Machine Assisted Reasoning for Multi-Threaded Java Bytecode

MIKAEL LAGERKVIST

Supervisor: Mads Dam

IMIT/LECS-2005-70

Mathematicians are a species of Frenchmen: if you say something to them they translate it into their own language, and presto! It is something entirely different.

Maxims and Reflections
Johann Wolfgang von Goethe

Abstract

In this thesis an operational semantics for a subset of the Java Virtual Machine (JVM) is developed and presented. The subset contains standard operations such as control flow, computation, and memory management. In addition, the subset contains a treatment of parallel threads of execution.

The operational semantics are embedded into a μ-calculus based proof assistant, called the VeriCode Proof Tool (VCPT). VCPT has been developed at the Swedish Institute of Computer Science (SICS), and has powerful features for proving inductive assertions.

Some examples of proving properties of programs using the embedding are presented.

Datorstödda resonemang om multi-trådad Java-bytekod

Examensarbete

Sammanfattning

I det här examensarbetet presenteras en operationell semantik för en delmängd av Javas virtuella maskin. Den delmängd som hanteras innehåller kontrollflöde, beräkningar och minneshantering. Vidare beskrivs semantiken för parallella exekveringstrådar.

Den operationella semantiken formaliseras i en bevisassistent for μ-kalkyl, VeriCode Proof Tool (VCPT). VCPT har utvecklats vid Swedish Instituite of Computer Science (SICS), och har kraftfulla tekniker för att bevisa induktiva påståenden.

Några exempel på bevis av egenskaper hos program användandes formaliseringen presenteras också.

Preface

This is a masters thesis from the Master of Science and Engineering in Computer Science and Engineering programme at the Royal Institute of Technology (KTH) in Stockholm, Sweden. The supervisor of the thesis is Associate Professor Mads Dam from the Department of Microelectronics and Information Technology at KTH.

The work has been performed at the Swedish Institute of Computer Science (SICS) in Stockholm. At SICS, Lars-Åke Fredlund has acted as advisor to the work.

Acknowledgements I would like to thank my supervisor Mads Dam for constant support and feedback during my work. Also, I would like to thank Lars-Åke Fredlund for answering all my questions, both large and small.

Contents

List of Tables

Chapter 1

Introduction

In the modern society, computers are a common part of our lives. They are everywhere and we trust them to perform many important tasks, even life-critical tasks. We have computers transferring our money, controlling medical equipment and regulating traffic. These computer systems must function correctly, even the smallest error may have devastating effects.

A prominent example of the risks connected to incorrectly functioning software is the explosion of the Ariane 5 launcher on the 4th of June 1996, approximately 37 seconds after lift-off. According to the press release, the explosion was due to "specification and design errors in the software" in the programs on the on-board computer. In the report from the official Inquiry Board, it is reported that the error occurred in a software module handling sensor data. The module was originally constructed for the Ariane 4, and had been imported into the Ariane 5 system essentially unchanged. The problem was that the module had been designed for execution up until lift-off, but had erroneously been active after lift-off. The assumptions made by the module about the sensor data it received was not valid, and the module crashed. There were back-up systems to handle malfunctioning modules. However, these were constructed to handle random hardware errors, not systematic software failures.

When dealing with safety-critical systems such as the on-board computer of Ariane 5, it is not enough to construct software that functions correctly 95% of the time, it must function correctly all the time. This goal is ambitious, but necessary. To achieve this goal, the Inquiry Board concluded the following.

> "...software should be considered to be faulty until[,] applying the currently accepted best practice methods[, we] can demonstrate that it is correct"

But what are these "best practice methods"? What kind of methods do we have that can demonstrate that a piece of software is correct? On a more fundamental level, what do we mean when we state that the software is correct?

Formal methods The area known as Formal Methods is concerned with applying mathematical tools to software and systems. This is often done by modelling software as mathematical objects, and showing properties about these objects. From the mathematical basis we get a precise interpretation of our models, accompanied by methods and techniques for handling them. The problematic statement "the software is correct" from the previous paragraph is, in this context, the question whether our model of the software has some property encoding its specification.

There are many variants and styles that conform to this basic scheme. The model may be a large and detailed one, or it may be a small one focusing on the core properties. It may be close to the system at hand (or even the actual system), or expressed in a specialised separate language. The actual verification may be manual or automatic. The properties that we can express as our specifications may be simple or complex. These variations all contribute to a very diverse field, with many different approaches to the problem. This diversity is a good thing; one way of handling things may be appropriate for one system and property, but not for another combination.

Why then did the constructors of Ariane 5 not employ formal methods, or for that matter, why does not everybody constructing software apply them? The simple answer is that formal methods are not easy or simple; verifying that a program has an interesting property is intrinsically hard. On the other hand, it is relatively cheap to construct a program that almost has the property we are interested in. This latter approach is the principal way in which programs are constructed in most of the industry. This, naturally, has led to it being the norm for all of the industry, even when formal methods are both feasible and necessary.

Java Java is a modern object-oriented, garbage-collected, multi-threaded, distributed, portable, interpreted programming language[1]. Java is used widely as a programming language for development in both business and research. Examples of areas where Java is a common language are internet applications, programs for mobile phones, and server-side business data processing. There is also a specification for a subset of Java tailored towards smart-cards, called JavaCard.

A large part of Java's success owes to the portability between different machines and operating systems. The portability of Java is based on the idea of a virtual machine interpreting a simple intermediate representation of the program, called byte-code. This machine is named the Java Virtual Machine (JVM). To achieve portability, the semantics of the byte-code is specified so that different implementations of the JVM can exist for different environments.

The byte-code of the JVM has simple stack-based computational instructions and (relatively) complex control-instructions encoding the object-model of Java. This ensures that a byte-code program, even though it is simple, does contain a direct representation of the high-level logical structure of a program.

[1] The Java language is in essence a buzzword-compliant language.

Studying the JVM is interesting since it is a relatively well-specified platform, it is very popular, and it is independent of the machine and operating system chosen for an application.

The VeriCode Proof Tool The Code Verification Group at the Swedish Institute of Computer Science (SICS) has developed a proof-assistant for first-order logic augmented with explicit fixpoint formulae (that is, recursive formulae) called the VeriCode Proof Tool (VCPT)[VCP]. This tool has support for advanced inductive and compositional reasoning, automation via tactics and functions, and representing transition relations.

VCPT stems from the Erlang Verification Tool (EVT)[ACD+03, Fre01]. EVT is a verification framework tailored specifically towards the Erlang programming language, with a direct embedding of the semantics. The proof-assistant part of the tool was then repackaged as a stand-alone program, with support for defining an arbitrary operational semantics.

The main features of VCPT that makes it interesting are the powerful mechanisms for specifying properties, the support for lazy discovery of induction schemas and the support for compositional reasoning.

VCPT was used by Barthe, Gurov, and Huismann in [BGH02] to model the call graphs of JavaCard programs. No further substantial work employing VCPT has been done.

1.1 This Thesis

This thesis is concerned with the formalisation of a subset of the Java Virtual Machine's semantics. This formalisation is embedded into VCPT, enabling reasoning about compiled Java-programs at the byte-code level. Given a program and a specification of a property, we can express these in VCPT and prove (or disprove) that the program has the property.

The usefulness of such a formalisation is manifold. The obvious use is to prove properties of programs, using the actual program as the model. We may also use the formalisation of the semantics as a means for discussing the semantics of the JVM, where the mechanisation aids as a tool for executing programs. An additional benefit of the work is that we get an evaluation of the suitability of the VCPT platform for describing an operational semantics.

The formalisation (or an extended one handling more of the JVM) could also be used as a platform for building tools for reasoning about programs and for increasing the safety of applications. One example is the concept of Proof Carrying Code (PCC) introduced by Necula and Lee in [NL96], where mobile code is accompanied by a proof of safety. One of the basic requirements for such a scheme is a proof-system for the language at hand, which is what is developed in this thesis.

1.1.1 Aim

The aim of this thesis is to produce a formalisation of a representative subset of the Java Virtual Machine as an operational semantics. The subset shall contain essential features such as control-flow, some data-handling, and parallel threads of execution.

The developed semantics shall be mechanised in the VeriCode Proof Tool to enable formal reasoning about programs conforming to the subset.

Some examples shall be presented to show the formalisation in action.

1.1.2 Scope

In this thesis we will restrict our attention to the core formalisation of the chosen subset. There is much work to do in producing an environment suitable for working with the formalisation, including tools such as pretty-printers and parsers, as well as lemma-libraries and advanced tactics. In the environment of this thesis, the transformation from a Java-program into a format suitable for the proof-tool is done by hand, the presentation of the system state in the tool is in the internal representation, and no general reusable lemmas or results have been produced.

We will further restrict our attention to systems without class-hierarchies, dynamic loading of classes, exceptions, and distributed computing. These restrictions are not necessary ones, in that the formalisation in no discernible way excludes the addition of such features. The problem lies instead with the time-constraints of the thesis.

1.1.3 Contributions

The main contributions of this thesis are the following.

- A formal operational semantics of a subset of the multi-threaded JVM, presented in a format that is common to most operational semantics.

- A mechanisation of the semantics in a proof assistant, supporting both exploration of the semantics as well as proofs of properties of actual programs.

- An evaluation of the suitability of the VeriCode Proof Tool as a tool for mechanising operational semantics.

1.1.4 Overview

This thesis starts off with a presentation of the relevant background information in Chapter 2. Topics discussed are semantics, the logic and tool used, and other possible approaches to both formal methods and to formalising Java, the JVM, or both. We continue in Chapter 3 with the first steps of development, namely some formulae and tactics for handling lists and natural numbers.

In Chapter 4 the Java Virtual Machine is described; the style of instructions, the model of concurrency, and the structure of the data. With a clear picture of the JVM, we can give the semantics for it, which is done in Chapter 5. The semantics are embedded into, and mechanised in, VCPT. This embedding is described in Chapter 6.

To illustrate our model of the JVM and the embedding of it in VCPT, some small examples of proving properties of programs are presented in Chapter 7. The work performed is evaluated and conclusions are drawn in Chapter 8. Directions for further work and possible extensions are also given.

In Appendix A some work relating to improving the user interface of the VeriCode Proof Tool is reported on. Appendix B contains a list of basic formulae for manipulating lists and their definitions.

Chapter 2

Background

This chapter presents background material necessary for this thesis. It describes a way of giving semantics to programs (the operational semantics in Section 2.1), the logic that is used as the formal basis for encoding the semantics and properties of programs (the μ-calculus in Section 2.2), and the tool used for mechanising the workings of the encoded semantics (the VeriCode Proof Tool in Section 2.3). There is also a discussion on related work, such as other efforts at formalising Java and the JVM, as well as other approaches and tools for formal methods in general (related work in Section 2.4).

2.1 Operational semantics

Operational semantics are mainly used to describe the semantics of programming languages. There are different flavours of operational semantics, where the common theme is to describe the meaning of a program via a symbolic execution of it. For introductory material on operational semantics, see [Plo81] or [Win93]; for a detailed formal description, see [AFV01]. The material in this section is based on the latter reference.

Operational semantics uses abstract states for the programs, and on these states, a transition-relation that describes the possible executions is defined. To describe this transition-relation in a succinct format, we often employ a rule-based format with the premises for the applicability of a rule above the line and the conclusion drawn below the line.

There are two major variants of operational semantics, often referred to as big-step—or natural—and small-step semantics. In big-step semantics, the meaning of a program is defined in terms of its transition from a program into its result when it terminates. This method is useful when we are dealing with terminating systems. In a small-step semantics we define the transition in terms of small operations, and the result it has on the state. This style is useful, for example, for reactive systems and for specifying interleaving semantics of parallel programs. In this thesis we will concentrate on the small-step variant, since we will describe an interleaving semantics for a parallel system.

2.1.1 Formal definition

A small-step operational semantics for a class of systems relates the states of the systems to their possible next states via labelled transitions. In the following section we define such transition-relations and how to specify them. The definition of operational semantics uses many-sorted signatures and valuations, which are both described in Section 2.2.1.

Definition 2.1.1 (LTS) *A* Labelled Transition System (LTS) *is a tuple* $\langle S, A, \rightarrow\rangle$*, where* S *is the set of possible states,* A *is the set of actions or labels, and* $\rightarrow \subseteq S \times A \times S$ *is a relation from states to states via actions.*

When a system in state s makes a transition into the state s' with the label α, we write $s \overset{\alpha}{\rightarrow} s'$ instead of $(s, \alpha, s') \in \rightarrow$. We adopt the convention that a transition $s \rightarrow s'$ is an abbreviation for a labelled transition with the special label τ, where τ represents the "silent" action.

To specify an LTS, we use a Transition System Specification (TSS), consisting of proof rules specifying the transitions that can be proved.

Definition 2.1.2 (Proof rule) *A* proof rule *is a triple* $\langle \textsc{Name}, C, \{P_1, \ldots, P_n\}\rangle.$ *, where* Name *is a name for the rule,* C *is the conclusion and the* P_i *form a set of premises. We write a proof rule graphically as*

$$\textsc{Name}\ \frac{P_1 \qquad \cdots \qquad P_n}{C}$$

If the set of premises is empty, we call the proof rule an axiom. The graphical representation of such a rule uses a single dot above the line to indicate the absence of premises. Sometimes there is some additional constraint associated with the rule. Such a constraint is written to the right of the line.

Definition 2.1.3 (TSS) *A* Transition System Specification (TSS) *over a signature* Σ*, with states from the sort* S *and actions from the sort* A*, consists of a set of proof-rules. Let* Q *be some truth valued predicate in an associated logic and let* x_{ij} *be from* $T(\Sigma, V)$*. The conclusions are transitions* $t \overset{\alpha}{\rightarrow} t'$*, and the premises are either of the form* $t_i \overset{\alpha_i}{\rightarrow} t'_i$ *(a transition-premise) or of the form* $Q_i\ x_{i1}\ \ldots\ x_{ik}$ *(a predicate-premise).*

Definition 2.1.4 (Proof of transition) *A proof by a* TSS *is a tree* T *with nodes labelled by either transitions and a name or a predicate, where the root has a transition-name label, and the following properties hold.*

- *For each node labelled by a predicate, it is a leaf and the predicate is true in the associated logic.*
- *For each node labelled by a transition $t \xrightarrow{\alpha} t'$ and a name N, there is a rule $\langle N, t_v \xrightarrow{\alpha} t'_v, \{P_1, \ldots, P_n\}\rangle$ in the TSS and a valuation $\rho : V \to T(\Sigma)$, such that $t_v\rho = t$, $t'_v\rho = t'$, and all P_i match under ρ with one child of the current node.*

The tree T is said to prove the transition in its root.

The meaning we associate with a TSS is an LTS, such that a transition $t \xrightarrow{\alpha} t'$ is included in the LTS if, and only if, there is a proof of the transition.

2.1.2 Operational semantics of concurrency

Operational semantics are common in process algebra, where concurrency must be specified. There are different ways to handle concurrency using operational semantics. Which approach to use depends on the goals of the semantics. A prominent distinction is between the specifications that introduce a total order of the actions, and those that only introduce a partial order.

In this thesis we will use an interleaving semantics for the threads, which means that the parallel execution of a number of threads will correspond to some interleaving of executions of atomic actions from the individual threads. Such a specification will introduce a total ordering of the actions of the system. The main assumption for such a specification to be correct is that the individual interleaved actions are in fact atomic, and that truly concurrent execution will not alter the possible outcomes.

Example 2.1.5 *A common way to specify parallel processes that exhibit an interleaving semantics in process algebra is the following. Assume that we have a TSS for non-parallel processes. We add an infix parallel composition operator $\|$. Let P and Q be two processes, then the addition of following two rules to the TSS will give an interleaving semantics for the parallel execution of processes.*

$$\text{Par}_l \; \frac{P \xrightarrow{\alpha} P'}{P \parallel Q \xrightarrow{\alpha} P' \parallel Q} \qquad\qquad \text{Par}_r \; \frac{Q \xrightarrow{\alpha} Q'}{P \parallel Q \xrightarrow{\alpha} P \parallel Q'}$$

$$\text{Comm} \; \frac{P \xrightarrow{\alpha} P' \qquad Q \xrightarrow{\overline{\alpha}} Q'}{P \parallel Q \to P' \parallel Q'} \; \alpha \text{ and } \overline{\alpha} \text{ complementary.}$$

The rules Par_l and Par_r allow either component of a parallel composition to compute. The rule Comm is for communication between processes. The rule has an additional condition, specifying that the labels α and $\overline{\alpha}$ are some sort of complementary actions.

It is important to note that the result of composing two processes results in a new process. This means that we can specify the parallel composition of the processes

Function	Type	Function	Type
0	nat	$n_\emptyset$	nat_l
s	$nat \rightarrow nat$	n	$nat \times nat_l \rightarrow nat_l$
pop	$inst$	$i_\emptyset$	$inst_l$
$push$	$nat \rightarrow inst$	i	$inst \times inst_l \rightarrow inst_l$
$goto_b$	$nat \rightarrow inst$	frm	$inst_l \times inst_l \times nat_l \rightarrow frame$
$gimp_b$	$nat \rightarrow inst$	τ	$action$

Table 2.1: Type annotations for the signature of the stack language fragment.

P, Q, and R as either $P \parallel (Q \parallel R)$ or $(P \parallel Q) \parallel R$. Note that, if the above rules are the only ones involving the parallel composition, then the two compositions are intuitively equivalent (the $\parallel$-operator is associative)

2.1.3 Example: A Fragment of a Stack Language

In this section, we will give an example of an operational semantics for a fragment of a stack-based language. We model the instructions $goto_b$, $push$, and pop. The intuitive meaning is that $goto_b$ jumps backwards in the instructions, $push$ pushes a value onto the stack, and pop pops a value of the stack. In a more complete language, instructions such as computation, forward jumps, a store, and conditionals would have to be added.

The signature for the data in this example is

$$\Sigma = \left\langle \begin{array}{l} \{inst, inst_l, nat, nat_l, frame, action\}, \\ \{0, s, goto_b, gimp_b, push, pop, n_\emptyset, n, i, i_\emptyset, frm, \tau\} \end{array} \right\rangle$$

The type annotations for the function symbols are shown in Table 2.1.

The intuitive meaning for the sorts and the function-symbols in the signature are as follows. A nat is a natural number, which is either 0 or a successor to a nat. A nat_l is a list of naturals and a $inst_l$ is a list of instructions. Both lists are either the constant for the empty list, or an element of data followed by a list. A $frame$ is built by two instruction lists and a list of naturals. The top element of the second list is the next instruction to execute. The extra instruction $gimp_b$ is used when moving backwards in the list of instructions. The name can be read as Goto IMPlementation.

The TSS that defines the operational semantics for this language is given in Table 2.2.

