbf{true}$                                   5.h1
5.2               $(if_g, \sigma_t) \xrightarrow{s} (mk\text{-}If(\mathbf{true}, r := i, \mathbf{nil}), \sigma_t)$             If-Eval(5.1)
5.3               $(mk\text{-}If(\mathbf{true}, r := i, \mathbf{nil}), \sigma_t) \xrightarrow{s} (mk\text{-}Assign(r, i), \sigma_t)$      If-T
5.4               $(i, \sigma_t) \xrightarrow{e} \sigma_t(i)$                                             Id(h3)
5.5               $(mk\text{-}Assign(r, i), \sigma_t) \xrightarrow{s} (\mathbf{nil}, \sigma_t \dagger \{r \mapsto \sigma_t(i)\})$       Assign(5.4)
5.6               $\sigma_n = \sigma_t \dagger \{r \mapsto \sigma_t(i)\}$                                  (h1, 5.5)
5.7               $\sigma_t(i) > 0$
5.8               $\sigma_n(r) > 0$                                                  (5.6, 5.7)
             infer $\sigma_n(r) \neq 0$                                               5.8
6            from $(r = 0, \sigma_t) \xrightarrow{e} \mathbf{false}$
6.1               $(a(i) = 5, \sigma_t) \xrightarrow{e} \mathbf{true} \vee (a(i) = 5, \sigma_0) \xrightarrow{e} \mathbf{false}$
6.2               from $(a(i) = 5, \sigma_t) \xrightarrow{e} \mathbf{false}$
6.2.1                  $(if_g, \sigma_t) \xrightarrow{s} (mk\text{-}If(\mathbf{false}, r := i, \mathbf{nil}), \sigma_t)$      If-Eval(6.2.h1)
6.2.2                  $(mk\text{-}If(\mathbf{false}, r := i, \mathbf{nil}), \sigma_t) \xrightarrow{s} (\mathbf{nil}, \sigma_t)$            If-F
6.2.3                  $\sigma_n = \sigma_t$                                                 (h1, 6.2.2)
                  infer $\sigma_n(r) \neq 0$                                          (6.2.3, 6.h1)
6.3               from $(a(i) = 5, \sigma_t) \xrightarrow{e} \mathbf{true}$
6.3.1                  $(if_g, \sigma_t) \xrightarrow{s} (mk\text{-}If(\mathbf{true}, r := i, \mathbf{nil}), \sigma_t)$       If-Eval(6.3.h1)
6.3.2                  $(mk\text{-}If(\mathbf{true}, r := i, \mathbf{nil}), \sigma_t) \xrightarrow{s}$
                           $(mk\text{-}Assign(r, i), \sigma_t)$                                  If-T
6.3.3                  $(i, \sigma_t) \xrightarrow{e} \sigma_t(i)$                                          Id(h3)
6.3.4                  $(mk\text{-}Assign(r, i), \sigma_t) \xrightarrow{s} (\mathbf{nil}, \sigma_t \dagger \{r \mapsto \sigma_t(i)\})$  Assign(6.3.3)
6.3.5                  $\sigma_n = \sigma_t \dagger \{r \mapsto \sigma_t(i)\}$                               (h1, 6.3.4)
6.3.6                  $\sigma_t(i) > 0$
6.3.7                  $\sigma_n(r) > 0$                                          (6.3.6, 6.3.5)
                  infer $\sigma_n(r) \neq 0$                                              6.3.7
             infer $\sigma_n(r) \neq 0$
        infer $\sigma_n(r) \neq 0$
```

Figure 4.13: Proof of Lemma 5.

From the precondition of the program, we know that there exists an element in the array that has the value of 5, so one or both of the threads will find it.

Proof of Program

Now all of the lemmas are worked out, they can be used in the proof of the whole program, shown in Figure 4.14. As with the other proofs in this section, the proof is not fully formalised.

	from $(mk\text{-}Seq(init, mk\text{-}Par(g, d)), \sigma) \xrightarrow{s}* (\mathbf{nil}, \sigma_\omega)$	
	$M(\exists k \in \{1, \ldots, a.length\} \cdot a(k) = 5, \sigma)$	
1	$(init, \sigma) \xrightarrow{s}* (\mathbf{nil}, \sigma \dagger \{r \mapsto 0, i \mapsto 1, j \mapsto a.length\})$	
2	$\sigma_1 = \sigma \dagger \{r \mapsto 0, i \mapsto 1, j \mapsto a.length\}$	
3	$M(\exists k \in \{i, \ldots, j\} \cdot a(k) = 5, \sigma_1)$	(h2, 2)
4	$(mk\text{-}Seq(init, mk\text{-}Par(g, d)), \sigma) \xrightarrow{s}*$ $(mk\text{-}Seq(\mathbf{nil}, mk\text{-}Par(g, d)), \sigma_1)$	Seq-Step∗(1)
5	$(mk\text{-}Seq(\mathbf{nil}, mk\text{-}Par(g, d)), \sigma_1) \xrightarrow{s} (mk\text{-}Par(g, d), \sigma_1)$	Seq-E
6	$(mk\text{-}Par(g, d), \sigma_1) \xrightarrow{s}* (mk\text{-}Par(\mathbf{nil}, d'), \sigma_n) \vee$ $(mk\text{-}Par(g, d), \sigma_1) \xrightarrow{s}* (mk\text{-}Par(g', \mathbf{nil}), \sigma_n)$	Lemma7
7	**from** $(mk\text{-}Par(g, d), \sigma_1) \xrightarrow{s}* (mk\text{-}Par(\mathbf{nil}, d'), \sigma_n)$	
7.1	$(g, \sigma_1) \xrightarrow{s}* (\mathbf{nil}, \sigma_n)$	7.h1
7.2	$\sigma_n(r) \neq 0$	Lemma5(7.1)
7.3	$(if_g, \sigma_m) \xrightarrow{s}* (\mathbf{nil}, \sigma_n)$	7.1
7.4	$\sigma_m(r) \neq \sigma_n(r) \;\Rightarrow\; \sigma_n(a)(\sigma_n(r)) = 5$	Lemma3(7.3)
7.5	$\sigma_m(r) = 0$	2
7.6	$\sigma_m(r) \neq \sigma_n(r)$	7.2, 7.5
7.7	$M(a(r) = 5, \sigma_n)$	⇒-E-L(7.4, 7.6)
7.8	$(mk\text{-}Par(\mathbf{nil}, d'), \sigma_n) \xrightarrow{s}* (mk\text{-}Par(\mathbf{nil}, \mathbf{nil}), \sigma_p)$	
7.9	$(mk\text{-}Par(\mathbf{nil}, \mathbf{nil}), \sigma_p) \xrightarrow{s} (\mathbf{nil}, \sigma_p)$	Par-E
7.10	$\sigma_p = \sigma_\omega$	7.9, h1
	infer $M(a(r) = 5, \sigma_\omega)$	
8	**from** $(mk\text{-}Par(g, d), \sigma_1) \xrightarrow{s}* (mk\text{-}Par(g', \mathbf{nil}), \sigma_n)$	
	$\vdots$	
	infer $M(a(r) = 5, \sigma_\omega)$	
	infer $M(a(r) = 5, \sigma_\omega)$	∨-E (6,7,8)

Figure 4.14: Proof of parallel array search program.

Lines 1 to 3 show that after the variables have been initialised, there exists an element in the array between i and j that has the property that we are looking for. Line 2 introduces σ_a as an abbreviation. It should be apparent that lines 1 and 3 could be expanded out and written formally using the semantics for assignment and the $=$ -*sub* rules, but this proof is informal and the abbreviated lines can be easily understood.

Lines 4 and 5 use the semantics of the *Seq* statement to show that once the variable initialisation part of the program has executed, we are left with the *Par* statement to execute. Line 4 shows that *init* will reach **nil** after one or more steps of the $\xrightarrow{s}$ transition relation. For the justification of this line we show that the line is justified by one or more applications of the Seq-Step semantic rule.

Line 6 states that after one or more steps of either or both of the Par-L or Par-R semantic rules, either the *left* or the *right* statement will finish executing and reach **nil**. The subproofs on lines 7 and 8 then show that in either case, the program will reach a final state in which the postcondition of the program will hold.

4.5 Parallel Array Search in Hoare logic

This section looks again at the parallel array search program and attempts to use axiomatic reasoning, in contrast to the direct operational semantic approach used in Section 4.4, so that the two approaches can be compared. We quickly see that a proof using pure Hoare axioms is not possible and present an argument that uses rely-guarantee reasoning.

The parallel array search program is repeated here in Figure 4.15 for convenience. As in the previous section, the left and right parts of the parallel statement are referred to as g and d respectively.

Inspection of either the g or d components makes it clear why false conclusions could be drawn from misuse of Hoare's rules. For example, looking at g in isolation it would be possible to prove that g would search through the array from the start until it found an appropriate value, providing no interference occurs. But we know that interference will occur and g and d actually count on interference so that one can terminate early if the other

```
{$\exists k \in \{1, \ldots, a.length\} \cdot a(k) = 5$}
r := 0 ;
i := 1 ;
j := a.length ;
(
  while $a(i) \neq 5 \wedge r = 0$
      $i++$ ;
      if $a(i) = 5$ then
          $r := i$
)
||
(
  while $a(j) \neq 5 \wedge r = 0$
      $j--$ ;
      if $a(j) = 5$ then
          $r := j$
)
{$a(r) = 5$}
```

Figure 4.15: Parallel array search program.

finds a 5 in the array first. However, if g was executed in an environment that could change r to a value that did not point to a 5 in the array, then g would terminate early but the value of r would be such that its postcondition does not hold.

This reinforces the point that Hoare's axioms were designed for sequential programs, not concurrent programs. Early approaches to proving properties about sequential programs using axiomatic methods attempted to prove properties about sequential components of the program in isolation, and then either show that the parallel components do not interfere (sometimes called "disjoint parallelism") [Hoa75, Jon83], or check that the sequential proofs are not interfered with by the other concurrent components. This latter approach was presented in its most worked out form by Susan Owicki and the method is known as the *Owkik/Gries* method (see [Owi75, OG76]).

In the case of the program in Figure 4.15, g and d would each be treated as sequential programs and proofs would be written to show that g satisfies its postcondition, *post-g*, and that d meets its postcondition, *post-d*. The Owicki/Gries method then requires a final *interference freedom* (or *"Einmischungsfrei"*) test to show non-interference.

The obvious proofs would fail this test and this is why Owicki's method is seen as non-compositional. It would then be necessary to re-do the earlier sequential proofs, which might just be possible if "auxiliary variables" were added.

We here take another approach and consider Jones' rely-guarantee idea which is broadly axiomatic but accepts the possibility of interference from the beginning.

Rely-guarantee reasoning [Jon83] makes it possible to reason about interference in the development of concurrent programs by augmenting the pre and post conditions introduced in Hoare logic with a rely-condition and a guarantee-condition for each unit specified. These four assertions are associated with a program text as follows,

$$\{P, R\}\ S\ \{G, Q\}$$

where P is the pre-condition, R is the rely-condition, S is the program, G is the guarantee-condition and Q is the post-condition.

Even for sequential programs, VDM always used relational post-conditions that allow the required outcome to be stated as a relation between initial and final states. Rely-guarantee reasoning is typically in a VDM framework, meaning the post-condition of a program is a predicate over the initial and final states of the program. Similarly, both rely and guarantee conditions are predicates over two states (the initial and final states of a single step of execution).

A program is assumed to be executed in an environment where it will run concurrently with other programs, and will both cause and be subject to interference. A rely-condition is the assumption that is made about the environment in which a program is running, and describes the interference that the program is able to tolerate from the environment. The guarantee-condition states the interference that the program will generate as part of the environment, under the assumption that the pre-condition and the rely-conditions are met. Both conditions should be reflexive and transitive to reflect the fact that there might be zero or many steps.

When multiple programs are executed concurrently, the rely-condition of each program must be met by the guarantee-condition of the other programs that make up the environment.

Going back to the program in Figure 4.15, we can use rely-guarantee reasoning to reason about the parallel execution of g and d. Figure 4.16 shows the specification[3] of g and d, including rely and guarantee conditions. The pre and post-conditions for g and d are the same as the pre and post-conditions of the parallel statement.

$\boldsymbol{g}$		$\boldsymbol{d}$	
rd:	$a : \mathbb{N} \xrightarrow{m} \mathbb{Z}$	**rd**:	$a : \mathbb{N} \xrightarrow{m} \mathbb{Z}$
wr:	$i : \mathbb{N}$	**wr**:	$j : \mathbb{N}$
	$r : \mathbb{N}$		$r : \mathbb{N}$
pre:	$\exists k \in \{i, \ldots, j\} \cdot a(k) = 5$	**pre**:	$\exists k \in \{i, \ldots, j\} \cdot a(k) = 5$
rely:	$a = \overleftarrow{a} \wedge i = \overleftarrow{i} \wedge$	**rely**:	$a = \overleftarrow{a} \wedge j = \overleftarrow{j} \wedge$
	$(r \neq \overleftarrow{r} \;\Rightarrow\; a(r) = 5)$		$(r \neq \overleftarrow{r} \;\Rightarrow\; a(r) = 5)$
guar:	$r \neq \overleftarrow{r} \;\Rightarrow\; a(r) = 5$	**guar**:	$r \neq \overleftarrow{r} \;\Rightarrow\; a(r) = 5$
post:	$a(r) = 5$	**post**:	$a(r) = 5$

Figure 4.16: Specification of g and d.

When g and d are executed in parallel, d is the environment for g and g is the environment for d.

Both g and d rely on a not being changed by the environment, and neither will modify a. Both rely on the value of r not being changed to an unexpected value (i.e. should only be changed to the value of an array index whose element has the value of 5), and both guarantee not to change the value of r to an unexpected value. The program g relies on i not being changed by the environment, and d does not access i. The program d relies

[3]Written in a VDM style.

on j not being changed by the environment, and g does not access j.

The relationship between the various conditions of the parallel programs is formalised in the parallel composition rule[4] (see Figure 4.17). Using this inference rule, we can show that parallel statement, $g \parallel d$, meets its postcondition.

$$\frac{\begin{array}{l}\{P, R \vee G_2\}\ S_1\ \{G_1, Q_1\} \\ \{P, R \vee G_1\}\ S_2\ \{G_2, Q_2\} \\ G_1 \vee G_2 \ \Rightarrow\ G \\ P \wedge Q_1 \wedge Q_2 \wedge (R \vee G_1 \vee G_2)^* \ \Rightarrow\ Q\end{array}}{\{P, R\}\ S_1 \parallel S_2\ \{G, Q\}}$$

Figure 4.17: Rely-Guarantee rule for parallel composition.

Since we know that both sections of the parallel statement can handle interference from the environment, and we know that there is at least one element of the array a with the value of 5, we are able to show that g or d will always find that value. Although the individual steps of this proof are trivial, notice that this is only the case because of the respective rely conditions precisely rule out any mischievous interference.

The rely-guarantee proof of the program can be made simpler by using the "local" idea of [JP08]. The variables i and j are not shared variables and so can be thought of as being local to their programs, making them effectively hidden from the environment and therefore not subject to interference.

4.5.1 Comparison

The purpose of this section was to work through an example using Hoare axioms, so that it could be compared to the direct operational semantic approached as used on the example in Section 4.4. It was quickly shown, however, that a traditional Hoare style approach would not work due to the interference caused by the parallel execution of g and d. Instead the program was shown to be correct using the rely-guarantee approach, which

[4]This version of the rule is borrowed from [Pie09]. Also used in [CJ07].

is basically axiomatic, but Jones' rely-guarantee idea is expressively weak in that there are forms of concurrency about which it cannot reason. There are extensions to the rely-guarantee approach such as Stoelen's addition of progress conditions in [Stø90], but even those have limitations. The approach demonstrated in this chapter places an immediate path to formal reasoning for any language that has an operational semantics.

4.6 Conclusion

From the examples in this chapter, we can see that when reasoning about a program using operational semantics, the proof consists of two basic steps. First, the operational semantic rules are used to show the effect that execution has on the initial state to give some final state. Next, we show that the postcondition of the program holds in the final state. This is most apparent in the variable swap example in Section 4.1, where lines 1-8 use the semantic rules to reach a final state, then lines 9 to the conclusion show that the postcondition holds in the final state. We can also see these steps in the proofs of the lemmas in Section 4.4, where they are repeated in each of the subproofs.

The *variable swap* examples show how changes to the operational semantic definition of a language can effect the proofs of a program in the language. Three equivalent programs were shown, each of which used different semantic definitions in their execution, resulting in different proofs for each program. The difference was in the part of the proof where the semantics are used to show the effect of execution on the state. In each proof there is very little difference in the part of the proofs that relate the final state to the postcondition of the program.

The last example (Section 4.4) uses a language that has a small step semantic definition, whereas the other languages used in this chapter have big step semantic definitions. From the proofs of the lemmas in the last example, we can see differences in the structure of a proof of a program, that stem from the size of the step.

The use of the *Seq* statement in the example languages means that a program is always a single statement, rather than a list of statements. In the examples using a big step semantics, when we apply the semantic rules to a program we start with the outer statement and work backwards until we have applied semantic rules to all parts of the program (as illustrated in Section 6.4). We can then show how the initial state is modified by the execution of the program. Working backwards this way is due to the semantic rule having hypotheses. The language used in Section 4.4 has a small step semantics and many of the semantic rules have no hypotheses, resulting in mostly using forward reasoning when writing a proof.

The proofs in the last example (Section 4.4) are longer than might have been expected, and are good examples of where a proof support tool managing the proofs would be of assistance. Using the proof support tool for such proofs will help the user by managing the proof layout and line numbers. It is also clear that a certain degree of automation would be useful. Chapter 6 suggests that the tool might do more than just suggest possible rules to justify a line, but also suggest appropriate proof lines to use as parameters to the justification, and if no appropriate lines exist, suggest lines to be added to the proof.

Such automation in the proof support tool would be straightforward to implement but would take care of what the user might consider to be the tedious details. The area in which human intuition is really required is in the choice of the lemmas and the structure of the proof. Having the proof support tool assist as much as possible and automate justification selection where possible, allows the user to focus on the more important or more difficult details of the proof.

4.7 Summary

The examples in this chapter illustrate the use of semantics of a language to reason about programs that are written in that language. These examples also show that the use of semantics to reason about a program is not only restricted to simple sequential programs, but can also be used to reason about programs that include more complex language features such as concurrency.

Later chapters describe the development of a prototype proof support tool for writing proofs using operational semantics as justifications. Each of the proofs of the examples in this chapter have been presented in the prototype tool.

Chapter 5

Formal Verification Tools

Formal verification is the use of mathematical techniques to show that some system performs or behaves as desired and will not reach some undesirable state, and can be applied to both hardware and software systems. In terms of software systems, formal verification techniques can be used to show that a computer program meets it specification, and also that it will not behave unexpectedly by, for example, indexing outside of an array.

The focus of this chapter is on the formal verification tools that are available, rather than on the mathematical techniques that they employ, with specific focus on formal reasoning tools for proving that computer programs satisfy their specifications.

5.1 Static and Runtime Verification

Static analysis involves the analysis of the source code of a program at compile time. Any compilation process will do some checking in order to determine if code can be generated from a source program. Such checks are normally things like the conformance of variable use to declarations. Obviously any checks performed by a compiler have to be decidable. Although compilers perform some static analysis on a program before it is compiled, such as syntactic and type checks, they are not generally considered static analysis tools. Static analysis tools perform more extensive checks on the source code of a program in an attempt to find sources of potential run-time errors.

Static analysis tools are not able to prove the correctness of a program, with respect to its specification, but can help to eliminate some sources of bugs from a program. In only a few languages (Eiffel [Eif11]) are specifications part of the program text, so that programs can be shown to satisfy its specification.

ESC/Java2 [esc11] is a static analysis tool that is referred to as an *extended static checker*. By including the specification of a program as JML annotations (see below), ESC/Java2 is able to try to prove the correctness of the program, with respect to its JML specification. The checking of the source code is performed at compile time and is fully automatic.

ESC/Java2 is good at proving the absence of sources of potential runtime exceptions, such as *NullPointerException*, and verifying relatively simple properties, but is not sound or complete.

Runtime Verification involves checking a program as it is running to ensure its behaviour is consistent with its specification.

The *Java Modelling Language* (JML) [jml10] is a language for annotating Java programs with specifications in the form of Hoare style pre and post conditions on methods, and invariants on variables. The annotations are written in comments and so will be ignored by the standard Java compiler. When compiled with the JML compiler, the JML annotations are converted into runtime assertions that will cause run-time exceptions to be thrown when the behaviour of the program does not match its specification.

5.2 Theorem Proving tools

A *theorem prover* is a program that attempts to automatically find a proof for a given conjecture, in a specific logical framework. Since the heuristics used in the proof search is not optimal for all problems [Bri10], user guidance may also be necessary. The degree of automation provided by theorem proving tools also depends on the complexity of the logic used. There is a trade off between the complexity and expressiveness of the logic used and the degree of automation possible.

Theorem provers generally fall into two categories – *automated* or *interactive* – depending on the degree of automation, and the degree of user interaction required. For both categories of tools it is often not important how a proof is found, but instead only important that it is found. Many modern systems offer a mix of automatic and interactive proof, but their emphasis varies.

5.2.1 Automated Theorem Proving Tools

Automated theorem proving (ATP) tools generally require little or no human intervention to guide the proof search. Many ATP tools use first-order logic, which is expressive enough to describe a wide range of problem, but also constrained enough to allow proof searching to be fully automated. ATPs are able to solve a wide variety of non-trivial problems, however, the complexity of many interesting problems cannot currently be solved within realistic resource limits. A key area of research is therefore developing more powerful ATPs that can solve more complex problems with the same resource limits [SS01]. Some ATPs save resources by not constructing a proof during its search, and so are unable to produce a proof of a theorem, but instead are able to provide assurance that a proof does exist. Other ATPs may provide a proof, or remember enough information from the search to be able to construct a proof if required.

Prover9 [McC10] is an automated theorem prover for first-order and equational logic and is the successor to the *Otter* theorem prover.

The user provides a goal and a set of assumptions to Prover9, and the system attempts to find a proof. If the system cannot prove a goal it will either time-out or halt with a message that a proof cannot be found. Determining why the proof could not be found is not always straightforward. If

Prover9 is able to find a proof for a theorem it can display the proof as a sequence of steps, providing the rationale for each step.

Although Prover9 has a fully automatic mode, some difficult problems may require user guidance.

Mace4 [McC10] is paired with Prover9, and can be run in parallel with Prover9. It searches for finite models and attempts to find counter examples.

SPASS [SPA11] is a sound and complete automated theorem prover for first-order logic with equality [HSW99], and is an implementation of a saturation-based resolution and superposition calculus with simplification.

SPASS takes a user's input file that includes a list of axioms and a list of conjectures, and produces an output that states if a proof was found. When running SPASS on the command line, an extra option can be used so that the proof is displayed in the output.

Some systems use ATPs as plug-ins to a more complex environment. An example of such a system is RODIN [rod10]. The RODIN tool is an open tool platform for the rigorous development of dependable software systems. The tool was built as an Eclipse plug-in, with the various components of the tool also being plug-ins. The tool allows a user to build abstract models of software and systems, and then to introduce refinements to produce more detailed and less abstract models of the system.

The tool automatically generates proof obligations from the user's model, and attempts to satisfy those obligations automatically using a collection of plug-in automated theorem provers. As the user continues to work on the tool it will work in the background to find proofs of the proof obligations, and will re-run any proofs that are affected by changes in the model. The theorem provers underneath the Rodin tools (PP and ML) are actually legacy code from the earlier Atelier-B effort. There is new work in place to map proof obligations to Isabelle.

The proof perspective of the RODIN tool allows the user to inspect the list of proof obligations for the current model and instantly tell which have been successfully proven and which have not by a green or red icon displayed next to each item on the list. Where an obligation cannot be automatically satisfied, the user can inspect the proof using the proof viewer, and can add additional hypotheses, add witness values or apply rewrite rules to assist the ATPs.

5.2.2 Interactive Theorem Proving Tools

Interactive (or semi-automated) theorem provers are a category of theorem provers that require a greater degree of user interaction and guidance to assist the tool in the search for a proof. The increased need for user interaction in interactive theorem provers is due to the use of more expressive logics, such as higher order logic, making automated proof searches more complex.

When proving a theorem in an interactive theorem prover, the user will direct the system to the goals using commands, rules or tactics. The system will ensure that rules are applied correctly and may automate what might be considered the simple tedious steps of a proof.

Interactive theorem provers have been applied to a variety of problems such as verification of deep mathematical theorems and software/hardware verification. This section summarises three well known theorem proving systems – HOL, Isabelle and Coq.

HOL4 is an ML-based environment for constructing proofs in higher order logic [SN08] and is the latest version of the HOL system developed by Gordon. The logic implemented by HOL4 is based on Church's Simple Type Theory [Chu40]. The HOL system was originally intended for use in the specification and verification of hardware designs, but also has applications in other areas, including software verification.

HOL is an LCF[1] style prover, meaning the system is built around a 'logical core' or 'kernel' of primitive inference rules. The logic implemented in the HOL kernel is kept intentionally small, with only a few simple axioms and rules of inference. The *metalanguage* (ML) of the proof system can be used to program more complex derived rules, which can be reduced by HOL to a

[1]LCF (Logic of Computable Functions) [GMW79] was on an early theorem prover.

sequence of kernel inferences. This is a powerful way of retaining consistency.

In HOL4, a proof is guided by the user at a high-level using the proof manager, with subsidiary proofs being passed to one of the automated reasoners provided. Tactics can be used to direct the proof construction by reducing the theorem to a series of subgoals to be proved. Once the theorem is proved then it can be stored in the current theory and can be used in subsequent proofs. The system also provides a wide collection of theories that can be used in proofs.

In his masters thesis, Sander Vermolen used HOL4 to automatically discharge proof obligations generated from VDM models [VHL10].

Other systems implementing HOL are *HOL Light* [Har11] and *ProofPower* [Art11].

Isabelle [PNW10] is a generic framework for implementing logic formalisms and interactive theorem proving. Isabelle is not based on a specific logic but instead has a meta-logic in which other logics are represented, allowing the system to be instantiated for different logics.

The meta-logic of Isabelle (*Pure*) is intuitionistic higher-order logic with implication, universal quantifiers and equality [NP92]. A logic that is represented using the meta-logic is called an *Object Logic*. Isabelle comes with several object logics that can be used as reasoning environments, the most developed being HOL. *Isabelle/HOL* is the specialisation of Isabelle for Higher Order Logic (HOL), and is based on Church's Simple Type Theory [Chu40]. The implementation of HOL in Isabelle defines implication, quantification and equality directly in terms of the meta-logic. Using the polymorphic type system, additional axioms are defined to form the logical core to the HOL theory, and other other inference rules are derived using these axioms.

In Isabelle the proof that is currently being worked on is termed a *goal*. A goal consists of a list of assumptions and a conclusions. Proof development is highly interactive and both forward and backwards reasoning are allowed and tactics are provided for both.

The standard user interaction with Isabelle is through the command line (terminal), with no proof support. A more user friendly option is to use Proof General [Pro11] which is a generic interface for proof assistants based on emacs.

Isabelle can be used with the *Isar* language[2] which is an interpreted language designed specifically for theory and proof development. It sits at the top-level and provides proof support, and provides a more structured and more readable proof output.

The *Archive of Formal Proofs* (AFP) is an online collection of proof libraries and examples that have been mechanically checked with Isabelle [WPN08]. Submissions to the archive are refereed.

It is also possible to run Isabelle fully automatically by using additional scripts and tools, along with certain tactics which are run one at a time until one succeeds or all fail [SBBT09].

One of the largest applications of Isabelle is the Bali Project [KNvO$^+$11] which was concerned with formalising aspects of the Java programming language using Isabelle/HOL. The successor to this project was the VerifiCard project [KNS$^+$11] which formalised aspects of JavaCard, a subset of Java. As an example of a deeper mathematical proof: Gödel's proof of the relative consistency of the axiom of choice has been mechanised using Isabelle/ZF [Pau99].

Coq [LG] is a proof assistant that is designed to develop programs and specifications and also to write formal proofs that a program meets its specification. The formal language of Coq is a *Calculus of Constructions* with *Inductive Definitions*, a lambda calculus with a rich type system [Wie06]. The kernel of Coq is implemented using Objective Caml, using only standard features of the language to make it less likely for bugs in the OCaml language to affect Coq. The kernel is also kept intentionally small in an attempt to reduce the risk of implementation bugs.

Programs are usually represented by functions and simple programs can be executed in Coq. It is possible to generate an executable program, in

[2] *Intelligible semi-automated reasoning*

Objective Caml or Haskell, from a specification.

Proof development in Coq is user guided using a tactic language. To construct a proof the user enters the statement to be proved, known as a theorem. The user can then use tactics to decompose the proof goal by introducing simpler subgoals. Once the proof is complete it is saved and can be used in other proofs.

The user can interact with the system using the command line or using one of the available user interfaces such as *CoqIde*, *Pcoq* or *Proof General*.

The expressiveness of the logic of Coq lends itself to formalisations of rich mathematical theorems. User contributed proofs in Coq include a proof of the four colour theorem, a proof of the three gap theorem, and a proof of Buchberger's algorithm.

5.3 Proof Support Tools

Proof checkers or *proof support* tools assist a user to construct a proof and can be used to mechanically check a proof for correctness, with checks usually being performed at the syntactic level. The user is responsible for finding the proof of a conjecture, not the tool. Generally the layout of the proof is intended to closely resemble how the proof would be written on paper.

5.3.1 Mural

The *mural* system was an interactive proof assistant developed in the 1980s at Manchester University and Rutherford Appleton Laboratories [JJLM91]. The tool is no longer available but it is worth discussing since it has many features that are of interest for the research set out in this thesis.

It is probably best to think of the *mural* project as an experiment in the design of interaction modes for the theorem proving task. At the time of its inception, almost all theorem provers were more or less built around the LCF style of interaction: the proof task was split into ever smaller goals until they could be discharged. The *mural* project attempted to change the mode of interaction. To a large extent, it was possible to envision and build the advanced HCI of *mural* because of the advanced workstations that were newly available in the late 1980s. Use of SmallTalk as an implementation language was also a key decision that made building advanced windowing features practical.

Using *mural*, a user could directly interact via an evolving natural deduction proof. The whole of a proof was potentially visible at the same time (with clever features for hiding portions that were not the current focus of the user's attention). Proof construction was user guided in *mural*, so the user was in control, and the tool would assist by checking the user's proof and helping to find rules that could be used as justifications for lines of the proof. The interaction style was very flexible: a user could for example add an unjustified line to the middle of a proof — but a proof was only considered complete when the inserted line was linked to both hypotheses and goal. It was also possible to justify a line in a proof based on some lemma, without having to prove the lemma first.

Another aspect of the generailty of *mural* was its provision of a Logical Frame (see Chapter 3 of [JJLM91]). The tool could be tailored to any (monotone) logic by providing its inference rules as data. In fact there were instantiations for both classical logic and LPF

The tool was developed from a formal description and this was both maintained throughout the project and is published as Appendix C in [JJLM91].

Mural maintained a store of theories, which were representations of mathematical theories. At the trivial level, theories could contain derived logic rules such as distribution of logical and over or; but the more useful role of theories was to organise whole collections of useful lemmas about data structures. The "rule" matching of *mural* spotted the derived rules from any parent theory that would help a user progress a proof.

The description of *mural* as an interaction study concedes the fact that a fully fledged TP system requires as many decision procedures as possible (e.g. Pressburger arithmetic); *mural* did not even have a resolution procedure built in.

5.3.2 Jape

Jape is probably the tool that most closely resembles *mural* but its interaction style was more restrictive that in *mural*.

Jape is another example of a proof support tool, which the creators of the tool term a 'proof calculator'. Jape aims to support the interactive development of proofs in a formal logic, and makes the step-by-step development of a proof easy for novices to use. The user interface presents the proofs in a box-and-line style, adapted from Fitch [Fit52], so that the proof resembles how it would look on paper. Jape is not committed to working in a particular logic, but can use any user defined logic, as long as the logic is expressed as inference rules in a two sided sequent calculus [BS99].

When working on a proof, the interaction style is restricted to forwards or backwards reasoning. A proof is constructed working forwards from hypotheses or working backwards from a conclusion, using proof rules from a library as justification, until the two ends of the proof match up and the proof is complete. A rule is applied as a justification by selecting the lines of the proof that match the hypotheses and conclusion of a rule. The rule options presented to the user for forward and backward steps are not changed according to the selected lines of the proof, but the tool will not allow the user to incorrectly apply a rule as a justification.

5.4 Summary

This chapter has summarised the categories of formal verification tools that are available, highlighting example tools for each.

In some of the tools discussed, it would be possible to encode an operational semantic description of a language. Nipkow has used Isabelle/HOL to provide a formal operational semantics of *Jinja*, a Java-like language [Nip06]. Isabelle/HOL was used to machine check proofs about the language, including type safety of the semantics. Jape can be used to animate the operational semantics of simple languages, as demonstrated in [SB98].

As discussed in Chapter 1, the acceptability of the approach to reasoning about programs introduced in this thesis will likely depend on a suitable support system. Although some of the tools covered in this chapter may be suitable, most have restrictive user interfaces and can be very difficult for novice users to use. For this approach to be accessible, the support system should be intuitive and not have a steep learning curve. The type of system considered to be most suitable for the purposes of this thesis is a proof support tool as described in Section 5.3.

The decision was taken to develop a new proof support tool, rather than using an existing tool. Using an existing tool has obvious advantages, but trying to develop tools that either are no longer actively under development (as is the case with Jape) or are no longer available (Mural) introduces difficulties. Developing a proof support tool from scratch, for the specific purpose of constructing natural deduction proofs using operational semantic, allows the tool to be tailored to the specific requirements of this approach.

The following chapters outline the features and requirements of the proof support tool, and describe its development. The tool is a Mural-like tool, in that interface and interaction style are similar.

Chapter 6

Outline of a Proof Support Tool

A proof support tool is intended to assist a user to write proofs, usually in a manner that reflects how the user would work on paper. The purpose of a proof support tool is to remove the difficulties involved with writing proofs by hand, allowing the user to concentrate on the content of the proof. The types of support provided to the user is generally restricted to managing the layout and formatting of the proof. Although proof support tools do not generally assist the user to complete the proof, they are usually able to check the correctness of the proof at the syntactic level.

As discussed in Section 5.4, the type of tool most suitable for the purposes of this thesis is a proof support tool.

This chapter explains why a proof support tool is necessary, outlines the requirements for the tool, and discusses what we consider to be important features for the tool to have.

6.1 Description of Proof Support Tool

The main purpose of the proof support tool is to provide assistance to the user as he writes a proof, rather than trying to do the proof for the user. With this in mind, proof construction should be guided by the user, and assisted by the tool, meaning the user is in control of how the proof is constructed, the order in which lines are added to the proof and how lines are justified.

The user should not be forced by the tool to work in a particular way when writing a proof. Section 2.5 shows three approaches to building a proof, and we would like the user to be able to construct a proof in any of these ways, or in any other way. The process of building a proof should feel as intuitive and unrestrictive in the tool as it would on paper.

The tool should assist the user by automatically formatting the user's proof as he is working on it, laying out the proof in the style that we use (see Chapter 2). Included in the formatting of the proof is the numbering of the proof lines — the tool should automatically number lines as they are added to a proof, and renumber any lines below the added lines, so that the lines stay numbered sequentially. The tool should also keep line number references used in justifications consistent, so that if a line is renumbered, any references to that line in justifications are also updated, to keep the justifications correct and consistent.

As well as formatting the proof, the tool should also assist the user by making suggestions, such as rules that can be used to justify a line of a proof, but the user should be free to carry on and ignore the suggestions without having to stop and accept or reject them.

The tool should provide the facility to store inference and semantic rules, so that the user can have easy access to them. The tool will require access to such a rule store to allow it to suggest rules to the user that can be used to justify proof lines.

6.2 Features of Proof Support Tool

This section outlines the features considered to be the most important to include in the proof support tool, many of which are listed in the paper [HJ08].

6.2.1 Proof Formatting

Numbering the lines in a proof is usually straightforward, but problems can occur when a new line is inserted into the proof, requiring the lines below it to be renumbered. When working on paper it can look quite messy crossing out and changing line numbers. The main problem comes when updating references to lines in the justifications. Failing to update a reference in a justification will mean the justification becomes incorrect as it is pointing to the wrong line of the proof. Allowing the tool automatically to number the lines that are added to a proof should prevent this problem from occurring. When a new line is inserted into a proof the tool will automatically renumber the lines below, and pointers to those lines in justifications will also be updated to keep them consistent.

When we are working on paper and we insert a line into a proof, we will sometimes give the line a temporary line number that may be out of sequence or include letters, rather than renumbering the lines below. When the proof is complete we might then rewrite the proof neatly and renumber the lines at the same time. As an alternative to automatic line numbering, we could allow the user to manage the line numbers as he adds lines to the proof. Since the user is able to add lines to a proof in any order he likes, he could use a 'temporary' line number when he adds a line[1], then when he has finished a proof or subproof, have the tool renumber the lines so they are numbered sequentially.

6.2.2 Proof Validation

As the user adds new lines to the proof, the tool should suggest rules that may be used as a justification of the line. These suggested rules will have conclusions that can be matched against the proof line. The appropriate rule to use from the list of suggested rules will depend on the hypotheses of the rule, so the user will need to select a rule and then specify which lines of the

[1]Or the tool could assign a temporary line number (which could actually be a letter)

proof match against the hypotheses of the selected rule. The line will then be justified by an appeal to the selected inference rule, with references to the specified lines as parameters to the justification.

We could also make the tool do more extensive checking of rules that can be used to justify a proof line, and find existing lines of the proof that can be matched against the hypotheses of a rule as well as the conclusion. If the tool is unable to match existing proof lines against the hypotheses of a rule it could give the user the option of adding new lines to the proof that will match against the hypotheses of the rule. If the user selects to add these lines, then the tool could proceed to find justifications for these new proof lines.

The tool should be able to check that a user's proof is valid and proven. A proof is valid if all lines of the proof have justifications that refer to rules or definitions that exist in the rule library, and are applied correctly. A proof is proven if it is valid and if all of the rules used in justifications are either axioms or have proofs that are proven. If the user has an invalid justification on a line, the tool should show the user that the justification is invalid in a way that does not stop the user from continuing to work on the proof. We don't, for example, want the tool to make the user correct the justification before he can continue working on the proof, but he should be free to come back and correct it later. The checking of a proof could be done in the background as the user works, as in the RODIN tool [rod10] (see Section 6.3.2).

6.2.3 Proof Support

The tool should have the facility to store rules in a *library* so that the user can have easy access to them. The rule library is also required so that the tool has a store of rules that it can check against lines in a user's proof and suggest rules that can be used to justify lines. Since some inference rules require proofs, the user should be able to check the proof of a rule and also add a proof if the rule does not already have one. This *rule library* might get quite large, but the user shouldn't have to search through the library for a rule every time he wants to add a justification, since the tool should be able to suggest an appropriate rule for a proof line.

The proofs that will be written with this tool may include subproofs, so the tool should be able to handle subproofs, and even nested subproofs. To increase the readability of long proofs with subproofs, the tool could include a feature that allows the user to collapse a subproof so that only its hypotheses and conclusion are visible. The user could expand the subproof when necessary to view or edit the lines of the subproof.

Any proof that we write can also be written as a derived rule [BFL$^+$94]. A derived rule acts as a shorthand for its proof, so we can use a derived rule to justify a line, rather than having to add multiple lines that correspond to the proof of the derived rule. This should be reflected in the tool, so that any proof that the user writes can be used as a derived rule to justify steps in other proofs.

Allowing the user to make use of existing proofs in this way means a large proof can be reduced into a series of smaller proofs. A proof of a program could be broken down into a number of smaller proofs that are used to justify steps in a main proof of the program. This approach is used in the example in Section 4.3.

6.3 Existing Tools

When specifying the requirements for the proof support tool, we looked at several existing theorem proving and model checking tools to see how these tools worked and to find features that would be useful for our tool. The tools that are of most interest are *mural* and RODIN, since they have a number of features that were needed in our proof support tool.

This section only describes the features of these tools that are of interest. The tools are described in more detail in Chapter 5.

6.3.1 Mural

Key features of *mural* [JJLM91] are described in Section 5.3. *Mural* was an interactive proof assistant developed in the 1980s at Manchester University and Rutherford Appleton Laboratories. Probably the most important single feature of *mural* that current research has attempted to copy is the user interaction style. It was clear from the beginning that something like the natural deduction style of proof was mandatory and that its explicit use as a medium for interaction was highly desirable. This way of thinking about proofs based on structural operational semantic descriptions and their incremental construction has pervaded the research reported in this thesis.

The idea in *mural* of constructing theories (of data types) would also play a large part in a fully developed tool for operational semantic proofs — in this latter case, the results might be general lemmas about program constructs. In the extreme case, these could approximate to Floyd/Hoare "axioms" to provide a proof style that comes close to the more common way of reasoning about programs from axiomatic definitions — but of course, it is central to the current research that for general programming languages such rules would not capture the whole language and any "axiomatic" style rules would have side conditions that showed when they could be applied.

It should have been possible to enter by hand into a *mural*-like "logical frame" inference rules that would have been codifications of AS/CC/SOS. Even had *mural* still been available, this would not have offered a very usable system and it was a major part of the demonstration tool that is described in this thesis to accept a more common representation of a language definition and to use the operational semantic rules of the language as inference rules, essentially tailoring the logical frame to the particular language. A system that was not designed for this purpose would always require work-arounds to acheive the desired results.