As a sample derivation, we will show the execution of one step of the program that pushes a value on the stack, pops it off, and then loops back. We will show the derivation of the jump backwards. The term that represents this program is the following.

$$frm(i(pop, i(push(0), i_\emptyset)), i(goto_b(s(s(0))), i_\emptyset), n_\emptyset)$$

$$\text{Push}\,\frac{\cdot}{frm(C_1, i(push(N), C_2), V) \to frm(i(push(N), C_1), C_2, n(N, V))}$$

$$\text{Pop}\,\frac{\cdot}{frm(C_1, i(pop, C_2), n(N, V)) \to frm(i(pop, C_1), C_2, V)}$$

$$\text{Goto}\,\frac{frm(C_1, i(gimp_b(N), i(goto_b(N), C_2), V) \to F}{frm(C_1, i(goto_b(N), C_2), V) \to F}$$

$$\text{Gimp}_N\,\frac{frm(i(inst, C_1), i(gimp_b(N), C_2), V) \to F}{frm(C_1, i(gimp_b(s(N)), i(inst, C_2)), V) \to F}$$

$$\text{Gimp}_0\,\frac{\cdot}{frm(C_1, i(gimp_b(0), C_2), V) \to frm(C_1, C_2, V)}$$

Table 2.2: TSS for the stack language fragment.

$$\cfrac{\cfrac{\cfrac{\cfrac{\cfrac{\cdot}{\begin{array}{c}frm(i_\emptyset, i(gimp_b(0), i(pop, i(push(0), i(goto_b(s(s(0))), i_\emptyset)))), n_\emptyset) \to\\ frm(i_\emptyset, i(pop, i(push(0), i(goto_b(s(s(0))), i_\emptyset))), n_\emptyset)\end{array}}\ \text{Gimp}_0}{\begin{array}{c}frm(i_\emptyset, i(gimp_b(0), i(pop, i(push(0), i(goto_b(s(s(0))), i_\emptyset)))), n_\emptyset) \to\\ frm(i_\emptyset, i(pop, i(push(0), i(goto_b(s(s(0))), i_\emptyset))), n_\emptyset)\end{array}}\ \text{Gimp}_N}{\begin{array}{c}frm(i(push(0), i_\emptyset), i(gimp_b(s(0)), i(pop, i(goto_b(s(s(0))), i_\emptyset))), n_\emptyset) \to\\ frm(i_\emptyset, i(pop, i(push(0), i(goto_b(s(s(0))), i_\emptyset))), n_\emptyset)\end{array}}\ \text{Gimp}_N}{\begin{array}{c}frm(i(pop, i(push(0), i_\emptyset), i(gimp_b(s(s(0))), i(goto_b(s(s(0))), i_\emptyset)), n_\emptyset) \to\\ frm(i_\emptyset, i(pop, i(push(0), i(goto_b(s(s(0))), i_\emptyset))), n_\emptyset)\end{array}}\ \text{Gimp}_N}{\begin{array}{c}frm(i(pop, i(push(0), i_\emptyset), i(goto_b(s(s(0))), i_\emptyset), n_\emptyset) \to\\ frm(i_\emptyset, i(pop, i(push(0), i(goto_b(s(s(0))), i_\emptyset))), n_\emptyset)\end{array}}\ \text{Goto}$$

Figure 2.1: The derivation of a jump backwards in the stack language fragment.

In Figure 2.1 the derivation-tree for the jump backwards is shown. We will describe the derivation as it may be constructed, not in terms of the validity according to the definition of proofs.

The derivations starts with applying the rule Goto, which introduces a $gimp_b$ instruction into the code stream, with the same argument as the current *goto* instruction. Using the rule Gimp_N repeatedly on this new state, we can move instructions from the right code-list to the left code-list, simultaneously decrementing the argument of $gimp_b$. For example, in the first application of Gimp_N, we move the *pop* instruction. When the argument to the $gimp_b$ instructions is zero, we remove it with the axiom-rule Gimp_0. The result of this transition, the frame without the $gimp_b$ instruction, is then propagated down through the derivation-tree.

2.2 The μ-calculus

VCPT, and thus this thesis, uses the μ-calculus with Gentzen style judgments. The μ-calculus is a first-order logic extended with fixed points of recursive formulae. This enables the description of infinite behaviours and infinite systems, which is very useful for handling reactive systems.

For a general introduction to the μ-calculus, see Stirling and Bradfield[BS01]. For a more in-depth account, see Stirling[Sti92]. The material in this chapter is based on the definitions in [Fre01], except where noted.

First we describe the signatures we use and some additional common concepts from logic, after which we give a full definition of the logic and the proof-system used.

2.2.1 Many-sorted First-order Structures

In the logic we need a systematic way to describe what kinds of data we have. In the framework of VCPT, elements of data are constructed as the terms of a many-sorted first-order structure. Constructing terms is done by applying a function-symbol to some other terms of the correct sort and number. The possible data are the terms that can be built from such applications.

For more detailed information about signatures and structures in a many-sorted context, see [MT92].

Definition 2.2.1 (Many-sorted First-order Structure) *A* Many-sorted First-order Structure *is given by a tuple $\Sigma = \langle S, F \rangle$ where S is a set of sorts and F is a set of function-symbols. For each f in F there is also an annotation for the type of the function $s_1 \times s_2 \times \cdots \times s_n \to s$, with $s_1, s_2, \ldots, s_n, s$ from S. The s_i, for $1 \leq i \leq n$, are the sorts of the arguments, and s is the sort of the result.*

Given a structure $\langle S, F \rangle$, we write $f(t_1, t_2, \ldots, t_n)$ for the application of f to the sub-terms $t_1, t_2 \ldots, t_n$ (all of the appropriate sort), which in turn yields a term of sort s. If the number of arguments for a function c is zero (i.e. it has the annotation s for some s), we call it a constant and write c instead of $c()$.

Definition 2.2.2 ($T(\Sigma, V)$) *The set $T(\Sigma, V)$ is the set of all terms in the signature Σ with variables from the set V, the variables themselves also annotated with sorts. We write $T(\Sigma, V)_s$ for the set $\{x \mid x \in T(\Sigma, V),\ x \text{ of sort } s\}$, i.e. the set of all terms of sort s. When V is empty, we omit the variables and are left with all the ground terms, designated $T(\Sigma)$. $T(\Sigma, V)$ is defined inductively using the following two rules.*

- *$v \in T(\Sigma, V)$ iff $v \in V$.*
- *$f(x_1, x_2, \ldots, x_n) \in T(\Sigma, V)_s$ for f with annotation $s_1 \times s_2 \times \cdots \times s_n \to s$ iff for all i from 1 to n, $x_i \in T(\Sigma, V)_{s_i}$.*

Example 2.2.3 *Suppose we are interested in sequences of natural numbers. We will give a many-sorted first-order structure for the two sorts nat and seq, representing natural numbers and sequences of natural numbers.*

A natural number nat is either the constant zero, given by the function zero, or it is the successor of another natural number, given by the function succ. A sequence seq is either the empty sequence (nil), or it is a natural number followed by a sequence (cons).

Our signature Σ *is thus* $\langle\{nat, seq\}, \{zero, succ, nil, cons\}\rangle$*, with the annotations*

Function	Type
zero	*nat*
succ	$nat \rightarrow nat$
nil	*seq*
cons	$nat \times seq \rightarrow seq$

A sequence $[1, 2, 0]$ *would, with the above signature, be given as*

$$cons(succ(zero), cons(succ(succ(zero)), cons(zero, nil)))$$

In the above example we have assumed that all terms that are syntactically different are also semantically different. Terms of sorts that have this property are called freely generated, or flat. Sometimes we want to equate terms that have different syntactic constructions. This is not handled directly in a structure, and thus must be handled in the logic.

2.2.2 Some Common Concepts from Logic

In the following sections some different concepts from logic that are used in this thesis are described and defined.

Substitutions and Valuations

A substitution is an operation that replaces all free occurrences of one symbol with some other symbol in a formula or term. Since we have many sorts for our terms and variables, all valid substitutions must preserve sorts. Assuming that we want to substitute X with Y in ϕ, we write $\phi[Y/X]$, read as "phi with Y for X". As a typical example, $(0 = X)[0/X]$ is a true statement.

A valuation is a mapping from the free variables to terms. In other words, a valuation ρ is a mapping $\rho : V \rightarrow T(\Sigma)$. We require that the valuation respects the sorting, i.e., a variable v of sort s will be mapped to an element of $T(\Sigma)_s$. Given a valuation ρ, the notation $\phi\rho$, read as "phi under rho", means that we take all the free variables in ϕ and substitute their occurrences with the associated value in ρ. For any ρ, the new expression $\rho[Y/X]$ is the valuation that agrees with ρ on all points, except possibly for X, where we have $\rho(X) = Y$.

We will use the identity-valuation *id*, which takes all elements to themselves.

Gentzen Systems

A Gentzen-style logical system is a system where we have sequents of the form $\Gamma \vdash \Delta$, where Γ and Δ are sets of formulae. The focus of Gentzen-style systems is the proof — the main objective is not to discover tautologies, but to prove sequents.

The formulae may have free variables in them, which means that we need a valuation from the variables to their value to evaluate such a judgment. Formally, a judgment is said to be valid if, and only if, for all valuations of the variables such that all formulae in Γ are true, then so is at least one formula in Δ. A judgment $\Gamma \vdash \Delta$ may be viewed as the conjunction of all formulae in Γ implying the disjunction of all the formulae in Δ under all valuations. Loosely, this could be coded as $\bigwedge \Gamma \Rightarrow \bigvee \Delta$, although we would lose the practical proof-properties.

We usually refer to the antecedent Γ as the the left-hand side, or as the assumptions, since those should be true. We refer to the succedent Δ simply as the right-hand side.

Gentzen systems are useful when we want to prove properties about programs. One possibility is to "store" assumptions about our world in the left-hand side, and have them available when we need them in our proofs. We can, of course, encode the same properties in a regular formula (as demonstrated earlier). The problem with the encoding approach, is that proofs and formulae become unwieldy very quickly. Another useful property of Gentzen systems is that they are particularly well suited to computer assisted reasoning[RS92].

Fixed Points

A very general view of a fixed point for a function f is that of a value d, such that $f(d) = d$. This formulation applies to all functions that have the same range as it has domain.

For monotone functions on complete lattices, there is a theorem by Knaster and Tarski[Win93] that guarantees the existence of both the least and the greatest fixed point, and that we can find them by repeated application of the function. The theorem states the following. Assume that we have a complete lattice $(L, \leq)$ and a function $f : L \to L$ with the property that for all $x, y \in L$, the ordering $x \leq y$ implies that $f(x) \leq f(y)$. Then the set of all fixed points of the function f form a complete lattice. A result of this theorem is the guaranteed existence of a least and a greatest fixed point for f, since every complete lattice has a least upper and a greatest lower bound.

A prominent, and for this thesis the most important, example of a complete lattice is the boolean lattice. A recursive predicate on this lattice is monotone if each occurrence of the predicate in the definition is under an even number of negations.

Modal Logics

In the standard first-order logic there is quantification over individuals (that is, over data). This can be seen as a rather static view of the world, where everything is fixed. In a modal logic, on the other hand, we can quantify over possibilities, such as possible worlds or possible next states. This is a more dynamic view of the world, where we acknowledge the fact that things may change in the future. The modal operators do not add any power to our language; they can be encoded in the standard first-order logic (as is done in the logic used in this thesis for example). The real motivation for us in adding modalities is to gain expressiveness, facilitating the construction of interesting properties.

The standard modal operators are the box ($\Box$) and the diamond ($\Diamond$). These operators are added to the logic, such that for any formula ϕ, the expressions $\Box\phi$ and $\Diamond\phi$ are also formulae. Loosely speaking, we say that the box stands for always and the diamond for sometimes. There are many variants, including "always in the future" and "always in the next state".

When dealing with programs, it is often useful to use multi-modal logics. In a multi-modal logic, we can have $[\alpha]\phi$ and $\langle\alpha\rangle\phi$ as formulae. Assuming that we have the state-based view of modalities, the formula $[\alpha]\phi$ means that ϕ is valid in all next states reachable via an α-action. If the formulae are first-order logic formulae, this logic is called Hennessy-Milner logic (HML), and was introduced in [HM80]. It is not a very useful logic in itself (e.g., it can not talk about infinite runs of programs), but it is a standard basis for many other branching time logics.

2.2.3 Syntax and Semantics

Given the information in the previous sections, we are now equipped to describe the actual logic used in VCPT. The logic is essentially the μ-calculus with approximated fixpoints of Kozen[Koz82], but where the modalities are encoded instead of primitive.

The logic is a first-order propositional logic with explicit fixpoint formulae defined by recursive predicates. We can specify both least and greatest fixed points. Informally, we say that least fixed points specify liveness properties (something must *eventually* happen) and that greatest fixed points specify safety properties (something must *continue* to hold). For more discussion about the intuitive meaning of μ-calculus formulae, see [BS01].

We will assume the following. A formula in the logic is to be interpreted over some signature $\Sigma = \langle S, F \rangle$, with terms from $T(\Sigma, V)$. We will let ϕ, ψ range over the possible formulae, x, y, z over the term variables in V, t over the terms in $T(\Sigma, V)$, X, Y, Z over the formula variables, κ over the ordinal variables, and s over the sorts in S.

ϕ	$:==$	X	predicate variable
	$\mid$	$t_1 = t_2$	term equality
	$\mid$	$\neg\phi$	negation
	$\mid$	$\phi \vee \psi$	disjunction
	$\mid$	$\exists x : s.\phi$	existential quantification
	$\mid$	$\lambda x : s.\phi$	lambda-abstraction
	$\mid$	$\phi\ t$	application
	$\mid$	$\mu X : type.\phi$	least fixed point
	$\mid$	$(\mu X : type.\phi)^\kappa$	approximated least fixed point
	$\mid$	$\kappa < \kappa'$	ordinal in-equation

Table 2.3: Syntax of the basic logic.

Syntax of the Logic

The basic logic has the usual constructs that we expect, such as disjunction, negation, lambda-abstractions, and first-order quantification. In addition, it has a recursive construction specifying least fixed points. To every formula ϕ we assign a type; the type of a formula is either `prop` or it is $s \rightarrow$ `type` for some sort s.

The base syntax of formulae is presented in Table 2.3. We restrict the construction $\mu U : type.\phi$ by requiring that every occurrence of the bound predicate-variable U must occur in ϕ under an even number of negations. This condition ensures that the semantics of ϕ is monotonic, thus ensuring that the least fixed point for the formula exists by the Knaster-Tarski theorem (Section 2.2.2).

Some examples of syntactically valid formulae are

$$\exists x : nat.0 = x \vee 1 = x$$

and

$$\mu U : nat \rightarrow prop.\lambda x : nat.x = 0 \vee \exists y : nat.\neg(\neg(U\ y) \vee \neg(x = y + 2)),$$

assuming that we have natural numbers as terms in the logic. As a side-note, the second formulae is true for all even natural numbers. The formulae $1{=}2\vee$ and $\mu U : prop.\neg U$ on the other hand, are not valid. The first since it can not be derived from the rules in Table 2.3, the second since the occurrence of U in the formula is under an odd number of negations.

The basic syntax is not enough to conveniently give logical specifications. We also need to have truth-variables, conjunction, implication and greatest fixed points, as well as some other conveniences. These extensions can all be found in Table 2.4.

Types in the logic

A type for a formula designates how many arguments may be applied to it, and what sorts these arguments take. As an example, the formula $\lambda x : nat.\lambda y.nat.x = y$

$\phi \wedge \psi$	$\triangleq$	$\neg(\neg\phi \vee \neg\psi)$	conjunction
$\phi_1 \Rightarrow \phi_2$	$\triangleq$	$\neg\phi_1 \vee \phi_2$	implication
$\forall x : s.\phi$	$\triangleq$	$\neg\exists x : s.\neg\phi$	universal quantification
$\nu X : \mathit{type}.\phi$	$\triangleq$	$\neg\mu X : \mathit{type}.\neg(\phi[\neg X/X])$	greatest fixed point
$\top$	$\triangleq$	$\neg\bot$	true
$\bot$	$\triangleq$	$\mu X : \mathit{prop}.X$	false
$t : \phi$	$\triangleq$	$\phi\ t$	satisfaction

Table 2.4: Extensions to the base logic.

has the type $nat \rightarrow nat \rightarrow \texttt{prop}$. For a more detailed account of the type-system of the logic, see [Fre01].

In the following we will assume that a formula is well-typed. In such a formula equality-checks between terms have the same sort on both sides and applications of a term of sort s to a formula of type $s' \rightarrow \texttt{type}$ is only done when $s = s'$. An example of a formula that is not well typed, and thus not correct, is the formula $\exists x : s_1.\exists y : s_2.x = y$, if the sorts s_1 and s_2 are different.

Semantics of the logic

We will now present the semantics of the logic, i.e., what each formula constructed from the syntactical rules means. The semantics are given in a denotational style, relating the syntactical formulae to either the empty set (denoting falsity), or to the singleton set containing the constant tt (denoting truth).

The semantics are defined relative to some valuation ρ, that is used to give meaning to predicate and term variables. The full semantics is given in Table 2.5

The semantics are rather straight-forward for the standard first order subset of the logic. For example, a disjunction is the union of the meaning of its constituents. The most interesting part is the semantics for recursive formulae. To a recursive formula ϕ we assign the meaning of the union of all ordinal approximants of ϕ. An approximation of a formula with approximant κ is in essence κ unfoldings of the formula, starting from false for the zeroth unfolding.

In [BS01], Bradfield and Stirling give some tips on how to read μ-calculus formulae. A fixpoint formula is to be read as a loop through the formula, recursing at the occurrences of the fixpoint variable. If it is a least fixed point, we should think of it as finite looping (meaning that we must terminate sometime), if it is a greatest fixed point, we allow infinite looping also.

Example 2.2.4 *The definition of $\bot$ is worth elaborating on as an example. A least fixed point must have a finite unfolding that is valid. The formulae $\mu X : prop.X$ clearly does not have such an unfolding, since it unfolds to itself.*

As a side-note, this definition, and the definition of $\top$, is different from the definitions in [Fre01]. There, $\top$ is defined as $\phi \vee \neg\phi$ for some unspecified formula

$$
\begin{array}{lcl}
\|t_1 = t_2\|_\rho & \triangleq & \text{if } t_1\rho = t_2\rho \text{ then } \{tt\} \text{ else } \emptyset \\
\|\phi \vee \psi\|_\rho & \triangleq & \|\phi\|_\rho \cup \|\psi\|_\rho \\
\|\neg\phi\|_\rho & \triangleq & \{tt\} \setminus \|\phi\|_\rho \\
\|\exists x : s.\phi\|_\rho & \triangleq & \bigcup_{v \in T(\Sigma)_s} \left(\|\phi\|_{\rho[v/x]}\right) \\
\|\lambda x : s.\phi\|_\rho & \triangleq & \lambda y : s.\|\phi\|_{\rho[y/x]} \\
\|\phi \; t\|_\rho & \triangleq & \|\phi\|_\rho \; t\rho \\
\|\mu X : type_\phi.\phi\|_\rho & \triangleq & \bigcup_\beta \|(\mu X : type_\phi.\phi)^\kappa\|_{\rho[\beta/\kappa]}, \quad \beta \;\; \text{any ordinal} \\
\|(\mu X : type_\phi.\phi)^\kappa\|_\rho & \triangleq & \begin{cases} \lambda x_1 : s_1 . \ldots . \lambda x_n : s_n . \emptyset \\ \text{if } \rho(\kappa) = 0 \text{ and } type_\phi = s_1 \rightarrow \cdots \rightarrow s_n \rightarrow \text{prop} \\ \\ \|\phi\|_{\rho\left[\|(\mu X : type_\phi.\phi)^\kappa\|_{\rho[\beta/\kappa]}/X\right]} \\ \text{if } \rho(\kappa) = \beta + 1 \\ \\ \bigcup_\beta \left\{\|(\mu X : type_\phi.\phi)^\kappa\|_{\rho[\beta/\kappa]} \;\mid\; \beta < \rho(\kappa)\right\} \\ \text{if } \rho(\kappa) \text{ is a limit ordinal} \end{cases} \\
\|U\|_\rho & \triangleq & \rho(U)
\end{array}
$$

Table 2.5: The semantics of formulae

ϕ.