There is of course also the overriding practical issue that the tool is no longer available! At the point at which the research reported in this thesis was begun, the SmallTalk code was not available. Thanks to Dr Bicarregui the actual code has been recovered but no one has yet been able to do anything with it. CBJ fears that the dependence on specific GUI methods in the implementation of SmallTalk will make this venture uneconomical. In fact, the author of this thesis worked on a reimplementation of from *mural*'s VDM-SL model for a 3rd year undergraduate project. The implementation provides a framework upon which a full working reimplementation of *mural* can be built.

Even with the benefit of hindsight, it is unlikely that the effort spent on resurrecting or reimplementing *mural* would have been cost-effective. That effort might well have distracted the focus of the research away from the provision of tools for easy inputting of language definitions.

It was the *mural* interaction style that was inspirational — getting the code running again would not have paid off.

6.3.2 RODIN

Although RODIN is not a proof support tool, it has some interesting features. The RODIN tool is described in more detail in Section 5.2.1.

RODIN automatically generates proof obligations from the user's model, and then attempts to satisfy those obligations automatically using a collection of plug-in proof checkers. The user can inspect the list of proof obligations for the current model and instantly tell which have been successfully proven and which have not by a green or red icon displayed next to each item on the list. The proof perspective of the RODIN tool allows the user to inspect the proofs of the obligations, and where an obligation cannot be automatically satisfied, the user can add additional hypotheses, add witness values or apply rewrite rules to assist the proof checker. The proof of each of the proof obligations can be viewed as a proof tree, which the user can inspect and 'prune' if necessary. Pruning a proof tree at a particular point removes everything below that point in the tree.

One of the features that we are interested in, with respect to our tool, is how the RODIN tools performs checks on a user's model as the user is working on it. The user does not have to click any buttons to have the model checked. If the user modifies the model then the proof obligations are automatically re-checked to make sure that they are still valid for the changed model. We would like our proof support tool to be able to check a user's proof as they write it, make suggestions to the user and flag lines that are incorrectly justified, all in the background without getting in the way.

The proof perspective is also a feature that is of interest, as the tool will need display proofs in a way that will be convenient for the user.

6.4 Interaction Style

Section 2.5 outlines the reasoning approaches or directions (forwards, backwards and mixed) that might be taken when writing a proof, and illustrates the approaches with examples. This section works through a simple proof to show how a proof could be written using the tool, and tries to describe the expected interaction between the user and a proof support tool when writing the proof. The example proof from Section 3.4.4 is used for this going through the steps that the user has to take when writing the proof, and also the steps that will be taken or suggested by the tool.

It is important that the proof support tool does not restrict the user when writing the proof, but should allow the user to work in any way that is convenient. The user should be able to work forward from the hypotheses or backwards for the conclusion, or just add a line in the middle of the proof.

The first step for the user to take is to specify the hypotheses and conclusion of the proof, as below.

$$\begin{array}{ll}
\textbf{from } (\textit{mk-If}(x < 0, \textit{mk-Assign}(r, -x), \textit{mk-Assign}(r, x)), \sigma_0) \xrightarrow{s} \sigma_f & \\
\quad\vdots & \\
\textbf{infer } \sigma_f(r) \geq 0 & ??
\end{array}$$

The first hypothesis states that the execution of the program in an initial state, σ_0, will result in a final state σ_f. The conclusion of the proof states that the postcondition of the program will hold in the final state.

Lines of the proof that are unjustified are labelled with '??' to show that they require a justification but have not been justified yet.

We know that the part of the *If* statement that is executed is dependent upon the value of its condition expression, so we start by stating that the condition will evaluate to either *true* or *false*.

	from $(\mathit{mk\text{-}If}(x < 0, \mathit{mk\text{-}Assign}(r, -x), \mathit{mk\text{-}Assign}(r, x)), \sigma_0) \xrightarrow{s} \sigma_f$	
1.	$x < 0 : \mathbb{B}$	??
2.	$(x < 0, \sigma_0) \xrightarrow{e} \mathbf{true} \vee (x < 0, \sigma_0) \xrightarrow{e} \mathbf{false}$	??
	infer $\sigma_f(r) \geq 0$	??

After adding line 2, the tool suggests that the *or elimination* rule be used to justify the conclusion using the conjunction and two subproofs. When we accept the suggestion, the tool then adds subproofs as lines 3 and 4, and adds the justification to the conclusion.

	from $(\mathit{mk\text{-}If}(x < 0, \mathit{mk\text{-}Assign}(r, -x), \mathit{mk\text{-}Assign}(r, x)), \sigma_0) \xrightarrow{s} \sigma_f$	
1.	$x < 0 : \mathbb{B}$	??
2.	$(x < 0, \sigma_0) \xrightarrow{e} \mathbf{true} \vee (x < 0, \sigma_0) \xrightarrow{e} \mathbf{false}$	??
3.	**from** $(x < 0, \sigma_0) \xrightarrow{e} \mathbf{true}$	
	$\vdots$	
	infer $\sigma_f(r) \geq 0$	??
4.	**from** $(x < 0, \sigma_0) \xrightarrow{e} \mathbf{false}$	
	$\vdots$	
	infer $\sigma_f(r) \geq 0$	??
	infer $\sigma_f(r) \geq 0$	$\vee$-E(2, 3, 4)

We can now start to fill out the first subproof, line 3, for which we are dealing with the case when the *If* statement's condition evaluates to *true*, so we can use the If-T rule. We add line 3.α to the sub proof, stating that the whole *If* statement will go to some new state that we call σ_1, and we justify the line using the If-T rule, and using the hypothesis of the subproof on line 3 for the first parameter. We will now just concentrate on the subproof on line 3, rather than repeating the whole of the proof.

3. **from** $(x < 0, \sigma_0) \xrightarrow{e}$ **true**
3.α $(\textit{mk-If}(x < 0, \textit{mk-Assign}(r, -x), \textit{mk-Assign}(r, x)), \sigma_0)$
$\xrightarrow{s} \sigma_1$ if-T(3.h1, ??)
infer $\sigma_f(r) \geq 0$??

We are using temporary line numbers for the subproof since we expect that line 3.α will be in the middle part of the subproof, and we will be working backwards from this line to the hypotheses. When working forwards it is easy enough to have the lines automatically labelled sequentially, but when working backwards it may be preferable to use temporary line numbers and update them later, so that the line numbers are not automatically updated every time a line is added.

Notice that there is a second parameter to the justification on line 3.α that we have not filled in yet. This refers to the hypothesis of the If-T rule that states that the 'then' part of the *If* statement will result in some new state under the $\xrightarrow{s}$ transition relation. In the case of this *If* statement, the 'then' part is an *assignment* statement, so we would expect the tool to suggest a new line containing the 'then' part of the *If* statement and use the Assign rule as a justification, and update the justification of line 3.α with a reference to this new line.

3. **from** $(x < 0, \sigma_0) \xrightarrow{e}$ **true**
3.β $(\textit{mk-Assign}(r, -x), \sigma_0) \xrightarrow{s} \sigma_0 \dagger \{r \mapsto -\sigma_0(x)\}$ Assign(??)
3.α $(\textit{mk-If}(x < 0, \textit{mk-Assign}(r, -x), \textit{mk-Assign}(r, x)), \sigma_0)$
$\xrightarrow{s} \sigma_0 \dagger \{r \mapsto -\sigma_0(x)\}$ if-T(3.h1, 3.β)
infer $\sigma_f(r) \geq 0$??

The tool can continue to suggest new lines that can be used to correctly apply a justification to a line, until we have reached a point where no more extra lines need to be added, as shown below.

from $(mk\text{-}If(x < 0, mk\text{-}Assign(r, -x), mk\text{-}Assign(r, x)), \sigma_0) \xrightarrow{s} \sigma_f, x\colon Id$

$\vdots$

3. **from** $(x < 0, \sigma_0) \xrightarrow{e}$ **true**

3.δ $(x, \sigma_0) \xrightarrow{e} \sigma_0(x)$ Id(h2)

3.γ $(-x, \sigma_0) \xrightarrow{e} -\sigma_0(x)$ UnaryExpr(3.δ)

3.β $(mk\text{-}Assign(r, -x), \sigma_0) \xrightarrow{s}$
$\sigma_0 \dagger \{r \mapsto -\sigma_0(x)\}$ Assign(3.γ)

3.α $(mk\text{-}If(x < 0, mk\text{-}Assign(r, -x), mk\text{-}Assign(r, x)), \sigma_0)$
$\xrightarrow{s} \sigma_0 \dagger \{r \mapsto -\sigma_0(x)\}$ if-T(3.h1, 3.β)

infer $\sigma_f(r) \geq 0$??

The justification of line 3.1, using the Id semantic rule, requires that we know that x is of type *Id* (rather than an *Expr*), so we add an additional hypothesis to the proof that states that x has the type *Id*.

At this point we will be working forwards from line 3.α, rather than backwards, so we can ask the tool to renumber the lines of the subproof to make them sequential and to replace the temporary line numbers that we were using. When the lines are renumbered, the references in justifications should also be updated by the tool with the appropriate line number to keep the proof consistent.

	from $(\textit{mk-If}(x < 0, \textit{mk-Assign}(r, -x), \textit{mk-Assign}(r, x)), \sigma_0) \xrightarrow{s} \sigma_f$,	
	$x{:}\, \textit{Id}$	
1.	$x < 0 : \mathbb{B}$	??
2.	$(x < 0, \sigma_0) \xrightarrow{e}$ **true** $\vee$ $(x < 0, \sigma_0) \xrightarrow{e}$ **false**	??
3.	**from** $(x < 0, \sigma_0) \xrightarrow{e}$ **true**	
3.1	$(x, \sigma_0) \xrightarrow{e} \sigma_0(x)$	Id(h2)
3.2	$(-x, \sigma_0) \xrightarrow{e} -\sigma_0(x)$	UnaryExpr(3.1)
3.3	$(\textit{mk-Assign}(r, -x), \sigma_0) \xrightarrow{s}$ $\sigma_0 \dagger \{r \mapsto -\sigma_0(x)\}$	Assign(3.2)
3.4	$(\textit{mk-If}(x < 0, \textit{mk-Assign}(r, -x), \textit{mk-Assign}(r, x)), \sigma_0)$ $\xrightarrow{s} \sigma_0 \dagger \{r \mapsto -\sigma_0(x)\}$	if-T(3.h1, 3.3)
	$\vdots$	
	infer $\sigma_f(r) \geq 0$	??
4.	**from** $(x < 0, \sigma_0) \xrightarrow{e}$ **false**	
	$\vdots$	
	infer $\sigma_f(r) \geq 0$	??
	infer $\sigma_f(r) \geq 0$	$\vee$-E(2, 3, 4)

The next step in completing the proof would be to relate the final state given in line 3.4 to the property that is stated in the goal of the subproof, and then complete the subproof of line 4. The completed proof can be found in Section 3.4.4.

6.5 Summary

This chapter outlines the features and goals of the prototype proof support tool that is developed as part of this thesis.

The chapter also illustrates the kind of interaction expected between the user and the tool when working on a proof.

The following chapter describes the development of the prototype proof support tool.

Chapter 7

Proof Support Tool: A Prototype

This chapter discusses the development of a prototype for a proof support tool that allows the use of operational semantic rules in proofs about programs. The intention of building a prototype proof support tool is to demonstrate that appropriate tool support benefits the approach to reasoning about a computer program using the operational semantics of a language.

The development of the prototype started with an abstract model of a proof system (as described in Section 7.1) which was modelled using the *Vienna Development Method* (VDM). This model was then used as the basis for the prototype.

The code generation feature of the VDM toolbox [CSK10] was used to produce a Java implementation of the model. The generated code provided a framework for the construction of the prototype tool, and is described in more detail in Section 7.2. The last step was to build a user interface and link it to the generated Java code.

7.1 VDM Model

The starting point in developing the proof support tool was to build a formal model of a basic proof system that modelled inference rules and proofs, and allowed for reasoning in propositional logic. The final model included support for quantified expressions.

The purpose of the model is to specify the structure and required functionality of the prototype proof support tool, that are independent of an implementation. Using a modelling language like VDM allows us to make precise descriptions about the desired behaviour and functionality of the tool [FLM+05]. A VDM model allows invariants to be specified on types and instance variables and pre- and post-conditions on operations and functions. These allow the formalisation of constraints on the model that will need to be followed in the implementation, and that can be be checked by the Toolbox. Writing such constraints for the model also forces us to think about such details at an early stage of the development cycle.

The model went through several iterations, starting as a very simple model written in VDM-SL, and then was eventually translated into VDM++[1], which is an object oriented version of the language. The main purpose in converting the model to VDM++ was to take advantage of the code generation feature of the Toolbox, and generate Java code directly from the model. Converting the model to VDM++ also made it more readable, since operations were grouped into classes, rather than having a single operation that had different cases to handle each class that could be passed to it.

This section describes the final version of the model, which was written in VDM++, but here classes are represented as records instead to make it easier to read. Record names relate to class names, and a record's fields relate to the *instance variables* of the class.

[1]The language definition for VDM-SL can be found in [LHB+96]. The VDM++ language description can be found in [CSK05], and [FLM+05] gives a good overview of using VDM++ in practical applications

7.1.1 Forms

In the VDM model, the structure that is used to write expressions is a *Form*, which is the parent-class for a number of classes.

$$\begin{aligned} \mathit{Form} = {} & \mathit{RelForm} \mid \mathit{Monad} \mid \mathit{Sequent} \mid \mathit{PHole} \mid \mathit{VSymb} \mid \mathit{BoolVal} \\ & \mid \mathit{MapLit} \mid \mathit{Configuration} \mid \mathit{IntLit} \mid \mathit{LangForm} \mid \mathit{LangFormHole} \\ & \mid \mathit{LookupForm} \mid \mathit{MapDefn} \mid \mathit{QForm} \mid \mathit{TypeDefn} \end{aligned}$$

Rather than explaining the structure of each of these classes, it would be more instructive to show examples of the kind of expressions that can be written with each of the subclasses.

RelForm	$a \wedge b$
Monad	$\neg a$
Sequent	$a, b \vdash c$
PHole	[1]
VSymb	a
BoolVal	**true**
MapLit	$\{a \mapsto b\}$
Configuration	(s, σ)
IntLit	5
LookupForm	$m(a)$
MapDefn	$a \xrightarrow{m} b$
QForm	$\forall x \in \mathbb{N} \cdot x > 5$
TypeDefn	$\textsc{Int} \mid \textsc{Bool}$

Many of these classes will contain *Form*s. For example, *RelForm* has a left-hand side and a right-hand side which are both of type *Form*. In the example used above, both sides consist of a *VSymb*, which is the class used to represent a variable name.

The class *PHole* is used to represent a placeholder, and is explained in Section 7.1.5. The classes *LangForm* and *LangFormHole* are both explained in Section 7.2.4.

7.1.2 Model of a Proof

The main focus of the VDM model of the proof system is to model the kind of proofs in Section 2.1, which are identical to the style of [Jon90].

The proof class itself is minimal; it simply has a *name*[2] and a *goal*.

$$\begin{array}{lrcl} \mathit{Proof} :: & \mathit{goal} & : & \mathit{Judgement} \\ & \mathit{name} & : & \mathit{char}^{+} \end{array}$$

A proof is modelled as a collection of *Judgement*s, where each judgement represents a 'line' in a proof. One of these judgements is selected as the goal of the proof, and a pointer[3] to it is stored in the *goal* field. The goal judgement is the root of the proof and is linked to all of the other *Judgement*s in the proof.

Appendix F shows an example proof written as judgements, showing the contents of each judgement field. The structure of a *Judgement* is shown below.

$$\begin{array}{lrcl} \mathit{Judgement} :: & \mathit{hyps} & : & \mathit{Judgement}^{*} \\ & \mathit{subJs} & : & \mathit{Judgement}\text{-}\mathbf{set} \\ & \mathit{concl} & : & \mathit{Form} \\ & \mathit{just} & : & \mathit{Justification} \\ & \mathit{parent} & : & [\mathit{Judgement}] \\ & \mathit{name} & : & \mathit{char}^{+} \end{array}$$

All *Judgement*s in the main body of a proof are subordinate to the goal judgement, so will have references in the goal judgement's *subJs* field. In the case of a subproof, only the conclusion line of the subproof is subordinate to the conclusion line of the main proof. The hypotheses and other lines in the subproof are linked to the rest of the proof through the conclusion line of the subproof.

[2]Notice that for the type of *name*, rather than a *token* which would be usual for a VDM specification, a sequence of characters (char^{+}) is used. This is so the name will be of type String when the Java code is generated from the model (See Section 7.4.1).

[3]Further explanation about how pointers are used in the VDM model are given in Section 7.4.1

The *Judgement*s of a proof form a doubly linked tree structure, with the *goal Judgement* as the root. The references to parent and child nodes are stored in the *Judgement*'s *subJs* and *parent* fields. The *parent* field is optional[4] as the root node has no parent, and so will have a *nil* value in its *parent* field.

Hypotheses are also subordinate to a goal judgement, but references to hypotheses are stored in a goal *Judgement*'s *hyps* field instead of in the *SubJs* field. A judgement may link to hypotheses if it is a conclusion line, either of the whole proof or of a subproof.

The body of the judgement, being an expression that represents the proof line, is the *concl* field. A *Form* is the parent class for any expressions. There are no restrictions on what can be a proof line (other than the *Form* must be well-formed), but a proof line will require a justification.

Each *Judgement* in a formal proof requires a justification, which will be stored in the *just* field. Justification are described in Section 7.1.3 below.

The *name* of a *Judgement* is used as an identifier when the *Judgement* is being referred to, such as in a justification. The name is also used as the line number of the *Judgement* when it is part of a proof.

7.1.3 Justifications

When constructing a formal proof, each line requires a justification. A justification should be attached to each judgement in a proof. The model uses the following justifications.[5]

$$\mathit{Justification} = \textsc{Hyp} \mid \mathit{RuleAppl} \mid \mathit{DefnAppl} \mid \textsc{EmptyJustif}$$

[4]An optional type means it can take a value or a **nil** value

[5]Here *Hyp* and *EmptyJustif* are written as quote literals, but in the VDM++ model they are actually classes since although they have no fields, they still have operations. Also, in a VDM++ model this *union type* would instead be implemented using class inheritance, so each of the four classes would be specified as being a subclass of *Justification*. This is another reason why *Hyp* and *EmptyJustif* need to be classes.

The justification field in a *Judgement* is not optional, so always requires a justification. The default justification to use is an *EmptyJustif*, meaning an empty justification, which is used when the judgement has not been justified yet. The justification field could be made optional and a *nil* used if the judgement is unjustified, but using an *EmptyJustif* allows us to implement operations in the class, rather than trying to handle a *nil* entry in the *Judgement* class. Also, hypotheses require no justification, so we want the tool to be able to distinguish between a *Judgement* that has no justification, and a *Judgement* that is a hypothesis and requires no justification.

To show that a *Judgement* is a hypothesis, we use HYP. When a proof is checked the tool will see a hypothesis as proven. A *Judgement* that has an *EmptyJustif* as its justification is unproven.

The usual way for a line to be justified in a proof is by reference to an inference rule. In the model this is called a justification by rule application. The *RuleAppl* points to the inference rule that is being used as a justification, and also the lines of the proof that the justification depends on.

$$\begin{array}{lrcl} \mathit{RuleAppl} :: & \mathit{rule} & : & \mathit{char}^+ \\ & \mathit{theory} & : & \mathit{char}^+ \\ & \mathit{dependsOn} & : & \mathit{Judgement}^* \end{array}$$

$$\begin{array}{l} \mathit{inv\text{-}RuleAppl}(\mathit{mk\text{-}RuleAppl}(\text{-}, \text{-}, \mathit{dep})) \quad \triangleq \\ \qquad \forall i, j \in \mathbf{inds}\ \mathit{dep} \cdot i \neq j \ \Rightarrow\ \mathit{dep}(i) \neq \mathit{dep}(j) \end{array}$$

The name of the rule being used as a justification is stored in the *rule* field[6], and the *theory* field stores the name of the theory (see Section 7.1.8) to which the rule belongs.

The *dependsOn* field stores pointers to the judgements that match against the hypotheses of the inference rule that is used as a justification. The invariant on *RuleAppl* states that the dependancies should be unique.

Section 2.4 shows that syntactic definitions can be defined that allow an

[6]See Section 7.4.1 for discussion on why name is used to reference a rule, rather than a pointer

expression to be re-written by *folding* or *unfolding* it. In the model, a folded or unfolded line is justified using a *DefnAppl*.

$$\begin{array}{lrcl} \mathit{DefnAppl} :: & \mathit{defn} & : & \mathit{char}^+ \\ & \mathit{theory} & : & \mathit{char}^+ \\ & \mathit{dependsOn} & : & \mathit{Judgement}^* \end{array}$$

$$\mathit{inv\text{-}DefnAppl}(\mathit{mk\text{-}DefnAppl}(\text{-}, \text{-}, \mathit{dep})) \quad \triangleq \quad \mathbf{len}\ \mathit{dep} = 1$$

The *defn* field stores the name of the definition being used as a justification, and the *theory* field stores the name of the theory to which the definition belongs.

The invariant on *DefnAppl* states that the number of *Judgement*s that the justification depends on should be exactly one, since a definition only takes one parameter. This is equivalent to making the type of the *dependsOn* field a single *Judgement* and having no invariant. This, however, would cause some difficulty in the implementation of the parser (see Section 7.4.1).

7.1.4 Inference Rules

Inference rules are used in proofs to describe what can be inferred from a set of knowns. They are explained in detail in Section 2.3. The main parts of an inference rule are the hypotheses and conclusion.

$$\begin{array}{lrcl} \mathit{Rule} :: & \mathit{hyps} & : & \mathit{Form}^* \\ & \mathit{concl} & : & \mathit{Form} \\ & \mathit{proof} & : & [\mathit{Proof} \mid \textsc{Axiom} \mid \textsc{sos} \mid \textsc{as} \mid \textsc{cc}] \\ & \mathit{name} & : & \mathit{char}^+ \end{array}$$

When an inference rule is used as a justification in a proof, lines of the proof are matched against hypotheses of the inference rule. The hypotheses of the inference rule are stored in a sequence since their order is important, making it possible to check that justifications are valid.

It can be determined if an inference rule is valid by referring to the *proof* field. An inference rule that is taken to be self-evident is known as an Axiom, and so its *proof* field will contain the AXIOM quote literal, and it will be taken to be valid and proven. Other inference rules require a proof to show that they are valid. In which case, a pointer to a proof of the rule will be stored in the *proof* field. An inference rule that has no proof will have *nil* in the proof field (since it is an optional field), making the rule unproven.

We also write the abstract syntax, context conditions and semantic rules of a programming language as inference rules (see Section 7.1.10). For these we use the quote literals SOS, AS and CC in the proof field. Inference rules with theses values in their proof field are taken to be valid and proven.

7.1.5 Placeholders and Binding

When we write an inference rule in our proof system, we define parts of the rule to be *placeholders*, which are represented in the model as the *PHole* class. These placeholders represent free variables and mark the points of the rule that can be instantiated, or filled.

$$PHole :: id : char^+$$

A *PHole* has an *id* to distinguish it from other *PHoles*[7]. Placeholders can be instantiated by any class that is a subclass of *Form* (see Section 7.1.1). Figure 7.1 shows the *and introduction* rule, with the variables marked as placeholders.

In Figure 7.1 the placeholder [1] could either be instantiated by the variable x, or by the expression $y \vee z$, or any other expression of type *Form*. The instantiation of a placeholder should be consistent throughout the rule. So both instances of the placeholder [1] should be instantiated with the same expression.

[7]Athought the id in *PHole* is defined as a sequence of characters (a string), a number is usually used for the placeholder id.

$$\boxed{\wedge\text{-I}}\ \frac{\begin{array}{l}[1]\\ [2]\end{array}}{[1] \wedge [2]}$$

Figure 7.1: ∧-Introduction rule, with placeholders.

The proof system used in this thesis uses inference rules as justifications to conjectures in a proof. A justification by application of an inference rule is essentially stating that the proof line is a valid instantiation of the conclusion line of the inference rule, and the lines it depends on for the justification are valid instantiations of the hypotheses of the inference rules.

A rule can only be added as a justification if the conclusion line of the rule matches the line of the proof. But invalid justifications are not disallowed, since only the conclusion of the rule is checked when the rule is selected. The justification could be made to depend on other lines of the proof that do not match the hypotheses of the rule. This approach is a little more flexible than only allowing valid justifications to be added to a proof. The system is however able to perform a more thorough check of a justification to determine if it is valid, but this check is only performed when we want to check if the proof is valid and complete, rather than as justifications are added.

To check the justification of a proof line we check that the proof line and rule's conclusion have the same form (both conjunctions or disjunctions etc.) and build a mapping of *PHole*s to *Form*s that shows how the placeholders should be filled. We then check the justification's dependencies against the rule's hypothesis and add to the mapping. If at any point we come across a contradictory mapping then we know that the line is not a valid instantiation of the part of the rule.

To show how this works, an incomplete proof is used as an example, with a line that uses the *and introduction* rule (see Figure 7.1) as a justification.

	from ...	
1	a	??
2	c	??
3	$a \wedge b$	$\wedge$-I(1,2)
	$\vdots$	
	infer ...	

To check that line 3 is justified correctly we first check that the form of the line matches the form of the conclusion of the inference rule. Since they are both conjunctions the line is a valid instantiation of the conclusion line of the inference rule. The next step is to build a mapping of *PHole*s to *Form*s from line 3 and the rule's conclusion. This gives us the mapping below.

$\{[1] \mapsto a, [2] \mapsto b\}$

We repeat this for line 1 and the first hypothesis of the *and introduction* rule, and get the following mapping.

$\{[1] \mapsto a\}$

We then need to combine this with our first mapping. Since [1] is already in our first mapping we check that they both map to the same term, which they do. We can then repeat for line 2 and the next hypothesis of the inference rule, which gives us the following mapping.

$\{[2] \mapsto c\}$

The placeholder [2] already exists in the first mapping so we compare the terms that they map to and find that they map to different terms. This means we have found a conflict, so that line 2 cannot be used in the justification of line 3, making the justification of line 3 invalid.

7.1.6 Sequent

A *sequent* states that some conclusion is derivable from some hypothesis. A sequent may also include *sequent varaibles* that are bound throughout the sequent. Sequents are covered in Section 2.2. Below is the definition for a sequent in the VDM model.

$$\begin{array}{lrcl} \mathit{Sequent} :: & \mathit{lhs} & : & \mathit{Form}^* \\ & \mathit{rhs} & : & \mathit{Form} \\ & \mathit{bindVars} & : & \mathit{VarSymb}^* \end{array}$$

The hypotheses of the sequent are contained in the *lhs* field, and the conclusion in the *rhs* field. Sequent variables that are bound in the sequent are stored in the *bindVars* field.

7.1.7 Quantified Expressions

This chapter does not go into any detail about how expressions are modelled, apart from the brief overview of *Form*s in Section 7.1.1, where there is a brief explanation of how quantified expressions are represented in the VDM model. It is worth here discussing in more detail how the model represents quantified expressions.

$\mathit{QSymb} = \textsc{UnivQ} \mid \textsc{ExistsQ}$

$$\begin{array}{lrcl} \mathit{QForm} :: & \mathit{symbol} & : & \mathit{QSymb} \\ & \mathit{bindVars} & : & \mathit{VarSymb}^* \\ & \mathit{bindTypes} & : & \mathit{Form}^* \\ & \mathit{pred} & : & \mathit{Form} \end{array}$$

$\mathit{inv\text{-}QForm}(\mathit{mk\text{-}QForm}(\text{-}, bv, bt, \text{-})) \quad \triangleq \quad \mathbf{len}\ bv = \mathbf{len}\ bt$

A quantified expression is represented in the model by *QForm*, and can be can be a universally or existentially quantified expression, depending on which *QSymb* it uses.

Variables that are bound by the quantified expression are stored in the *bindVars* and *bindTypes* fields. The *bindVars* field contains the names of the variables that are bound by the expression; *bindTypes* contains the types of the bound variables. In the *bindTypes* field, the type *Form* is used to represent the type of the bound variable so that, for example, we can state that the variable is in the domain of a mapping. Storing the variables and types in two sequences (rather than a map) allows us to keep the order of the list of bound variables, which is essential when instantiating the placeholders in a quantified expression. The invariant on the datatype ensures that the length of both sequences is always the same.

The *pred* field stores the predicate of the quantified expression. The predicate may contain variables that are bound by the quantified expression, as well as free variables.

7.1.8 Theory

A *Theory* is a collection of axioms, inference rules and definitions that are related in some way [BFL+94].

$$\begin{array}{lrcl} \mathit{Theory} :: & \mathit{rules} & : & \mathit{Rule}\text{-}\mathbf{set} \\ & \mathit{proofs} & : & \mathit{Proof}\text{-}\mathbf{set} \\ & \mathit{defns} & : & \mathit{Defn}\text{-}\mathbf{set} \\ & \mathit{name} & : & \mathit{char}^{+} \end{array}$$

A *Theory* in the model also includes a collection of proofs that are related to the theory, by either being a proof of an inference rule or just mostly using inference rules from the theory.

7.1.9 Library

A *library* is a convenient data structure for grouping all of the theories together.

$$\mathit{Library} :: \mathit{theories} : \mathit{Theory}\text{-}\mathbf{set}$$

7.1.10 Modelling a language

In the proof system that is being modelled we want to be able to write the kinds of proofs introduced in Section 3.4.4, where we can reason about programs using operational semantics. To do this we need to have the language specification, including the semantics, available to be used as justifications.

In an early model, once simple proofs had been modelled in propositional logic, the model was extended, adding additional classes to allow language descriptions to be added to the rule library, with semantic rules being different from inference rules. The final model was greatly simplified by specifying the language using standard inference rules (see Section 7.1.4), rather than adding new classes and types to handle the language specification and justifications using semantics.

The early sections in Chapter 3 cover the usual way of writing the abstract syntax, context conditions and semantics for a language. For a language specification to be useful to the model we need to write it as a collection of inference rules. Below is a demonstration of how the abstract syntax, context conditions and semantics for an *If* statement are written as inference rules.

Abstract Syntax

The usual way of writing an abstract syntax is to use the VDM record type. For an *If* statement we would write

$$\begin{array}{lcl} \mathit{If} :: \mathit{test} & : & \mathit{Expr} \\ \mathit{th} & : & \mathit{Stmt} \\ \mathit{el} & : & \mathit{Stmt} \end{array}$$

This is quite straightforward to convert to an inference rule, shown below.

$$\boxed{\textsf{as-If}}\ \frac{\begin{array}{l} \mathit{test} \in \mathit{Expr} \\ \mathit{th} \in \mathit{Stmt} \\ \mathit{el} \in \mathit{Stmt} \end{array}}{\mathit{mk\text{-}If}(\mathit{test}, \mathit{th}, \mathit{el}) \in \mathit{If}}$$

The definition becomes a rule about the types of the various parts of the *If* statement. From the rule we can infer that if all of the parts of the

If statement have the appropriate type, and they are combined using the constructor, then we have an *If* statement.

Context Conditions

Context conditions are written as VDM functions that are able to determine if a statement is valid or not. The language definition in Appendix B uses the function *wf-Stmt* to determine if a statement is valid. The function takes a statement and a mapping of declared variables and their types, and returns *true* or *false*. Below we show the signature for the *wf-Stmt* function, and the case that deals with an *If* statement.

$$\mathit{wf\text{-}Stmt} : \mathit{Stmt} \times \mathit{Id} \xrightarrow{m} \mathit{Type} \to \mathbb{B}$$

$$\begin{aligned}
&\mathit{wf\text{-}Stmt}(\mathit{mk\text{-}IfStmt}(\mathit{test}, \mathit{th}, \mathit{el}), \mathit{tpm}) \quad \triangleq \\
&\qquad \mathit{typeof}(\mathit{test}, \mathit{tpm}) = \textsc{Bool} \wedge \\
&\qquad \mathit{wf\text{-}Stmt}(\mathit{th}, \mathit{tpm}) \wedge \\
&\qquad \mathit{wf\text{-}Stmt}(\mathit{el}, \mathit{tpm})
\end{aligned}$$

The *typeof* function used in the *wf-stmt* function takes an expression and a mapping of declared variables to their type (*Id* to *Type*), and returns the type of the expression. This is determined by looking at the relations and variables used in the expression. The language definition in Appendix B includes the definition of the *typeof* function.

To write this context conditions as inference rules we introduce a new type, *wfStmt*, which is a subtype of *Stmt*. A *Stmt* is also a *wfStmt* if it is *well-formed*, meaning all of its component parts are also well-formed. A separate rule is defined for each of the cases of the *wf-Stmt* function. The above case of the *wf-Stmt* function would be written as an inference rule as below.

$$\boxed{\textsf{cc-If}}\ \frac{\begin{array}{l} tpm\colon Id \xrightarrow{m} Type \\ tpm \vdash typeof(test)\colon BoolTp \\ tpm \vdash th\colon wfStmt \\ tpm \vdash el\colon wfStmt \end{array}}{tpm \vdash mk\text{-}IfStmt(test, th, el)\colon wfStmt}$$

Comparing *cc-IF* rule with the equivalent function, we can see that the pattern matching condition of the function forms part of the conclusion, and the body of the function becomes the hypotheses.

The *tpm* variable used in the inference rule has the same type as the type map used in the *wf-Stmt* function, and is used in the same way. The *tpm* mapping comes from a *Program* and is a mapping of declared variable names to their types. The contents of the type map has a bearing on whether a statement is well-formed or not, since, for example, an assignment to an undeclared variable is not well-formed. So the context condition rule is essentially stating that a statement is well-formed with respect to a given type-map.

The *cc-If* rule states that if the *test* part of the *If* statement is a boolean expression, and the *th* and *el* parts are well-formed in the type map *tpm*, then the whole *If* statement will be well-formed in *tpm*, and will therefore be of type *wfStmt*.

Semantics

There was no need to change how semantic rules are written, as they are already written as rules.

$$\boxed{\textsf{sos-If-t}}\ \frac{\begin{array}{l} (test, \sigma) \xrightarrow{e} true \\ (th, \sigma) \xrightarrow{s} \sigma' \end{array}}{(mk\text{-}IfStmt(test, th, el), \sigma) \xrightarrow{s} \sigma'}$$

It was, however, necessary to make additions to the model so that $\xrightarrow{s}$

and $\xrightarrow{e}$ could be used as relation symbols.

7.2 Java Prototype

The section covers the development of the Java prototype of the proof support tool. The prototype includes the key features that were described in Chapter 6.

We wanted to be able to make use of the formal model of the proof system in the development of the prototype. The VDM Toolbox provides two ways of doing this. The first is to use the *dynamic link facility* of the VDM Toolbox, that allows VDM models to be linked to a Java or C++ program. Method calls at the program level are translated to function or operation calls on the model, with results being passed back to the program.

We decided not to use this method since the overhead involved in translating commands and passing them to the model, then passing the results back to the program, would cause delays in the responsiveness of the interface.

Another approach is to use the code generation [CSK08] facility of the Toolbox to generate Java code from the model. The advantage of this approach is that Java code can be generated directly from the model, compiled and then run, without having to link between the model and a user interface.

We decided to use the second approach, to generate Java directly from the VDM++model using the VDM Toolbox. We see this way as being more convenient, and it should execute faster than using the dynamic link facility. Generating Java from the model does not mean that the model is discarded once the code is generated, but we can still use the model and make necessary changes to it, and then generate the Java code again.

7.2.1 Java Code Generation

Using the Java code generation facility of the VDM Toolbox, it is possible to convert the model to Java classes, once it has been syntax and type checked without any errors. This process is quite straightforward — the Java code is generated with the click of a button, and the generated code mostly compiles without any problems.

One of the reasons for converting the model to VDM++ from VDM-SL was to be able to use the VDM Toolbox to generate Java code from the model. A class in the VDM++ model becomes a class in the generated Java code.

The Java code generated from the model gave a collection of classes[8] that provided a starting point for the prototype, providing the datatypes and classes needed to represent inference rules, proofs etc. The next step was to write a parser (see Section 7.2.2) that could read in a file and build abstract syntax trees (ASTs) using the generated Java classes.

Using the code generation facility of the toolbox to generate Java code from the VDM model has several advantages over writing the java classes from scratch, one of them being the time that is saved by not having to write a Java implementation of the model. Also, we did not have to worry about any errors or inconsistencies that might have been introduced in a hand written Java implementation.

If at any point in the implementation a change needs to be made to the model, the required changes can be made to the VDM model, and then the Java implementation can be instantly updated with the new changes by re-generating the Java code from the updated model.

During the implementation of the prototype there were a few changes that needed to be made to the model that were specifically for the benefit of the generated Java code. Some of these issues are discussed in Section 7.4.1.

When a Java class is generated from a VDM model, the class uses other classes that are in the VDM Java Library package provided by the VDM Toolbox. The import statement is automatically added to the Java class. Any other classes that are written that make use of the generated class may also have to import this package and handle the custom exceptions added by the VDM Toolbox.

[8]This collection of classes generated from the VDM++ model will be referred to as a framework

7.2.2 Parser

Once the framework of Java classes had been generated from the VDM model, the next thing to do was build a parser to make use of the classes and instantiate them with rules and proofs, etc.

The first thing to do was to decide on a concrete syntax to use, and then write a file containing inference rules, definitions and proofs, to give some test data to use when building a parser.

Using the scanner (or lexical analyser) generator JFlex [KRD10], a scanner was created that could read all of the symbols used in the concrete syntax of the file. Then the parser generator CUP [HFP$^+$10] was used to make a parser that could parse the file, and make use of the Java classes. The completed parser was able to read in rules, definitions and proofs from a file and generate a Java AST that could then be used by a proof support tool.

As well as reading in from a file,the tool should be able to save any changes or additions to the rules and proofs to file. To do this a method called *output* was added to each class of the Java framework. When called on a class, this method returns a string representation of the class, in a format that can be read by the parser. The method call is passed down the abstract syntax tree, so the returned value is actually a string representation of the class and its child classes. For example, when *output* is called on a proof, the method call will be passed on to the root judgement of the proof, and then on to each of the judgements in the proof, and so on until the terminal nodes of the abstract syntax tree are reached and values are passed back up the tree, resulting in a string representing the whole proof.

File Format

It is worth explaining here a little about the “file format” used to store the rules, definitions and proofs. This is only intended to be used by the tool to store its library, to read the library in from and write the library to a file. The file format is hidden from users so they don’t have to learn the syntax to write a rule, since the user interface will provide a way to add rules and write proofs.

In the file, rules, definitions and proofs should be grouped together in a theory, identified by the keyword *theory*, followed by the name of the theory. A file may contain multiple theories, and everything that follows the name of the theory is taken to be part of that theory, until the next use of the *theory* keyword. A theory contains a list of rules, definitions and proofs, in that order; or it may contain just rules or rules and definitions or rules and proofs, as long as they are written in the appropriate order.

When saved to file, the *or introduction left* rule would look as follows