Example 2.2.5 *As an example of a more complex μ-calculus formula, we give the formula relating two natural numbers to their sum. The formula is*

$$
\begin{array}{l}
\mu U \;:\; nat \rightarrow nat \rightarrow nat \rightarrow prop. \\
\quad \lambda x : nat.\lambda y : nat.\lambda z : nat. \\
\qquad\qquad (x = 0 \wedge z = y) \\
\qquad \vee \left(\begin{array}{rl} \exists x' : nat.\exists z' : nat. & \\ & x = x' + 1 \\ \wedge & z = z' + 1 \\ \wedge & U \; x' \; y \; z' \end{array} \right)
\end{array}
$$

This formula can be read as follows. It is a least fixed point of a function, taking three natural numbers (x, y, and z) as arguments. The function is true if one of the following cases are true. Either the first argument is zero and y and z are equal, in which case we clearly know that $x + y = z$. Otherwise, there must exists some other natural numbers x' and z', both one less than x and z. This corresponds to subtracting one from both sides of the equation $x + y = z$. Furthermore, we must have that $x' + y = z'$, represented as a recursive use of the formula.

Example 2.2.6 *To illustrate the close relationship with logic programming, we will show the above relation as a program in Prolog. First we define the relation* `add` *in in Prolog (assuming the same representation for numbers as above).*

```
add(X, Y, Z) :- X=0, Y=Z.
add(X, Y, Z) :- X'+1=X, Z'+1=Z, add(X', Y, Z').
```

This definition gives us two opportunities for proving the clause `add(3, W, 10)`*: we can match with the first clause, requiring* `X` *to be equal to zero (which is false); we can match against the second clause, requiring the existence of new values* `X'` *and* `Z'` *one less than their counterparts (which is true), and that we can prove* `add(2, W, 9)`*.*

This formulation resembles the one above very closely, if the implicit logical connectives are shown. Writing these explicitly, the definition of `add` *would look like*

```
add(X, Y, Z) ≜
    (X=0 ∧ Y=Z)
  ∨ (∃X'. ∃Z'.
        X'+1=X ∧ Z'+1=Z ∧ add(X', Y, Z').
```

The semantics of Prolog specifies the meaning of a relation as the least fixed point of the unfolding of the relation, which would translate into a μ-definition for the relation, effectively arriving at the same formula as the one in the above example.

For more information on Prolog, see, for example, the textbook The Art of Prolog[SS94].

2.2.4 A Proof-system for the μ-calculus

To handle the logic described, we need a proof-system for proving the validity of sequents. The proof-system consists of local proof rules for the classical first-order logic part, along with some local and one global rule for handling recursive formulae.

We use proof rules (Definition 2.1.2) with both conclusions and premises of the form $\Gamma \vdash \Delta$, where Γ and Δ are sets of μ-calculus formulae. The sequents are the Gentzen sequents introduced in Section 2.2.2. The intuition behind a proof rule is that whenever all the premises are valid (i.e. provable), so is the conclusion. If there are no premises, the proof rule is an axiom, and the conclusion is always valid.

To specify the proof rules, we will use schemas for collections of rules, where a specific rule may be obtained by instantiating a schema with the appropriate formulae. As an example, we take the schema for introducing disjunction on the left side:

$$\vee_l \; \frac{\Gamma, \phi \vdash \Delta \qquad \Gamma, \psi \vdash \Delta}{\Gamma, \phi \vee \psi \vdash \Delta}$$

$$\text{Id}\ \frac{\cdot}{\Gamma, \phi \vdash \phi, \Delta}$$

$$\text{W}_l\ \frac{\Gamma \vdash \Delta}{\Gamma, \phi \vdash \Delta} \qquad\qquad \text{W}_r\ \frac{\Gamma \vdash \Delta}{\Gamma \vdash \phi, \Delta}$$

$$\text{Cut}\ \frac{\Gamma \vdash \phi, \Delta \qquad \Gamma, \phi \vdash \Delta}{\Gamma \vdash \Delta}$$

Table 2.6: Structural proof-rules.

where we use the notation Γ, ϕ instead of the more verbose $\Gamma \cup \{\phi\}$. An actual proof rule is obtained by substituting ϕ and ψ for some formulae and Γ and Δ for some sets of formulae.

In some schemas there will be side-conditions, written to the right of the line, indicating some extra properties that we need for an instantiation to be valid. When we write x *fresh* as a condition, with x a term variable, it means that x does not occur free in the conclusion. When we write κ *fresh*, it means that the ordinal variable κ must be globally fresh to the proof-tree. To simplify the terminology, we will refer to a schema for proof rules simply as a proof rule.

Proof rules for classical first-order logic

The rules for the classical first order part of the logic are the rules that deal with non-recursive formulae. The rules are categorised as follows.

Structural rules The structural rules (see Table 2.6) manipulate the structure of sequents without regard to the actual formulae. The weakening rules allow us to infer a sequent if we can infer the same sequent without a member-formula of Γ or Δ. The identity rule is an axiom that allows us to infer a sequent if a formula occurs in both the right- and left-hand side.

The cut-rule is an interesting rule. For example, it may be viewed as the introduction of a lemma ϕ, such that we prove the lemma from the left-hand side (the validity of the lemma), and we prove the right-hand side using the lemma. Strictly speaking, the cut rule can be eliminated from a proof[GTL89], but not without a loss of perspicuity.

The Γ and Δ of $\Gamma \vdash \Delta$ are sets of formulae, and the notation Γ, ϕ is just a shorthand for $\Gamma \cup \{\phi\}$. Therefore we do not need rules to reorder the formulae of a sequent.

Logical rules The logical rules (see Table 2.7) deal with the introduction of the basic logical connectives ($\neg$, $\vee$ and $\exists$), as well as applications of arguments to λ-abstractions.

The rules for application, negation, and disjunction are simple consequences of their semantics. To infer an existential quantification to the right we need a witness

$$\neg_l\ \frac{\Gamma \vdash \phi, \Delta}{\Gamma, \neg\phi \vdash \Delta} \qquad\qquad \neg_r\ \frac{\Gamma, \phi \vdash \Delta}{\Gamma \vdash \neg\phi, \Delta}$$

$$\vee_l\ \frac{\Gamma, \phi \vdash \Delta \qquad \Gamma, \psi \vdash \Delta}{\Gamma, \phi \vee \psi \vdash \Delta} \qquad\qquad \vee_r\ \frac{\Gamma \vdash \phi, \psi, \Delta}{\Gamma \vdash \phi \vee \psi, \Delta}$$

$$\exists_l\ \frac{\Gamma, \phi[x'/x] \vdash \Delta}{\Gamma, \exists x : s.\phi \vdash \Delta}\ x' \text{ fresh} \qquad\qquad \exists_r\ \frac{\Gamma \vdash \phi[t/x], \Delta}{\Gamma \vdash \exists x : s.\phi, \Delta}\ t \in T(\Sigma, V)_s$$

$$\text{Apply}_l\ \frac{\Gamma, \phi[t/x] \vdash \Delta}{\Gamma, (\lambda x : s.\phi)\ t \vdash \Delta} \qquad\qquad \text{Apply}_r\ \frac{\Gamma \vdash \phi[t/x], \Delta}{\Gamma \vdash (\lambda x : s.\phi)\ t, \Delta}$$

Table 2.7: Logical proof rules.

$$\text{Refl}\ \frac{\cdot}{\Gamma \vdash t = t, \Delta} \qquad\qquad \text{Subst}\ \frac{\Gamma[t'/x] \vdash \Delta[t'/x]}{\Gamma[t/x], t = t' \vdash \Delta[t/x]}$$

Table 2.8: Equality proof rules

to it, to infer the quantification to the left, we need to prove it for some new system variable x'.

Equality rules The rules for equality (see Table 2.8) handle equality between terms of any sort.

The rules state that equality between equal terms on the right is an axiom (reflexivity), and that the equality of two terms on the left is true if we can prove the same statement with the left term substituted for the right term in the rest of the sequent.

Transitivity and symmetry of equality are not needed as basic rules, since they can be derived using these rules and the structural rules.

Equality rules for freely generated sorts The equality rules for freely generated sorts (see Table 2.9) are additional rules for equality. These rules apply whenever the associated sort of the result for the head-constructor of a term does not stipulate equality between syntactically different terms.

With CEq$_l$ and CEq$_r$ we may deconstruct terms, moving the equality-test to point-wise equality between the arguments. Using CIneq, we may conclude inequality when the head-constructor differs.

Derived proof rules

The basic proof rules given in the previous section is not enough. We must also be able to handle the extensions listed in Table 2.4 in a convenient manner. These

$$\text{CEq}_l\ \frac{\Gamma, t_1 = t_1', \ldots, t_n = t_n' \vdash \Delta}{\Gamma, op(t_1, \ldots, t_n) = op(t_1' \ldots, t_n') \vdash \Delta}$$

$$\text{CEq}_r\ \frac{\Gamma, \vdash t_1 = t_1', \Delta \qquad \cdots \qquad \Gamma, \vdash t_n = t_n', \Delta}{\Gamma \vdash op(t_1, \ldots, t_n) = op(t_1' \ldots, t_n'), \Delta}$$

$$\text{CIneq}\ \frac{\cdot}{\Gamma, op(t_1, \ldots, t_n) = op'(t_1' \ldots, t_m') \vdash \Delta}\ op \neq op'$$

Table 2.9: Equality proof rules for terms of freely generated sorts.

$$\text{C}_l\ \frac{\Gamma, \phi, \phi \vdash \Delta}{\Gamma, \phi \vdash \Delta} \qquad \text{C}_r\ \frac{\Gamma \vdash \phi, \phi, \Delta}{\Gamma \vdash \phi, \Delta}$$

$$\bot_l\ \frac{\cdot}{\Gamma, \bot \vdash \Delta} \qquad \top_r\ \frac{\cdot}{\Gamma \vdash \top, \Delta}$$

$$\wedge_l\ \frac{\Gamma, \phi, \psi \vdash \Delta}{\Gamma, \phi \wedge \psi \vdash \Delta} \qquad \wedge_r\ \frac{\Gamma \vdash \phi, \Delta \qquad \Gamma \vdash \psi, \Delta}{\Gamma \vdash \phi \wedge \psi, \Delta}$$

$$\forall_l\ \frac{\Gamma, \phi[t/x] \vdash \Delta}{\Gamma, \forall x : s.\phi \vdash \Delta}\ t \in T(\Sigma, V)_s \qquad \forall_r\ \frac{\Gamma \vdash \phi[x'/x], \Delta}{\Gamma \vdash \forall x : s.\phi, \Delta}\ x' \text{ fresh}$$

$$\Rightarrow_l\ \frac{\Gamma \vdash \phi, \Delta \qquad \Gamma, \psi \vdash \Delta}{\Gamma, \phi \Rightarrow \psi \vdash \Delta} \qquad \Rightarrow_r\ \frac{\Gamma, \phi \vdash \psi, \Delta}{\Gamma \vdash \phi \Rightarrow \psi, \Delta}$$

$$\text{Symm}_l\ \frac{\Gamma, t' = t \vdash \Delta}{\Gamma, t = t' \vdash \Delta} \qquad \text{Symm}_r\ \frac{\Gamma \vdash t' = t, \Delta}{\Gamma \vdash t = t', \Delta}$$

$$\text{Trans}_l\ \frac{\Gamma, t_1 = t_3 \vdash \Delta}{\Gamma, t_1 = t_2, t_2 = t_3 \vdash \Delta} \qquad \text{Trans}_r\ \frac{\Gamma \vdash t_1 = t_3, \Delta}{\Gamma \vdash t_1 = t_2, t_2 = t_3, \Delta}$$

Table 2.10: Derived proof rules

derived proof rules are given in Table 2.10, along with some further structural rules and rules for handling the transitivity and symmetry of equalities.

As an example of such a derivation, we present a derivation of the rule C_l, which is contraction of a repeated formula to the left.

$$\frac{\dfrac{\cdot}{\Gamma, \phi \vdash \phi, \Delta}\ \text{Id} \qquad \Gamma, \phi, \phi \vdash \Delta}{\Gamma, \phi \vdash \Delta}\ \text{Cut}$$

This derivation is not a complete proof. Instead, we can view it as a proof-snippet which could be inserted into any proof using the C_l proof rule, without altering the proof. This is possible since this derivation does not rely on any special property of Γ, Δ, or ϕ.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\cdot}{\phi \vdash \neg\psi, \phi}\,\mathrm{ID}}{\vdash \neg\phi, \neg\psi, \phi}\,\neg_r \qquad \dfrac{\dfrac{\cdot}{\psi \vdash \neg\phi, \psi}\,\mathrm{ID}}{\vdash \neg\phi, \neg\psi, \psi}\,\neg_r}{\vdash \neg\phi, \neg\psi, \phi \wedge \psi}\,\wedge_r}{\vdash \neg\phi \vee \neg\psi, \phi \wedge \psi}\,\vee_r}{\neg(\phi \wedge \psi) \vdash \neg\phi \vee \neg\psi}\,\neg_l}{\vdash \neg(\phi \wedge \psi) \Rightarrow (\neg\phi \vee \neg\psi)}\,\Rightarrow_r \qquad \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\cdot}{\phi, \psi \vdash \phi}\,\mathrm{ID}}{\neg\phi, \phi, \psi \vdash}\,\neg_l \qquad \dfrac{\dfrac{\cdot}{\phi, \psi \vdash \psi}\,\mathrm{ID}}{\neg\psi, \phi, \psi \vdash}\,\neg_l}{\neg\phi \vee \neg\psi, \phi, \psi \vdash}\,\vee_l}{\neg\phi \vee \neg\psi, \phi \wedge \psi \vdash}\,\wedge_r}{\neg\phi \vee \neg\psi \vdash \neg(\phi \wedge \psi)}\,\neg_r}{\vdash (\neg\phi \vee \neg\psi) \Rightarrow \neg(\phi \wedge \psi)}\,\Rightarrow_r}{\vdash \neg(\phi \wedge \psi) \Rightarrow (\neg\phi \vee \neg) \wedge (\neg\phi \vee \neg) \Rightarrow \neg(\phi \wedge \psi)}\,\wedge_r}{\vdash \neg(\phi \wedge \psi) \Leftrightarrow (\neg\phi \vee \neg\psi)}\,\text{abbrv.}$$

Figure 2.2: Derivation of the validity of $\vdash \neg(\phi \wedge \psi) \Leftrightarrow (\neg\phi \vee \neg\psi)$.

Example: $\vdash \neg(\phi \wedge \psi) \Leftrightarrow (\neg\phi \vee \neg\psi)$

As an example of the proof rules for the logic, we give the derivation of the double implication $\neg(\phi \wedge \psi) \Leftrightarrow (\neg\phi \vee \neg\psi)$, a classical DeMorgan rule.[1] The complete derivation is given in Figure 2.2. The statement that we prove consists of the formula placed to the right of the turnstile, which means that the implication must always be true.

We start by expanding the abbreviation of the double implication, and proceed by splitting the conjunction. This result in two goals to prove, both proved in essentially the same way, and the proofs are rather straightforward.

Worth noting is that the use of the rules $\wedge_r$ and $\vee_l$ are both used as late as possible in the proof. They could both be used before $\neg_l$ and $\neg_r$, but then we would have more applications of rules. This is not an essential part in proving the truth of the formula, but can be a useful strategy in larger proofs to minimize the size of the proof.

Proofs of recursive formulae

To handle the recursive construction there are some local proof rules and one global rule of discharge. The local rules allow sound unfoldings of fixpoint formulae. Using the global rule on a locally sound proof, we can identify recurrent nodes that define a well-founded induction on the ordinal variables.

The local rules for least fixed points are given in table 2.11 and the derived rules for greatest fixed points in Table 2.12. These rules enable the approximation and unfolding of fixpoint formulae. There are also some more rules, shown in Table 2.13, based on the monotonicity of ordinal approximations (rules IdMon1 and IdMon2) as well as transitivity of ordinal variables (rule OrdTrans).