```
rule orIL, [2], [1] or [2], Axiom;
```

The keyword *rule* states that a rule is being defined, and is followed by the name of the rule. Then the hypotheses, conclusion and justification are given, all separated by commas. The *or introduction left* rule is an axiom, so its justification has the keyword *Axiom*. If the rule was an inference rule that required a proof, then the justification would be the name of the proof. A semi-colon is used as a terminator to mark the end of the rule definition.

The numbers surrounded by square brackets are *placeholders*, and mark the parts of the rule that can be instantiated by variables or expressions, when the rule is used as a justification.

The layout of a proof in the file is quite similar to how it would be written on paper. Below is the proof of the *and introduction* inference rule.

```
proof andIProof
from a | b
  1  not not a                  by prop.notnotI(h1)
  2  not not b                  by prop.notnotI(h2)
  3  not ( not a or not b )     by prop.notorI(1, 2)
infer a and b                   by defn prop.defand(3)
;
```

A proof starts with the keyword *proof*, followed by the name of the proof. Hypotheses are preceded by the keyword *from*, and separated by a vertical bar. The conclusion of the proof should be the last line and start with the keyword *infer*. A line of the proof consists of a line number, the body of the line, and a justification, all separated by white space.

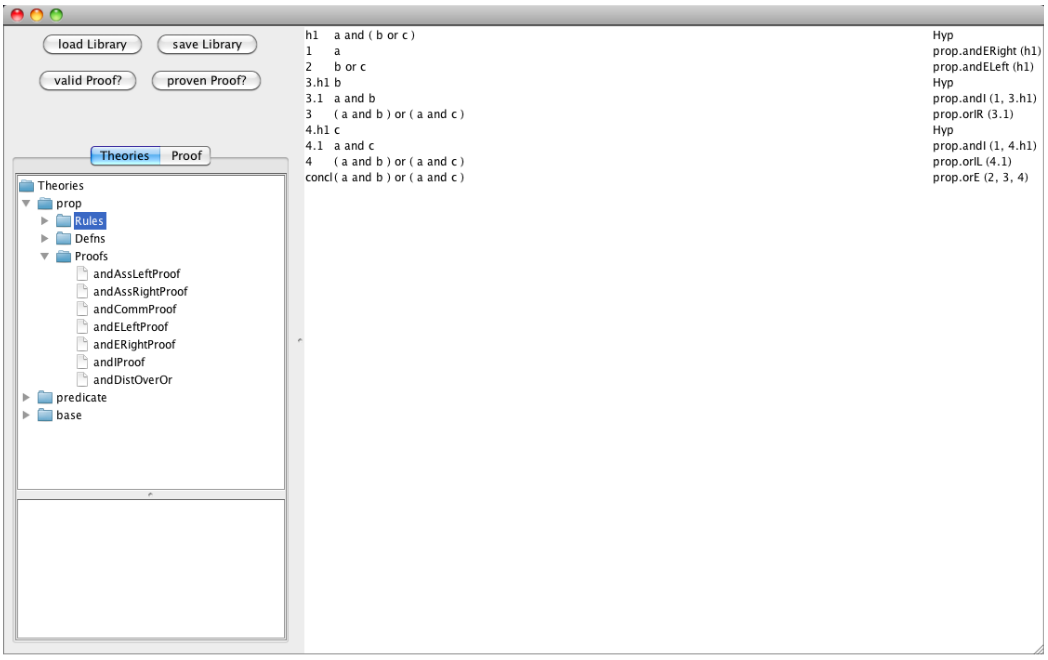
A line justification can be justified by an inference rule or by a definition[9]. Rule justifications start with the keyword *by*, then state the name of a theory, then a *dot*, the name of the rule and then parameters in parentheses. A definition justification has the same format, but has the additional keyword, *defn*.

7.2.3 User Interface

Once theories, rules, proofs and definitions could be read from a file and produce a Java AST, the next step was to start to build a user interface and link it to the existing classes. The interface was written in Java, using the components provided by the Swing and AWT libraries. The main reason for choosing to use Java to build the user interface was due to the code generated from the VDM model being Java, making it easier to integrate a Java user interface with the Java classes from the VDM model.

Figure 7.2 shows a screen dump of the prototype's user interface. The user interface consists of the *theory viewer* and the *proof builder*. The theory viewer displays the theory library and each theory can be expanded to show rules, definitions and proofs that are part of that theory. Existing proofs can be selected and viewed in the proof builder, allowing the user to work on a proof. The proof builder part of the user interface was designed so it would display a proof in the way they are written on paper (see Section 2.1), but with some slight changes to make the proof easier to work with in the proof builder. First of all, each hypothesis is displayed on a separate line, and labelled (rather than having them all on the *from* line), and *from* and *infer* are not used in the proof. Judgements are each displayed on a separate line, and the convention is followed that hypotheses go at the top, and the conclusion at the bottom of the proof. As a result of these changes, the proof

[9]Justification using a definition is referred to as *folding* or *unfolding* (see Section 2.4)



```
h1    a and ( b or c )                    Hyp
1     a                                   prop.andERight (h1)
2     b or c                              prop.andELeft (h1)
3.h1  b                                   Hyp
3.1   a and b                             prop.andI (1, 3.h1)
3     ( a and b ) or ( a and c )          prop.orIR (3.1)
4.h1  c                                   Hyp
4.1   a and c                             prop.andI (1, 4.h1)
4     ( a and b ) or ( a and c )          prop.orIL (4.1)
concl ( a and b ) or ( a and c )          prop.orE (2, 3, 4)
```

Figure 7.2: User interface for the prototype.

layout looks more like that used by Fitch in [Fit52]. Section 7.3.3 explains the specifics of how the tool can be used to build a proof.

7.2.4 The User's Language

When writing a proof that contains a program, we will tend to write the program in a concrete syntax to make it shorter and more readable. But this gives the problem of how the tool should read this concrete syntax, since it is for a user defined language, and we want the tool to be able to use any language defined by the user. The solution here was to use a separate parser that can be plugged-in to the tool that can parse the user's language into a standard format that the tool can understand and use.

To make the parser for this language compatible with the prototype a Java interface class is provided that the abstract syntax trees generated by the parser should implement. This interface, called *UserLangObject*, contains several methods that must be implemented, and provide a way for the prototype tool to interact with the ASTs from the parser. These methods are shown below.