[1] We use the standard abbreviation $\phi \Leftrightarrow \psi \triangleq (\phi \Rightarrow \psi) \wedge (\psi \Rightarrow \phi)$ in this example.

$$\text{Approx}_l\ \frac{\Gamma, ((\mu U : \mathit{type}_\phi.\phi)^\kappa)\ t_1\ \ldots\ t_n \vdash \Delta}{\Gamma, (\mu U : \mathit{type}_\phi.\phi)\ t_1\ \ldots\ t_n \vdash \Delta}\ \kappa \text{ fresh}$$

$$\text{Unf1}_l\ \frac{\Gamma, (\phi[\mu U : \mathit{type}_\phi.\phi/U])\ t_1\ \ldots\ t_n \vdash \Delta}{\Gamma, (\mu U : \mathit{type}_\phi.\phi)\ t_1\ \ldots\ t_n \vdash \Delta}$$

$$\text{Unf1}_r\ \frac{\Gamma \vdash (\phi[\mu U : \mathit{type}_\phi.\phi/U])\ t_1\ \ldots\ t_n, \Delta}{\Gamma \vdash (\mu U : \mathit{type}_\phi.\phi)\ t_1\ \ldots\ t_n, \Delta}$$

$$\text{Unf2}_l\ \frac{\Gamma, (\phi[(\mu U : \mathit{type}_\phi.\phi)^{\kappa'}/U])\ t_1\ \ldots\ t_n, \kappa' < \kappa \vdash \Delta}{\Gamma, ((\mu U : \mathit{type}_\phi.\phi)^\kappa)\ t_1\ \ldots\ t_n \vdash \Delta}\ \kappa' \text{ fresh}$$

$$\text{Unf3}_r\ \frac{\Gamma \vdash \kappa' < \kappa, \Delta \qquad \Gamma \vdash (\phi[(\mu U : \mathit{type}_\phi.\phi)^{\kappa'}/U])\ t_1\ \ldots\ t_n, \Delta}{\Gamma \vdash ((\mu U : \mathit{type}_\phi.\phi)^\kappa)\ t_1\ \ldots\ t_n, \Delta}$$

Table 2.11: Local proof rules for least fixed points.

$$\text{Approx}_r\ \frac{\Gamma \vdash ((\nu U : \mathit{type}_\phi.\phi)^\kappa)\ t_1\ \ldots\ t_n, \Delta}{\Gamma \vdash (\nu U : \mathit{type}_\phi.\phi)\ t_1\ \ldots\ t_n, \Delta}\ \kappa \text{ fresh}$$

$$\text{Unf1}\nu_l\ \frac{\Gamma, (\phi[\nu U : \mathit{type}_\phi.\phi/U])\ t_1\ \ldots\ t_n \vdash \Delta}{\Gamma, (\nu U : \mathit{type}_\phi.\phi)\ t_1\ \ldots\ t_n \vdash \Delta}$$

$$\text{Unf1}\nu_r\ \frac{\Gamma \vdash (\phi[\nu U : \mathit{type}_\phi.\phi/U])\ t_1\ \ldots\ t_n, \Delta}{\Gamma \vdash (\nu U : \mathit{type}_\phi.\phi)\ t_1\ \ldots\ t_n, \Delta}$$

$$\text{Unf2}_r\ \frac{\Gamma, \kappa' < \kappa \vdash (\phi[(\nu U : \mathit{type}_\phi.\phi)^{\kappa'}/U])\ t_1\ \ldots\ t_n, \Delta}{\Gamma \vdash ((\nu U : \mathit{type}_\phi.\phi)^\kappa)\ t_1\ \ldots\ t_n, \Delta}\ \kappa' \text{ fresh}$$

$$\text{Unf3}_l\ \frac{\Gamma \vdash \kappa' < \kappa, \Delta \qquad \Gamma, (\phi[(\nu U : \mathit{type}_\phi.\phi)^{\kappa'}/U])\ t_1\ \ldots\ t_n \vdash \Delta}{\Gamma, ((\nu U : \mathit{type}_\phi.\phi)^\kappa)\ t_1\ \ldots\ t_n \vdash \Delta}$$

Table 2.12: Derived local proof rules for greatest fixed points.

$$\text{OrdTrans}\ \frac{\Gamma, \kappa < \kappa', \kappa' < \kappa'', \kappa < \kappa'' \vdash \Delta}{\Gamma, \kappa < \kappa', \kappa' < \kappa'' \vdash \Delta}$$

$$\text{IdMon1}\ \frac{\Gamma \vdash \kappa \leq \kappa', \Delta}{\Gamma, (\mu U : \mathit{type}_\phi.\phi)^\kappa\ t_1\ \ldots\ t_n \vdash (\mu U : \mathit{type}_\phi.\phi)^{\kappa'}\ t_1\ \ldots\ t_n, \Delta}$$

$$\text{IdMon2}\ \frac{\Gamma \vdash \kappa' \leq \kappa, \Delta}{\Gamma, (\nu U : \mathit{type}_\phi.\phi)^\kappa\ t_1\ \ldots\ t_n \vdash (\nu U : \mathit{type}_\phi.\phi)^{\kappa'}\ t_1\ \ldots\ t_n, \Delta}$$

Table 2.13: Monotonicity and transitivity rules for ordinals.

To prove an inductive assertion, there is a global rule of discharge. That the rule is global means that it is defined on the whole proof-tree. This rule enables fixed-point induction using well-founded induction on the approximating ordinals. The rule is defined on unfinished proofs with proof-nodes that are recurring. A recurring node is a pair of nodes in the proof-tree, one a leaf and the other, the *companion*, a node on the path from the root to the leaf. The global discharge condition requires that all the recurring nodes are part of a well-founded induction schema on the approximation variables.

The meaning of this condition is that the proof-graph (which now has back edges), has a well-founded unfolding of the fixpoint formulae so that we may loop through the graph, and when doing so we are guaranteed to make some progress. For a complete description of the global discharge, see [SD03]. In addition to defining the rule of discharge, Sprenger and Dam proves the equivalence between this proof system and proof-systems based on local induction proof-rules, by showing a translation between the two systems.

Example 2.2.7 *To illustrate the most basic use of the discharge rule, we give the derivation of the* $\top_r$*-proof rule.*

$$
\dfrac{\dfrac{\dfrac{\dfrac{\overset{\textit{Discharge against }\circledast}{\Gamma, (\mu X : prop.X)^{\kappa'}, \kappa' < \kappa \vdash \Delta}}{\circledast \quad \Gamma, (\mu X : prop.X)^{\kappa} \vdash \Delta}\,\textit{Unf2}_l}{\Gamma, \mu X : prop.X \vdash \Delta}\,\textit{Approx}_l}{\Gamma \vdash \neg \mu X : prop.X, \Delta}\,\neg_r}{\Gamma \vdash \top, \Delta}\,\textit{abbrv.}
$$

After expanding the abbreviation and removing the negation, we have a least fixed point to the left of the turnstile. We approximate the formula to get the result $(\mu X : prop.X)^{\kappa}$*, and then we unfold the approximation. The new approximated formula* $(\mu X : prop.X)^{\kappa'}$ *has an approximant that is strictly less than the one in the previous goal. We can discharge against it since it is a least fixed point that is unfolded to the left, with a well-founded ordering of the approximants. This intuitively means that we can continue to unfold the formula until we reach* $(\mu X : prop.X)^0$*, which is a false statement according to the semantics.*

For a larger example that uses discharge, see Section 2.3.4.

Conventions

For the logic to be a usable tool for specifying properties about programs, we need some further features to construct formulae, such as modalities and a simpler type-system. These features are described below.

Satisfactions As shown in Table 2.4 on extensions to the base logic, we will let a formula of the form $s : \phi$ stand for the application of s to ϕ (i.e. $\phi\ s$). This is used to separate the properties (and the arguments that parametrises the property)

$$\Box_l \; \frac{\Gamma \vdash t \xrightarrow{\alpha} t', \Delta \qquad \Gamma, t' : \phi \vdash \Delta}{\Gamma, t : [\alpha]\phi \vdash \Delta} \; t' \in T(\Sigma, V)_S$$

$$\Box_r \; \frac{\Gamma, t \xrightarrow{\alpha} X \vdash X : \phi, \Delta}{\Gamma \vdash t : [\alpha]\phi, \Delta} \; X \text{ fresh}$$

$$\Diamond_l \; \frac{\Gamma \vdash t \xrightarrow{\alpha} X, \Delta \qquad \Gamma, X : \phi \vdash \Delta}{\Gamma, t : \langle\alpha\rangle\phi \vdash \Delta} \; X \text{ fresh}$$

$$\Diamond_r \; \frac{\Gamma \vdash t \xrightarrow{\alpha} t', \Delta \qquad \Gamma \vdash t' : \phi, \Delta}{\Gamma, t : \langle\alpha\rangle\phi \vdash \Delta} \; t' \in T(\Sigma, V)_S$$

Table 2.14: Derived proof rules for modalities.

from the systems that have the property in question. Thus, we will only use the satisfaction-form for systems s and properties of systems ϕ.

Modalities The logic described does not contain any primitive modal operators. The view is taken that an operational semantics transition-relation $\to \subseteq S \times A \times S$ is given as a predicate in the logic. Using this predicate it is possible to encode the standard Hennessy-Milner multi-modal operators as follows.

$$\begin{aligned} [\alpha]\phi &\triangleq \lambda sys : S.\, \forall sys' : S.\ sys \xrightarrow{\alpha} sys' \Rightarrow sys' : \phi \\ \langle\alpha\rangle\phi &\triangleq \lambda sys : S.\, \exists sys' : S.\ sys \xrightarrow{\alpha} sys' \wedge sys' : \phi \end{aligned}$$

The box-modality definition specifies that it is a function taking a system (the sort S) as argument. This function is true if, for all systems that we can reach from the argument via an α transition, the property ϕ is true. Similarly for the diamond modality, it is true if there is some system which we can reach with an α-transition, such that it has the property ϕ.

To handle these two extensions, we need further proof rules in the style of the previous sections. These are given in Table 2.14, assuming that the transition-relation that we reason about is given by $\to$.

Example 2.2.8 *The μ-calculus is very expressive language for specifying properties about programs using modalities. If we have a transition system where the only action is τ, the formula*

$$\mu U : system \to prop.[\tau]U$$

specifies that a system will, at some point in time, halt. The definition of the box modality specifies that we should follow all execution paths when we unfold this formula, and for each such path there is a recursive use of the formula. The only *way to end a path, and thus satisy the least fixed point, is if the system halts (i.e., there are no more valid transitions).*

Example 2.2.9 *Sometimes we want to assume that some system we are interested in has some fairness property. One such property is that on any path where the action α is enabled infinitely often, it will be executed. This property, often referred to as strong fairness, is expressible in the μ-calculus.*

In this example, we let the notation $[-\alpha]$ stand for all possible actions except α. Thus, if the possible actions are α, β, and γ, then $[-\alpha]\phi$ stands for $[\beta]\phi \wedge [\gamma]\phi$.

We can now express the property with the following formula from [BS01]. We omit types for the fix-point operators for clarity.

$$\nu X.\mu Y.\nu Z.[\alpha]X \wedge \big((\langle\alpha\rangle\lambda x : S.\top) \rightarrow [-\alpha]Y\big) \wedge [-\alpha]Z$$

The formula can be explained as follows. After any α action, we always have that when we loop through non α actions via the inner greatest fixpoint, and α is possible to execute infinitely often (thus satisfying the antecedent in the middle clause), we must sometime take an α action, since otherwise the middle least fixpoint would be looped through infinitely often, and thus false.

Lifting abstractions The type-system briefly described earlier combined with the above definitions forces all formulas used after a modal operator or a satisfaction to have the type $S \rightarrow \mathit{prop}$. This leads to problems for modalities such that the classical formula $\langle\alpha\rangle\top$ (indicating that there is some α transition available) is not a valid formula. Instead, it must be written as $\langle\alpha\rangle\lambda x : S.\top$. This notation is inconvenient and distracts from the essential properties of the formula. To remedy this, we allow the silent addition of abstractions to formulae. This is done by applying a *lift*-function to formulae occurring after a modal operators or satisfactions, so that $\langle\alpha\rangle\phi$ is handled as if it were written as $\langle\alpha\rangle \mathit{lift}(\phi)$. The *lift*-function is defined according to the following rules over the basic formulae.

$$\begin{array}{ll}
\mathit{lift}(t_1 = t_2) & = \lambda x : S.t_1 = t_2 \\
\mathit{lift}(\phi \vee \psi) & = \lambda x : S.\mathit{lift}(\phi)\ x \vee \mathit{lift}(\psi)\ x \\
\mathit{lift}(\neg\phi) & = \lambda x : S.\neg(\mathit{lift}(\phi)\ x) \\
\mathit{lift}(\exists x' : s'.\phi) & = \lambda x : S.\exists x' : s'.\mathit{lift}(\phi)\ x \\
\mathit{lift}(\phi) & = \begin{cases} \lambda x : S.\phi & \text{if } \phi \text{ has type } \mathit{prop} \\ \phi & \text{otherwise} \end{cases}
\end{array}$$

We assume that x is fresh in ϕ, ψ, t_1, t_2. The definition here is slightly different from the definition in [Fre01], so that it works with our new definitions of $\top$ and $\bot$. The difference is in the last clause, where we introduce abstractions around for formulae of type *prop*. This is also consistent with the way VCPT behaves.

Using this scheme, a formula such as $\langle\tau\rangle(\langle\tau\rangle\top \vee \langle\tau\rangle\langle\tau\rangle\bot)$ will be handled as if it were defined as $\langle\tau\rangle\lambda x : S.((\langle\tau\rangle\neg\lambda y : S.(\lambda z : S.\bot)\ y)\ x \vee (\langle\tau\rangle\langle\tau\rangle\lambda w : S.\bot)\ x)$ (unfolding the definition of $\top$ but not $\bot$). The meaning of the formula is that for some τ-transition, we can do one more τ-transition, but not two more.

2.3 The VeriCode Proof Tool

The VeriCode Proof Tool (VCPT) [VCP] is a proof-assistant for the μ-calculus. It supports lazy incremental discovery of induction schemas, postponing the choice of witnesses, structured collections of definitions, graphical display of proofs, and extensive automation through tactics and programs.

Like many other proof-assistants it is implemented in Standard ML (SML), a functional language originally created specifically for proof-assistants and theorem-provers[GMM$^+$78]. The language was originally called the Metalanguage of the theorem-prover, which was subsequently shortened to ML. There are many different dialects in the ML family (e.g, Standard ML and Ocaml). For more information about SML, see the textbook [MCP93]. For an introduction to proof-assistants, see [Pau92], where a basic proof-assistant is implemented in SML.

The user interface of the proof-assistant is the top-level environment of SML, which has a read-eval-print loop. By evaluating expressions containing functions defined in VCPT, it is possible to change the global state of the system and the proof. SML uses type-inference, making almost all type-annotations unnecessary. This greatly simplifies the interactive use of the language and in-line function definitions.

A proof in VCPT is constructed backwards, i.e., we start with the goal that is to be proved. We elaborate on this goal by applying proof rules with matching conclusions. If these rules have premises, these appear as new goals. In contrast to other proof-assistants, the whole proof-tree, instead of just the current proof-goals, are saved. This is done to implement the rule of discharge which is defined on the whole tree.

As a proof-assistant, VCPT automates the application of proof rules to goals, the collections of available definitions, delaying a choice of witness, and incremental checks for the rule of discharge. However, VCPT is not an automatic theorem-prover, with large, sophisticated methods to handle logical formulae without interaction.

By automating the application of proof rules, the soundness of a proof is ensured, reducing the risk of errors when constructing a proof. The rule of discharge is implemented as an incremental check on the proof-tree, instead of the abstract formulation that relies on the whole proof-tree to be available. This allows the incremental development of inductive proofs.

Choosing a witness is needed, for example, in the rule for introducing an existential quantifier to the right. Delaying this choice and introducing a so-called meta-variable instead, enables us to carry on with the proof without committing to a specific witness. When we have more information about which witness that could be profitable to use, we can instantiate the meta-variable with this value. This is very useful for programming automated proof-searches.

The graphical display of proofs is based on automatic interaction with the DaVinci graph visualiser[uDr]. For example, by evaluating `display_proof();` in VCPT, the DaVinci system will be started and the current proof displayed. This enables visual inspection of dependencies between different parts of the proof, as well as using it for simpler navigation through the proof.

Since VCPT is implemented in SML and uses the top-level environment for interaction, we can also program VCPT using SML. This enables the development of programs that searches for proofs.

Below, we will elaborate on the parts of VCPT used for defining an operational semantics. This includes such things as the syntax of formulae, describing an operational semantics, and how to automate small proofs. For a full account, see the reference manual for VCPT [VCP02]. We will conclude with a description of using VCPT to prove a simple property, to illustrate the style of reasoning employed.

2.3.1 Syntax and Theories

When writing non-interactive input to VCPT we collect our definitions of predicates, theorems, and data into theories. A theory will, typically, consist of some many-sorted first-order structure and some definitions of predicates. From these definitions we can then give theorems[2]. We may also give SML-code containing tactics or supporting code in a theory. The SML-code does, in most cases, reside in external files. When a theory-file is read, the C pre-processor will be run, allowing for macro-expansion of formulae elements, supporting cleaner definitions of properties.

In the tool, multiple theories may be loaded, incrementally building up the data, predicates, and lemmas that are available. Furthermore, a theory may itself use other theories.

Data When specifying the signature of data in the tool, we use a syntax that is simpler and more compact than the specifications of many-sorted first-order structures in the earlier sections. If we want to specify a new sort s with constructors $f_1, \ldots, f_n$ with annotations $t_1, \ldots, t_n$, we specify it as

$$\begin{array}{lcl}
\texttt{datatype}\ s & = & f_1\ \texttt{of}\ t_1 \\
 & | & f_2\ \texttt{of}\ t_2 \\
 & & \vdots \\
 & | & f_n\ \texttt{of}\ t_n
\end{array}$$

If the annotation t_i is empty, we omit it along with the word *of.* The new sort s is considered to be a freely generated sort unless it is specified that it is not using the modifier *nonflat* before the datatype declaration.

[2] In VCPT the class given to a statement that should be proved is not significant. One can choose – and we do so – freely from the names Theorem, Proposition, Lemma, Conjecture, Guess, and Axiom.

For every sort s there is also a sort s *list*, representing lists of elements of sort s. The syntax for writing an element of the sort s *list* is $[H_1, H_2, \ldots, H_n \mid T]$, for a list starting with the elements $H_1, H_2, \ldots, H_n$, all of sort s, and continuing with the tail T of sort s *list*. If no tail is given, the given head elements are all of the list, and we omit the bar. One must give at least one head-element if the tail is given. We can also specify tuples of sorts $s_1, \ldots, s_n$. Such a tuple has sort $s_1 \times \cdots \times s_n$, and an element of the tuple is given as $(e_1, \ldots, e_n)$ for elements e_i of the appropriate sort.

There are more advanced features, such as specifying sub-sorts of sorts using predicates, but these features were not used in this thesis and will not be further elaborated on here.

Syntactical extensions To simplify the construction of complex formulae, some new constructs are added to VCPT for handling some common usages.

In VCPT it is possible to write a formula

$$\begin{array}{lll}
\texttt{if} & \phi & \texttt{then} \\
 & \psi & \\
\texttt{else} & & \\
 & \gamma &
\end{array}$$

as an abbreviation for the formula $\phi \rightarrow \psi \wedge \neg\phi \rightarrow \gamma$. This simple construction shows the intent of a constructed formula more clearly than using the "pure" version directly.

Since we have algebraic datatypes, it is useful to have a construction for matching an element of data against some alternatives. The syntax used is the following.

$$\begin{array}{llll}
\multicolumn{4}{l}{\texttt{cases}\ t\ \texttt{of}} \\
 & t_1 & \Rightarrow & \phi_1 \\
\mid & t_2 & \Rightarrow & \phi_2 \\
 & & \vdots & \\
\mid & t_n & \Rightarrow & \phi_n \\
\multicolumn{4}{l}{\texttt{end}}
\end{array}$$

The meaning of such a formula is that t is tested for equality against all patterns t_i. This is done under an existential quantification of all the free variables in the patterns. Letting $V_1, \ldots, V_k$ of sorts $s_1, \ldots, s_k$ be the free variables in $t_1, \ldots, t_n$, the above formula is equivalent to the following formula.

$$\exists v_1 : s_1 . \ldots . \exists x_k : s_k . \bigwedge_{i=1}^{n} t = t_i \Rightarrow \phi_i$$

If a formula ϕ_i is simply $\top$, we may omit it and the arrow after the pattern.

Predicates Predicates may be specified as simple definitions, as least fixed points, as greatest fixed points, or simply declared. We may use a scope-construction to group some related definitions into a common scope with a single exported predicate.

A predicate *pred* taking arguments of sorts s_1 to s_n, is given as

$$\begin{array}{l} \mathit{pred} \;:\; s_1 \to \cdots \to s_n \to \mathit{prop} \mathrel{\hat{=}} \\ \quad \lambda T_1 : s_1. \ldots . \lambda T_n : s_n. \\ \qquad \phi \end{array}$$

for some formula ϕ. We let $\ulcorner \hat{=} \urcorner$ equal $\ulcorner = \urcorner$ for regular definitions, $\ulcorner \Leftarrow \urcorner$ for least fixed points, and $\ulcorner \Rightarrow \urcorner$ for greatest fixed points. The arguments T_1 to T_n will be bound term-variables in ϕ. Additionally, if the definition is a fixpoint definition, *pred* will be a bound predicate variable in ϕ.