```
public String concreteOut();
public String getId();
public Vector getParams();
public boolean equiv(UserLangObject f);
```

The method *concreteOut* returns a string representation of an object in the concrete syntax of its language. This method is used when displaying the object in a proof, or when saving a proof to file.

The *getId* method returns the name of the construct, so for an *if* statement the returned string would be “If”.

The *getParams* method should return a list of any parameters the construct might have. The names of the parameters will be returned as Strings inside a Vector. For an *If* statement, the result might be $\{test, th, el\}$.

We want to be able to compare objects, so each class should implement the *equiv* method and be able to compare itself for equality against other *UserLangObject*s.

When writing some concrete syntax in a proof, the text should be written in double quotes so that the parser knows that it is part of the user's language, and can pass the text to the appropriate parser, and get back an instance of a class that implements the *UserLangObject* interface and is an abstract syntax representation of the parsed text.

Modelling the User's Language

A new class was added to the VDM model called *UserLangObject*, which contains the above methods with the bodies of the methods being defined as ‘subclass responsibility’[10]. With the class having no instance variables and no implemented operations, it was possible to designate it to be an interface when generated as a Java class.

[10]This is similar to defining a Java method as abstract in an abstract class. The implementation of the method becomes the responsibility of a subclass

This class is incorporated into the model using the *LangForm* class, which is a subclass of *Form*, and represents any piece of code in the user's language. The *LangForm* class is essentially a 'wrapper' class that just contains an instance of a *UserLangObject*, and provides operations to interact with the *UserLangObject*, as well as implementing operations required by *Form*.

$$\mathit{LangForm} :: \mathit{ast} : \mathit{UserLangObject}$$

When specifying the semantics of a language, statements are added from the user's language to a semantic rule. To use the rule as a justification we need to be able to specify placeholders as in Section 7.1.5. To allow the user to do this a new class was added to the VDM model that allows a statement in the user's language to be written abstractly, and which can be instantiated by a concrete representation of the statement.

A *LangFormHole* is a particular type of placeholder that can only be filled by a *LangForm*. The *LangFormHole* class has a name, and a parameter list.

$$\begin{array}{lrl} \mathit{LangFormHole} :: & \mathit{name} : & \mathit{char}^{+} \\ & \mathit{params} : & \mathit{VarSymb}^{*} \end{array}$$

A *LangForm* can only instantiate a *LangFormHole* if it has a matching name and parameter list. The *LangFormHole*'s name is compared to the returned value of the *getId* method called on the *LangForm*'s *UserLangObject*, to make sure the correct construct is filling the placeholder. Checking the parameter list of the *LangForm* against the *getParams* method of the *LangForm*'s *ast* field, ensures that the various parts of the construct also match.

Since the type of *params* is a *VarSymb*, the parameters can be either a *VSymb* or a *PHole*, allowing the parameters to be placeholders.

7.3 Using the Tool

This section presents the process of using the proof support tool, from adding a language description, to constructing a proof about a program.

7.3.1 Specifying a Language

Before semantic rules can be used as justification, they first must be added to the tool. To add a language description to the tool the language description must be written in the form of inference rules (as explained in Section 7.1.10), which are added to the tool's library in a new theory. This gives an abstract definition of the language and allows the rules to be used as justifications in a proof.

As discussed in Section 7.2.4, to be able to use the concrete syntax of a user's language in a proof, we need to be able to parse the text and make use of the returned abstract syntax tree. This requires a parser for the user's language. To assist in making the parser for the user language, three tools are used: *Classgen* [WD10], *JFlex* [KRD10] and *CUP* [HFP^{+}10].

Classgen is a tool that allows users to write descriptions of classes by specifying their global variables, methods and inheritance relations, and then generate the Java classes from the description. This tool is used to generate that Java classes of the abstract syntax tree of the language. The tools JFlex and CUP are also used to create a parser for the user's language. The parser will make use of the Java classes generated by the Classgen tool to create an abstract syntax representation of some parsed text.

7.3.2 Building a Theory

A *Theory* is a collection of axioms, inference rules, definitions and proofs that are related in some way (see Section 7.1.8). In the prototype tool, all theories are listed in the *theory viewer*, which uses a tree structure. Expanding a theory shows three groups, *Rules*, *Defns* and *Proofs*.

Inference rules, axioms and language rules are all grouped together in the *Rules* part of the theory. The various types of rules can be distinguished by the small icon that is displayed next to the name of the rule. For example, any axioms in the list will have an icon displaying the letters 'Ax'.

Selecting a rule in the list will display the rule in the *rule viewer*, just below the theory viewer. The rule viewer will also show if the displayed rule is proven or not[11].

To add a new theory, users right-click on the root of the tree in the theory viewer (labelled 'Theories'), then select 'Add Theory'; the tool will then ask for a theory name. Once a name has been entered, an empty theory will be created and added to the theory viewer.

Adding Rules

To add a rule to a theory, users right-click on the 'Rules' node in the theory and select 'Add New Rule' from the menu. In the current version of the prototype the user specifies the new rule by typing the text representation of the rule, as used in the tool's file format (see Section 7.2.2). The text is passed to the parser and, if the text is syntactically correct, the returned rule is saved to the selected theory.

It would be better if the user did not have to type the rule in this way, and one of the 'wish list' (see Section 8.2) items is to improve the way a rule is entered so that it is easier and more intuitive for the user.

Adding Definitions

The process of adding a definition is similar to adding a new rule. Users first select the theory that the definition is to be added to, then right-click on the 'Defns' node and select 'Add New Defn' from the menu. An input dialog is displayed, and the user types the text representation of the new definition,

[11]In the case of inference rules, they are proven if they have an associated proof that is proven. Axioms, abstract syntax, context conditions and semantic rules are all taken to be proven

as used in the tool's file format (see Section 7.2.2). When the user clicks 'ok' on the input dialog the text will be sent to the parser and, if the text is syntactically correct, a definition will be returned and saved in the selected theory.

As with the method of adding a new rule, ideally this should improved so that no knowledge of the syntax of the tool's file format is required.

Adding a Proof to a Rule

Inference rules that are not axioms are required to be proven, and may have an attached proof. If an inference rule does not yet have a proof the user may add a proof of the rule by right-clicking on the rule and selecting 'Add Rule Proof'. The tool will then create a new proof with the hypotheses and conclusion of the rule, and add a pointer to the proof to the rule's *proof* field. Once the proof has been proven, then the rule is marked as proven.

7.3.3 Building a Proof in the Tool

The first thing to do when creating a proof is to add a new proof to a theory in the library. This is done by right-clicking on the *proofs* node of a theory and selecting "Add New Proof". The tool will first ask for a name for the proof, and then will ask for the proof goal, which is the conclusion line of the proof. A proof consisting of just a conclusion will then be added to the list of proofs and the proof can be viewed by double-clicking on the name of the proof in the list.

So far the proof consists of just a conclusion, so the next step is to add the hypotheses. This is done by right-clicking on the conclusion line of a proof in the proof editor and selecting "Insert Hyp". The conclusion line must be selected since only conclusions can have hypotheses. Selecting the conclusion line lets the tool know we are adding the hypothesis to the proof, not to a subproof. When inserting a new hypothesis the user will first be asked for a line number. There is no restriction on this, but the convention should be followed of labelling hypotheses with a 'h', so the first hypothesis would be ‘h1’, and the next ‘h2’. The user will then be asked to add the proof line, which will then be parsed and added to the proof as a hypothesis. The justification will automatically be set to *Hyp*, meaning it is a hypothesis.

A new line is added to to the proof by right-clicking on the proof editor and selecting “Insert line” from the menu. As with the hypothesis, the user will be asked to provide the label and the body of the new line. The new line will be added to the proof with an empty justification (which is displayed as “??”).

A line number can be changed at any point in the proof, by right-clicking on the line number and selecting “Renumber line”. The new line number must not already be in use. Any references to the old line number in justifications are automatically updated in the displayed proof, since the justifications refer to the object reference of a *Judgment* rather than the *String* representation of its line number.

To add a justification to a line, users right-click on the justification part of the line and in “Suggested Rules” sub menu a list of rules and definitions that can be applied to the line will be shown, broken down into theories. The tool decides if a rule can be applied by comparing the selected line against the conclusion of the rule, and if they match, the rule will be added to the list of suggested rules. The tool only allows the addition of a rule from the list of suggestions. Once a rule has been selected the tool will then ask for the line numbers of the lines that the justification depends on, one line number for each of the hypotheses in the rule. These will be lines of the proof that match against the hypotheses of the selected rule. The order in which the line numbers are given is important, since the first line number will match against the first hypothesis of the rule, etc.

When adding a justification to a line, the tool will not check the dependent lines of the justification against the hypotheses of the rule. These are only checked when the user uses the *proof checker*, as described below.

Due to restrictions in the Java Swing components used to implement the proof viewer, a proof is not directly editable using the proof viewer. To edit a line of the proof the user must right-click on the line and select "Edit Line", which brings up a dialog box that allows the user to edit the text of the line. Once the text has been edited it will then be re-parsed and the *Judgement* will be updated.

A subproof can be easily added to a proof by first adding the conclusion line of the subproof as a regular line. Users right-click on this new line and select "Make line a subproof". The tool will then ask the user for a hypothesis, which will be added as a hypothesis to the selected line, and will be labelled 'x.h1', where 'x' is the number of the selected line. Additional hypotheses and lines can be added to the subproof by right-clicking on the conclusion line of the subproof and selecting the appropriate option from the menu.

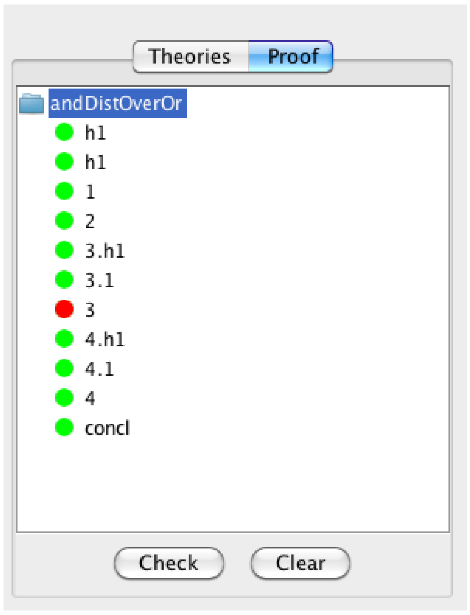


Figure 7.3: Proof checker.

At any point during the proof construction, the user can check that the proof is valid and identify lines of the proof that are either not justified, or are incorrectly justified. This *proof checker* (see Figure 7.3) is found in the 'Proof' tab, in the same pane as the theory library. The user can check the proof that is currently in the proof viewer by clicking the ‘Check’ button. Once the proof has been checked by the tool, then a list of line numbers will be displayed along with either a red or a green icon. A line number that has a green icon has a valid justification. A line number with a red icon has either an invalid or empty justification.

7.4 Discussion

This section discusses some of the problems that were faced during the development of the VDM model and the prototype proof support tool.

7.4.1 Model Compromises

During the development of the prototype some difficulties were encountered that were due to the method of building the prototype; that is, writing a VDM model and then generating Java from the model. While working on the Java implementation of the prototype, some issues arose that required small changes to the VDM model. They have been termed compromises since the purpose in making the change to the model was for the benefit of the generated Java code. The changes usually involved a slightly different way of specifying something, and did not affect the integrity of the model.

Pointers and References

When writing a specification in VDM-SL it is common to specify a type that acts as a pointer, then create a mapping of ‘pointers’ to ‘objects’[12], the object being an instance of a record type. This allows objects and pointers to be modelled in a specification. For example, *RuleId* might be defined as a *token* type and used as a pointer to a *Rule*, and elsewhere in the specification, where the rules are stored, there would be a mapping of *RuleId* to *Rule*. The

[12]They are not actually objects, since VDM-SL is not object-oriented

early model of the proof system[13] used such *Id* types to act as pointers to 'objects'.

If we are interested in generating Java code from a model, then writing specifications using this kind of 'pointer' makes the generated code more complicated. Since Java is an object-oriented language, objects have pointers[14] and we don't need to have this extra class that represents a pointer in the specification.

The VDM++ language, unlike VDM-SL, is an object-oriented language, and objects have references [FLM+05]. So in a VDM++ model, anytime we refer to an instance of an object, we are using a reference to the instance, rather than the instance itself.

When the model was converted from a VDM-SL to a VDM++ model, most record types were converted to classes, meaning *Id* types were no longer necessary, so they were removed from the model. This simplified the Java code generated from from the model, since object references in the model translate to object references in the Java code. For example, the collection of rules becomes a set of *Rule*, which in Java becomes a HashSet.

Names and Identifiers

Many of the classes in the VDM++ model have a name field so that an instance of a class can be distinguished from other instances and identified by the user. The usual type to use for such an identifier in a VDM model is the *token* type.

When Java code is generated from a model, a *token* type becomes an *Object* type, which is not as useful as a *String* type would be (since strings are generally used for names). Instead of using the *token* type for a class name, $char^+$ (non-empty sequence of characters) is used, which becomes a *String* in the generated Java code.

[13]See Section 7.1

[14]Or *references*, to use the Java terminology

Equality and toString

Many of the Java classes generated from the VDM model required extra methods to integrate them with the prototype and parser. Since these methods were only required in the Java prototype, it did not make sense to add them to the VDM model then generate the Java code, as this would only have unnecessarily complicated the VDM model.

One method that was required in many of the classes was an *equals* method that would override the default *equals* method of the class, allowing us to write our own definitions for class equality.

Another method that was required was the *output* method, which is used to output the user's theory library to a file. The *output* method is called on the *Library* class, which calls the *output* method on the classes that are stored in the library. Each class returns a string that represents the class, and that can be read by the parser to re-create the class (See Section 7.2.2).

When a Java class is regenerated from the model, any changes made to the class file will be overwritten. To prevent this from happening, special tags can added in comments before and after the addition methods. These tags are read by the toolbox when it re-generates a Java class from a VDM++ class, and tell the toolbox not to remove the section of code from the class. Without these tags the toolbox would just rebuild the Java class file from scratch, ignoring any changes that had been made since it was last generated.

Class Inheritance

The VDM++ language supports multiple inheritance, and a class can be a subclass of more than one superclass. Inheritance in Java, however, works differently; a class can only have one superclass, but can also implement multiple interfaces. The Toolbox only allows Java code to be generated if the inheritance structure of the classes in the model meet the requirements of inheritance in Java. Using the Toolbox, the user may choose to have classes generated as Java interfaces. So a Java class can only be generated from a VDM++ class if the class has at most one superclass that will be generated as a class, and any other superclasses are set to be generated as interfaces.

Since we wanted to generate Java code from the VDM++ model, we had to restrict inheritance in the model so that a class only had one superclass. This wasn't a huge restriction, but it did require us to make some adjustments to the inheritance structure of the model to allow Java code to be generated from it.

Sequences and Vectors

A *sequence* in VDM is an ordered collection of elements of the same type. When a Java class is generated from a VDM model, a sequence is represented as a Vector object. The Java code generator does not make use of Java generics, so the Vector just provides us with an ordered collection of elements of type *Object*, which is a weaker type description than is given by the model.

If we have a Vector that we expect to contain *Form*s, if we want to call any methods on an element in the Vector, then we need to first cast it to a *Form*. If we cast an element to the wrong class then we will cause a *ClassCastException* to be thrown. Using generics, we could add a restriction to the Vector so that it can only contain *Form*s, or classes that inherit from *Form*. Adding the restriction also means it is not necessary to cast the elements of the Vector.

Not using generics to restrict collections should not cause any problems since the Toolbox makes sure that the model is type correct before any Java code is generated. Where a problem might occur is when other Java classes are written that integrate with the generated code, as is the case with the tool's parser. In the example of the *Vector*, we should be aware of what type of class we expect to be stored in the *Vector*, and care needs to be taken in this code to make sure that we are adding the right type of classes to the Vector (or any other *collection* that might be used), to avoid potential exceptions being thrown.

In the parser, we make use of this weakening of the representation of a sequence to our advantage, since any class can be added to a *Vector*. When a proof is being parsed, the dependencies of the justifications are represented as strings, but we want them to be references to instances of *Judgement*s. However, we have to parse the whole proof before we can create all of the *Judgement*s in the proof and have access to the reference to the *Judgement*s. So the parser will create the justification and provide an instance of a *Vector* containing strings as the dependencies of the justification. Once the proof is parsed, the parser will make another pass over the *Judgement*s and replace the strings in the dependency list of each justification with a reference to the appropriate *Judgement*.

Instantiating justifications in this way means that a modification to the VDM model was needed to benefit the Java implementation. In the model, a *DefnAppl* can only have one *Judgement* as a dependency, but the type of the *dependsOn* field is defined as a sequence of *Judgement*s, and then an invariant is used to restrict the length of the sequence to exactly one. If the type of the field was a *Judgement*, then a *Judgement* must be provided when an instance of *DefnAppl* is created in the Java implementation, and it would not be possible to pass it a string. Making the type of the *dependsOn* field a sequence in the VDM model means that it will be a *Vector* in the generated Java code, allowing a string to be passed when an instance of *DefnAppl* is created, and it can then be modified later by replacing the string with a reference to the appropriate *Judgement*.

7.4.2 Other Challenges

One of the biggest challenges in the development of the VDM model and the prototype was the problem of how to represent a user's language in the model, and how to allow the user to write statements in his language in semantic rules and proofs, and have the tool understand it. The user should be able to use the tool to write proofs about a program in any language that he can write a description for, so the interface to the user's language needs to be general, so that different languages can be used.

The solution to this is outlined in Sections 7.2.4 and 7.3.1. It may be possible to extend the solution by writing an additional tool that makes use of all the tools that are used here. So the user only has to write a description of the language, and the tool would automatically generate the parser, java classes and inference rules. This would certainly make defining a new language for use with the tool easier, and could be added to the 'wish list' (see Section 8.2).

7.5 Summary

This chapter covers the development of a prototype proof support tool for reasoning about programs using operational semantics. The first stage of the development was to produce a VDM++ model of a proof system. Using the VDM toolbox, Java code was generated from the VDM++ model, providing the basis for the Java prototype, to which a graphical user interface was added.

Chapter 8

Conclusions

The purpose of this thesis is to investigate an alternative method of reasoning about computer programs, by using operational semantic descriptions directly. This thesis also argues that such proofs would be benefited by adequate tool support.

The specific goals of the thesis are stated in the thesis hypotheses in the introduction to the thesis (Chapter 1), which are copied here for convenience.

1. The operational semantics of a programming language can be used as a basis for reasoning about programs written in the language — this approach is applicable to a wider class of languages than that covered by axiomatic semantics.

2. Although some setup is required when basing program reasoning on operational semantics, this effort can be made acceptable with a suitable support system.

This chapter will review the achievements and contributions of this thesis, and show how the thesis supports the thesis hypotheses.

8.1 Main Contribution

The approach to using operational semantic rules in a natural deduction proof of a program was introduced in Section 3.4.4, and expanded on with further examples in Chapter 4. Three example programming languages[1] are used in this thesis to allow the examples to be diverse.

The examples provided in this thesis support the argument for reasoning about programs directly with operational semantic descriptions, and show that this method of reasoning about programs is not just restricted to simple sequential programs, but can be applied to programs that use more complex language features, including concurrency.

Section 4.4 uses the operational semantic approach to show that a simple parallel program meets its specification. In Section 4.5, the same program is used, but the example attempts to use an axiomatic approach, and shows that it is not possible to prove the program correct due to the interference between the parallel threads. These sections therefore show that the operational approach can be used for concurrent programs with interference, whereas a strict Hoare style axiomatic approach is unable deal with interference.

In terms of the second hypothesis, one of the achievements of this thesis is the development of a prototype proof support tool for reasoning about computer programs using operational semantic descriptions. The user is able to use the tool to specify a language and use the language definition in proofs of programs written in that language. As well as a language definition, additional setup is required in the form of a parser for the concrete syntax of the user defined language (Sections 7.2.4 and 7.3.1). This is needed to allow the user to write concrete syntax in a proof. Section 8.2 suggests that only allowing an abstract syntax to be used in a proof would make language definition and setup easier, but may make proofs longer and more difficult to read.

The prototype tool was instantiated for each of the three languages used in this thesis, allowing all of the proofs in the thesis to be checked in the

[1] *Base*(Appendix B), *Simple*(Appendix C) and *Parallel*(Appendix D).

tool. The tool allows proofs, inference rules, language description, etc., to be saved to file and then opened up later. The data is saved to file in a syntax that is intended to be easy to read. Appendix E shows the three language descriptions exported from the tool in this syntax, along with the examples proofs from this thesis.

Early on the decision was made to develop a proof support tool, rather than using an existing tool, or try to rebuild the *Mural* tool. It was hoped that this would result in a tool that was more suited to the approach of using operational semantic rules as inference rules in natural deduction proofs. As mentioned in Section 5.4 the tools that were identified as being potentially most suitable were *Jape* and *Mural*. Using an existing proof tool has obvious advantages, but also has its share of difficulties. Although Jape meets many of the requirements outlined in Chapter 6, its interaction style is quite restrictive and does not allow middle-out reasoning (see Section 2.5). Mural had a less restrictive interaction style, allowing unjustified lines to be added to the middle of a proof without linking it to existing lines, and only considered a proof complete when the inserted lines were linked to both hypothesis and goal. The *Mural* tool is no longer available and the effort to rebuild it may not have paid off (see Section 6.3.1). But *Mural*'s intereaction style did provide inspiration for to proof support tool.

The tool is designed to be a generic proof support tool that can be instantiated for different user defined languages. Once the user has provided the language description to the tool, they are then able to use the semantic rules of the language to prove properties about programs written in the language. Chapter 6 outlines what are considered to be important features for the tool to have.

The development of the prototype tool started with a VDM model of the proof system. This allowed the structure and key features of the system, that are independent of an implementation, to be modelled formally and defined early in the development cycle. The final version of the model was more general than initially expected. Rather than including special constructs for the various parts of a language description, the decision was made to instead change how a language description was represented, so that the abstract syntax and context conditions are written as rules. This resulted in the VDM model only modelling inference rules and their use within natural deduction proofs, with a language description being treated as a collection of inference rules.

Using the VDM toolbox, Java classes were generated from the VDM model, forming the basis of the Java implementation. If any changes were required to the model during the development, the changes could be made to the VDM model, and then the Java code re-generated from the model. The generated Java code was never modified except to add additional Java specific methods such as the *.equals()* method. This meant that the Java implementation would always reflect the model. Choosing to work in this way did introduce some challenges (see Section 7.4.1), but it also had many advantages, such as the time saved in constructing an implementation of the model and the assurance that generated classes would be a valid implementation of the VDM model.

The development of a proof support tool for reasoning about programs using operational semantics was not as straightforward as the author initially thought. However, the prototype tool does still demonstrate the usefulness of tool support for this type of reasoning. The tool becomes especially useful when dealing with large proofs such as the proof of the parallel search program in Section 4.4.

This thesis also introduces a new approach to defining the context conditions of a language, using rules rather than functions (see Section 7.1.10).

8.2 Further work

The proof support tool developed was a prototype that only contained the essential features of a final version of the tool. The next step for this work would be to implement a proof support tool that contained more of the features identified as being useful for the tool (see Section 6.2), as well as other features identified during the implementation of the prototype as being useful additions to the tool.

This 'wish list' is an explanation of features or possible extensions to the tool that were not added due to the time constraints of this work, but could be added in the future. The lack of these features in the prototype tool does not affect the use of the tool, but once implemented, they would be a benefit to the tool. Many of the features described here were identified during the implementation of the prototype tool.

A feature listed in the requirements that was not implemented in the prototype was the option to collapse a subproof so that only hypotheses and conclusion are left visible. This would be as a useful feature to increase the readability of the proof, especially on computers where screen size is limited. This was available in Mural and would be a straightforward addition.

When the user adds a new line to a proof, the tool is able to suggest rules that may be used to justify the line. Ideally, suggestions should be extended further so that once the user selects a rule from the list of suggestions, the tool also suggests adding new lines to the proof that match the hypotheses of the selected rule, or suggests existing proof lines that match the hypotheses of the selected rule.

If the user selects to have the tool add the new lines to the proof to match the hypotheses of the selected rule, the tool will need to identify an appropriate instantiation of placeholders in the inference rule to terms in the proof. An instantiation of placeholders to variables or terms will be created from the selected proof line and the conclusion line of the inference rule. If a hypothesis contains a placeholder that is not included in the instantiation, then the placeholder will be left in the new proof line, and the user will have to fill it manually.

This feature could be implemented quite easily and it would likely be a benefit to the user, and may also help the user to write proofs faster. The examples in Chapter 4 have shown us that a certain degree of automation would help the user focus on the more important details of the proof, while the tool could assist by automatically completing what the user might consider the tedious details of the proof.

The way a user specifies rules and definitions in the tool could also be improved. At the moment, the user writes the rule or definition in the concrete syntax of the tool's parser, then the tool parses the text and adds the rule or definition to the library. Adding a more usable way of specifying rules and definitions would mean that the user is not required to learn the syntax of the tool's parser.

The way that the user specifies a language definition could be improved so that the proof support tool can be used to write proofs about programs written in that language. First, the user must provide the abstract syntax, context conditions and operational semantic of the language, all written as rules. Adding these rules to the rule library allows them to be used to justify steps in a proof, and also allows the tool to suggest the rules as justifications. Second, the user needs to define a concrete syntax for the language and write a parser that can be used by the proof support tool. This will allow the user to write program statements in a proof in a concrete syntax. The writing of a parser is assisted by the use of the tools JFlex, CUP and Classgen. The process of defining a language to be used in the proof support tool is described in more detail in Sections 7.2.4 and 7.3.1.

This process could be improved by writing an addition to the tool that allows the user to write a language specification, and then the tool automatically generates the rules for the language, along with the parser for the concrete syntax of the language. This would certainly make the process of adding a language description to the tool a lot easier.

The difficulties involved in allowing the use of a concrete syntax in a proof could be avoided by only allowing an abstract syntax to be used in proofs. The user would then only have to provide the abstract syntax, context conditions and operational semantics of a language as inference rule to the rule library. The reason for not restricting the notation to abstract syntax in proofs is that allowing concrete syntax provides a shorthand for writing language statements and makes proofs easier to read, and also easier to write[2]. Providing a user with both of these options might also be a good solution.

Another extension to the tool could be to link the tool to existing automated theorem proving tools such as HOL or Isabelle. For simple conjectures the user could select an option to have the tool send the conjecture to an external theorem prover that would run in the background and attempt to prove it. If successful, the conjecture would be labelled as being justified by use of the external theorem prover.

Additionally, an option could be included for the user to export a proof to an external theorem prover such as Spass, to see if it can be automatically proven. This would involve a translation of the language description and the proof into a form that would be understood by the external tool.

The prototype allows the user to save proof and rule data to file and reload it later. Ideally, the tool could also have the facility to export proofs and rules to LaTeX.

An ambitious extension to the proof support tool could be to have the tool "learn" proof strategies from the user (a similar idea is expressed in [BGJ09]). As the user works on a proof, the tool would learn the approach that the user takes to write proofs about different kinds of programs and develop strategies that it can use to attempt to automatically write proofs.

[2]Keeping track of brackets when writing a long program statement in an abstract syntax can become confusing.

Bibliography

[Art11] Rob Arthan. ProofPower, May 2011. http://www.lemma-one.com/ProofPower/index/.

[BFL^{+}94] Juan C. Bicarregui, John S. Fitzgerald, Peter A. Lindsay, Richard Moore, and Brian Ritchie. *Proof in VDM: A Practitioner's Guide*. Springer-Verlag, 1994.

[BGJ09] Alan Bundy, Gudmund Grov, and Cliff B. Jones. An outline of a proposed system that learns from experts how to discharge proof obligations automatically. In Jean-Raymond Abrial, Michael Butler, Rajeev Joshi, Elena Troubitsyna, and Jim C. P. Woodcock, editors, *Dagstuhl 09381: Refinement Based Methods for the Construction of Dependable Systems*, pages 38–42, 2009.

[BH91] Rod Burstall and Furio Honsell. Operational semantics in a natural deduction setting. In *Logical frameworks*, pages 185–214. Cambridge University Press, New York, NY, USA, 1991.

[BJ82] Dines Bjørner and Cliff B. Jones. *Formal Specification and Software Development*. Prentice Hall International, 1982.

[Bri10] James P. Bridge. Machine learning and automated theorem proving. Technical Report UCAM-CL-TR-792, University of Cambridge, Computer Laboratory, November 2010.

[BS99] Richard Bornat and Bernard Sufrin. Animating formal proof at the surface: The jape proof calculator. *The Computer Journal*, 42(3):177–192, 1999.

[Chu40] Alonzo Church. A formulation of the simple theory of types. *Journal Symbolic Logic*, 5(2):56–68, 1940.

[CJ07] Joey W. Coleman and Cliff B. Jones. A structural proof of the soundness of rely/guarantee rules. *J. Log. and Comput.*, 17(4):807–841, 2007.

[Col08] Joseph William Coleman. *Constructing a Tractable Reasoning Framework upon a Fine-Grained Structural Operational Semantics.* PhD thesis, School of Computing Science, http://www.cs.ncl.ac.uk/publications/trs/papers/1083.pdf, January 2008.

[CSK05] CSK. *The VDM++ Language.* Technical report, CSK Corporation, 2005.

[CSK08] CSK. *The VDM++ Java Code Generator ver.1.1.* Technical report, CSK Systems Corporation, 2008.

[CSK10] CSK. *VDM Toolbox*, Feburary 2010. http://www.vdmtools.jp/en/.

[Daw91] John Dawes. *The VDM-SL Reference Guide.* Pitman, 1991.

[dRdBH$^+$01] Willem-Paul de Roever, Frank de Boer, Ulrich Hannemann, Jozef Hooman, Yassine Lakhnech, Mannes Poel, and Job Zwiers. *Concurrency Verification: Introduction to Compositional and Noncompositional Methods.* Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 2001.

[Eif11] Eiffel software, May 2011. http://www.eiffel.com/.

[esc11] *Extended Static Checker for Java version 2 (ESC/Java2)*, February 2011. http://sort.ucd.ie/products/opensource/ESCJava2/.

[Fit52] Frederic Brenton Fitch. *Symbolic Logic: An Introduction*. The Ronald Press Company, 1952.

[FL98] John Fitzgerald and Peter Gorm Larsen. *Modelling Systems: Practical Tools and Techniques in Software Development*. Cambridge University Press, 1998.

[FLM$^+$05] John Fitzgerald, Peter Gorm Larsen, Paul Mukherjee, Nico Plat, and Marcel Verhoef. *Validated Designs for Object-oriented Systems*. Springer-Verlag, 2005.

[Flo67] Robert W. Floyd. Assigning meanings to programs. In J. T. Schwartz, editor, *Proceedings of a Symposium on Applied Mathematics*, volume 19 of *Mathematical Aspects of Computer Science*, pages 19–31. American Mathematical Society, 1967.

[Gen69] Gerhard Gentzen. *The Collected Papers of Gerhard Gentzen*. Studies in Logic and The Foundations of Mathematics. North-Holland Publishing Company, 1969.

[GMW79] Michael J. C. Gordon, R. Milner, and C. P. Wadsworth. *Edinburgh LCF: A Mechanised Logic of Computation*. Springer-Verlag, 1979.

[GvN47] H. H. Goldstine and J. von Neumann. Planning and coding of problems for an electronic computing instrument. Part II, Vol. 1 of a report prepared for U.S. Army Ord. Dept., 1947.

[Har11] John Harrison. The hol light theorem prover, May 2011. http://www.cl.cam.ac.uk/ jrh13/hol-light/.

[Heh84] Eric C. R. Hehner. *The Logic of Programming.* Prentice Hall International, 1984.

[HFP$^+$10] Scott E. Hudson, Andrea Flexeder, Michael Petter, C. Scott Ananian, Frank Flannery, Dan Wang, and Andrew W. Appel. *CUP : LALR Parser Generator in Java*, February 2010. http://www2.cs.tum.edu/projects/cup/.

[HJ08] John R. D. Hughes and Cliff B. Jones. Reasoning about programs via operational semantics: requirements for a support system. *Automated Software Engineering*, 15(3-4):299–312, 2008.

[HMRC88] Richard C. Holt, Philip A. Matthews, J. Alan Rosselet, and James R. Cordy. *The Turing Programming Language: Design and Definition.* Prentice Hall International, 1988.

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

[Hoa75] C. A. R. Hoare. Parallel programming: an axiomatic approach. *Computer Languages*, 1(2):151–160, 1975.

[Hoa09] C. A. R. Hoare. Retrospective: An axiomatic basis for computer programming. *Communications of the ACM*, 52(10):30–32, October 2009.

[HSW99] U. Hustadt, R. A. Schmidt, and C. Weidenbach. MSPASS: Subsumption testing with SPASS. In P. Lambrix, A. Borgida, M. Lenzerini, R. Möller, and P. Patel-Schneider, editors, *Proc. of Intern. Workshop on Description Logics'99*, pages 136–137. Linköping University, 1999.

[HW73] C. A. R. Hoare and N. Wirth. An axiomatic definition of the programming language pascal. *Acta Informatica*, 2(4):335–355, December 1973.

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

[jml10] Java Modelling Language (JML), June 2010. http://sourceforge.net/projects/jmlspecs/.

[Jon81] C. B. Jones. *Development Methods for Computer Programs including a Notion of Interference*. PhD thesis, Oxford University, June 1981. Printed as: Programming Research Group, Technical Monograph 25.

[Jon83] C. B. Jones. Tentative steps toward a development method for interfering programs. *Transactions on Programming Languages and System*, 5(4):596–619, 1983.

[Jon90] Cliff B. Jones. *Systematic Software Development using VDM*. Prentice Hall International, second edition, 1990.

[Jon03a] C. B. Jones. Wanted: a compositional approach to concurrency. In Annabelle McIver and Carroll Morgan, editors, *Programming Methodology*, pages 1–15. Springer Verlag, 2003.

[Jon03b] Cliff B. Jones. The early search for tractable ways of reasoning about programs. *IEEE, Annals of the History of Computing*, 25(2):26–49, 2003.

[Jon03c] Cliff B. Jones. Operational semantics: concepts and their expression. *Information Processing Letters*, 88(1-2):27–32, 2003.

[JP08] Cliff Jones and Ken Pierce. Splitting atoms with rely/guarantee conditions coupled with data reification. In Egon Börger, Michael Butler, Jonathan Bowen, and Paul Boca, editors, *Abstract State Machines, B and Z*, volume 5238 of *Lecture Notes in Computer Science*, pages 360–377. Springer Berlin / Heidelberg, 2008.

[Kah87] Gilles Kahn. Natural semantics. In *STACS '87: Proceedings of the 4th Annual Symposium on Theoretical Aspects of Computer Science*, pages 22–39, London, UK, 1987. Springer-Verlag.

[KNS+11] Gerwin Klein, Tobias Nipkow, Norbert Schirmer, Martin Strecker, and Martin Wildmoser. Verificard, May 2011. http://isabelle.in.tum.de/VerifiCard/.

[KNvO+11] Gerwin Klein, Tobias Nipkow, David von Oheimb, Norbert Schirmer, and Leonor Prensa Nieto. Isabelle: Project bali, May 2011. http://isabelle.in.tum.de/bali/.

[KRD10] Gerwin Klein, Steve Rowe, and Régis Décamps. *JFlex : The Fast Scanner Generator for Java*, February 2010. http://jflex.de/.

[LG] INRIA Logical Group. The coq proof assistant. http://coq.inria.fr/.

[LHB+96] P.G. Larsen, B.S. Hansen, H. Brunn, N. Plat, H. Toetenel, D.J. Andrews, J. Dawes, and G. Parkin. Information technology - programming languages, their environment and system software interfaces - Vienna Development Method — specification language part 1 : Base language. International standard ISO/IEC 13817-1, International Organisation for Standardisation, Dec 1996.

[McC66] J. McCarthy. A formal description of a subset of ALGOL. In *Formal Language Description Languages for Computer Programming*, pages 1–12. North-Holland, 1966.

[McC10] W. McCune. Prover9 and mace4. `http://www.cs.unm.edu/~mccune/prover9/`, 2005–2010.

[Nip06] Tobias Nipkow. Jinja: Towards a comprehensive formal semantics for a Java-like language. In H. Schwichtenberg and K. Spies, editors, *Proof Technology and Computation*, pages 247–277. IOS Press, 2006.

[NN07] Hanne Riis Nielson and Flemming Nielson. *Semantics with Applications*. Springer-Verlag, 2007.

[NP92] Tobias Nipkow and Lawrence C. Paulson. Isabelle-91. In *In Proceedings of the 11th International Conference on Automated Deduction, D. Kapur, Ed. Springer-Verlag LNAI 607*, pages 673–676, 1992.

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

[Owi75] Susan Owicki. *Axiomatic proof techniques for parallel programs.* PhD thesis, Department of Computer Science, Cornell University, Ithaca, NY, USA, 1975.

[Pau99] Lawrence C. Paulson. The relative consistency of the axiom of choice – mechanized using isabelle/zf. *LMS Journal of Computation and Mathematics*, 6:2003, 1999.

[Pie09] K. G. Pierce. *Enhancing the Usability of Rely-Guarantee Conditions for Atomicity Refinement.* PhD thesis, School of Computing Science, 2009.

[Plo76] G. D. Plotkin. A powerdomain construction. *SIAM Journal on Computing*, 5(3):452–487, 1976.

[Plo81] G. D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Aarhus University, 1981.

[Plo04] Gordon D. Plotkin. The origins of structural operational semantics. *Journal of Logic and Algebraic Programming*, 60–61:3–15, July–December 2004.

[PNW10] Larry Paulson, Tobias Nipkow, and Makarius Wenzel. Isabelle, August 2010. http://www.cl.cam.ac.uk/research/hvg/Isabelle/.

[Pra06] Dag Prawitz. *Natural Deduction. A Proof-Theoretical Study.* Dover Publications, Inc., 2006. ISBN 0-486-44655-7.

[Pro11] Proof general, May 2011. http://proofgeneral.inf.ed.ac.uk/.

[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.

[rod10] *RODIN – Rigorous Open Development Environment for Complex Systems*, February 2010. http://rodin.cs.ncl.ac.uk/.

[SB98] Bernard Sufrin and Richard Bornat. Animating operational semantics with JAPE, 1998.

[SBBT09] Geoff Sutcliffe, Christoph Benzmüller, Chad E. Brown, and Frank Theiss. Progress in the development of automated theorem proving for higher-order logic. In *Proceedings of the 22nd International Conference on Automated Deduction*, CADE-22, pages 116–130, Berlin, Heidelberg, 2009. Springer-Verlag.

[SN08] Konrad Slind and Michael Norrish. A brief overview of hol4. In Otmane Mohamed, César Muñoz, and Sofiène Tahar, editors, *Theorem Proving in Higher Order Logics*, volume 5170 of *Lecture Notes in Computer Science*, pages 28–32. Springer Berlin / Heidelberg, 2008.

[SN10] Peter Sewell and Francesco Zappa Nardelli. Ott, August 2010. http://www.cl.cam.ac.uk/∼pes20/ott/.

[SPA11] *SPASS: An Automated Theorem Prover for First-Order Logic with Equality*, February 2011. http://www.spass-prover.org/.

[SS01] Geoff Sutcliffe and Christian Suttner. Evaluating general purpose automated theorem proving systems. *Artificial Intelligence*, 131(1-2):39 – 54, 2001.

[Sto77] Joseph E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory.* The MIT Press, 1977.

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

[Tur49] A. M. Turing. Checking a large routine. In *Report of a Conference on High Speed Automatic Calculating Machines*, pages 67–69. University Mathematical Laboratory, Cambridge, June 1949.

[vD94] Dirk van Dalen. *Logic and Structure.* Springer-Verlag, third edition, 1994.

[VHL10] Sander D. Vermolen, Jozef Hooman, and Peter Gorm Larsen. Proving consistency of vdm models using hol. In Sung Y. Shin, Sascha Ossowski, Michael Schumacher, Mathew J. Palakal, and Chih-Cheng Hung, editors, *Proceedings of the 2010 ACM Symposium on Applied Computing (SAC), Sierre, Switzerland, March 22-26, 2010*, pages 2503–2510. ACM, 2010.

[WD10] Sebastian Winter and Florian Deißenböck. *Classgen*, February 2010. http://classgen.sourceforge.net/.

[Wie06] Freek Wiedijk. *The Seventeen Provers of the World: Foreword by Dana S. Scott (Lecture Notes in Computer Science / Lecture Notes in Artificial Intelligence).* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[Win94] Glynn Winskel. *The Formal Semantics of Programming Languages : An Introduction.* Massachusetts Institute of Technology, second edition, 1994.

[WPN08] Makarius Wenzel, Lawrence Paulson, and Tobias Nipkow. The isabelle framework. In Otmane Mohamed, César Muñoz, and Sofiène Tahar, editors, *Theorem Proving in Higher Order Logics*, volume 5170 of *Lecture Notes in Computer Science*, pages 33–38. Springer Berlin / Heidelberg, 2008.

Part I

Appendices

Appendix A

Inference Rules

A.1 Propositional Logic

Axioms

$$\boxed{\vee\text{-I-left}}\,\frac{e_2}{e_1 \vee e_2}$$

$$\boxed{\vee\text{-I-right}}\,\frac{e_1}{e_1 \vee e_2}$$

$$\boxed{\vee\text{-E}}\,\frac{e_1 \vee e_2;\, e_1 \vdash e;\, e_2 \vdash e}{e}$$

$$\boxed{\neg\neg\text{-I}}\,\frac{e}{\neg\neg e}$$

$$\boxed{\neg\neg\text{-E}}\,\frac{\neg\neg e}{e}$$

$$\boxed{\neg\text{-}\vee\text{-I}}\,\frac{\neg e_1;\, \neg e_2}{\neg(e_1 \vee e_2)}$$

$$\boxed{\neg\text{-}\vee\text{-E-left}}\,\frac{\neg(e_1 \vee e_2)}{\neg e_2}$$

$$\boxed{\neg\text{-}\vee\text{-E-right}}\,\frac{\neg(e_1 \vee e_2)}{\neg e_1}$$

Definitions

$$a \wedge b \overset{def}{=} \neg(\neg a \vee \neg b)$$

$$a \Rightarrow b \overset{def}{=} \neg a \vee b$$

$$a \Leftrightarrow b \overset{def}{=} (a \Rightarrow b) \wedge (b \Rightarrow a)$$

Derived Rules

$$\boxed{\wedge\text{-I}}\,\frac{e_1; e_2}{e_1 \wedge e_2}$$

$$\boxed{\wedge\text{-E-Left}}\,\frac{a \wedge b}{b}$$

$$\boxed{\wedge\text{-E-Right}}\,\frac{a \wedge b}{a}$$

$$\boxed{\wedge\text{-subs-left}}\,\frac{e_1 \wedge e_2; e_1 \vdash e}{e \wedge e_2}$$

$$\boxed{\Rightarrow\text{-E-L}}\,\frac{a \Rightarrow b; a}{b}$$

$$\boxed{\Rightarrow\text{-I-L-vac}}\,\frac{b}{a \Rightarrow b}$$

$$\boxed{\Rightarrow\text{-I-L-vac}}\,\frac{\neg a}{a \Rightarrow b}$$

A.2 Predicate Logic

Axioms

$$\boxed{\exists\text{-E}}\ \frac{\begin{array}{l}\exists x \in A \cdot P(x)\\ y \in A, P(y) \vdash e\end{array}}{e}$$

$$\boxed{\exists\text{-I}}\ \frac{\begin{array}{l}a \in A\\ P(a)\end{array}}{\exists x \in A \cdot P(x)}$$

$$\boxed{\neg\text{-}\exists\text{-E}}\ \frac{a \in A; \neg(\exists x \in A \cdot P(x))}{\neg P(a)}$$

Definitions

$$\forall x \in A \cdot P(x) \overset{def}{=} \neg\exists x \in A \cdot \neg P(x)$$

A.3 Equality

Axioms

$$\boxed{=\text{-subs-left}}\ \frac{e_1 = e_2; P(e_2)}{P(e_1)}$$

$$\boxed{=\text{-subs-right}}\ \frac{e_1 = e_2; P(e_1)}{P(e_2)}$$

Appendix B

Base Language

This appendix contains the description of what is referred to as the *Base* Language — a simple sequential language that has a simple state that maps variable names to integer values.

B.1 Abstract Syntax

$$\begin{array}{lll} \mathit{Program} :: & \mathit{vars} & : \mathit{Id} \xrightarrow{m} \mathit{Type} \\ & \mathit{body} & : \mathit{Stmt} \end{array}$$

$$\mathit{Type} = \textsc{IntType}$$

$$\mathit{Stmt} = \mathit{If} \mid \mathit{Assign} \mid \mathit{While} \mid \mathit{Seq}$$

$$\begin{array}{lrl} \mathit{Assign} :: & \mathit{lhs} & : \mathit{Id} \\ & \mathit{rhs} & : \mathit{Expr} \end{array}$$

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

$$\begin{array}{lrl} \mathit{If} :: & \mathit{test} & : \mathit{Expr} \\ & \mathit{th} & : \mathit{Stmt} \\ & \mathit{el} & : \mathit{Stmt} \end{array}$$

$$\begin{array}{lrl} \mathit{While} :: & \mathit{test} & : \mathit{Expr} \\ & \mathit{body} & : \mathit{Stmt} \end{array}$$

$$\mathit{Expr} = \mathit{Id} \mid \mathit{Value} \mid \mathit{ArithExpr} \mid \mathit{RelExpr} \mid \mathit{UnaryExpr}$$

$$\begin{array}{lrl} \mathit{ArithExpr} :: & \mathit{opd}1 & : \mathit{Expr} \\ & \mathit{op} & : \textsc{Plus} \mid \textsc{Minus} \\ & \mathit{opd}2 & : \mathit{Expr} \end{array}$$

$$\begin{aligned} \mathit{RelExpr} :: \mathit{opd}1 &: \mathit{Expr} \\ \mathit{op} &: \textsc{Equals} \mid \textsc{NotEquals} \mid \mathrm{LT} \mid \mathrm{GT} \mid \mathrm{LTEQ} \mid \mathrm{GTEQ} \\ \mathit{opd}2 &: \mathit{Expr} \end{aligned}$$

$$\begin{aligned} \mathit{UnaryExpr} :: \mathit{opd} &: \mathit{Expr} \\ \mathit{op} &: \textsc{Minus} \end{aligned}$$

$\mathit{Value} = \mathbb{Z} \mid \mathbb{B}$

B.2 Context Conditions

$\mathit{wf\text{-}Program}\colon \mathit{Program} \to \mathbb{B}$
$\mathit{wf\text{-}Program}(\mathit{mk\text{-}Program}(\mathit{vars}, \mathit{body}) \triangleq$
$\quad \mathit{wf\text{-}Stmt}(\mathit{body}, \mathit{vars})$

$\mathit{wf\text{-}Stmt}\colon (\mathit{Stmt} \times \mathit{Id} \xrightarrow{m} \mathit{Type}) \to \mathbb{B}$

$\mathit{wf\text{-}Stmt}(\mathit{mk\text{-}Assign}(\mathit{lhs}, \mathit{rhs}), \mathit{tpm}) \triangleq$
$\quad \mathit{lhs} \in \mathbf{dom}\ \mathit{tpm} \land$
$\quad \mathit{typeof}(\mathit{rhs}, \mathit{tpm}) = \textsc{IntType}$

$\mathit{wf\text{-}Stmt}(\mathit{mk\text{-}Seq}(\mathit{sl}, \mathit{sr}), \mathit{tpm}) \triangleq$
$\quad \mathit{wf\text{-}Stmt}(\mathit{sl}, \mathit{tpm}) \land$
$\quad \mathit{wf\text{-}Stmt}(\mathit{sr}, \mathit{tpm})$

$\mathit{wf\text{-}Stmt}(\mathit{mk\text{-}While}(b, s), \mathit{tpm}) \triangleq$
$\quad \mathit{typeof}(b, \mathit{tpm}) = \textsc{BoolType} \land$
$\quad \mathit{wf\text{-}Stmt}(s, \mathit{tpm})$