When we want to have a definition containing nested fixpoint formulae, or if we want to make the definition more modular but not pollute the global namespace, we can use a scope-construction of the form

```
LET def WHERE def-list END.
```

Here we have a single definition *def* exported, but where the definitions in *def-list* are available in *def* and vice-versa.

It is also possible to specify fixed points directly with essentially the same syntax as the logic, if we want to give a fixpoint formula in-line.

A declared predicate introduces a predicate but does not give any meaning to it. In particular, it can be used to describe abstract concepts that are handled in some way outside of the logic, using for example SML code.

Formula macros It is not possible to define predicates that take predicates as arguments, but there is a simple form of macro expansion available. An example of this usage is when we define macros that expand to common logical formulae, such as the following macro specifying that a formula ϕ is always satisfied along every possible τ-action sequence.

```
#define Always(φ)
        (νX:system → prop .
             (φ ∧
              [τ]X))
```

Given this definition we may write $S \;:\; \mathit{Always}(\mathit{prp})$, to state that the system S should always have the property *prp*. If there are any other actions than τ, we could add those by adding more clauses to the conjunction.

Notation In the tool, we only have access to the basic ASCII alphabet. This constrains our notational possibilities, as we must find equivalents to the logical symbols. In Table 2.15, the notation used for non-ASCII characters are given.

Symbol	Equivalent	Symbol	Equivalent
$\Leftarrow$	`<=`	$\vee$	`\/` or `or`
$\Rightarrow$	`=>`	$\wedge$	`/\` or `and`
$\top$	`tt`	$\rightarrow$	`-->`
$\bot$	`ff`	$\neg$	`not`
μ	`lfp`	$\exists$	`exists`
ν	`gfp`	$\forall$	`forall`
λ	`\`	$\vdash$	`|-`
$\times$	`*`	$\|$	`|`

Table 2.15: ASCII notation for symbols in VCPT

2.3.2 Embedding an operational semantics

VCPT has some support for defining operational semantics. The program views an operational semantics as a predicate $pred : S \rightarrow A \rightarrow S \rightarrow prop$ (cf. Definition 2.1.1), where S is the sort of the states and A is the sort of the actions.

When such a predicate is specified, we declare it as an operational semantics-relation using the declaration

```
MODALITIES pred-name END.
```

This will enable reasoning about systems using modalities and satisfactions. The modalities are the multi-modal prefix-operators $[a]$ and $\langle a \rangle$, for any a in A. As described in Section 2.2.4, these are encoded and not primitive operators.

A useful connection between this support for predicates as transition-relations and the μ-calculus, is that the notion of a provable derivation in a TSS (Definition 2.1.3) corresponds to the truth of a least fixed point of a predicate encoding the TSS.

2.3.3 The VCPT Interface

The interface to VCPT are the functions defined that are available to the user. These may be called from the top-level environment or from a function. This section is divided into three parts. The first two talk about proof-rules and the third about other parts of the system.

Tactics and tacticals To prove something, we must use the proof rules presented in Section 2.2.4. These are available in the proof-tool in the form of tactics. A tactic `t` may be applied to the current goal by evaluating the expression `by(t);`. The proof rules are implemented as functions taking an integer argument specifying which formula to handle and returning a tactic. As an example, the type of the $\exists_l$-rule is `exists_l : int -> tactic`. There are tactics for all of the proof rules.

The rules that use a specific term take an optional argument specifying the term. If the argument is not given (i.e., the argument NONE is given), a new meta-variable is created. Thus, the choice of witness is postponed. A meta-variable may later be assigned a value, and this assignment will propagate through the entire proof-tree.

If a tactic is not applicable (the conclusion does not match the formula or a side-condition does not hold), it will throw an exception and the tactic will fail. If a tactic fails, no alterations to the proof-tree are done.

For the extensions listed in Section 2.3.1, there are tactics defined that will operate on them. An if-then-else formula will simply be split, and the implications removed. For a cases-formula, the work done is more extensive. The quantifiers will be removed, and the formula will be split. The equalities in the antecedents of the disjuncts are then evaluated. If the tactic can deduce equality using the possible simple substitutions, this will be done and that case chosen.

As is usual in proof-assistants, there are tacticals. These are functions that allow the creation of new tactics that are sound, by combining the basic tactics. Below, we present some representative examples of tacticals from VCPT.

t_skip Does nothing and always succeeds.

t_compose2 Takes two tactics as arguments. First it applies the first argument, then it applies the second argument to all resulting goals.

t_orelse Takes two tactics as arguments. If the application of the first argument fails, it tries with the second argument. If that fails to, the tactics fails.

t_rhs Takes an argument that is a function from integers to tactics. Tries to apply the argument anywhere in the right-hand side, and fails if it fails everywhere.

t_while Applies its tactic-argument iteratively as long as it does not fail. This tactic will always succeed.

t_fix A combinator that can be used to construct recursive tactics.

Example 2.3.1 *If we want to apply the tactics* `or_r` *and* `and_r` *to all positions in the right-hand side, we can define a tactic* `split_and_or_r` *as follows:*

```
val split_and_or_r =
  t_while
    (t_rhs (fn pos =>
             t_orelse
               (or_r pos)
               (and_r pos)
           )
    )
```

This tactic will iteratively split all disjunctions and conjunctions that are present, or becomes present, as top-level constructions on the right-hand side.

Since VCPT is controlled from the top-level of SML, is it not sufficient to simply use the SML features for programming the use of tactics? For example, the function

```
fun orand pos = t_compose2 (or_r pos) (and_r pos)
```

applies first $\vee_r$ and then $\wedge_r$ to position pos. This function uses the tacticals presented. Instead, we could write the SML-function

```
fun orand pos = (by(or_r pos); by(and_r pos)).
```

This function applies the same rules to the proof-tree. The difference is that the version constructed using the tacticals will not keep the resulting intermediate nodes in the proof-tree. This is significant when proofs grow, since VCPT will keep the whole proof-tree in memory.

Example 2.3.2 *The tactical* t_fix *is used to construct recursive tactics, and is best explained by example. The following tactic will split all disjunctions in the formula at position* pos *on the right-hand side, without affecting the formulae in the rest of the right hand side.*

```
fun splitor_r pos =
  t_fix 0 (fn add => (fn recurse =>
      (t_orelse
        (t_compose2 (or_r (pos+add))
                    (t_compose2 (recurse (add+1))
                                (recurse add)
                    )
        )
        t_skip)
  ))
```

Inside the tactic, recurse *can be called with an integer argument, available as* add*, as a recursive invocation of the tactic. In the initial invocation,* add *will have the value of 0. The tactic itself works as follows. If it is possible to split a disjunction at* pos+add*, this is done, otherwise we just succeed. If a disjunction is split, we recursively split the disjunctions in the two new formulae (located at* pos+add+1 *and* pos+add*).*

For the interested, the combinator t_fix *is implemented in the tool using the following function definition.*

```
fun t_fix a tt s = tt a (fn a => t_fix a tt) s
```

When SML is given this definition, it calculates that the most general type for t_fix *is* $\alpha \to (\alpha \to (\alpha \to \beta \to \gamma) \to \beta \to \gamma) \to \beta \to \gamma$ *for some types* α*,* β*, and* γ*. However, in VCPT, the type* $\beta \to \gamma$ *is constrained to be the type tactic.*

Non-tactic proof-rules Not all the proof rules are implemented as tactics. In particular, the global rule of discharge is implemented as a function.

The discharge rule is called using the expression `discharge0 n`, where `n` is a proof-tree node. The function then checks if the global discharge-condition is satisfied for the current goal with `n` as the companion node. There is also a convenience function `discharge1` that checks the current goal against all possible companion nodes.

When dealing with interleaving operational semantics of parallel programs, there is a general problem of different interleavings that result in the same state. In the worst case, we can get exponentially many derivations of a single state. To handle this problem, there is a function called copydischarge that merges two goals if they are the same[3]. There are two variants of copydischarge, invoked using the expressions `copydischarge0 n` and `copydischarge1()`, with similar behaviour as discharge.

Other functions In addition to the rules and tacticals presented above for applying proof-rules, there are other functions defined in VCPT. These functions are for handling theories, navigating the proof-tree and the different claims, and for accessing information about the environment. We will not discuss these features, an account is available in the reference manual[VCP02].

2.3.4 Example: There is no greatest natural number

This section describes a simple proof of a theorem in the proof-assistant. The purpose is to show the flavour of reasoning employed, and how a proof may be discovered.

We prove the theorem $\vdash \neg(\exists N : nat. \forall N' : nat.\ natLessEq\ N'\ N)$, which states that there is no greatest natural number.[4] The formula `natLessEq` is defined in the initial environment, and describes the less-than-or-equal relation. The definition is shown in Figure 2.3 in both input notation and mathematical notation.

The idea behind the definition of `natLessEq` is that we can decrease the first argument to zero faster than we can decrease the second argument. This is reflected in the definition, which is true if we have zero as the first argument, or if we can decrease both arguments by one and the relation still holds for these new values.

We start the proof with the rule `not_r`, which strips of the negation and moves the formula to the other side of the turnstile, resulting in the goal

$$\exists N : nat. \forall N' : nat.\ natLessEq\ N'\ N \ \vdash \ .$$

Now our goal is to show that the formula before the turnstile is invalid (thus showing that the goal is valid). The existential quantifier is removed of using `exists_l`,

[3] Copydischarge is more commonly known as the rule of subsumption.

[4] The judgement is written as `|- not(exists X:nat. forall Y:nat. natLessEq Y X)` in the proof-tool.

```
natLessEq : nat --> nat --> prop <=
  \X:nat. \Y:nat.
    cases X of
      0    => tt
      X'+1 => exists Y':nat.
                  (   Y=Y'+1
                   /\ natLessEq X' Y')
    end
end
```

$$
\begin{array}{l}
\mathit{natLessEq}\ :\ \mathit{nat} \rightarrow \mathit{nat} \rightarrow \mathit{prop} \Leftarrow \\
\quad \lambda X : \mathit{nat}.\ \lambda Y : \mathit{nat}. \\
\qquad\qquad X = 0 \\
\qquad \vee \quad \exists X' : \mathit{nat}. \exists Y' : \mathit{nat}. \\
\qquad\qquad \left(\begin{array}{ll} & X = X' + 1 \\ \wedge & Y = Y' + 1 \\ \wedge & \mathit{natLessEq}\ X'\ Y' \end{array} \right)
\end{array}
$$

Figure 2.3: The formula `natLessEq`.

which introduces a fresh existential variable ${X_0}^5$. We then use the rule `forall_l` to handle the universal quantifier. This rule takes an optional term representing our witness to the fact that the formula is false. By inspecting the current goal, we see that the term $X_0 + 1$ is such a witness, and we supply it to the rule[6]. The current goal is now

$$\mathit{natLessEq}\ X_0 + 1\ X_0\ \vdash\ .$$

To prove a goal such as the one we have now, we use inductive reasoning on the least fixed point `natLessEq`. First we unfold the formula, using `unfold_l`, so that we get access to the definition of `natLessEq`. Then we approximate the fixed point using `approx_l`. This results in the goal

$$\mathit{natLessEq}^{\kappa}\ X_0 + 1\ X_0\ \vdash\ , \tag{2.1}$$

where κ is the ordinal variable for the approximation. Now we may go on and unfold the approximated definition (`unfold_app_l`) and apply the function to the

[5] The index of the variable X_0 may vary depending on if any other existential variables has been introduced before.

[6] Supplying a term argument to a rule can, for example, be done using the function `pt` that parses a string and returns a term. Thus, the expression evaluated is `by(forall_l (SOME (pt "X_0+1")) 1);`

arguments, by using `apply_l` twice. This will give us the goal

$$\begin{array}{llll} \texttt{cases} & X_0+1 & \texttt{of} & \\ & 0 & \Rightarrow & \texttt{tt} \\ | & X'+1 & \Rightarrow & \exists Y' : nat. \\ &&& \left(\begin{array}{ll} & X_0 = Y'+1 \\ \wedge & natLessEq^{\kappa'}\ X'\ Y' \end{array}\right) \\ \texttt{end}, &&& \\ \kappa' < \kappa &&& \\ \vdash\ . &&& \end{array}$$

The ordinal in-equation $\kappa' < \kappa$ tells us that the recursive occurrence of the formula $\texttt{natLessEq}^{\kappa'}$, present in the second case-part, is in some sense less than the formula that we are examining currently. To handle the cases, we employ the predefined tactic `cases_l`. This tactic finds that the second case-part is the appropriate one, and discards the first case-part, leaving us with

$$\exists Y' : nat. X_0 = Y' + 1 \wedge natLessEq^{\kappa'}\ X_0\ Y', \quad \kappa' < \kappa\ \vdash\ .$$

We proceed by removing the existential quantifier (`exists_l`), splitting the conjunction (`and_l`), and substituting $X_1 + 1$ for X_0 in the rest of the formula (`eq_subst`). This gives us the goal

$$natLessEq^{\kappa'}\ X_1{+}1\ \ X_1, \quad \kappa' < \kappa\ \vdash\ . \tag{2.2}$$

The goal 2.2 is an instance of the earlier node 2.1, given the substitution $\sigma = id[X_1/X_0]$. Furthermore, a least fixed point has been unfolded to the left of the turnstile (shown by the in-equation $\kappa' < \kappa$). This means that we may successfully apply the command `discharge1`, which finds the companion node (i.e., the node 2.1) and discharge the current goal against it. All goals have now been taken care of, and thus the proof is finished.

The proof-graph generated by these proof-steps is shown in Figure 2.4, as displayed by the DaVinci graph-visualiser. The arrow drawn with a broken line, represents the relation between the discharged node and its companion node.

2.4 Related work

There exists a great deal of interest in using formal methods for programs, and specifically for various parts of the Java family. In this section an overview of related work is given, both in the types of formal methods that are used and different approaches to formalising Java and the JVM.

2.4.1 Formal methods

As stated in the introduction, there are many different methods included in the collection known as formal methods. Here we give some more insight into some distinctions, with comments on when different approaches can be used.

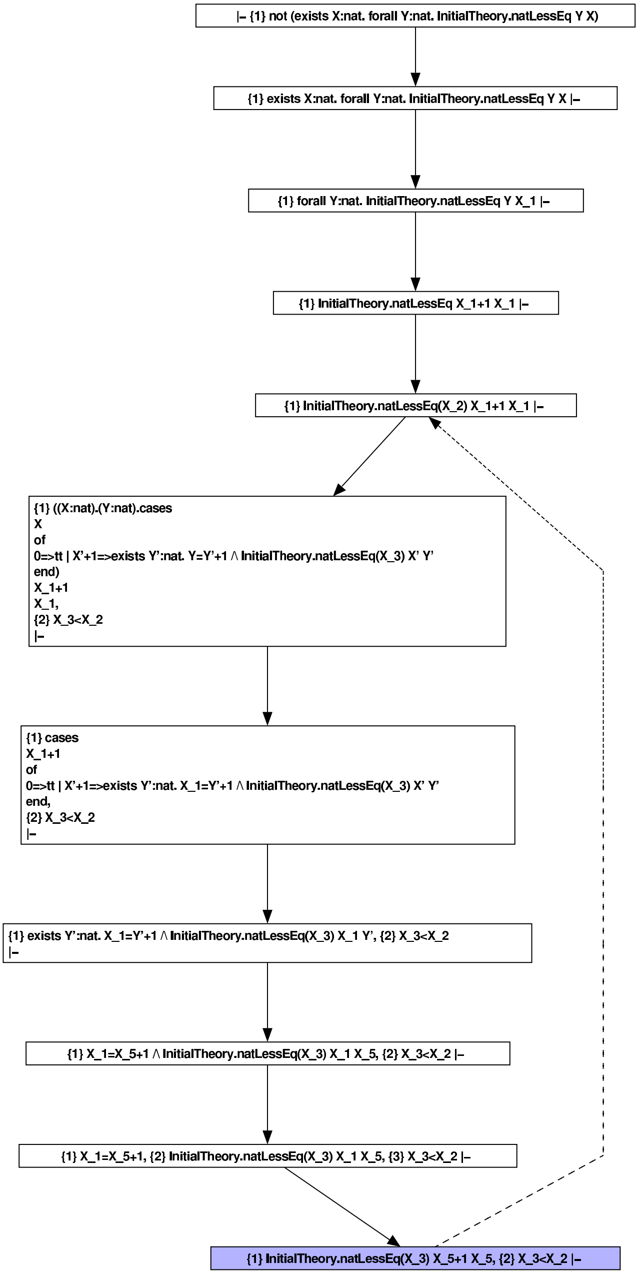
|– {1} not (exists X:nat. forall Y:nat. InitialTheory.natLessEq Y X)
{1} exists X:nat. forall Y:nat. InitialTheory.natLessEq Y X |–
{1} forall Y:nat. InitialTheory.natLessEq Y X_1 |–
{1} InitialTheory.natLessEq X_1+1 X_1 |–
{1} InitialTheory.natLessEq(X_2) X_1+1 X_1 |–
{1} ((X:nat).(Y:nat).cases
X
of
0=>tt | X’+1=>exists Y’:nat. Y=Y’+1 ∧ InitialTheory.natLessEq(X_3) X’ Y’
end)
X_1+1
X_1,
{2} X_3<X_2
|–
{1} cases
X_1+1
of
0=>tt | X’+1=>exists Y’:nat. X_1=Y’+1 ∧ InitialTheory.natLessEq(X_3) X’ Y’
end,
{2} X_3<X_2
|–
{1} exists Y’:nat. X_1=Y’+1 ∧ InitialTheory.natLessEq(X_3) X_1 Y’, {2} X_3<X_2
|–
{1} X_1=X_5+1 ∧ InitialTheory.natLessEq(X_3) X_1 X_5, {2} X_3<X_2 |–
{1} X_1=X_5+1, {2} InitialTheory.natLessEq(X_3) X_1 X_5, {3} X_3<X_2 |–
{1} InitialTheory.natLessEq(X_3) X_5+1 X_5, {2} X_3<X_2 |–

Figure 2.4: The proof graph

For a general introduction to formal methods and their use for specifying and verifying properties of systems, the textbook [Pel01] by Peled is a good place to start.

Light- or Heavy-weight The distinction between light-weight and heavy-weight methods is not a clear one – there is a whole range of possibilities. On one end of the spectrum, a formal method may be just writing down a behavioural specification in a sufficiently precise manner to accommodate some control over the system. On the other end, we have methods that are designed for full-system modelling and verification, with all aspects handled. The distinction is as much a technical one as it is a managerial one, in that it heavily reflects the perceived difficulty of using the method in the software development process.

A prominent example of the light-weight method are the simple applications of assertions in code. This method is used by many practicing software developers, and can be applied where needed. This common work-tool originates from the axiomatic approach to verification[Hoa69], where it is used as a full-featured approach for proving properties about programs. On the other side, the system developed in this thesis may be classified as a heavy-weight method. It models complete systems and allows for the verification of extremely detailed properties, it is very formal, and it requires a specialist to use it.

Terminating or Reactive Systems In the early days of formal methods, the emphasis was on studying terminating systems. A program was commonly viewed as a simple input-output relation. An example is [Hoa69], where the axiomatic semantics of programs are described. In this style, we write a specification for a program P as $\{Pre\}P\{Post\}$, for some pre- and post-conditions. The statement has the interpretation, that if P is started in a state that satisfies Pre, and if it terminates, then the state in which it terminates will have the property $Post$. Syntax directed rules are then given for inferring such relations.

This style of input-output reasoning is less relevant when we want to study reactive systems – systems that work in cooperation with their environment, where termination often is a sign of failure. For these systems, we need a different approach. A common approach is to specify that a program has a certain property, expressed in some form of modal logic. For more information on this kind of reasoning, see [BCMS01].

Expressible properties It is not enough that we can model our system and show that is has a property, we must also be able to express the properties we are interested in verifying. One starting point is the standard first-order logic. There are many natural things that can be expressed in this logic, but we can not talk about infinite structures such as the runs of a reactive program. Second order logic, on the other hand, is more powerful and many more statements are expressible. But the logic is so powerful that it is not feasible to handle in a mechanised way.

Different logics dealing with the trade-off between expressiveness and possibility to handle has been proposed, some for the explicit purpose of handling program-properties, some for the purpose of researching the boundaries between first- and second-order logic.

Two examples of logics for specifying properties over programs are LTL and CTL*. These logics are both temporal logics, they deal with time. They differ in their view of time; is time a linear series of events, or is it a branching structure of possibilities[Lam80]. In LTL (Linear Temporal Logic) we can quantify over runs over the program. Using the standard modal operators we can give always- and sometime-properties. In addition, there is also an until operator, written as $\phi U \psi$, which states that ϕ is true until ψ is true, and that ψ must be true sometime in the the future. For more information on LTL, see [Pel01]. In contrast, CTL* (Computation Tree Logic)[EH83] is a logic where we can quantify over the possible paths a program can take, as opposed to the sequence of states in a run of the program. In CTL* we have the operators O (in the next state), U (same as in LTL), and E (existential quantification over paths). Other operators are then derived, such as $F\phi \triangleq \top U \phi$ and $G\phi \triangleq \neg F \neg \phi$. Using these operators, we can state properties such as $EGF\phi$, which can be frased in english as *"along some path from here, ϕ will hold infinitely often"*.

The modal μ-calculus, which is used in this thesis, is a branching-time logic. The logic subsumes many other logics used for specifying properties about programs, but it is not so powerful that we get the same problems as in second-order logic. As an example of the expressiveness, it is easy to express halting in μ-calculus (Example 2.2.8). It is also possible to express properties that only apply to the fair executions of an unfair system (Example 2.2.9). There is a linear time variant of the μ-calculus, but it is not as interesting as the branching time version.

A big problem with μ-calculus is readability; when alternating fixpoint definitions are nested and interdependent, the formula can be very hard to understand (cf. Example 2.2.9). One possibility is to use a simpler logic such as CTL* for most of the specifications, translating the formulae into μ-calculus when we want to prove something, and to use μ-calculus formulae when we need something above and beyond the chosen base-language. This is possible since CTL* is a subset of the μ-calculus, as can be seen in [Dam92] where a direct translation from CTL* into μ-calculus is given.

The power of the language that we express our properties in also influences the model that we talk about. Petri nets are a model for concurrent systems, where markers move between containers. Petri nets do not have the ability to simulate Turing machines – for that they would have to test for the case of no markers in a container, which is impossible for a Petri net. But if we use a μ-calculus formula for a property of a Petri net, we can restrict our attention to the possible "executions" of a Petri net that respect the needed test, and this restricted version will simulate a Turing Machine. See [BS01] for some further discussion about this.

One interesting alternative approach for expressing the desired behaviour comes from process algebra. There, it is common to use properties of the form *"is the* *same as* `Spec`*"*, where `Spec` is a system in the same language that the system we reason about. The method of determining if two systems are the same varies. The most commonly used method is bisimulation, which roughly states that the two systems must be able to simulate each other. See [Mil89] for more information about bisimulation equivalences and its uses. For more information about the possible values of *same* (including bisimulation), see [Gla01].

Finite or Infinite A prominent distinction is the difference between finite and infinite systems, or rather between systems of bounded or unbounded size.

When a system is finite, such as an abstract protocol, we can model it as a simple finite state machine. If the number of states is low enough that we can handle it in a reasonable amount of time, we may test the model exhaustively, this is called model-checking. An example of a test is if the system ever deadlocks. A more interesting test is whether it is fair, e.g., do we necessarily progress? An interesting property may be specified in some sort of modal logic, and then the test is if the model of the system is a model of the given property. This kind of verification is very useful, since it is automatic. If a system is to large we sometimes may be able to simplify it, by abstracting away unnecessary clutter, but still retain a model that contains the essence of the system.

An example of a program and a method for modelling finite systems and performing tests on them is the SPIN model checker[SPI]. With this tool the user can code a finite model of the system in a programming-language with threads, called Promela. The model can be tested against LTL formulae, that it does not deadlock, and that it does not live-lock (by annotating certain points in the program as progressing points). It is possible to test all executions or just the fair executions.

An unbounded system is intrinsically hard to handle. For some systems, the interesting properties can be abstracted out as a finite system, and is thus subject to model checking. Furthermore, some simple classes of infinite systems can be handled automatically. See [BCMS01] for more information on such systems and automatic verification of properties of them. When these two escapes fail, we must use manual methods that are tailored towards handling unbounded systems, such as proof-assistants. For these methods, it is important to raise the level of discourse to the conceptual level, abstracting away from technical details. For example, small sub-problems that arise when dealing with an unbounded system may be sufficiently simple to be automated.

Abstract or concrete model What do our formal methods apply to? There are methods that exists solely as specifications in an abstract world, and those designed to deal with the actual implementation, or even to be integrated into the implementation.

A fully abstract method is the method of process algebra, where we model concurrent systems and prove theorems about them. These methods are useful for describing the abstract view of the system. On the other end, there are systems such as ESC/Java2[ESC], which employs a specialised system of comments for formal specifications. These specifications may either be compiled as assertions into the code, or they may be used as guides for attempts at verification.

2.4.2 Formalising Java and the JVM

Many different formalisations of Java and the JVM exist, with different amount of details and with different goals. In this section some different approaches, with different objectives for their formalisations, are presented.

Bali The Bali project[Bal] is aimed at formalising both Java and the JVM in the Isabelle theorem prover[Isa]. The formalisation of the JVM has roughly the same instructions as the ones this thesis includes, but lacks threads. On the other hand, they include object hierarchies and exceptions. The main use of the Bali model has been in proving properties of the bytecode verifier.

KeY The KeY project[KeY] integrates a dynamic logic of the Java Card language into a commercial Computer Aided Software Engineering (CASE) tool. Specifications for classes are written in the Object Constraint Language, and these constraints are subsequently translated into dynamic logic as proof obligations on the programs. The aim of the project is to enable formal reasoning about programs in an environment familiar to software developers.

M5 The M5 model of the JVM[M5] is constructed in the logic of the ACL2 theorem prover[ACL]. The name ACL2 stands for A Computational Logic for Applicative Common Lisp, reflecting the fact that a model in ACL2 is a Lisp program. Since it is written in Lisp, the model is also essentially an implementation of the JVM. The ACL2 system has mainly seen use in formalising different processors or parts of processors.

The M5 model includes threads, bounded arithmetic, and object hierarchies, but not exceptions. The aim of the formalisation is to be a relatively complete and very concrete model of the JVM. For example, they model the `aload_0` up to `aload_5` instructions as well as the generic `aload` instruction (which takes an integer argument), instead of translating the occurrences of `aload_`i into the `aload` instruction with i as an argument.

Abstract State Machines In [SBS01], Börger et al. present an Abstract State Machine of both the Java language and the JVM. The description aims at being a complete specification of the language and the virtual machine, with a definition of a compilation of the language into byte-code. The specification is also and executable model of both Java and the JVM. However, the JVM specification does not deal with threads, although the authors claim that this is no problem, and their approach from the Java formalisation can be copied into the JVM formalisation.

A main part of the book deals with the verification of the correctness of the compilation specified from the Java model into the JVM model, and of various issues regarding type-safety and the bytecode verifier. An interesting result is that a compiled Java program will pass the bytecode verification and will execute without runtime errors. This means that an offensive and a defensive JVM will behave in the same way on a well-formed input program.

ESC/Java2 The ESC/Java2 system[ESC] is an Extended Static Checker for Java originally developed by Compaq, but is now maintained and developed by Nijmegen University. The goal of the ESC/Java2 system is not to be a formalisation of Java, but to support the practical usage of axiomatic semantics and static checking in Java. The program works by using annotations in the code and automatic theorem proving techniques, to catch and detect many common errors.

By extending the annotations to be sufficiently complete, it is possible to verify programs using pre- and post-conditions.

Applet Interaction The Applet Interaction model by Barthe et al.[BGH02] is an abstract semantics of the call-graphs of applets that are loaded onto JavaCard smart-cards. This model has been embedded as a theory in VCPT. This embedding can be used for verification of interaction-properties of the applets, such as verifying that an applet will not call other applets that it is not supposed to call.

Model Checking Java The are various systems that are designed to extract finite-state models of Java systems, in order to model check them. Two such examples are Bandera[HP00] and Java PathFinder (JPF)[CDH$^+$00]. These two systems are both capable of exporting the finite-state model constructed as a model in the Promela language, which may subsequently be used by the SPIN model checker[SPI].

Featherweight Java Featherweight Java[IPW01] is an extremely small subset of the Java language, including only class declarations containing method definitions, object creation, sub-typing, casting, return statements, and field access. The purpose of this very small language (which is equivalent in computational power to a Turing machine), is to study the type system of Java and in particular possible extensions to the type system. The goal of the subset is to be as small as possible, without sacrificing the essential and interesting properties of the Java type system.

2.4.3 The position of this thesis

In the land of formal methods and of models of Java and the JVM presented, where does the work in this thesis fit in? What characteristics does it have, and what close relatives are there.

The environment of the model in this thesis is a proof-assistant tool. Thus, we have a heavyweight method; it is not intended for the casual application of the method to a real-world problem. On the other hand, we have a fairly concrete model, which do correspond directly to real-world programs. The specification logic of the tool, the modal μ-calculus, is very expressive, and is capable of formulating many interesting properties.

It is interesting to reason about the JVM, instead of directly about Java, since the JVM is a platform for many languages, despite its name. This means that an effective proof system for the JVM would not only facilitate proofs about Java, but also about the many other languages that can be compiled to the JVM, such as C, Python, Smalltalk, Groovy, Nice, etc.

The model of the JVM that is developed in this thesis includes threads, which is not common in formal models of Java. Most models are concerned with either questions about the type-system of Java, or with extensions of the base language. The M5 model is the only model of the JVM in a framework for infinite processes that does incorporate threads. As a comparison, the style of our model is most closely related to the Bali model, while in purpose, it is most closely related to the M5 model.

Chapter 3

Basic formulae and tactics

In this chapter a guide to some of the basic and more generic formulae developed for the formalisation is given. The style of development for both formulae and tactics presented in this chapter is used for all of the work presented in this thesis.

There are formulae to handle lists accompanied with tactics for these. Furthermore, some tactics for computing with natural numbers are presented, and some simple and general simplification-tactics. Most of the formulae and tactics are used in the subsequent development of the operational semantics of the JVM in the following chapters.

3.1 Automating computation

The arithmetic system in VCPT is based on a unary representation of the natural numbers. That is, they are represented using the constant zero and the successor function. A definition for the sort *nat* (if it were not already defined in the tool with nicer syntax), would be the following.

```
datatype nat = zero
             | succ of nat
```

It is easy to see that `zero`, `succ(zero)`, etc. can be viewed as the natural numbers. In the following, we will write the decimal number 2 instead of the verbose `succ(succ(zero))`, as is done in VCPT. We may form numbers by writing the corresponding constant down, or by taking any number (e.g. a variable of the sort *nat*) and adding a constant to it. Adding a constant to a variable represents the repeated application of `succ` to the variable.

The unary representation of numbers is a very easy system for most parts, but it is not very efficient. There are a number of formulae defined in the initial environment, that describe computation with numbers in this representation. There are formulae describing addition (`natPlus`, see Figure 3.1), subtraction (`natMinus`), and multiplication (`natTimes`). Furthermore, there are also predicates for the rela-

```
natPlus : nat → nat → nat → prop ⇐
  λX:nat. λY:nat. λZ:nat.
  cases X of
    0 ⇒ Z=Y
  | X'+1 ⇒ ∃ Z':nat. Z=Z'+1
                    ∧ natPlus X' Y Z'
  end
end
```

Figure 3.1: The definition of the predicate `natPlus`.

tions less than (`natLess`) and less than or equal (`natLessEq`, see Figure 2.3). These formulae are all fine for describing the properties of arithmetic logically, but the use of them in any proof requires a lot of manual labour. In order to ease this problem, tactics automating these formulae for some common cases have been developed.

3.1.1 Cases handled

The cases handled for the arithmetic formulae are simple true-false evaluation of concrete data and calculation of values into existential variables.

With concrete data, such as a formula expressing the fact $2+3=5$ (written as `natPlus 2 3 5`), we can apply the corresponding tactic `natplus_`*s*, with *s* either `l` or `r`. On the right hand side, the result is a successful termination of the goal; on the left hand side, the result is elimination of the formula. If the formula is false, such as `natPlus 1 1 3`, the results would have been switched (termination on the right hand side, elimination on the left hand side).

When the third argument to a formula in the left hand side is an existential variable or a meta-variable, the tactic will compute the value of the expression, and substitute the value into the variable. When on the right hand side, a meta-variable (not just a variable) is required to successfully compute the value.

The case handled for the relations formulae is the concrete case, i.e. when both values are concrete numbers. The tactics will compute the truth of the formula, with appropriate results based on which side and if it is true or false. The tactics will also work when the arguments are sufficiently concrete, e.g. a formula akin to $natLess\ 2\ X{+}3$ will be correctly identified as a true formula.

3.1.2 Implementation

The implementation of these tactics is for most parts straightforward. As a representative example, we will present the handling of addition. Compared to the formula in Example 2.2.5, the definition of addition is somewhat simpler since it uses the cases-construction. The definition of `natPlus` used in the tool is shown in Figure 3.1.

```
fun natplus_r pos =
    t_fix () (fn _ => (fn recurse =>
        t_compose_l
            [unfapp_r pos,
             cases_r pos,
             t_orelse_l
                 [exteq_r pos,
                  eq_flat_elim_r pos,
                  t_compose2
                       (exists_r NONE pos)
                       (t_compose
                            (and_r pos)
                            [exteq_r pos,
                             recurse()
                             ])]]))
```

Figure 3.2: The definition of the tactic `natplus_r`. This tactic automates computation of addition to the right of the turnstile.

The tactic for handling `natPlus`-formulae to the right is shown in in Figure 3.2. The tactic unfolds the formula and applies the arguments (see Section 3.3 for a description of `unfapp_r`), after which it uses the cases-tactic. We use `t_orelse_l` to try the different alternatives. Either we have bottomed out on the first argument, which means that we can test if the predicate was true (by using `exteq_r`, to check for existential equality to the right), or if it was false (by using `eq_flat_elim_r`, to see if the equality has reduced to `zero=succ(W)` for some variable or constant `W` of sort *nat*). If the second case was the correct, the two previous tactic-applications will fail, since the top-level construction is then an existential quantification. We remove this and introduce a fresh meta-variable by applying `exists_r NONE`. Using `exteq_r` we let VCPT deduce the correct binding of the meta-variable introduced earlier, and bind the variable to it. The remaining recursive occurrence of the `natPlus`-predicate, with decreased arguments, is handled by recursively invoking the tactic. If the recursive invocation of the tactic fails, then the whole tactic fails.

As stated earlier, there are tactics for plus and minus, less than, and less than or equal, for both sides of the turnstile. The implementation of these tactics all follow a similar pattern as the `natplus_r`-tactic. The tactics that handle computation to the the left of the turnstile, all take an argument for the recursive `t_fix` construction, that specifies an offset from the original argument `pos`. For an example of how this can be programmed and used, see Example 2.3.2.

3.2 Managing lists

Lists are a simple but useful general way of structuring data, and it is available in VCPT. The basic operations on lists are appending a node at the front, and accessing the first node or the tail (which is a list). There is also some syntactic sugar for accessing or appending any constant number of elements to the front of the list. From these basic constructs several useful operations has been developed, described as predicates matching input with output.

The handling of these predicates is automated for some common cases, so that they may be used more easily. The tactics assume that the first arguments for the predicate are "input arguments" and the last (for almost all predicates) is an "output argument". This means that for the first arguments concrete data must be supplied, while for the last argument it can be either a concrete value or a value with some parts (or the whole) as a variable.

A serious problem in constructing these predicates is that lists are polymorphic, and predicates are not. This has been solved using the simple macro expansion capabilities of the C pre-processor (CPP). To use these predicates in a theory, a user can include their definitions in a file, and instantiate them for every desired sort of element, accompanied by a suitable suffix for the instantiated predicates. The suffix is needed, since there is no overloading in VCPT. For example, to handle lists of natural numbers (the sort *nat*), and to refer to the predicates using the suffix `_n`, an instantiation

```
arraypredicates(_n, nat);
```

will create all the necessary predicates called `at_n`, `set_n`, et.c.

In the following discussion we will refrain from writing these suffixes and from worrying about the sort of data in a list, assuming the sort *elem* for the elements of a list. The predicates assume that the node-elements are from a flat datatype, meaning that simple syntactic equality is sufficient for determining semantic equality. For a full listing of the definitions of the predicates, see Appendix B.

Array access to elements Using the `at`-predicate, accessing the elements of a list indexed by a natural number is possible. The definition of this predicate is given in Figure 3.3. We see that the predicate works by recursively decrementing the index by one, accompanied with popping the head of the list, until we bottom out at index zero. At this point, a check is made to ensure that the top of the list is equal to the desired element. As an example, the sequent $\vdash$ *at* $[1, 2, 3]$ 1 2 is valid, since the element at place one in $[1, 2, 3]$ is 2.

To modify a list, we can use the `set`-predicate. The predicate has the type *elem list* → *nat* → *elem* → *elem list*, where the first argument is the original list, and the last argument is the new list. The implementation of the predicate works along the same principles as the `at`-predicate. There is also a `set_from`-predicate, with the type *elem list* → *nat* → *elem list* → *elem list*. It is used to set the values

```
at : elem list → nat → elem → prop ⇐
  λL:elem list. λN:nat. λE:elem.
    cases (N,L) of
      (0,[E|Tail])          ⇒ ⊤
    | (N'+1,[Head|Tail]) ⇒ at Tail N' E
    end
end
```

Figure 3.3: The `at`-predicate.

in the first argument to the values in the third argument from the position of the second argument.

Manipulating lists To use lists effectively, we need some additional predicates to ease the use of them in the semantics. In this category, we have `size`, `extend`, `remove`, `reverse`, `split`, and `split_rev`.

The `size`-predicate describes the length of a list. The `extend`-predicate is used to extend one list with a new element at the end of the list. Using this predicate, we can do some simple operations on lists, treating them as queues. Using `remove`, we can remove an arbitrary element indexed by a natural number from the list.