```
$\mathit{typeof}\colon \mathit{Expr} \times \mathit{Id} \xrightarrow{m} \mathit{Type} \to \textsc{Int} \mid \textsc{Bool}$
$\mathit{typeof}(e, \mathit{tpm}) \triangleq$
    $\mathbf{cases}\ e\ \mathbf{of}$
        $\mathit{mk\text{-}ArithExpr}(\text{-}, \text{-}, \text{-}) \to \textsc{IntType}$
        $\mathit{mk\text{-}RelExpr}(\text{-}, \text{-}, \text{-}) \to \textsc{BoolType}$
        $\mathit{mk\text{-}UnaryExpr}(\text{-}, \text{-}) \to \textsc{IntType}$
        $e \in \mathbb{B} \to \textsc{BoolType}$
        $e \in \mathbb{Z} \to \textsc{IntType}$
        $e\colon \mathit{Id} \to \mathit{tpm}(e)$
    $\mathbf{end}$
```

B.3 Semantics

B.3.1 Semantic Objects

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

B.3.2 Program Semantics

The program semantics initialises the declared variables and sets their values to zero.

$$\xrightarrow{p} : \mathcal{P}\left(\mathit{Program} \times \textsc{Done}\right)$$

$$\boxed{\textsf{Program}}\ \frac{\begin{array}{l}\sigma = \{id \mapsto 0 \mid id \in \mathbf{dom}\ vars\}\\ (body, \sigma) \xrightarrow{s} \sigma'\end{array}}{(\mathit{mk\text{-}Program}(vars, body)) \xrightarrow{p} \textsc{Done}}$$

B.3.3 Statement Semantics

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

$$\boxed{\textsf{If-T}}\ \frac{\begin{array}{l}(test, \sigma) \xrightarrow{e} \mathit{true}\\ (th, \sigma) \xrightarrow{s} \sigma'\end{array}}{(\mathit{mk\text{-}If}(test, th, el), \sigma) \xrightarrow{s} \sigma'}$$

$$\boxed{\textsf{If-F}}\ \frac{\begin{array}{l}(test, \sigma) \xrightarrow{e} \mathit{false}\\ (el, \sigma) \xrightarrow{s} \sigma'\end{array}}{(\mathit{mk\text{-}If}(test, th, el), \sigma) \xrightarrow{s} \sigma'}$$

$$\boxed{\textsf{Assign}}\ \frac{(rhs, \sigma) \xrightarrow{e} v}{(\mathit{mk\text{-}Assign}(lhs, rhs), \sigma) \xrightarrow{s} \sigma \dagger \{lhs \mapsto v\}}$$

$$\boxed{\textsf{Seq}}\ \frac{\begin{array}{l}(sl, \sigma) \xrightarrow{s} \sigma'\\ (sr, \sigma') \xrightarrow{s} \sigma''\end{array}}{(\mathit{mk\text{-}Seq}(sl, sr), \sigma) \xrightarrow{s} \sigma''}$$

$$\boxed{\textsf{While-T}}\ \frac{\begin{array}{l}(test, \sigma) \xrightarrow{e} true\\ (body, \sigma) \xrightarrow{s} \sigma'\\ (mk\text{-}While(test, body), \sigma') \xrightarrow{s} \sigma''\end{array}}{(mk\text{-}While(test, body), \sigma) \xrightarrow{s} \sigma''}$$

$$\boxed{\textsf{While-F}}\ \frac{(test, \sigma) \xrightarrow{e} false}{(mk\text{-}While(test, body), \sigma) \xrightarrow{s} \sigma}$$

B.3.4 Expression Semantics

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

$$\boxed{\textsf{Arith-Plus}}\ \frac{\begin{array}{l}(e1, \sigma) \xrightarrow{e} v1\\ (e2, \sigma) \xrightarrow{e} v2\end{array}}{(mk\text{-}ArithExpr(e1, \textsc{Plus}, e2), \sigma) \xrightarrow{e} v1 + v2}$$

$$\boxed{\textsf{Arith-Minus}}\ \frac{\begin{array}{l}(e1, \sigma) \xrightarrow{e} v1\\ (e2, \sigma) \xrightarrow{e} v2\end{array}}{(mk\text{-}ArithExpr(e1, \textsc{Minus}, e2), \sigma) \xrightarrow{e} v1\text{-}v2}$$

$$\boxed{\textsf{Rel-Equals}}\ \frac{\begin{array}{l}(e1, \sigma) \xrightarrow{e} v1\\ (e2, \sigma) \xrightarrow{e} v2\end{array}}{(mk\text{-}RelExpr(e1, \textsc{Equals}, e2), \sigma) \xrightarrow{e} v1 = v2}$$

$$\boxed{\textsf{Rel-NotEquals}}\ \frac{\begin{array}{l}(e1, \sigma) \xrightarrow{e} v1\\ (e2, \sigma) \xrightarrow{e} v2\end{array}}{(mk\text{-}RelExpr(e1, \textsc{NotEquals}, e2), \sigma) \xrightarrow{e} v1 \neq v2}$$

$$\boxed{\textsf{Rel-GT}}\ \frac{\begin{array}{l}(e1, \sigma) \xrightarrow{e} v1\\ (e2, \sigma) \xrightarrow{e} v2\end{array}}{(mk\text{-}RelExpr(e1, \mathrm{GT}, e2), \sigma) \xrightarrow{e} v1 > v2}$$

$$\boxed{\textsf{Rel-LT}}\ \frac{(e1,\sigma) \xrightarrow{e} v1 \qquad (e2,\sigma) \xrightarrow{e} v2}{(\textit{mk-RelExpr}(e1, \mathrm{LT}, e2), \sigma) \xrightarrow{e} v1 < v2}$$

$$\boxed{\textsf{Rel-GTEQ}}\ \frac{(e1,\sigma) \xrightarrow{e} v1 \qquad (e2,\sigma) \xrightarrow{e} v2}{(\textit{mk-RelExpr}(e1, \mathrm{GTEQ}, e2), \sigma) \xrightarrow{e} v1 \geq v2}$$

$$\boxed{\textsf{Rel-LTEQ}}\ \frac{(e1,\sigma) \xrightarrow{e} v1 \qquad (e2,\sigma) \xrightarrow{e} v2}{(\textit{mk-RelExpr}(e1, \mathrm{LTEQ}, e2), \sigma) \xrightarrow{e} v1 \leq v2}$$

$$\boxed{\textsf{sos-UnaryExpr}}\ \frac{(e,\sigma) \xrightarrow{e} v}{(\textit{mk-UnaryExpr}(\textsc{Minus}, e), \sigma) \xrightarrow{e} -v}$$

$$\boxed{\textsf{sos-id}}\ \frac{id\colon Id}{(id,\sigma) \xrightarrow{e} \sigma(id)}$$

B.4 Inference Rules

$$\boxed{\sigma\textsf{-Equiv}}\ \frac{(s,\sigma) \xrightarrow{s} \sigma' \qquad (s,\sigma) \xrightarrow{s} \sigma''}{\sigma' = \sigma''}$$

$$\boxed{\dagger\textsf{-Equiv}}\ \frac{\sigma' = \sigma \dagger \{x_1 \mapsto v_1, \ldots, x_n \mapsto v_n\}}{\sigma'(x_i) = v_i}$$

$$\boxed{\textsf{GT-F}}\ \frac{(\textit{mk-RelExpr}(x, \mathrm{GT}, y), \sigma) \xrightarrow{e} \mathbf{false}}{\sigma(x) \leq \sigma(y)}$$

$$\boxed{\textsf{GT-T}}\ \frac{(\textit{mk-RelExpr}(x, \mathrm{GT}, y), \sigma) \xrightarrow{e} \mathbf{true}}{\sigma(x) > \sigma(y)}$$

$$\boxed{\textsf{LT-F}}\frac{(\textit{mk-RelExpr}(x, \mathrm{LT}, y), \sigma) \xrightarrow{e} \mathbf{false}}{\sigma(x) \geq \sigma(y)}$$

$$\boxed{\textsf{LT-T}}\frac{(\textit{mk-RelExpr}(x, \mathrm{LT}, y), \sigma) \xrightarrow{e} \mathbf{true}}{\sigma(x) < \sigma(y)}$$

Appendix C

Simple Language

C.1 Abstract Syntax

$$\begin{aligned} Program &:: \; vars : Id \xrightarrow{m} Type \\ & \; body : Stmt \end{aligned}$$

$Type = \textsc{IntType} \mid \textsc{ArrayType}$

$Stmt = If \mid Assign \mid While \mid Seq \mid Swap \mid GDo$

$$\begin{aligned} Assign &:: \; lhs : VarRef \\ & \; rhs : Expr \end{aligned}$$

$$\begin{aligned} Seq &:: \; sl : Stmt \\ & \; sr : Stmt \end{aligned}$$

$$\begin{aligned} If &:: \; test : Expr \\ & \; th : Stmt \\ & \; el : Stmt \end{aligned}$$

$$\begin{aligned} While &:: \; test : Expr \\ & \; body : Stmt \end{aligned}$$

$$\begin{aligned} Swap &:: \; lhs : Id \\ & \; rhs : Id \end{aligned}$$

$GDo :: body : Clause^*$

$$\begin{aligned} Clause &:: \; b : Expr \\ & \; s : Statement \end{aligned}$$

$Expr = Value \mid ArithExpr \mid RelExpr \mid UnaryExpr \mid VarRef$

$VarRef = Id \mid ArrayRef$

$$\begin{aligned}
&\mathit{ArrayRef} :: \mathit{array} : \mathit{Id}\\
&\qquad\qquad \mathit{index} : \mathit{Expr}
\end{aligned}$$

$$\begin{aligned}
&\mathit{ArithExpr} :: \mathit{opd}1 : \mathit{Expr}\\
&\qquad\qquad \mathit{op} : \textsc{Plus} \mid \textsc{Minus}\\
&\qquad\qquad \mathit{opd}2 : \mathit{Expr}
\end{aligned}$$

$$\begin{aligned}
&\mathit{RelExpr} :: \mathit{opd}1 : \mathit{Expr}\\
&\qquad\qquad \mathit{op} : \textsc{Equals} \mid \textsc{NotEquals} \mid \mathrm{LT} \mid \mathrm{GT} \mid \mathrm{GTEQ} \mid \mathrm{LTEQ}\\
&\qquad\qquad \mathit{opd}2 : \mathit{Expr}
\end{aligned}$$

$$\begin{aligned}
&\mathit{UnaryExpr} :: \mathit{opd} : \mathit{Expr}\\
&\qquad\qquad \mathit{op} : \textsc{Minus}
\end{aligned}$$

$$\mathit{Value} = \mathbb{Z} \mid \mathbb{B} \mid \mathit{ArrayValue}$$

$$\mathit{ArrayValue} = \mathbb{N} \xrightarrow{m} \mathbb{Z}$$

C.2 Context Conditions

$$\begin{aligned}
&\mathit{wf\text{-}Program} : \mathit{Program} \to \mathbb{B}\\
&\mathit{wf\text{-}Program}(\mathit{mk\text{-}Program}(\mathit{vars}, \mathit{body}) \triangleq\\
&\qquad \mathit{wf\text{-}Stmt}(\mathit{body}, \mathit{vars})
\end{aligned}$$

$$\mathit{wf\text{-}Stmt} : (\mathit{Stmt} \times \mathit{Id} \xrightarrow{m} \mathit{Type}) \to \mathbb{B}$$

$$\begin{aligned}
&\mathit{wf\text{-}Stmt}(\mathit{mk\text{-}Assign}(\mathit{lhs}, \mathit{rhs}), \mathit{tpm}) \triangleq\\
&\qquad \mathit{lhs} \in \mathbf{dom}\, \mathit{tpm} \wedge\\
&\qquad \mathit{typeof}(\mathit{rhs}, \mathit{tpm}) = \textsc{IntType}
\end{aligned}$$

$$\begin{aligned}
&\mathit{wf\text{-}Stmt}(\mathit{mk\text{-}Seq}(\mathit{sl}, \mathit{sr}), \mathit{tpm}) \triangleq\\
&\qquad \mathit{wf\text{-}Stmt}(\mathit{sl}, \mathit{tpm}) \wedge\\
&\qquad \mathit{wf\text{-}Stmt}(\mathit{sr}, \mathit{tpm})
\end{aligned}$$

$$\begin{aligned}
&\mathit{wf\text{-}Stmt}(\mathit{mk\text{-}While}(b, s), \mathit{tpm}) \triangleq\\
&\qquad \mathit{typeof}(b, \mathit{tpm}) = \textsc{BoolType} \wedge\\
&\qquad \mathit{wf\text{-}Stmt}(s, \mathit{tpm})
\end{aligned}$$

$$\begin{aligned}
&\mathit{wf\text{-}Stmt}(\mathit{mk\text{-}Swap}(\mathit{lhs}, \mathit{rhs}), \mathit{tpm}) \triangleq\\
&\qquad \mathit{lhs} \in \mathbf{dom}\, \mathit{tpm} \wedge\\
&\qquad \mathit{rhs} \in \mathbf{dom}\, \mathit{tpm}
\end{aligned}$$

CCs for GDo

```
$typeof : Expr \times Id \xrightarrow{m} Type \to \textsc{Int} \mid \textsc{Bool}$
$typeof(e, tpm) \triangleq$
    cases $e$ of
        $mk\text{-}ArithExpr(\text{-},\text{-},\text{-}) \to \textsc{IntType}$
        $mk\text{-}RelExpr(\text{-},\text{-},\text{-}) \to \textsc{BoolType}$
        $mk\text{-}UnaryExpr(\text{-},\text{-}) \to \textsc{IntType}$
        $mk\text{-}ArrayRef(\text{-},\text{-}) \to \textsc{IntType}$
        $e \in \mathbb{B} \to \textsc{BoolType}$
        $e \in \mathbb{Z} \to \textsc{IntType}$
        $e : Id \to tpm(e)$
    end
```

C.3 Semantics

C.3.1 Semantic Objects

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

C.3.2 Program Semantics

The program semantics initialises the declared variables and sets their values to zero.

$$\xrightarrow{p} : \mathcal{P}\,(Program \times \textsc{Done})$$

$$\boxed{\textsf{Program}}\ \frac{\sigma = \{id \mapsto 0 \mid id \in \mathbf{dom}\ vars\} \qquad (body, \sigma) \xrightarrow{s} \sigma'}{(mk\text{-}Program(vars, body)) \xrightarrow{p} \textsc{Done}}$$

C.3.3 Statement Semantics

$$\xrightarrow{s} : \mathcal{P}\,((Stmt \times \Sigma) \times \Sigma)$$

$$\boxed{\textsf{If-T}}\ \frac{(test, \sigma) \xrightarrow{e} true \qquad (th, \sigma) \xrightarrow{s} \sigma'}{(mk\text{-}If(test, th, el), \sigma) \xrightarrow{s} \sigma'}$$

$$\boxed{\text{If-F}}\ \frac{\begin{array}{l}(test, \sigma) \xrightarrow{e} false\\ (el, \sigma) \xrightarrow{s} \sigma'\end{array}}{(mk\text{-}If(test, th, el), \sigma) \xrightarrow{s} \sigma'}$$

$$\boxed{\text{Assign}}\ \frac{(rhs, \sigma) \xrightarrow{e} v}{(mk\text{-}Assign(lhs, rhs), \sigma) \xrightarrow{s} \sigma \dagger \{lhs \mapsto v\}}$$

$$\boxed{\text{Assign-Array}}\ \frac{\begin{array}{l}(i, \sigma) \xrightarrow{e} index\\ index \in \mathbf{dom}\, \sigma(a)\\ (e, \sigma) \xrightarrow{e} v\end{array}}{(mk\text{-}Assign(mk\text{-}ArrayRef(a, i), e), \sigma) \xrightarrow{s} \sigma \dagger \{a \mapsto \sigma(a) \dagger \{index \mapsto v\}\}}$$

$$\boxed{\text{Seq}}\ \frac{\begin{array}{l}(sl, \sigma) \xrightarrow{s} \sigma'\\ (sr, \sigma') \xrightarrow{s} \sigma''\end{array}}{(mk\text{-}Seq(sl, sr), \sigma) \xrightarrow{s} \sigma''}$$

$$\boxed{\text{While-T}}\ \frac{\begin{array}{l}(test, \sigma) \xrightarrow{e} true\\ (body, \sigma) \xrightarrow{s} \sigma'\\ (mk\text{-}While(test, body), \sigma') \xrightarrow{s} \sigma''\end{array}}{(mk\text{-}While(test, body), \sigma) \xrightarrow{s} \sigma''}$$

$$\boxed{\text{While-F}}\ \frac{(test, \sigma) \xrightarrow{e} false}{(mk\text{-}While(test, body), \sigma) \xrightarrow{s} \sigma}$$

$$\boxed{\text{Swap}}\ \frac{\begin{array}{l}(lhs, \sigma) \xrightarrow{e} v_1\\ (rhs, \sigma) \xrightarrow{e} v_2\end{array}}{(mk\text{-}Swap(lhs, rhs), \sigma) \xrightarrow{s} \sigma \dagger \{lhs \mapsto v_2, rhs \mapsto v_1\}}$$

$$\boxed{\text{GDo-F}}\ \frac{\forall i \in \mathbf{inds}\ cs \cdot (cs(i).b, \sigma) \xrightarrow{e} \mathbf{false}}{(mk\text{-}GDo(cs), \sigma) \xrightarrow{s} \sigma}$$

$$\boxed{\textsf{GDo-T}}\ \frac{\begin{array}{l} i \in \mathbf{inds}\ cs \\ (cs(i).b, \sigma) \xrightarrow{e} \mathbf{true} \\ (cs(i).s, \sigma) \xrightarrow{s} \sigma' \\ (\textit{mk-GDo}(cs), \sigma') \xrightarrow{s} \sigma'' \end{array}}{(\textit{mk-GDo}(cs), \sigma) \xrightarrow{s} \sigma''}$$

C.3.4 Expression Semantics

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

$$\boxed{\textsf{Arith-Plus}}\ \frac{\begin{array}{l} (e1, \sigma) \xrightarrow{e} v1 \\ (e2, \sigma) \xrightarrow{e} v2 \end{array}}{(\textit{mk-ArithExpr}(e1, \textsc{Plus}, e2), \sigma) \xrightarrow{e} v1 + v2}$$

$$\boxed{\textsf{Arith-Minus}}\ \frac{\begin{array}{l} (e1, \sigma) \xrightarrow{e} v1 \\ (e2, \sigma) \xrightarrow{e} v2 \end{array}}{(\textit{mk-ArithExpr}(e1, \textsc{Minus}, e2), \sigma) \xrightarrow{e} v1\text{-}v2}$$

$$\boxed{\textsf{Rel-Equals}}\ \frac{\begin{array}{l} (e1, \sigma) \xrightarrow{e} v1 \\ (e2, \sigma) \xrightarrow{e} v2 \end{array}}{(\textit{mk-RelExpr}(e1, \textsc{Equals}, e2), \sigma) \xrightarrow{e} v1 = v2}$$

$$\boxed{\textsf{Rel-NotEquals}}\ \frac{\begin{array}{l} (e1, \sigma) \xrightarrow{e} v1 \\ (e2, \sigma) \xrightarrow{e} v2 \end{array}}{(\textit{mk-RelExpr}(e1, \textsc{NotEquals}, e2), \sigma) \xrightarrow{e} v1 \neq v2}$$

$$\boxed{\textsf{Rel-GT}}\ \frac{\begin{array}{l} (e1, \sigma) \xrightarrow{e} v1 \\ (e2, \sigma) \xrightarrow{e} v2 \end{array}}{(\textit{mk-RelExpr}(e1, \mathrm{GT}, e2), \sigma) \xrightarrow{e} v1 > v2}$$

$$\boxed{\textsf{Rel-LT}}\ \frac{\begin{array}{l} (e1, \sigma) \xrightarrow{e} v1 \\ (e2, \sigma) \xrightarrow{e} v2 \end{array}}{(\textit{mk-RelExpr}(e1, \mathrm{LT}, e2), \sigma) \xrightarrow{e} v1 < v2}$$

$$\boxed{\textsf{Rel-GTEQ}}\ \frac{(e1,\sigma) \xrightarrow{e} v1 \qquad (e2,\sigma) \xrightarrow{e} v2}{(\textit{mk-RelExpr}(e1, \mathrm{GTEQ}, e2), \sigma) \xrightarrow{e} v1 \geq v2}$$

$$\boxed{\textsf{Rel-LTEQ}}\ \frac{(e1,\sigma) \xrightarrow{e} v1 \qquad (e2,\sigma) \xrightarrow{e} v2}{(\textit{mk-RelExpr}(e1, \mathrm{LTEQ}, e2), \sigma) \xrightarrow{e} v1 \leq v2}$$

$$\boxed{\textsf{sos-UnaryExpr}}\ \frac{(e,\sigma) \xrightarrow{e} v}{(\textit{mk-UnaryExpr}(\textsc{Minus}, e), \sigma) \xrightarrow{e} -v}$$

$$\boxed{\textsf{ArrayRef}}\ \frac{(i,\sigma) \xrightarrow{e} \textit{index} \qquad \textit{index} \in \mathbf{dom}\,\sigma(a)}{(\textit{mk-ArrayRef}(a, i), \sigma) \xrightarrow{e} \sigma(a)(\textit{index})}$$

$$\boxed{\textsf{sos-id}}\ \frac{\textit{id}\colon \textit{Id}}{(\textit{id},\sigma) \xrightarrow{e} \sigma(\textit{id})}$$

C.4 Inference Rules

$$\boxed{\sigma\textsf{-Equiv}}\ \frac{(s,\sigma) \xrightarrow{s} \sigma' \qquad (s,\sigma) \xrightarrow{s} \sigma''}{\sigma' = \sigma''}$$

$$\boxed{\dagger\textsf{-Equiv}}\ \frac{\sigma' = \sigma \dagger \{x_1 \mapsto v_1, \ldots, x_n \mapsto v_n\}}{\sigma'(x_i) = v_i}$$

$$\boxed{\textsf{GT-F}}\ \frac{(\textit{mk-RelExpr}(x, \mathrm{GT}, y), \sigma) \xrightarrow{e} \mathbf{false}}{\sigma(x) \leq \sigma(y)}$$

$$\boxed{\textsf{GT-T}}\ \frac{(\textit{mk-RelExpr}(x, \mathrm{GT}, y), \sigma) \xrightarrow{e} \mathbf{true}}{\sigma(x) > \sigma(y)}$$

$$\boxed{\textsf{LT-F}}\,\frac{(\textit{mk-RelExpr}(x, \mathrm{LT}, y), \sigma) \xrightarrow{e} \mathbf{false}}{\sigma(x) \geq \sigma(y)}$$

$$\boxed{\textsf{LT-T}}\,\frac{(\textit{mk-RelExpr}(x, \mathrm{LT}, y), \sigma) \xrightarrow{e} \mathbf{true}}{\sigma(x) < \sigma(y)}$$

Appendix D

Parallel Language

This appendix contains the full language description of the *Parallel* language. The semantics of this language are defined in a small step semantics, so that we can model the interference that occurs between parallel threads of execution.

This language includes a simple one-dimensional array that can only contain integer values, just like the *Array* language.

The language includes an atomic assignment statement that evaluates an expression and then stores the result to the state in a single step, so it cannot be interfered with by a parallel thread of execution. The semantics for this atomic assignment statement is given in a big step semantics, since it should be executed in a single step.

Concurrency is introduced into the language by the *Par* statement, which has a left and a right statement that are executed in parallel.

D.1 Abstract Syntax

$$\begin{aligned} \mathit{Program} :: \ & \mathit{vars} : \mathit{Id} \xrightarrow{m} \mathit{Type} \\ & \mathit{body} : \mathit{Stmt} \end{aligned}$$

$$\mathit{Type} = \textsc{IntType} \mid \textsc{ArrayType}$$

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

$$\begin{aligned} \mathit{Assign} :: \ & \mathit{lhs} : \mathit{VarRef} \\ & \mathit{rhs} : \mathit{Expr} \end{aligned}$$

$$\begin{aligned} \mathit{Seq} :: \ & \mathit{sl} : \mathit{Stmt} \\ & \mathit{sr} : \mathit{Stmt} \end{aligned}$$

$$\begin{aligned} \mathit{Par} :: \ & \mathit{sl} : \mathit{Stmt} \\ & \mathit{sr} : \mathit{Stmt} \end{aligned}$$

$$\begin{aligned} \mathit{If} \;::\; \mathit{test} &: \mathit{Expr} \\ \mathit{th} &: \mathit{Stmt} \\ \mathit{el} &: \mathit{Stmt} \end{aligned}$$

$$\begin{aligned} \mathit{While} \;::\; \mathit{test} &: \mathit{Expr} \\ \mathit{body} &: \mathit{Stmt} \end{aligned}$$

$\mathit{Expr} = \mathit{Value} \mid \mathit{ArithExpr} \mid \mathit{RelExpr} \mid \mathit{UnaryExpr} \mid \mathit{VarRef}$

$\mathit{VarRef} = \mathit{Id} \mid \mathit{ArrayRef}$

$$\begin{aligned} \mathit{ArrayRef} \;::\; \mathit{array} &: \mathit{Id} \\ \mathit{index} &: \mathit{Expr} \end{aligned}$$

$$\begin{aligned} \mathit{ArithExpr} \;::\; \mathit{opd}1 &: \mathit{Expr} \\ \mathit{op} &: \text{PLUS} \mid \text{MINUS} \\ \mathit{opd}2 &: \mathit{Expr} \end{aligned}$$

$$\begin{aligned} \mathit{RelExpr} \;::\; \mathit{opd}1 &: \mathit{Expr} \\ \mathit{op} &: \text{EQUALS} \mid \text{NOTEQUALS} \mid \text{LT} \mid \text{GT} \mid \text{GTEQ} \mid \text{LTEQ} \mid \text{AND} \mid \text{OR} \\ \mathit{opd}2 &: \mathit{Expr} \end{aligned}$$

$$\begin{aligned} \mathit{UnaryExpr} \;::\; \mathit{opd} &: \mathit{Expr} \\ \mathit{op} &: \text{MINUS} — \text{LEN} \end{aligned}$$

$\mathit{Value} = \mathbb{Z} \mid \mathbb{B} \mid \mathit{ArrayValue}$

$\mathit{ArrayValue} = \mathbb{N} \xrightarrow{m} \mathbb{Z}$

D.2 Context Conditions

$\mathit{wf\text{-}Program}\colon \mathit{Program} \to \mathbb{B}$
$\mathit{wf\text{-}Program}(\mathit{mk\text{-}Program}(\mathit{vars}, \mathit{body}) \triangleq$
$\quad \mathit{wf\text{-}Stmt}(\mathit{body}, \mathit{vars})$

$\mathit{wf\text{-}Stmt}\colon (\mathit{Stmt} \times \mathit{Id} \xrightarrow{m} \mathit{Type}) \to \mathbb{B}$

$\mathit{wf\text{-}Stmt}(\mathit{mk\text{-}Assign}(\mathit{lhs}, \mathit{rhs}), \mathit{tpm}) \triangleq$
$\quad \mathit{lhs} \in \mathbf{dom}\, \mathit{tpm} \wedge$
$\quad \mathit{typeof}(\mathit{rhs}, \mathit{tpm}) = \text{INT}$

$\mathit{wf\text{-}Stmt}(\mathit{mk\text{-}Seq}(\mathit{sl}, \mathit{sr}), \mathit{tpm}) \triangleq$
$\quad \mathit{wf\text{-}Stmt}(\mathit{sl}, \mathit{tpm}) \wedge$

$$\begin{array}{l}
\quad \textit{wf-Stmt}(sr, tpm) \\
\\
\textit{wf-Stmt}(\textit{mk-Par}(sl, sr), tpm) \triangleq \\
\quad \textit{wf-Stmt}(sl, tpm) \wedge \\
\quad \textit{wf-Stmt}(sr, tpm) \\
\\
\textit{wf-Stmt}(\textit{mk-While}(b, s), tpm) \triangleq \\
\quad \textit{typeof}(b, tpm) = \textsc{Bool} \wedge \\
\quad \textit{wf-Stmt}(s, tpm)
\end{array}$$

$$\begin{array}{l}
\textit{typeof}: \textit{Expr} \times \textit{Id} \xrightarrow{m} \textit{Type} \to \textsc{Int} \mid \textsc{Bool} \\
\textit{typeof}(e, tpm) \triangleq \\
\quad \textbf{cases } e \textbf{ of} \\
\quad\quad \begin{array}{r}
\textit{mk-ArithExpr}(\text{-},\text{-},\text{-}) \\
\textit{mk-RelExpr}(\text{-},\text{-},\text{-}) \\
\textit{mk-UnaryExpr}(\text{-},\text{-}) \\
e \in \mathbb{B} \\
e \in \mathbb{Z} \\
e: \textit{Id}
\end{array}
\begin{array}{l}
\to \textsc{IntType} \\
\to \textsc{BoolType} \\
\to \textsc{IntType} \\
\to \textsc{BoolType} \\
\to \textsc{IntType} \\
\to tpm(e)
\end{array} \\
\quad \textbf{end}
\end{array}$$

D.3 Semantics

D.3.1 Semantic Objects

$\Sigma = \textit{Id} \xrightarrow{m} \mathbb{Z} \mid \textit{ArrayValue}$

D.3.2 Program Semantics

The program semantics initialises the variables that are of type Int, and sets their values to zero. Variables of type ArrayType are not initialised.

$$\xrightarrow{p}: \mathcal{P}(\textit{Program} \times \textsc{Done})$$

$$\boxed{\textsf{Program}}\ \frac{\begin{array}{l}\sigma = \{id \mapsto 0 \mid id \in \textbf{dom}\ vars \wedge vars(id) = \textsc{ArrayType}\} \\ (body, \sigma) \xrightarrow{s} (\textbf{nil}, \sigma')\end{array}}{(\textit{mk-Program}(vars, body)) \xrightarrow{p} \textsc{Done}}$$

D.3.3 Statement Semantics

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

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

$$\boxed{\textsf{Assign-Array}}\ \frac{\begin{array}{l}(e, \sigma) \xrightarrow{e} v \\ (i, \sigma) \xrightarrow{e} index\end{array}}{(mk\text{-}Assign(mk\text{-}ArrayRef(a, i), e), \sigma) \xrightarrow{s} (\mathbf{nil}, \sigma \dagger \{a \mapsto \sigma(a) \dagger \{index \mapsto v\}\})}$$

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

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

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

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

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

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

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

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

$$\boxed{\text{While-E}}\ \frac{}{(mk\text{-}While(test, body), \sigma) \xrightarrow{s} (mk\text{-}If(test, mk\text{-}Seq(body, mk\text{-}While(test, body)), \mathbf{nil}), \sigma)}$$

$$\boxed{\mathbf{nil}}\ \frac{}{(\mathbf{nil}, \sigma) \xrightarrow{s} (\mathbf{nil}, \sigma)}$$

D.3.4 Expression Semantics

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

$$\boxed{\text{Arith-L}}\ \frac{(e1, \sigma) \xrightarrow{e} e1'}{(mk\text{-}ArithExpr(e1, op, e2), \sigma) \xrightarrow{e} mk\text{-}ArithExpr(e1', op, e2)}$$

$$\boxed{\text{Arith-R}}\ \frac{(e2, \sigma) \xrightarrow{e} e2'}{(mk\text{-}ArithExpr(e1, op, e2), \sigma) \xrightarrow{e} mk\text{-}ArithExpr(e1, op, e2')}$$

$$\boxed{\text{Arith-E}}\ \frac{\begin{array}{l} e1 \colon \mathbb{Z} \\ e2 \colon \mathbb{Z} \end{array}}{(mk\text{-}ArithExpr(e1, op, e2), \sigma) \xrightarrow{e} [\![op]\!](e1, e2)}$$

$$\boxed{\text{Rel-L}}\ \frac{(e1, \sigma) \xrightarrow{e} e1'}{(mk\text{-}RelExpr(e1, op, e2), \sigma) \xrightarrow{e} mk\text{-}RelExpr(e1', op, e2)}$$

$$\boxed{\text{Rel-R}}\ \frac{(e2, \sigma) \xrightarrow{e} e2'}{(mk\text{-}RelExpr(e1, op, e2), \sigma) \xrightarrow{e} mk\text{-}RelExpr(e1, op, e2')}$$

$$\boxed{\text{Rel-E}}\ \frac{\begin{array}{l} e1 \colon \mathbb{Z} \\ e2 \colon \mathbb{Z} \end{array}}{(mk\text{-}RelExpr(e1, op, e2), \sigma) \xrightarrow{e} [\![op]\!](e1, e2)}$$

$$\boxed{\text{Unary-Eval}}\ \frac{(e, \sigma) \xrightarrow{e} (e', \sigma')}{(mk\text{-}UnaryExpr(op, e), \sigma) \xrightarrow{e} mk\text{-}UnaryExpr(op, e')}$$

$$\boxed{\textsf{Unary-E}}\ \frac{e\colon\mathbb{Z}}{(\textit{mk-UnaryExpr}(\textsc{Minus}, e), \sigma) \xrightarrow{e} -e}$$

$$\boxed{\textsf{Unary-E}}\ \frac{e\colon\mathbb{Z}}{(\textit{mk-UnaryExpr}(\textsc{Len}, e), \sigma) \xrightarrow{e} e}$$

$$\boxed{\textsf{ArrayRef}}\ \frac{(i, \sigma) \xrightarrow{e} i'}{(\textit{mk-ArrayRef}(a, i), \sigma) \xrightarrow{e} \textit{mk-ArrayRef}(a, i')}$$

$$\boxed{\textsf{ArrayRef-E}}\ \frac{\begin{array}{l} i\colon\mathbb{N} \\ i \in \mathbf{dom}\,\sigma(a) \end{array}}{(\textit{mk-ArrayRef}(a, i), \sigma) \xrightarrow{e} \sigma(a)(\textit{index})}$$

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

Appendix E

Languages in the Proof Tool

The prototype proof assistant tool uses its own syntax to save rule libraries and proofs to file so that they can be loaded in again later. This appendix presents the language definitions of the three languages used in the thesis, along with the example proofs from Chapter 4.

E.1 Base Language