The `reverse`-predicate is used to reverse a list. The naive construction of this predicate iterates through its first argument, and appends the head value to the result (by using `extend`). The problem with this approach is that it is quadratic in complexity. Using an accumulator for the result, the predicate achieves linear complexity with a slightly more complicated definition.

To split a list into two lists, the first list of a specific length (given as an argument to the predicate), we can use the predicate `split`. To accommodate a later need, the slightly unorthodox predicate `split_rev` has also been developed. This predicate works in the same way as the previous one, but will in addition reverse the first part. This predicate is both in theory and implementation a combination of `split` and `reverse`.

Key-Value search It is not always wise to use placement indexes into a list for referencing information. Using the `contains` predicate, we can search in a list of pairs of keys and values, for the correct value for a specific key.

This predicate takes more arguments than the others do, when instantiated using CPP. These additional arguments specify the types of the key and the value, as well as the constructor used to form pairs (if any; if left empty it will become the standard pairing mechanism).

3.3 Simplification

An important aspect of Gentzen-style systems combined with backwards reasoning, is that they naturally lend themselves to many automatic simplifications. Essentially, many of the proof rules where no choice has to be made (e.g. no witness is to be chosen, no formulae is to be introduced, no substitution is made) are safe to apply to a goal, and will result in a simpler goal. In the spirit of this, a simple simplifier that applies these safe rules to all formulae in a sequent has been constructed. The tactic will be applied recursively to all resulting goals, as long as some progress can be made. Furthermore, tactics for unfolding a predicate and applying the arguments has been added.

The rules applied in the simplification are the axioms; removing negation; splitting disjunction, conjunction, and implication; the case and if-then-else tactics; deconstructing flat data; handling universal quantification and box-modalities to the right and existential quantification and diamond-modalities to the left. Furthermore, a user may supply additional tactics that should be used in simplification upon invocation of the simplifier. These further tactics are supplied in two lists to the simplifier, each consisting of tactics that take an integer argument as the position that it should be applied to. The difference between the two lists, is that the tactics in the left list will be applied to the left of the turnstile, and the tactics in the right list will be applied to the right of the turnstile.

A common task is to unfold a definition, and to apply some arguments to it. This has been collected into a single tactic, called `unfapp_`s, with s either `l` or `r`. The tactic takes an argument specifying where it is to be applied. The tactic will not unfold/apply arguments to a modality, since these formulae are better handled by the specialised box and diamond tactics.

Example 3.3.1 *To illustrate the use of these tactics, consider the goal*

$$\vdash natLessEq\ 10\ 20.$$

To prove this goal, we can use the tactic invocation

```
simplify_tacs [] [unfapp_r, exists_r NONE, exteq_r].
```

In addition to the rules the simplifier applies as standard, this invocation will also do the following. Any definitions will be unfolded and their arguments (if any) be applied, existential quantifiers will be removed and meta-variables introduced (delaying the choice of a witness), and equality-checks will be done while allowing the binding of meta-variables in order to achieve equality. The tactic `exists_r` *takes two arguments normally. In this example, the first argument is bound to* `NONE`*, resulting in a tactic that takes one argument.*

While this invocation works and proves the goal, it is slower than using the specialised tactics from Section 3.1. Furthermore, the simplifier will affect all of the formulae in a sequent, which is not always desirable.

3.4 Discussion

The simplifier is a very handy tool for removing much irritating basic work from proving goals, and can even prove simple goals automatically (for example, the example in Section 2.2.4 would be handled directly using the simplifier). There are many features that could be added to make the versatility greater, such as an option to localise the simplification, where only one formula in a goal would be touched.

For a simplifier to be really useful, we would also need a much more suitable framework for adding simple lemmas that can be used in the simplification process, such as the facilities that are present in Isabelle[NPW02].

The usage of lists in a theory has many more applications when coupled with some predicates to describe more advanced features such as the ones presented above. These predicates work very well for the case of concrete data, and for some other cases also, when the arguments are “sufficiently concrete”.

One problem is when we want to prove a complex statement, which often requires reasoning on a higher level than these predicates and tactics allow. To enable this, a library of general lemmas could be used, especially if the simplifier could use such a library efficiently.

Are these tactics developed good tactics? They handle the case of concrete data well. On the other hand, when the arguments are undetermined we may get an infinite loop in the tactic, which of course is very bad behaviour. This behaviour is especially bad when we want to use the tactics in larger tactics, e.g. to handle addition in the JVM.

To make the tactics into full-fledged tactics for all occasions, checks must be added so this bad behaviour is detected and fails the application of the tactic.

Chapter 4

The Java Virtual Machine

The Java Virtual Machine (JVM) is the platform used by, and designed for, the Java language to enable platform independence. A general definition is given in a book by Lindholm and Yellin[LY99]. It is a stack based machine with a rather direct encoding of the object system and heap model of Java. For example, there is an instruction `invokevirtual` that is defined to handle all the intricacies of virtual method-invocation.

In this chapter an overview of the structure of the JVM is given, along with the main points that are relevant to the developed semantics.

4.1 Structure

The JVM is a virtual machine that executes a set of classes. As a virtual machine, the JVM specifies the instructions available for the code, and their semantics. The semantics of a program is not solely determined by the definition of the instructions, however. There is also a concept of native methods, methods that are implemented outside the JVM. These methods may change the configuration of the JVM in ways that are not possible to express using the instructions. For example, there is no way to start a new thread of computation in the JVM without using a native method.

The instructions of the JVM are high-level regarding the method-calls and object-handling, and low-level regarding the actual computations performed. This makes it interesting to study in a formal setting, since a compiled program will contain all of the high-level logical structure, such as methods and classes, but no complex data operations. Since the structure of the program is explicit, it simplifies the task of compositional reasoning.

The JVM operates on elements of data that are of either single (32-bit) or double (64-bit) size. This is in contrast to the Java language, which has booleans (conceptually single bit values), bytes (single byte values), and so on. The data may be either integers or floating point values.

Dynamic allocation of memory is done by allocating new instances of classes.

There is no explicit deallocation possible, instead a JVM must perform garbage-collection when more new memory is needed.

4.2 What is a program?

The JVM platform does not have the traditional concept of a program. Instead, a JVM will start with an initial class to load. When a class is loaded, the JVM will parse the class-file and load all definitions. When an unloaded class is referenced and needed, it will in turn be loaded and added to the loaded classes. After the loading of the initial class is done, the static method main in the initial class is invoked in an initial thread.

The state of a running JVM consists of the loaded classes, a heap of dynamically allocated data, and the running threads. Each thread has an associated class and a list of activation-frames for the invoked methods, the call-stack.

Any started JVM will contain some instances of the class `java.lang.Class`, where each instance represents a type of a loaded class. These instances are created automatically when a class is loaded.

4.3 What is a class-file?

A class file contains all the definitions necessary to describe one class for the JVM. The class file begins with a constant pool and some class information. After this, there are lists of interfaces implemented, fields in the class, methods in the class, and attributes associated with the class.

The *constant pool* contains a list of all the constants used in the definition of the class. Examples of constants are string values, numeric constants, and class- and method-names. The use of a constant pool enables the subsequent definitions in a constant pool to use indexes into the constant pool instead of in-lining them. A main benefit is in simplifying the structure of the class file.

The *class information* in a class file, specifies the super class of the class (except if it the class file for the class `java.lang.Object`) and the class that the file specifies. Furthermore, the access modifiers of the class are given (`public`, `protected`, etc.). The *interface list* of the file lists all the interfaces that are implemented by the class.

The *list of fields* has information about the types, access modifiers, and names of all the fields that are declared in a class. The *list of methods* is a list of all the methods a class contains, such as instance methods, static methods, and instance initialisation methods (i.e., constructors). If a method is not native or abstract, the description of the method includes the byte-code of the method.

The *attributes* of a class gives additional descriptive information about the class. An attribute is not allowed to alter the semantics of a class.

4.4 Parallel computing

Parallel computing inside the JVM is based on threads. These threads are instances of thread-objects, that are managed via an API implemented as native methods. This is an example of extensions to the base machine that reside in a library.

A thread object is created when a class is instantiated that is a subtype of the `java.lang.Thread`-class. This allocation does not initiate a new thread, that is done when the native method `start()` is called on the created object. This invocation will create the thread of computation, and call the thread-objects method `run()` as the initial method of the class. The thread class also provides a constructor to wrap an instance of a class implementing the `java.lang.Runnable` interface. This is a convenience done to support running classes as separate threads, when the actual class has to inherit from another class than the thread class.

Memory model An important part of any parallel implementation is the memory model that is used. A memory model specifies how the memory will behave, when concurrent reads and writes of memory are performed. The specification of the memory model of the JVM has changed from the original specification in the new Java 1.5 release of the JVM. The new memory model, reported on in [MPA05], fixes some problems with the earlier specification. The main promise the new memory model gives, is that for a *correctly synchronized* program, the memory will show *sequential consistency.* A correctly synchronized program is a program that does not contain any data races. A data-race is defined as conflicting accesses to a variable by multiple threads, where the accesses are not ordered. The guaranteed property of sequential consistency, introduced as a memory model by Lamport in [Lam79], means that the memory accesses in a program conceptually occurs in some total order (an order that does not contain parallel accesses to memory) and that the order of memory accesses from a thread is the same as the order that they occur in the program. This notion, when applied without the restriction of the program being correctly synchronized, is a very strong property that does not admit many important optimizations. The benefit of the model is that it is a convenient model for the programmer to reason about. With the restriction of correct synchronization added, the designers of this new memory model argue that there is more of a balance between convenient semantic guarantees and the optimizations that are allowed.

Locking and synchronizing The JVM platform has support for a limited version of the concept of monitors[Hoa74]. There is inherent support for synchronizing the invocation of methods on an object, using a mutual exclusion lock that is associated with the object (this lock is sometimes called a monitor in the literature on the JVM), via a flag associated with the method. The lock may also be locked at will, by issuing a `monitorenter`-instruction (this is a done, for example, when a synchronized-block in Java is compiled).

The condition variables that are commonly used with monitors are also present in Java, although there is only one such variable per object. This variable is included with all objects, and is present implicitly (not as an actual Java variable). The semantics of the condition-variable is specified through native methods that are defined in the Object class. The methods available are `wait()`, `notify()`, and `notifyAll()`, corresponding to the wait, signal, and signal all operations on condition variables.

Modes for threads When a thread is created, it is initially unscheduled. This means that it will not be eligible for execution. The thread enters the pool of thread that can be executed, the scheduled threads, when the `start()` method of the thread is called. When a thread is waiting for a mutual exclusion lock, it is blocked, and can not be executed. When the thread is waiting for a condition variable, it is said to be waiting. A thread that has finished execution is said to be terminated.

Scheduling The actual scheduling used by a JVM is largely up to the implementation to decide. The JVM specification does not even require the presence of time-slicing. There is a priority associated with each thread, that is an integer between one and ten. The only promise made, is that the thread with the highest priority will at some time get to run.

The purpose of such a loose specification is to allow the implementation of the JVM on severely constrained platforms, where time-slicing is not a feasible solution. However, many of the actual JVM implementations that exist use more advanced scheduling techniques, including the use of time-slicing. The result is, that it is easy to write a multi-threaded Java program, that does not work on all standard compliant platforms, just on the platforms that does time-slicing. For a multi-threaded application targeting the JVM platform, the programming of the multiple threads must be done in a cooperative fashion, where the threads (when performing long computations) yield to other threads occasionally.

Chapter 5

The Semantics of the JVM

In this chapter a formal operational semantics is given for a subset of the JVM. The subset chosen is presented, after which the representation of programs and the semantics are given. We conclude with a discussion about the semantics, possible alternatives and extensions, and about the missing features.

5.1 Subset chosen

The semantics only deals with a subset of the JVM, but it contains many essential features nonetheless. In this section an account of the chosen subset is given. We will assume that a program is a correctly synchronized program. This means that the execution of a JVM instruction is atomic, with the result that we can model the execution of the JVM as an interleaving of the instructions. The usage of such a semantic model will give a total ordering of the instructions executed. The M5 model of the JVM also makes this assumption, in order to guarantee that the interleaving model is appropriate.

The main reductions in complexity are a simple flat class-hierarchy (no inheritance and no interfaces), no exceptions, unbounded arithmetic, unsigned natural numbers as the only basic datatype, the system is static (e.g. no class-loading) and local (i.e. no distribution), and the only operation on threads is starting them. Furthermore, only a representative subset of the instructions are modelled. Below, we will elaborate on these points.

A class in this formalisation is simply an identity and the number of fields. An instantiation is a list of these fields in the heap. The class does not have any static data, and no static methods exist. Furthermore, the flat class-hierarchy allows a much simplified method-lookup.

The chosen basic datatype, unbounded natural numbers, is used for both references and integers. This means that we do not consider negative integers nor bounded arithmetic, where operations can overflow. Restricting us to unbounded numbers also means that we only have one size of integer. The datatypes that we do not model at all, are floating point values, character values, and arrays of basic values.

We model only static systems, i.e., systems where there is no dynamic loading of classes. We assume that all the classes that a system references are loaded at the start, and that no new classes are constructed during the execution of the program.

A thread in the subset is simply the list of activated frames. A thread can be started and can lock and unlock the locks associated with class instances. Waiting and notifying on the condition variable of an class instance is not supported.

The instructions modeled form a simple but representative subset of the instructions of the JVM. Instructions for allocating data, control-flow, managing threads, loading and storing data, and performing actual computations are present. Absent are many more operations on the actual data, instructions for working on different types of data, some stack-manipulations, instructions for referencing static values and methods, the differences between the different method invocation instructions, and such instructions as `instanceof`, `isnull`, `throws`, etc.

5.2 Representation of a program

The representation of programs is, for most parts, a simple construction using the memory layout suggested in the JVM specification[LY99]. A program, or a system, consists of a set of threads, a heap, and the constant-pool.

The *set of threads* is the currently active threads in the system that are eligible for execution, i.e. those threads that are not waiting for a lock. Each *thread* is a list of the frames for the methods called and an identifier linking the thread to the class instance it was started from. For a invoked method, the *frame* will contain the code for the method, a program counter, a value stack, and a list of local variables. We have a copy of the code in each frame to enable the local execution of method-local instructions without referring to the global constant-pool.

The *heap* is a list of the class instantiations that have been made and the associated data. Each *class instantiation* contains the identity of the class instantiated, a list of local variables (that is, the list of instance variables), a lock for the associated mutex, and a list of threads waiting for the lock to be released.

The *constant pool* is a set of definitions of classes and methods. A *class definition* contains the identity of the class and the number of member fields of the class. A *method definition* contains method identity, the code of the method, the number of local variables, and the initial value of the program counter. A *method identifier* contains the number of arguments to the method. Note that this constant pool is not the same as the constant pool described in Section 4.3. This constant pool is for a running JVM with possibly many classes loaded, while the constant pool in Section 4.3 is for a single class.

5.3 Operational semantics of the JVM

The operational semantics of the JVM is defined in two levels, one for the execution of single methods, designated $\rightarrow_m$, and one for the execution of complete systems, designated $\rightarrow$. The relations are of type `frame` × `action` × `frame` and `system` × `action` × `system` respectively.

The overall model is that the system transition chooses a thread to execute. The current frame in the chosen thread is then executed by the method transition. The result of this execution is processed by the system transition if it is something other than a simple computation.

The specification of the semantics use the formulae described in Section 3.2. We will disregard from the fact that there is no overloading defined for these, and use the generic names in the specification.

Method transitions The method-transition relation uses the labels on the transitions for communicating with the system-transition when the instruction executed requires it, while the other transitions use the label for silent actions. In Table 5.1, the transition system specification for the method transition relation is given. In part one of the table, the semantics for method-local instructions are given. These instructions operate solely on the state of the current frame, which is reflected in that the labels are all τ. In part two of the table, the instructions with some global effect are given. These instructions use the label on the arrow to communicate with the surrounding environment.

For the specification, we will use the following conventions. Identifiers starting with a capital letter are variables, and those starting with lowercase letters are constants or functions. For the variables, we let `CS` be some code; `PC` be a program counter; `N` a natural number; `Ref` a reference; `Val` and `V` be any value (a number or a reference); `VS` a stack of values; `LS` the local variables; `Args` a list of argument values; `MId` a method identifier; `ClsId` a class identifier; and finally `Idx` an index, which is some natural number. We will subscript or prime the variables to indicate that they are different entities, but of the same kind.