```
/* -------------------
 * -- base language --
 * -------------------
 */
theory base~;

//Abstract Syntax
rule asProgram, [vars] in mapof{Id}{Type} | [body] in Stmt, mkh_Program([vars], [body]) in Program, AS;
rule asIf, [test] in Expr | [th] in Stmt | [el] in Stmt, mkh_IfStmt([test], [th], [el]) in IfStmt, AS;
rule asWhile, [b] in Expr | [body] in Stmt, mkh_WhileStmt([b], [body]) in WhileStmt, AS;
rule asSeq, [ls] in Stmt | [rs] in Stmt, mkh_SeqStmt([ls], [rs]) in SeqStmt, AS;
rule asAssign, [lhs] in Id | [rhs] in Stmt, mkh_AssignStmt([lhs], [rhs]) in AssignStmt, AS;
rule asInt_Type, _, <IntType> in Type, AS;
rule asIf_Stmt, [s] in IfStmt, [s] in Stmt, AS;
rule asWhile_Stmt, [s] in WhileStmt, [s] in Stmt, AS;
rule asSeq_Stmt, [s] in SeqStmt, [s] in Stmt, AS;
rule asAssign_Stmt, [s] in AssignStmt, [s] in Stmt, AS;
rule asId_I, _, [i] in Id, AS;
rule as_ArithExpr, [opd1] in Expr | [op] in </<Plus> ^ <Minus> /> |
    [opd2] in Expr, mkh_ArithExpr([opd1], [op], [opd2]) in ArithExpr, AS;
rule as_RelExpr, [opd1] in Expr | [op] in </<Equals> ^ <NotEquals> ^ <LT> ^ <GT> ^ <LTEQ> ^ <GTEQ>/> |
    [opd2] in Expr, mkh_ArithRel([opd1], [op], [opd2]) in RelExpr, AS;
rule as_UnaryExpr, [e] in Expr, mkh_UnaryExpr(<Minus>, [e]) in UnaryExpr, AS;
rule as_Expr_Arith, [e] in ArithExpr, [e] in Expr, AS;
rule as_Expr_Rel, [e] in RelExpr, [e] in Expr, AS;
rule as_Expr_Unary, [e] in UnaryExpr, [e] in Expr, AS;
rule as_Expr_Id, [e] in Id, [e] in Expr, AS;
rule as_Expr_Value, [e] in Value, [e] in Expr, AS;
rule as_Bool_Value, [e] in Bool, [e] in Value, AS;
rule as_Int_Value, [e] in Int, [e] in Value, AS;

//Context Conditions
rule cc2, [1] in dom [2] | [1] in Id, !([2], ctp([1]), [2]([1])), CC;
rule cc3, [1] in Int, !([2], ctp([1]), IntTp), CC;
rule cc4, [1] in Bool, !([2], ctp([1]), BoolTp), CC;
rule cc5, !([1], ctp([2]), IntTp) | !([1], ctp([3]), IntTp),
    !([1], ctp(mkh_ArithExpr([2],[4],[3])), IntTp), CC;
rule cc6, !([1], ctp([2]), IntTp) | !([1], ctp([3]), IntTp),
    !([1], ctp(mkh_RelExpr([2],[4],[3])), BoolTp), CC;
rule cc7, !([1], ctp([2]), IntTp), !([1], ctp(mkh_UnaryExpr([3],[2])), IntTp), CC;
rule cc8, [1] in dom [2] | !([2], ctp([3]), tpm([1])), !([2], mkh_AssignStmt([1],[3]), wfStmt), CC;
rule cc9, !([1], ctp([2]), BoolTp) | !([1], [3], wfStmt) | !([1], [4], wfStmt),
    !([1], mkh_IfStmt([2],[3],[4]), wfStmt), CC;
rule cc10, !([1], ctp([2]), BoolTp) | !([1], [3], wfStmt), !([1], mkh_WhileStmt([2],[3]), wfStmt), CC;
rule cc11, !([1], [2], wfStmt) | !([1], [3], wfStmt), !([1], mkh_SeqStmt([2],[3]), wfStmt), CC;

//Semantic Objects
rule sem_State, [s] in mapof{Id}{Int}, [s] in Sigma, SOS;

//SOS Rules
rule sos_If_t, ([test], [sigma]) -e-> true | ([th], [sigma]) -s-> [sigma'],
    (mkh_IfStmt([test], [th], [el]), [sigma]) -s-> [sigma'], SOS;
rule sos_If_f, ([test], [sigma]) -e-> false | ([el], [sigma]) -s-> [sigma'],
    (mkh_IfStmt([test], [th], [el]), [sigma]) -s-> [sigma'], SOS;
rule sos_assign, ([rhs], [sigma]) -e-> [v],
    (mkh_AssignStmt([lhs], [rhs]), [sigma]) -s-> [sigma] ++ {[lhs] |-> [v]}, SOS;
rule sos_seq, ([sl], [sigma]) -s-> [sigma'] | ([sr],[sigma']) -s-> [sigma''],
    (mkh_SeqStmt([sl], [sr]), [sigma]) -s-> [sigma''], SOS;
rule sos_while_t, ([test], [sigma]) -e-> true | ([body], [sigma]) -s-> [sigma'] |
    (mkh_WhileStmt([test], [body]),
    [sigma']) -s-> sigma'', (mkh_WhileStmt([test], [body]), [sigma]) -s-> [sigma''], SOS;
rule sos_while_f, ([test], [sigma]) -e-> false,
    (mkh_WhileStmt([test], [body]), [sigma]) -s-> [sigma], SOS;
rule sos_Arith_Plus, ([e1], [sigma]) -e-> [v1] | ([e2], [sigma]) -e-> [v2],
    (mkh_ArithExpr([e1], <Plus>, [e2]), [sigma]) -s-> [v1] + [v2], SOS;
rule sos_Arith_Minus, ([e1], [sigma]) -e-> [v1] | ([e2], [sigma]) -e-> [v2],
    (mkh_ArithExpr([e1], <Minus>, [e2]), [sigma]) -s-> [v1] - [v2], SOS;
rule sos_Rel_Equals, ([e1], [sigma]) -e-> [v1] | ([e2], [sigma]) -e-> [v2] | [v1] = [v2],
    (mkh_RelExpr([e1], <Equals>, [e2]), [sigma]) -s-> true, SOS;
rule sos_Rel_NotEquals, ([e1], [sigma]) -e-> [v1] | ([e2], [sigma]) -e-> [v2] | [v1] = [v2],
    (mkh_RelExpr([e1], <NotEquals>, [e2]), [sigma]) -s-> false, SOS;
rule sos_Rel_GT, ([e1], [sigma]) -e-> [v1] | ([e2], [sigma]) -e-> [v2],
    (mkh_RelExpr([e1], <GT>, [e2]), [sigma]) -e-> [v1] > [v2], SOS;
rule sos_Rel_LT, ([e1], [sigma]) -e-> [v1] | ([e2], [sigma]) -e-> [v2],
    (mkh_RelExpr([e1], <LT>, [e2]), [sigma]) -e-> [v1] < [v2], SOS;
rule sos_Rel_GTEQ, ([e1], [sigma]) -e-> [v1] | ([e2], [sigma]) -e-> [v2],
    (mkh_RelExpr([e1], <GTEQ>, [e2]), [sigma]) -e-> [v1] >= [v2], SOS;
rule sos_Rel_LTEQ, ([e1], [sigma]) -e-> [v1] | ([e2], [sigma]) -e-> [v2],
    (mkh_RelExpr([e1], <LTEQ>, [e2]), [sigma]) -e-> [v1] <= [v2], SOS;
rule sos_UnaryExpr, ([e], [sigma]) -e-> [v], (mkh_UnaryExpr(<Minus>, [e]), [sigma]) -e-> -[v], SOS;
rule sos_id, [id] in Id, ([id], [sigma]) -e-> [id]::[sigma], SOS;

//Inference rles
rule sigma_equiv, ([s], [sigma]) -s-> [sigma’] | ([s], [sigma]) -s-> [sigma’’],
    [sigma’] = [sigma’’], Axiom;
rule GT_F, (mkh_RelExpr([x], <GT>, [y]), [sigma]) -e-> false, [x]::[sigma] <= [y]::[sigma], Axiom;
rule GT_T, (mkh_RelExpr([x], <GT>, [y]), [sigma]) -e-> true, [x]::[sigma] > [y]::[sigma], Axiom;
rule LT_F, (mkh_RelExpr([x], <LT>, [y]), [sigma]) -e-> false, [x]::[sigma] >= [y]::[sigma], Axiom;
rule LT_T, (mkh_RelExpr([x], <LT>, [y]), [sigma]) -e-> true, [x]::[sigma] < [y]::[sigma], Axiom;
```

```
proof abs

from ("if x < 0 then r := -x else r := x fi", sigma_0) -s-> sigma_f
1 (("x < 0", sigma_0) -e-> true) or (("x < 0", sigma_0) -e-> false) by ??

2  from ("x < 0", sigma_0) -e-> true
2.1   -x::sigma_0 >= 0 by ?? //inst {|->}
2.2   ("r := -x", sigma_0) -s-> sigma_0 ++ { r |-> -sigma(x) } by ??
2.3   ("if x < 0 then r := -x else r := x fi", sigma_0) -s-> sigma_0 ++ { r |-> -x::sigma } by ??
2.4   sigma_f = sigma_0 ++ { r |-> -x::sigma } by ??
infer r::sigma_f >= 0 by ??

3 from ("x < 0", sigma_0) -e-> false
3.1   x::sigma_0 >= 0 by ??
3.2   ("r := x", sigma_0) -s-> sigma_0 ++ { r |-> sigma(x) } by ??
3.3   ("if x < 0 then r := -x else r := x fi", sigma_0) -s-> sigma_0 ++ { r |-> x::sigma } by ??
3.4   sigma_f = sigma_0 ++ { r |-> x::sigma } by ??
infer r::sigma_f >= 0 by ??
infer r::sigma_f >= 0 by ??
;
```

E.2 Simple Language

```
/* --------------------
 * -- Simple Language --
 * --------------------
 */
theory simple~;

//Abstract Syntax
rule asProgram, [vars] in mapof{Id}{Type} | [body] in Stmt, mkh_Program([vars], [body]) in Program, AS;
rule asIf, [test] in Expr | [th] in Stmt | [el] in Stmt, mkh_IfStmt([test], [th], [el]) in IfStmt, AS;
rule asWhile, [b] in Expr | [body] in Stmt, mkh_WhileStmt([b], [body]) in WhileStmt, AS;
rule asSeq, [ls] in Stmt | [rs] in Stmt, mkh_SeqStmt([ls], [rs]) in SeqStmt, AS;
rule asAssign, [lhs] in Id | [rhs] in Stmt, mkh_AssignStmt([lhs], [rhs]) in AssignStmt, AS;
rule asSwap, [lhs] in Id | [rhs] in Id, mkh_SwapStmt([lhs], [rhs]) in SwapStmt, AS;
rule asInt_Type, _, <IntType> in Type, AS;
rule asArray_Type, _, <ArrayType> in Type, AS;
rule asIf_Stmt, [s] in IfStmt, [s] in Stmt, AS;
rule asWhile_Stmt, [s] in WhileStmt, [s] in Stmt, AS;
rule asSeq_Stmt, [s] in SeqStmt, [s] in Stmt, AS;
rule asAssign_Stmt, [s] in AssignStmt, [s] in Stmt, AS;
rule asSwap_Stmt, [s] in SwapStmt, [s] in Stmt, As;
rule asId_I, _, [i] in Id, AS;
rule as_Value_Id, [e] in Id, [e] in VarRef, AS;
rule as_Value_ArrayRef, [e] in ArrayRef, [e] in VarRef, AS;
rule as_Expr_Arith, [e] in ArithExpr, [e] in Expr, AS;
rule as_Expr_Rel, [e] in RelExpr, [e] in Expr, AS;
rule as_Expr_Unary, [e] in UnaryExpr, [e] in Expr, AS;
rule as_Expr_Value, [e] in Value, [e] in Expr, AS;
rule as_Expr_VarRef, [e] in VarRef, [e] in Expr, AS;
rule as_ArrayRef, [array] in Id | [index] in Expr, mkh_ArrayRef([array], [index]) in ArrayRef, AS;
rule as_ArithExpr, [opd1] in Expr | [op] in </<Plus> ^ <Minus> /> | [opd2] in Expr,
    mkh_ArithExpr([opd1], [op], [opd2]) in ArithExpr, AS;
rule as_RelExpr, [opd1] in Expr | [op] in </<Equals> ^ <NotEquals> ^ <LT> ^ <GT> ^ <GTEQ> ^ <LTEQ>/> |
    [opd2] in Expr, mkh_ArithRel([opd1], [op], [opd2]) in RelExpr, AS;
rule as_UnaryExpr, [e] in Expr, mkh_UnaryExpr(<Minus>, [e]) in UnaryExpr, AS;
rule as_Bool_Value, [e] in Bool, [e] in Value, AS;
rule as_Int_Value, [e] in Int, [e] in Value, AS;
rule as_ArrayValue_Value, [e] in ArrayValue, [e] in Value, AS;
rule as_ArrayValue, [e] in mapof{Nat}{Bool}, [e] in ArrayValue, AS;

//Context Conditions
rule cc2, [1] in dom [2] | [1] in Id, !([2], ctp([1]), [2]([1])), CC;
rule cc3, [1] in Int, !([2], ctp([1]), IntTp), CC;
rule cc4, [1] in Bool, !([2], ctp([1]), BoolTp), CC;
rule cc5, !([1], ctp([2]), IntTp) | !([1], ctp([3]), IntTp),
    !([1], ctp(mkh_ArithExpr([2],[4],[3])), IntTp), CC;
rule cc6, !([1], ctp([2]), IntTp) | !([1], ctp([3]), IntTp),
    !([1], ctp(mkh_RelExpr([2],[4],[3])), BoolTp), CC;
rule cc7, !([1], ctp([2]), IntTp), !([1], ctp(mkh_UnaryExpr([3],[2])), IntTp), CC;
rule cc8, [1] in dom [2] | !([2], ctp([3]), tpm([1])), !([2], mkh_AssignStmt([1],[3]), wfStmt), CC;
rule cc9, !([1], ctp([2]), BoolTp) | !([1], [3], wfStmt) | !([1], [4], wfStmt),
    !([1], mkh_IfStmt([2],[3],[4]), wfStmt), CC;
rule cc10, !([1], ctp([2]), BoolTp) | !([1], [3], wfStmt), !([1], mkh_WhileStmt([2],[3]), wfStmt), CC;
rule cc11, !([1], [2], wfStmt) | !([1], [3], wfStmt), !([1], mkh_SeqStmt([2],[3]), wfStmt), CC;
```

```
//Semantic Objects
rule sem_State, [s] in mapof{Id}{Value}, [s] in Sigma, SOS;

//SOS Rules
rule sos_If_t, ([test], [sigma]) -e-> true | ([th], [sigma]) -s-> [sigma’],
    (mkh_IfStmt([test], [th], [el]), [sigma]) -s-> [sigma’], SOS;
rule sos_If_f, ([test], [sigma]) -e-> false | ([el], [sigma]) -s-> [sigma’],
    (mkh_IfStmt([test], [th], [el]), [sigma]) -s-> [sigma’], SOS;
rule sos_assign, ([rhs], [sigma]) -e-> [v], (mkh_AssignStmt([lhs], [rhs]),
    [sigma]) -s-> [sigma] ++ {[lhs] |-> [v]}, SOS;
rule sos_assign_array, ([i], [sigma]) -e-> [index] | [index] in dom [a]::[sigma] |
    ([e], [sigma]) -e-> [v], (mkh_Assign(mkh_ArrayRef([a], [i]), [e]), [sigma]) -s-> [sigma] ++
    {[a] |-> [a]::[sigma] ++ {[index] |-> [v]}}, SOS;
rule sos_seq, ([sl], [sigma]) -s-> [sigma’] | ([sr],[sigma’]) -s-> [sigma’’],
    (mkh_SeqStmt([sl], [sr]), [sigma]) -s-> [sigma’’], SOS;
rule sos_while_t, ([test], [sigma]) -e-> true | ([body], [sigma]) -s-> [sigma’] |
    (mkh_WhileStmt([test], [body]), [sigma’]) -s-> sigma’’,
    (mkh_WhileStmt([test], [body]), [sigma]) -s-> [sigma’’], SOS;
rule sos_while_f, ([test], [sigma]) -e-> false, (mkh_WhileStmt([test], [body]),
    [sigma]) -s-> [sigma], SOS;
rule sos_swap, ([lhs], [sigma]) -e-> [v1] | ([rhs], [sigma]) -e-> [v2], (mkh_SwapStmt([lhs], [rhs]),
    [sigma]) -s-> [sigma] ++ {[lhs] |-> [v2], [rhs] |-> [v1]}, SOS;
rule sos_swap_2, _ , (mkh_SwapStmt([lhs], [rhs]), [sigma]) -s-> [sigma] ++
    {[lhs] |-> [rhs]::[sigma], [rhs] |-> [lhs]::[sigma]}, SOS;
rule sos_Arith_Plus, ([e1], [sigma]) -e-> [v1] | ([e2], [sigma]) -e-> [v2],
    (mkh_ArithExpr([e1], <Plus>, [e2]), [sigma]) -s-> [v1] + [v2], SOS;
rule sos_Arith_Minus, ([e1], [sigma]) -e-> [v1] | ([e2], [sigma]) -e-> [v2],
    (mkh_ArithExpr([e1], <Minus>, [e2]), [sigma]) -s-> [v1] - [v2], SOS;
```

```
rule sos_Rel_Equals, ([e1], [sigma]) -e-> [v1] | ([e2], [sigma]) -e-> [v2] | [v1] = [v2],
    (mkh_RelExpr([e1], <Equals>, [e2]), [sigma]) -s-> true, SOS;
rule sos_Rel_NotEquals, ([e1], [sigma]) -e-> [v1] | ([e2], [sigma]) -e-> [v2] | [v1] = [v2],
    (mkh_RelExpr([e1], <NotEquals>, [e2]), [sigma]) -s-> false, SOS;
rule sos_Rel_GT, ([e1], [sigma]) -e-> [v1] | ([e2], [sigma]) -e-> [v2],
    (mkh_RelExpr([e1], <GT>, [e2]), [sigma]) -e-> [v1] > [v2], SOS;
rule sos_Rel_LT, ([e1], [sigma]) -e-> [v1] | ([e2], [sigma]) -e-> [v2],
    (mkh_RelExpr([e1], <LT>, [e2]), [sigma]) -e-> [v1] < [v2], SOS;
rule sos_Rel_GTEQ, ([e1], [sigma]) -e-> [v1] | ([e2], [sigma]) -e-> [v2],
    (mkh_RelExpr([e1], <GTEQ>, [e2]), [sigma]) -e-> [v1] >= [v2], SOS;
rule sos_Rel_LTEQ, ([e1], [sigma]) -e-> [v1] | ([e2], [sigma]) -e-> [v2],
    (mkh_RelExpr([e1], <LTEQ>, [e2]), [sigma]) -e-> [v1] <= [v2], SOS;
rule sos_UnaryExpr, ([e], [sigma]) -e-> [v], (mkh_UnaryExpr(<Minus>, [e]), [sigma]) -e-> -[v], SOS;
rule sos_ArrayRef, ([i], [sigma]) -e-> [index] | [index] in dom [a]::[sigma],
    (mkh_ArrayRef([a], [i]), [sigma]) -e-> [a]::[sigma](index), SOS;
rule sos_id, [id] in Id, ([id], [sigma]) -e-> [id]::[sigma], SOS;

//Inference rules
rule sigma_equiv, ([s], [sigma]) -s-> [sigma'] | ([s], [sigma]) -s-> [sigma''],
    [sigma'] = [sigma''], Axiom;
rule GT_F, (mkh_RelExpr([x], <GT>, [y]), [sigma]) -e-> false, [x]::[sigma] <= [y]::[sigma], Axiom;
rule GT_T, (mkh_RelExpr([x], <GT>, [y]), [sigma]) -e-> true, [x]::[sigma] > [y]::[sigma], Axiom;
rule LT_F, (mkh_RelExpr([x], <LT>, [y]), [sigma]) -e-> false, [x]::[sigma] >= [y]::[sigma], Axiom;
rule LT_T, (mkh_RelExpr([x], <LT>, [y]), [sigma]) -e-> true, [x]::[sigma] < [y]::[sigma], Axiom;

//proofs
proof array_assign
```

```
from i != j | a::sigma_0(j) = 2 | i::sigma_0 in ( dom a::sigma_0 ) | j::sigma in ( dom a::sigma_0 ) |
    i in Id | j in Id | ("a(i) := 5", sigma_0) -s-> sigma_f
1 (i, sigma_0) -e-> i::sigma_0  by ??
2 (5, sigma_0) -e-> 5  by ??
3 ("a(i) := 5", sigma_0) -s-> sigma_0 ++ { a |-> a::sigma_0 ++ { i::sigma_0 |-> 5}} by ??
4 sigma_f = sigma_1  by ??
5 a::sigma_f = a::sigma_0 ++ { i::sigma_0 |-> 5 } by ??
6 a::sigma_f(i) = 5  by ??
7 a::sigma_f(j) = 2  by ??
infer  ( a::sigma_f(j) = 2 ) and ( a::sigma_f(i) = 5 )  by ??
;

proof SwapV1

from ("r := x ; x := y ; y := r", sigma_0) -s-> sigma_f | x in Id | y in Id | r in Id
1 (x, sigma_0) -e-> x::sigma_0  by base.sos_id (h2)
2 ("r := x", sigma_0) -s-> ( sigma_0 ++ {r |-> x::sigma_0} )  by base.sos_assign (1)
3 (y, sigma_0 ++ {r |-> x::sigma_0}) -e-> y::sigma_0  by base.sos_id (h3)
4 ("x := y", sigma_0 ++ {r |-> x::sigma_0}) -s->
        ( sigma_0 ++ {r |-> x::sigma_0, x |-> y::sigma_0} )  by ??
5 (r, sigma_0 ++ {r |-> x::sigma_0, x |-> y::sigma_0}) -e-> x::sigma_0  by base.sos_id (h4)
6 ("y := r", sigma_0 ++ {r |-> x::sigma_0, x |-> y::sigma_0}) -s->
        ( sigma_0 ++ {r |-> x::sigma_0, x |-> y::sigma_0, y |-> x::sigma_0} )  by ??
7 ("x := y ; y := r", sigma_0 ++ {r |-> x::sigma_0}) -s->
        ( sigma_0 ++ {r |-> x::sigma_0, x |-> y::sigma_0, y |-> x::sigma_0} )  by base.sos_seq (4, 6)
8 ("r := x ; x := y ; y := r", sigma_0) -s->
        ( sigma_0 ++ {r |-> x::sigma_0, x |-> y::sigma_0, y |-> x::sigma_0} )  by base.sos_seq (2, 7)
9 sigma_f = ( sigma_0 ++ {r |-> x::sigma_0, x |-> y::sigma_0, y |-> x::sigma_0} )
```

```
        by base.sigma_equiv (h1, 8)
10 x::sigma_f = y::sigma_0  by ??
11 y::sigma_f = x::sigma_0  by ??
infer  ( x::sigma_f = y::sigma_0 ) and ( y::sigma_f = x::sigma_0 )  by prop.andI (10, 11)
;

proof SwapV2

from ("x $ y", sigma_0) -s-> sigma_f | x in Id | y in Id
1 (x, sigma_0) -e-> x::sigma_0  by base.sos_id (h2)
2 (y, sigma_0) -e-> y::sigma_0  by base.sos_id (h3)
3 ("x $ y", sigma_0) -s-> ( sigma_0 ++ {x |-> y::sigma_0, y |-> x::sigma_0} )
        by base.sos_swap (1, 2)
4 sigma_f = ( sigma_0 ++ {x |-> y::sigma_0, y |-> x::sigma_0} )  by base.sigma_equiv (h1, 3)
5 x::sigma_f = y::sigma_0  by ??
6 y::sigma_f = x::sigma_0  by ??
infer  ( x::sigma_f = y::sigma_0 ) and ( y::sigma_f = x::sigma_0 )  by prop.andI (5, 6)
;

proof SwapV3
from ("x $ y", sigma_0) -s-> sigma_f | x in Id | y in Id
1 ("x $ y", sigma_0) -s-> ( sigma_0 ++ {x |-> y::sigma_0, y |-> x::sigma_0} )  by base.sos_swap_2 ()
2 sigma_f = ( sigma_0 ++ {x |-> y::sigma_0, y |-> x::sigma_0} )  by base.sigma_equiv (h1, 1)
3 x::sigma_f = y::sigma_0  by ??
4 y::sigma_f = x::sigma_0  by ??
infer  ( x::sigma_f = y::sigma_0 ) and ( y::sigma_f = x::sigma_0 )  by prop.andI (3, 4)
;
```

E.3 Parallel Language

```
/* -----------------------
 * -- Parallel Language --
 * -----------------------
 */
theory par~;

//Abstract Syntax
rule asProgram, [vars] in mapof{Id}{Type} | [body] in Stmt, mkh_Program([vars], [body]) in Program, AS;
rule asIf, [test] in Expr | [th] in Stmt | [el] in Stmt, mkh_IfStmt([test], [th], [el]) in IfStmt, AS;
rule asWhile, [b] in Expr | [body] in Stmt, mkh_WhileStmt([b], [body]) in WhileStmt, AS;
rule asSeq, [ls] in Stmt | [rs] in Stmt, mkh_SeqStmt([ls], [rs]) in SeqStmt, AS;
rule asPar, [ls] in Stmt | [rs] in Stmt, mkh_ParStmt([ls], [rs]) in ParStmt, AS;
rule asAssign, [lhs] in Id | [rhs] in Stmt, mkh_AssignStmt([lhs], [rhs]) in AssignStmt, AS;
rule asInt_Type, _, <IntType> in Type, AS;
rule asArray_Type, _, <ArrayType> in Type, AS;
rule asIf_Stmt, [s] in IfStmt, [s] in Stmt, AS;
rule asWhile_Stmt, [s] in WhileStmt, [s] in Stmt, AS;
rule asSeq_Stmt, [s] in SeqStmt, [s] in Stmt, AS;
rule asPar_Stmt, [s] in ParStmt, [s] in Stmt, AS;
rule asAssign_Stmt, [s] in AssignStmt, [s] in Stmt, AS;
rule asNil_Stmt, [s] in Nil, [s] in Stmt, AS;
rule asId_I, _, [i] in Id, AS;
rule as_Value_Id, [e] in Id, [e] in VarRef, AS;
rule as_Value_ArrayRef, [e] in ArrayRef, [e] in VarRef, AS;
rule as_Expr_Arith, [e] in ArithExpr, [e] in Expr, AS;
rule as_Expr_Rel, [e] in RelExpr, [e] in Expr, AS;
rule as_Expr_Unary, [e] in UnaryExpr, [e] in Expr, AS;
rule as_Expr_Value, [e] in Value, [e] in Expr, AS;
rule as_Expr_VarRef, [e] in VarRef, [e] in Expr, AS;
rule as_ArrayRef, [array] in Id | [index] in Expr, mkh_ArrayRef([array], [index]) in Arrayref, AS;
rule as_ArithExpr, [opd1] in Expr | [op] in </<Plus> ^ <Minus>/> | [opd2] in Expr,
    mkh_ArithExpr([opd1], [op], [opd2]) in ArithExpr, AS;
rule as_RelExpr, [opd1] in Expr |
    [op] in </<Equals> ^ <NotEquals> ^ <LT> ^ <GT> ^ <GTEQ> ^ <LTEQ> ^ <AND> ^ <OR>/> |
    [opd2] in Expr, mkh_ArithRel([opd1], [op], [opd2]) in RelExpr, AS;
rule as_UnaryExpr, [e] in Expr, mkh_UnaryExpr(<Minus>, [e]) in UnaryExpr, AS;
rule as_Bool_Value, [e] in Bool, [e] in Value, AS;
rule as_int_Value, [e] in Int, [e] in Value, AS;
rule as_ArrayValue_Value, [e] in ArrayValue, [e] in Value, AS;
rule as_ArrayValue, [e] in mapof{Nat}{Bool}, [e] in ArrayValue, AS;

//Context Conditions
rule cc2, [1] in dom [2] | [1] in Id, !([2], ctp([1]), [2]([1])), CC;
rule cc3, [1] in Int, !([2], ctp([1]), IntTp), CC;
rule cc4, [1] in Bool, !([2], ctp([1]), BoolTp), CC;
rule cc5, !([1], ctp([2]), IntTp) | !([1], ctp([3]), IntTp),
    !([1], ctp(mkh_ArithExpr([2],[4],[3])), IntTp), CC;
rule cc6, !([1], ctp([2]), IntTp) | !([1], ctp([3]), IntTp),
    !([1], ctp(mkh_RelExpr([2],[4],[3])), BoolTp), CC;
rule cc7, !([1], ctp([2]), IntTp), !([1], ctp(mkh_UnaryExpr([3],[2])), IntTp), CC;
rule cc8, [1] in dom [2] | !([2], ctp([3]), tpm([1])), !([2], mkh_AssignStmt([1],[3]), wfStmt), CC;
rule cc9, !([1], ctp([2]), BoolTp) | !([1], [3], wfStmt) |
    !([1], [4], wfStmt), !([1], mkh_IfStmt([2],[3],[4]), wfStmt), CC;
rule cc10, !([1], ctp([2]), BoolTp) | !([1], [3], wfStmt),
    !([1], mkh_WhileStmt([2],[3]), wfStmt), CC;
rule cc11, !([1], [2], wfStmt) | !([1], [3], wfStmt),
    !([1], mkh_SeqStmt([2],[3]), wfStmt), CC;

//Semantic Objects
rule sem_State, [s] in mapof{Id}{Value}, [s] in Sigma, SOS;

//SOS Rules
rule sos_assign, ([e], [sigma]) -e-> [v],
    (mkh_AssignStmt([id], [e]), [sigma]) -s-> ("nil", [sigma] ++ {[id] |-> [v]}), SOS;
rule sos_assign_array, ([e], [sigma]) -e-> [v] | ([i], [sigma]) -e-> [index],
    (mkh_AssignStmt(mkh_ArrayRef([a], [i]), [e]), [sigma]) -s->
    ("nil", [sigma] ++ {[a] |-> [a]::[sigma] ++ {[index] |-> [v]}}), SOS;
rule sos_seq_step, ([sl], [sigma]) -s-> ([sl'], [sigma']),
    (mkh_SeqStmt([sl], [sr]), [sigma]) -s-> (mkh_SeqStmt([sl'], [sr]), [sigma']), SOS;
rule sos_seq_E, _, (mkh_SeqStmt("nil", [sr]), [sigma]) -s-> ([sr], [sigma]), SOS;
rule sos_par_L, ([sl], [sigma]) -s-> ([sl'], [sigma']),
    (mkh_ParStmt([sl], [sr]), [sigma]) -s-> (mkh_ParStmt([sl'], [sr]), [sigma']), SOS;
rule sos_par_R, ([sr], [sigma]) -s-> ([sr'], [sigma']),
    (mkh_ParStmt([sl], [sr]), [sigma]) -s-> (mkh_ParStmt([sl], [sr']), [sigma']), SOS;
rule sos_par_E, _, (mkh_ParStmt("nil", "nil"), [sigma]) -s-> ("nil", [sigma]), SOS;
rule sos_If_Eval, ([test], [sigma]) -e-> test',
    (mkh_IfStmt([test], [th], [el]), [sigma]) -s-> (mkh_IfStmt([test'], [th], [el]), [sigma]), SOS;
rule sos_If_T, _, (mkh_IfStmt("true", [th], [el]), [sigma]) -s-> ([th], [sigma]), SOS;
rule sos_If_F, _, (mkh_IfStmt("false", [th], [el]), [sigma]) -s-> ([th], [sigma]), SOS;
rule sos_while_E, _, (mkh_WhileStmt([test], [body]), [sigma]) -s->
    (mkh_IfStmt([test], mkh_SeqStmt([body], mkh_WhileStmt([test], [body])), "nil"), [sigma]), SOS;
rule sos_Arith_L, ([e1], [sigma]) -e-> [e1'],
    (mkh_ArithExpr([e1], [op], [e2]), [sigma]) -e-> mkh_ArithExpr([e1'], [op], [e2]), SOS;
rule sos_Arith_R, ([e2], [sigma]) -e-> [e2'],
    (mkh_ArithExpr([e1], [op], [e2]), [sigma]) -e-> mkh_ArithExpr([e1], [op], [e2']), SOS;
rule sos_Arith_Plus_E, [e1] in Int | [e2] in Int,
    (mkh_ArithExpr([e1], [op], [e2]), [sigma]) -e-> [e1] + [e2'], SOS;
rule sos_Arith_Minus_E, [e1] in Int | [e2] in Int,
    (mkh_ArithExpr([e1], [op], [e2]), [sigma]) -e-> [e1] - [e2'], SOS;
rule sos_Rel_L, ([e1], [sigma]) -e-> [e1'],
    (mkh_RelExpr([e1], [op], [e2]), [sigma]) -e-> mkh_RelExpr([e1'], [op], [e2]), SOS;
rule sos_Rel_R, ([e2], [sigma]) -e-> [e2'],
    (mkh_RelExpr([e1], [op], [e2]), [sigma]) -e-> mkh_RelExpr([e1], [op], [e2']), SOS;
    (mkh_RelExpr([e1], [op], [e2]), [sigma]) -e-> [[op]]([e1] [e2']), SOS;
rule sos_Unary_Eval, ([e], [sigma]) -e-> ([e'], [sigma']),
    (mkh_UnaryExpr(<Minus>, [e]), [sigma]) -e-> mkh_UnaryExpr(<Minus>, [e']), SOS;
rule sos_UnaryExpr_E, [e] in Int, (mkh_UnaryExpr(<Minus>, [e]), [sigma]) -e-> -[e], SOS;
rule sos_ArrayRef, ([i], [sigma]) -e-> [i'],
    (mkh_ArrayRef([a], [i]), [sigma]) -e-> mkh_ArrayRef([a], [i']), SOS;
rule sos_ArrayRef_E, [i] in Nat | i in dom [a]::[sigma],
    (mkh_ArrayRef([a], [i]), [sigma]) -e-> [a]::[sigma](index), SOS;
rule sos_id, [id] in Id, ([id], [sigma]) -e-> [id]::[sigma], SOS;
rule sos_nil, _, ("nil", [sigma]) -s-> ("nil", [sigma]), SOS;

//Inference rules
rule sos_seq_step_2, ([sl], [sigma]) -s->* ([sl'], [sigma']),
    (mkh_SeqStmt([sl], [sr]), [sigma]) -s->* (mkh_SeqStmt([sl'], [sr]), [sigma']), SOS;
rule sigma_equiv, ([s], [sigma]) -s-> [sigma'] | ([s], [sigma]) -s-> [sigma''],
    [sigma'] = [sigma''], Axiom;
rule GT_F, (mkh_RelExpr([x], <GT>, [y]), [sigma]) -e-> false, [x]::[sigma] <= [y]::[sigma], Axiom;
rule GT_T, (mkh_RelExpr([x], <GT>, [y]), [sigma]) -e-> true, [x]::[sigma] > [y]::[sigma], Axiom;
rule LT_F, (mkh_RelExpr([x], <LT>, [y]), [sigma]) -e-> false, [x]::[sigma] >= [y]::[sigma], Axiom;
rule LT_T, (mkh_RelExpr([x], <LT>, [y]), [sigma]) -e-> true, [x]::[sigma] < [y]::[sigma], Axiom;
rule EQ_T, ([1] = [2], [sigma]) -s-> true, [1]::[sigma] = [2], Axiom;
rule EQ_T_2, ([1]([2]) = [3], [sigma]) -s-> true, [1]::[sigma]([3]) = [3], Axiom;

//proofs
proof lemma_1

from ("while a(i) != 5 and r = 0 do i := i + 1 od", sigma_m) -s->*
    ("while a(i) != 5 and r = 0 do i := i + 1 od", sigma_n) or
    ("while a(i) != 5 and r = 0 do i := i + 1 od", sigma_m) -s->*
    ("nil", sigma_n) | exists k in {i,...,j} & a::sigma_m(k) = 5

1 (a(i) != 5 and r = 0, sigma_m) -e-> true or (a(i) != 5 and r = 0, sigma_m) -e-> false by IJ
2 from (a(i) != 5 and r = 0, sigma_m) -e-> true
2.1 ("while a(i) != 5 and r = 0 do i := 1 + 1 od", sigma_m) -s->
    ("if a(i) != 5 and r = 0 then i := 1 + 1 ;
    while a(i) != 5 and r = 0 do i := 1 + 1 od fi", sigma_m) by par.sos_while_E ()
2.2 ("if a(i) != 5 and r = 0 then i := 1 + 1 ;
    while a(i) != 5 and r = 0 do i := 1 + 1 od fi", sigma_m) -s->
    ("if true then i := 1 + 1 ; while a(i) != 5 and r = 0 do i := 1 + 1 od fi", sigma_m)
    by par.sos_If_Eval (2.h1)
2.3 ("if true then i := 1 + 1 ; while a(i) != 5 and r = 0 do i := 1 + 1 od fi", sigma_m) -s->
    ("i := i + 1 ; while a(i) != 5 and r = 0 do i := 1 + 1 od", sigma_m) by par.sos_If_T ()
2.4 ("i := i + 1", sigma_m) -e-> sigma_m(i) + 1 by IJ
2.5 ("i := i + 1", sigma_m) -s-> ("nil", sigma_m ++ { i |-> sigma_m(i) + 1})
    by par.sos_assign (?)
```

```
2.6 ("i := i + 1 ; while a(i) != 5 and r = 0 do i := 1 + 1 od", sigma_m) -s->
    ("nil ; while a(i) != 5 and r = 0 do i := 1 + 1 od", sigma_m ++ { i |-> sigma_m(i) + 1})
        by par.sos_seq_step (2.5)
2.7 ("nil ; while a(i) != 5 and r = 0 do i := 1 + 1 od", sigma_m ++ { i |-> sigma_m(i) + 1}) -s->
    ("while a(i) != 5 and r = 0 do i := 1 + 1 od", sigma_m ++ { i |-> sigma_m(i) + 1})
    by par.sos_seq_E ()
2.8 sigma_n = sigma_m ++ { i |-> sigma_m(i) + 1} by IJ
2.9 sigma_n(i) = sigma_m(i) + 1 by IJ (2.8)
2.10 a::sigma_m(i) != 5 by IJ (2.h1)
2.11 exists k in {i + 1,...,j} & a::sigma_m(k) = 5 by IJ (h2, 2.9)
  infer exists k in {i,...,j} & a::sigma_n(k) = 5 by IJ (2.8, 2.10)
3 from (a(i) != 5 and r = 0, sigma_m) -e-> false
3.1 ("while a(i) != 5 and r = 0 do i := 1 + 1 od", sigma_m) -s->
    ("if a(i) != 5 and r = 0 then i := i + 1 ;
    while a(i) != 5 and r = 0 do i := 1 + 1 od fi", sigma_m) by par.sos_while_E ()
3.2 ("if a(i) != 5 and r = 0 then i := i + 1 ;
    while a(i) != 5 and r = 0 do i := 1 + 1 od fi", sigma_m) -s->
    ("if false then i := i + 1 ; while a(i) != 5 and r = 0 do i := 1 + 1 od fi", sigma_m)
    by par.sos_If_Eval (3.h1)
3.3 ("if false then i := i + 1 ; while a(i) != 5 and r = 0 do i := 1 + 1 od fi", sigma_m) -s->
    ("nil", sigma_m) by par.sos_If_F ()
3.4 sigma_n = sigma_m by IJ
  infer exists k in {i,...,j} & a::sigma_n(k) = 5 by IJ (3.4, h2)
infer exists k in {i,...,j} & a::sigma_n(k) = 5 by prop.orE (1, 2, 3)
;

proof lemma_3
```

```
from ("if a(i) = 5 then r := i fi", sigma_0) -s->* ("nil", sigma_f) | i in Id
1 (a(i) = 5, sigma_0) -e-> true or (a(i) = 5, sigma_0) -e-> false by IJ
2 from (a(i) = 5, sigma_0) -e-> true
2.1 ("if a(i) = 5 then r := i fi", sigma_0) -s-> ("if true then r := i fi", sigma_0)
    by par.sos_If_Eval (2.h1)
2.2 ("if true then r := i fi", sigma_0) -s-> ("r := i", sigma_0) by par.sos_If_T ()
2.3 (i, sigma_0) -e-> i::sigma_0 by par.sos_id (h2)
2.4 ("r := i", sigma_0) -s-> ("nil", sigma_0 ++ { r |-> i::sigma_0 }) by par.sos_assign (2.3)
2.5 sigma_f = sigma_0 ++ { r |-> i::sigma_0 } by IJ
2.6 a::sigma_0(i::sigma_0) = 5 by par.EQ_T_2 (2.h1)
2.7 r::sigma_f = i::sigma_0 by IJ (2.5)
2.8 a::sigma_f = a::sigma_0 by IJ (2.5)
2.9 a::sigma_0(r::sigma_f) = 5 by IJ (2.7, 2.6)
2.10 a::sigma_f(r::sigma_f) = 5 by IJ (2.8, 2.6)
  infer r::sigma_0 != r::sigma_r => a::sigma_f(r::sigma_f) = 5 by prop.implILvac (2.10)
3 from (a(i) = 5, sigma_0) -s-> false
3.1 ("if a(i) = 5 then r := i fi", sigma_0) -s-> ("if false then r := i fi", sigma_0)
    by par.sos_If_Eval (3.h1)
3.2 ("if false then r := i fi", sigma_0) -s-> ("nil", sigma_0) by par.sos_If_F ()
3.3 sigma_f = sigma_0 by IJ (3.2, h1)
3.4 r::sigma_f = r::sigma_0 by IJ (3.3)
3.5 not (r::sigma_f != r::sigma_0) by IJ (3.4)
  infer r::sigma_0 != r::sigma_r => a::sigma_f(r::sigma_f) = 5 by prop.implIRvac (2.5)
infer r::sigma_0 != r::sigma_r => a::sigma_f(r::sigma_f) = 5 by prop.orE (1, 2, 3)
;

proof lemma_5
```

```
from ("while a(i) != 5 and r = 0 do i := i + 1 od ; if a(i) = 5 then r := i fi", sigma_m) -s->*
     ("nil", sigma_n) | exists k in {i,...,j} & a::sigma_n(k) = 5 | i in Id
1 ("while a(i) != 5 and r = 0 do i := i + 1 od ; if a(i) = 5 then r := i fi", sigma_m) -s->*
    ("nil ; if a(i) = 5 then r := i fi", sigma_t) by IJ
2 (a(i) != 5 and r = 0, sigma_t) -e-> false by IJ (1)
3 (a(i) != 5, sigma_t) -e-> false or (r = 0, sigma_t) -e-> false by IJ (2)
4 ("nil ; if a(i) = 5 then r := i fi", sigma_t) -s-> ("if a(i) = 5 then r := i fi", sigma_t)
    by par.sos_seq_E ()
5 from (a(i) != 5, sigma_t) -e-> false
5.1 (a(i) = 5, sigma_t) -e-> true by IJ (5.h1)
5.2 ("if a(i) = 5 then r := i fi", sigma_t) -s-> ("if true then r := i fi", sigma_t)
    by par.sos_If_Eval (5.1)
5.3 ("if true then r := i fi", sigma_t) -s-> ("r := i", sigma_t) by par.sos_If_T ()
5.4 (i, sigma_t) -e-> i::sigma_t by par.sos_id (h3)
5.5 ("r := i", sigma_t) -s-> ("nil", sigma_t ++ { r |-> i::sigma_t }) by par.sos_assign (5.4)
5.6 sigma_n = sigma_t ++ { r |-> i::sigma_t } by IJ (h1, 5.5)
5.7 i::sigma_t > 0 by IJ
5.8 r::sigma_n > 0 by IJ (5.6, 5.7)
  infer r::sigma_n != 0 by IJ (5.8)
6 from (r = 0, sigma_t) -e-> false
6.1  (a(i) = 5, sigma_t) -e-> true or (a(i) = 5, sigma_0) -e-> false by IJ
6.2  from (a(i) = 5, sigma_t) -e-> false
6.2.1  ("if a(i) = 5 then r := i fi", sigma_t) -s-> ("if false then r := i fi", sigma_t)
           by par.sos_If_Eval (6.2.h1)
6.2.2 ("if false then r := i fi", sigma_t) -s-> ("nil", sigma_t) by par.sos_If_T ()
6.2.3 sigma_n = sigma_t by IJ (h1, 6.2.2)
infer  r::sigma_n != 0 by IJ (6.2.3, 6.h1)
6.3 from (a(i) = 5, sigma_t) -e-> true
```

```
6.3.1 ("if a(i) = 5 then r := i fi", sigma_t) -s-> ("if true then r := i fi", sigma_t)
            by par.sos_If_Eval (6.3.h1)
6.3.2 ("if true then r := i fi", sigma_t) -s-> ("r := i", sigma_t) by par.sos_If_T ()
6.3.3 (i, sigma_t) -e-> i::sigma_t by par.sos_id (h3)
6.3.4 ("r := i", sigma_t) -s-> ("nil", sigma_t ++ { r |-> i::sigma_t })
            by par.sos_assign (6.3.3)
6.3.5 sigma_n = sigma_t ++ { r |-> i::sigma_t} by IJ (h1, 6.3.4)
6.3.6 i::sigma_i > 0 by IJ
6.3.7 r::sigma_n > 0 by IJ (6.3.6, 6.3.5)
infer r::sigma_n != 0 by IJ (6.3.7)
  infer r::sigma_n != 0 by prop.orE (6.1, 6.2, 6.3)
infer r::sigma_n != 0 by ??
;

proof par_search_abstract

from (mk_SeqStmt(init, mk_ParStmt(g,d)), sigma) -s->*
     ("nil", sigma_w) | exists k in {i,...,a.length} & a::sigma(k) = 5
1 (init, sigma) -s->* ("nil", sigma ++ { r |-> 0, i |-> 1, j |-> a.length}) by IJ
2 sigma_1 = sigma ++ { r |-> 0, i |-> 1, j |-> a.length} by IJ (1)
3 exists k in {i,...,j} & a::sigma_1(k) = 5 by IJ (h2, 2)
4 (mk_SeqStmt(init, mk_ParStmt(g,d)), sigma) -s->* (mk_SeqStmt("nil", mk_ParStmt(g,d)), sigma_1)
    by par.sos_seq_step_2 (1)
5 (mk_SeqStmt("nil", mk_ParStmt(g,d)), sigma_1) -s-> (mk_ParStmt(g,d), sigma_1) by par.sos_seq_E ()
6 (mk_ParStmt(g,d), sigma_1) -s->* (mk_ParStmt("nil", d2), sigma_n) or
    (mk_ParStmt(g, d), sigma_1) -s->* (mk_ParStmt(g2, "nil"), sigma_n) by IJ
7 from (mk_ParStmt(g,d), sigma_1) -s->* (mk_ParStmt("nil", d2), sigma_n)
7.1 (g, sigma_1) -s->* ("nil", sigma_n) by IJ (7.h1)
```

```
7.2 r::sigma_n != 0 by par.Lemma5 (7.1)
7.3 (ifg, sigma_m) -s->* ("nil", sigma_n) by IJ (7.1)
7.4 (r::sigma_m != r::sigma_n) => a::sigma_n(r::sigma_n) = 5 by IJ
7.5 r::sigma_m = 0 by IJ (2)
7.6 r::sigma_m != r ::sigma_n by IJ (7.2, 7.5)
7.7 a::sigma_n(r::sigma_n) = 5 by prop.implEL (7.4, 7.6)
7.8 (mk_ParStmt("nil",d2), sigma_n) -s->* (mk_ParStmt("nil", "nil"), sigma_p) by IJ
7.9 (mk_ParStmt("nil", "nil"), sigma_p) -s-> ("nil", sigma_p) by par.sos_par_E ()
7.10 sigma_p = sigma_w by IJ (7.9, h1)
  infer a::sigma_w(r)  = 5 by IJ (7.7, 7.10)
8 from (mk_ParStmt(g, d), sigma-1) -s->* (mk_ParStmt(g2, "nil"), sigma_n)
  infer a::sigma_w(r) = 5 by IJ
infer a::sigma_w(r) = 5 by prop.orE (6, 7, 8)
;
```

Appendix F

Proofs and Judgements

In the model of the proof system, a line of a proof is represented as a *Judgement*. Figure F.2 takes the proof of *and distributes over or* and shows each *Judgement* of the proof, and the values of each field of the judgements. The proof of *and distributes over or* is given in Fig. F.1

	from $a \wedge (b \vee c)$	
1	a	$\wedge$-E-Right(h1)
2	$b \vee c$	$\wedge$-E-Left(h1)
3	**from** b	
3.1	$a \wedge b$	$\wedge$-I(1, 3.h1)
	infer $(a \wedge b) \vee (a \wedge c)$	$\vee$-I-Right(3.1)
4	**from** c	
4.1	$a \wedge c$	$\wedge$-I(1, 4.h1)
	infer $(a \wedge b) \vee (a \wedge c)$	$\vee$-I-Left(4.1)
	infer $(a \wedge b) \vee (a \wedge c)$	$\vee$-E(2,3,4)

Figure F.1: Proof for $\wedge$ distributes over $\vee$

	Hyps	***Conclusion***	***Justification***	***Parent***	***subordinates***	***name***
h1	[ ]	“a ∧ (b ∨ c)”	Hyp	conc	{}	“h1”
1	[ ]	“a”,	∧-E-R(h1)	conc	{}	“1”
2	[ ]	“b ∨ c”	∨-E-L(h1)	conc	{}	‘2”
3.h1	[ ]	“b”	Hyp	3.conc	{}	“3.h1”
3.1	[ ]	“a ∧ b”	∧-I(1, 3.h)	3.conc	{ }	“3.1”
3.conc	[3.h1]	“(a ∧ b) ∨ (a ∧ c)”	∨-I(3.1),	conc	{3.h1, 3.1}	“3.conc”
4.h1	[ ]	“c”,	Hyp	4.conc	{}	“4.h1”
4.1	[ ]	“a ∧ b”	∧-I(1, 4.h1)	4.conc	{}	“4.1”
4.conc	[4.h1]	“(a ∧ b) ∨ (a ∧ c)”	∨-I(4.1)	conc	{4.h1, 4.1}	“4.conc”
conc	[h1]	“(a ∧ b) ∨ (a ∧ c)”	∨-E(2, 3.conc, 4.conc)	**nil**	{h1, 1, 2, 3.conc, 4.conc}	“conc”

Figure F.2: Judgements of the proof of ∧ distributes over ∨