For the method-local instructions we will elaborate on the semantics of `iceq`, to be read as Integer Compare EQuality, as a representative example. The instruction is modelled after the JVM instruction `icmp_eq`, that pops two values and compares them for equality. The result of the comparison determines if a jump should be made or not. The semantics are specified using two rules, Iceq$_t$ and Iceq$_f$. These rules may be used when the element at index `PC` in the code is an `iceq` instruction (as specified by the *at*-formula premise, see Section 3.2 for a description of *at*). The first is applicable when the top two elements of the stack are equal, with the result that the program-counter is set to the argument of the instruction. If the top two elements are different, the second rule is applicable, with the result that execution continues with the next instruction. In both cases the values are popped from the stack. The rules for the other method-local instructions are also simple encodings of their behaviour.

$$\text{Push}\,\frac{at\ CS\ PC\ push(Val)}{\langle CS, PC, VS, LS\rangle \rightarrow_m \langle CS, PC+1, [Val|VS], LS\rangle}$$

$$\text{Pop}\,\frac{at\ CS\ PC\ pop}{\langle CS, PC, [V|VS], LS\rangle \rightarrow_m \langle CS, PC+1, VS, LS\rangle}$$

$$\text{Dup}\,\frac{at\ CS\ PC\ dup}{\langle CS, PC, [V|VS], LS\rangle \rightarrow_m \langle CS, PC+1, [V, V|VS], LS\rangle}$$

$$\text{Swap}\,\frac{at\ CS\ PC\ swap}{\langle CS, PC, [V_1, V_2|VS], LS\rangle \rightarrow_m \langle CS, PC+1, [V_2, V_1|VS], LS\rangle}$$

$$\text{Nop}\,\frac{at\ CS\ PC\ nop}{\langle CS, PC, VS, LS\rangle \rightarrow_m \langle CS, PC+1, VS, LS\rangle}$$

$$\text{IAdd}\,\frac{at\ CS\ PC\ iadd \qquad\qquad N_1 + N_2 = N}{\langle CS, PC, [N_1, N_2|VS], LS\rangle \rightarrow_m \langle CS, PC+1, [N|VS], LS\rangle}$$

$$\text{ISub}\,\frac{at\ CS\ PC\ isub \qquad\qquad N_1 - N_2 = N}{\langle CS, PC, [N_1, N_2|VS], LS\rangle \rightarrow_m \langle CS, PC+1, [N|VS], LS\rangle}$$

$$\text{Goto}\,\frac{at\ CS\ PC\ goto(N)}{\langle CS, PC, VS, LS\rangle \rightarrow_m \langle CS, N, VS, LS\rangle}$$

$$\text{Iceq}_t\,\frac{at\ CS\ PC\ iceq(N) \qquad\qquad N_1 = N_2}{\langle CS, PC, [N_1, N_2|VS], LS\rangle \rightarrow_m \langle CS, N, VS, LS\rangle}$$

$$\text{Iceq}_f\,\frac{at\ CS\ PC\ iceq(N) \qquad\qquad N_1 \neq N_2}{\langle CS, PC, [N_1, N_2|VS], LS\rangle \rightarrow_m \langle CS, PC+1, VS, LS\rangle}$$

$$\text{Iclt}_t\,\frac{at\ CS\ PC\ iclt(N) \qquad\qquad N_1 < N_2}{\langle CS, PC, [N_1, N_2|VS], LS\rangle \rightarrow_m \langle CS, N, VS, LS\rangle}$$

$$\text{Iclt}_f\,\frac{at\ CS\ PC\ iclt(N) \qquad\qquad N_1 \not< N_2}{\langle CS, PC, [N_1, N_2|VS], LS\rangle \rightarrow_m \langle CS, PC+1, VS, LS\rangle}$$

$$\text{Store}\,\frac{at\ CS\ PC\ store(Idx) \qquad set\ LS\ Idx\ V\ LS'}{\langle CS, PC, [V|VS], LS\rangle \rightarrow_m \langle CS, PC+1, VS, LS'\rangle}$$

$$\text{Load}\,\frac{at\ CS\ PC\ load(Idx) \qquad\qquad at\ LS\ Idx\ N}{\langle CS, PC, VS, LS\rangle \rightarrow_m \langle CS, PC+1, [N|VS], LS\rangle}$$

Table 5.1: The method transition relation part 1, method local instructions.

$$\text{Invoke}\ \frac{at\ CS\ PC\ invoke(\langle Id, N\rangle) \qquad split\_rev\ VS\ N\ Args\ VS'}{\langle CS, PC, VS, LS\rangle \xrightarrow{call(\langle Id,N\rangle,Args)}_m \langle CS, PC+1, VS', LS\rangle}$$

$$\text{Return}\ \frac{at\ CS\ PC\ return}{\langle CS, PC, VS, LS\rangle \xrightarrow{result}_m \langle CS, PC+1, VS, LS\rangle}$$

$$\text{VReturn}\ \frac{at\ CS\ PC\ vreturn}{\langle CS, PC, [V|VS], LS\rangle \xrightarrow{vresult(V)}_m \langle CS, PC+1, VS, LS\rangle}$$

$$\text{Getfield}\ \frac{at\ CS\ PC\ getfield(Idx)}{\langle CS, PC, [Ref|VS], LS\rangle \xrightarrow{get(Ref,Idx,V)}_m \langle CS, PC+1, [V|VS], LS\rangle}$$

$$\text{Putfield}\ \frac{at\ CS\ PC\ putfield(Idx)}{\langle CS, PC, [V, Ref|VS], LS\rangle \xrightarrow{put(Ref,Idx,V)}_m \langle CS, PC+1, VS, LS\rangle}$$

$$\text{New}\ \frac{at\ CS\ PC\ new(ClsId)}{\langle CS, PC, VS, LS\rangle \xrightarrow{allocate(ClsId,Ref)}_m \langle CS, PC+1, [Ref|VS], LS\rangle}$$

$$\text{Start}\ \frac{at\ CS\ PC\ start(MId)}{\langle CS, PC, [Ref|VS], LS\rangle \xrightarrow{spawn(Ref,MId)}_m \langle CS, PC+1, VS, LS\rangle}$$

$$\text{MEnter}\ \frac{at\ CS\ PC\ monitorenter}{\langle CS, PC, [Ref|VS], LS\rangle \xrightarrow{enter(Ref)}_m \langle CS, PC+1, VS, LS\rangle}$$

$$\text{MExit}\ \frac{at\ CS\ PC\ monitorexit}{\langle CS, PC, [Ref|VS], LS\rangle \xrightarrow{exit(Ref)}_m \langle CS, PC+1, VS, LS\rangle}$$

Table 5.1: The method transition relation part 2, instructions with global effects.

The instructions with global effect are inherently more complicated, since they are defined in cooperation with the system-transitions. The instructions are of different complexity. On one hand, the semantics for, for example, the `start` instruction are rather simple. The necessary information is collected (the starting method and the reference to the object), and a `spawn`-action is emitted. On the other hand, a more complex example is the instruction `new`. The action emitted when a `new`-instruction is executed is `allocate(ClsId, Ref)`, where `ClsId` is the identity of the class to allocate, and `Ref` is a reference that is "returned" by the environment of the frame executed. For a single method-transition, this value can be any value at all, since there is nothing that constrains the choice of value for it. But when we look at the rule in the system transition relation that allows `allocate`-actions to occur, we will see that it will only allow the method-transition that has a value for the reference corresponding to the index into the heap of the newly allocated class instance.

The Invoke rule has a premise that uses the formula *split_rev* described in Section 3.2. This premise is used to gather the arguments to the invoked method from the value stack. The arguments are also reversed so that they are in the right order for storing as local variables in the invoked method.

System transitions All system transitions will have the label τ (the silent action), reflecting the fact that we are dealing with closed systems that do not interact with their environment. The complete transition system specification is given in Table 5.2. In the first part of the table, transitions that model the execution of the non-threaded aspects are given. These rules could easily be adapted into a model for a single threaded JVM. In the second part of the table, the transitions that handle multi-threaded actions are given.

In the specifications we will, in addition to the conventions for the method-transitions, use the following conventions: `Th` is a thread; `Ths`, `Q`, and `QT` are lists of threads; `T` is a list of frames; `F` is a frame; `Hp` is a heap; `TId` is a reference into the heap; `CP` is the constant pool; `L` is a lock; `Fields` a list of values; and `I` is and index.

The first thing to note about these transition-rules, is that they have many premises. A suggested order of reading them is top to bottom, left to right, which will give an imperative style to the rules, if the variables are thought to be bound at their first occurrence (this is true in most cases). All rules have the initial premise $at\ Ths\ I\ [F|T]$. Since the index I is unbound, the premise is true for any choice of I such that the I:th element of Ths is a thread. In particular, it is only true for a finite amount of values, since lists are finite in length. This unconstrained scheme for choosing the thread to execute describes all interleavings of the atomic actions. This will for any specific run model some scheduler, while for questions about all runs it models all possible schedulers. All the rules match against the possible transitions of the top frame of the chosen thread.

Since these rules are more complex and varied than the rules for method-execution, we will discuss all of them. The descriptions use the imperative reading of the rules. It is important to note, that stepping a program according to one of these rules, is only done if the chosen thread can perform that action. Consider the Compute rule. We can not apply this rule, with a choice of a thread to execute, if the top frame of the thread does not have a τ-transition possible.

Compute Compute is a simple wrapper allowing a thread to execute method-local instructions. In the first premise, a thread is chosen for execution. In the middle premise the top frame of the thread chosen is executed using the method-transition. The modified thread (with a new top frame) is then "stored" using the `set`-predicate into a new thread list.

Call The Call-rule handles method-invocations. From the information about the

$$\text{Compute}\ \frac{\mathit{at}\ \mathit{Ths}\ I\ \langle \mathit{TId}, [F|T]\rangle \quad F \rightarrow_m F' \quad \mathit{set}\ \mathit{Ths}\ I\ \langle \mathit{TId}, [F'|T]\rangle\ \mathit{Ths}'}{\langle \mathit{Ths}, \mathit{Hp}, \mathit{CP}\rangle \rightarrow \langle \mathit{Ths}', \mathit{Hp}, \mathit{CP}\rangle}$$

$$\text{Result}\ \frac{\mathit{at}\ \mathit{Ths}\ I\ \langle \mathit{TId}, [F|T]\rangle \quad F \xrightarrow{\mathit{result}}_m F' \quad \mathit{set}\ \mathit{Ths}\ I\ \langle \mathit{TId}, T\rangle\ \mathit{Ths}'}{\langle \mathit{Ths}, \mathit{Hp}, \mathit{CP}\rangle \rightarrow \langle \mathit{Ths}', \mathit{Hp}, \mathit{CP}\rangle}$$

$$\text{VResult}\ \frac{\begin{array}{c} \mathit{at}\ \mathit{Ths}\ I\ \langle \mathit{TId}, [F, \langle \mathit{CS}, \mathit{PC}, \mathit{VS}, \mathit{LS}\rangle|T]\rangle \quad F \xrightarrow{\mathit{vresult}(V)}_m F' \\ \mathit{set}\ \mathit{Ths}\ I\ \langle \mathit{TId}, [\langle \mathit{CS}, \mathit{PC}, [V|\mathit{VS}], \mathit{LS}\rangle|T]\rangle\ \mathit{Ths}' \end{array}}{\langle \mathit{Ths}, \mathit{Hp}, \mathit{CP}\rangle \rightarrow \langle \mathit{Ths}', \mathit{Hp}, \mathit{CP}\rangle}$$

$$\text{Call}\ \frac{\begin{array}{c} \mathit{at}\ \mathit{Ths}\ I\ \langle \mathit{TId}, [F|T]\rangle \quad F \xrightarrow{\mathit{call}(\mathit{MId}, \mathit{Args})}_m F' \\ \langle \mathit{MId}, \mathit{CS}, \mathit{NL}, \mathit{PC}\rangle \in \mathit{CP} \quad \mathit{create}\ \mathit{NL}\ 0\ \mathit{LS}' \\ \mathit{set\_from}\ \mathit{LS}'\ 0\ \mathit{Args}\ \mathit{LS} \\ \mathit{set}\ \mathit{Ths}\ I\ \langle \mathit{TId}, [\langle \mathit{CS}, \mathit{PC}, [], \mathit{LS}\rangle, F|T]\rangle\ \mathit{Ths}' \end{array}}{\langle \mathit{Ths}, \mathit{Hp}, \mathit{CP}\rangle \rightarrow \langle \mathit{Ths}', \mathit{Hp}, \mathit{CP}\rangle}$$

$$\text{Get}\ \frac{\begin{array}{c} \mathit{at}\ \mathit{Ths}\ I\ \langle \mathit{TId}, [F|T]\rangle \qquad F \xrightarrow{\mathit{get}(\mathit{Ref}, \mathit{Idx}, \mathit{Val})}_m F' \\ \mathit{at}\ \mathit{Hp}\ \mathit{Ref}\ \langle \mathit{ClsId}, \mathit{Fields}, L, Q\rangle \quad \mathit{at}\ \mathit{Fields}\ \mathit{Idx}\ \mathit{Val} \\ \mathit{set}\ \mathit{Ths}\ I\ \langle \mathit{TId}, [F'|T]\rangle\ \mathit{Ths}' \end{array}}{\langle \mathit{Ths}, \mathit{Hp}, \mathit{CP}\rangle \rightarrow \langle \mathit{Ths}', \mathit{Hp}, \mathit{CP}\rangle}$$

$$\text{Put}\ \frac{\begin{array}{cc} \mathit{at}\ \mathit{Ths}\ I\ \langle \mathit{TId}, [F|T]\rangle & F \xrightarrow{\mathit{put}(\mathit{Ref}, \mathit{Idx}, \mathit{Val})}_m F' \\ \mathit{at}\ \mathit{Hp}\ \mathit{Ref}\ \langle \mathit{CID}, \mathit{Fields}, L, Q\rangle & \mathit{set}\ \mathit{Fields}\ \mathit{Idx}\ \mathit{Val}\ \mathit{Fields}' \\ \mathit{set}\ \mathit{Hp}\ \mathit{Ref}\ \langle \mathit{CID}, \mathit{Fields}', L, Q\rangle\ \mathit{Hp}' & \mathit{set}\ \mathit{Ths}\ I\ \langle \mathit{TId}, [F'|T]\rangle\ \mathit{Ths}' \end{array}}{\langle \mathit{Ths}, \mathit{Hp}, \mathit{CP}\rangle \rightarrow \langle \mathit{Ths}', \mathit{Hp}', \mathit{CP}\rangle}$$

$$\text{Allocate}\ \frac{\begin{array}{c} \mathit{at}\ \mathit{Ths}\ I\ \langle \mathit{TId}, [F|T]\rangle \quad F \xrightarrow{\mathit{allocate}(\mathit{ClsId}, \mathit{Ref})}_m F' \\ \langle \mathit{ClsId}, \mathit{NFields}\rangle \in \mathit{CP} \quad \mathit{create}\ \mathit{NFields}\ 0\ \mathit{Fields} \\ \mathit{extend}\ \mathit{Hp}\ \langle \mathit{ClsId}, \mathit{Fields}, \mathit{unlocked}, []\rangle\ \mathit{Hp}' \\ \mathit{size}\ \mathit{Hp}'\ \mathit{Ref}+1 \quad \mathit{set}\ \mathit{Ths}\ I\ \langle \mathit{TId}, [F'|T]\rangle\ \mathit{Ths}' \end{array}}{\langle \mathit{Ths}, \mathit{Hp}, \mathit{CP}\rangle \rightarrow \langle \mathit{Ths}', \mathit{Hp}', \mathit{CP}\rangle}$$

Table 5.2: The system transition relation, part 1.

$$\text{Spawn}\;\frac{\begin{array}{c}\mathit{at}\ \mathit{Ths}\ I\ \langle \mathit{TId},[F|T]\rangle \qquad F \xrightarrow{\mathit{spawn}(\mathit{Ref},\mathit{RunId})}_m F' \\ \langle \mathit{RunId}, \mathit{CS}, \mathit{NL}, \mathit{PC}\rangle \in \mathit{CP} \qquad \mathit{create}\ \mathit{NL}\ 0\ \mathit{LS}' \\ \mathit{set}\ \mathit{LS}'\ 0\ \mathit{Ref}\ \mathit{LS} \quad \mathit{extend}\ \mathit{Ths}\ \langle \mathit{Ref},[\langle \mathit{CS},\mathit{PC},[],\mathit{LS}\rangle]\rangle\ \mathit{Ths}'' \\ \mathit{set}\ \mathit{Ths}''\ I\ \langle \mathit{TId},[F|T]\rangle\ \mathit{Ths}'\end{array}}{\langle \mathit{Ths},\mathit{Hp},\mathit{CP}\rangle \rightarrow \langle \mathit{Ths}',\mathit{Hp},\mathit{CP}\rangle}$$

$$\text{Enter}_l\;\frac{\begin{array}{c}\mathit{at}\ \mathit{Ths}\ I\ \langle \mathit{TId},[F|T]\rangle \qquad F \xrightarrow{\mathit{enter}(\mathit{Ref})}_m F' \\ \mathit{at}\ \mathit{Hp}\ \mathit{Ref}\ \langle \mathit{Id},L,\mathit{locked},Q\rangle \quad \mathit{extend}\ Q\ \langle \mathit{TId},[F'|T]\rangle\ Q' \\ \mathit{set}\ \mathit{Hp}\ \mathit{Ref}\ \langle \mathit{Id},L,\mathit{locked},Q'\rangle\ \mathit{Hp}' \quad \mathit{remove}\ \mathit{Ths}\ I\ \mathit{Ths}'\end{array}}{\langle \mathit{Ths},\mathit{Hp},\mathit{CP}\rangle \rightarrow \langle \mathit{Ths}',\mathit{Hp}',\mathit{CP}\rangle}$$

$$\text{Enter}_u\;\frac{\begin{array}{c}\mathit{at}\ \mathit{Ths}\ I\ \langle \mathit{TId},[F|T]\rangle \qquad F \xrightarrow{\mathit{enter}(\mathit{Ref})}_m F' \\ \mathit{at}\ \mathit{Hp}\ \mathit{Ref}\ \langle \mathit{Id},L,\mathit{unlocked},Q\rangle \\ \mathit{set}\ \mathit{Hp}\ \mathit{Ref}\ \langle \mathit{Id},L,\mathit{locked},Q'\rangle\ \mathit{Hp}' \\ \mathit{set}\ \mathit{Ths}\ I\ \langle \mathit{TId},[F'|T]\rangle\ \mathit{Ths}'\end{array}}{\langle \mathit{Ths},\mathit{Hp},\mathit{CP}\rangle \rightarrow \langle \mathit{Ths}',\mathit{Hp}',\mathit{CP}\rangle}$$

$$\text{Exit}_Q\;\frac{\begin{array}{c}\mathit{at}\ \mathit{Ths}\ I\ \langle \mathit{TId},[F|T]\rangle \qquad F \xrightarrow{\mathit{exit}(\mathit{Ref})}_m F' \\ \mathit{at}\ \mathit{Hp}\ \mathit{Ref}\ \langle \mathit{Id},L,\mathit{locked},[\mathit{Th}|Q]\rangle \\ \mathit{set}\ \mathit{Hp}\ \mathit{Ref}\ \langle \mathit{Id},L,\mathit{locked},Q\rangle\ \mathit{Hp}' \\ \mathit{extend}\ \mathit{Ths}\ \mathit{Th}\ \mathit{Ths}'' \quad \mathit{set}\ \mathit{Ths}''\ I\ \langle \mathit{TId},[F'|T]\rangle\ \mathit{Ths}'\end{array}}{\langle \mathit{Ths},\mathit{Hp},\mathit{CP}\rangle \rightarrow \langle \mathit{Ths}',\mathit{Hp}',\mathit{CP}\rangle}$$

$$\text{Exit}_{EQ}\;\frac{\begin{array}{c}\mathit{at}\ \mathit{Ths}\ I\ \langle \mathit{TId},[F|T]\rangle \qquad F \xrightarrow{\mathit{exit}(\mathit{Ref})}_m F' \\ \mathit{at}\ \mathit{Hp}\ \mathit{Ref}\ \langle \mathit{Id},L,\mathit{locked},[]\rangle \\ \mathit{set}\ \mathit{Hp}\ \mathit{Ref}\ \langle \mathit{Id},L,\mathit{unlocked},[]\rangle\ \mathit{Hp}' \\ \mathit{set}\ \mathit{Ths}''\ I\ \langle \mathit{TId},[F'|T]\rangle\ \mathit{Ths}'\end{array}}{\langle \mathit{Ths},\mathit{Hp},\mathit{CP}\rangle \rightarrow \langle \mathit{Ths}',\mathit{Hp}',\mathit{CP}\rangle}$$

Table 5.2: The system transition relation, part 2.

method in the constant-pool designated by the method identifier *MId*, a new frame can be constructed and pushed onto the stack of the thread. The arguments supplied are inserted at the start of the list of local variables using the set_from formula.

Result and VResult These two rules handle frames that return to their caller. This is done by removing them from top of the frame list of the executing thread. If a result is returned (the VResult rule), it is pushed on the stack of the resuming frame. Since the program counter is incremented when a method-transition invokes a method, it is already in the correct position when the invoked method returns.

Get and Put These two rules handle modifications of the contents of the heap. Both rules have method-transition labels with the arguments Ref, Idx, and Val. The difference between the two rules is that the value of Val for a `put`-action is uniquely determined by the frame, while the value of Val for a `get`-action is unconstrained by the method-transition, and is determined by the use of the `at`-formula that references the field of the class instance referred to.

Allocate To allocate new memory, a frame emits an `allocate`-action, with the `ClsId` of the class to instantiate and an unconstrained reference. The number of fields in the class are fetched from the constant-pool and a new instance of the class is created at the end of the heap. The reference from the action is then matched against the size of the heap, resulting in the address of the newly created instance.

Spawn When a spawn-action occurs, a new thread of control is created, and an initial frame is created using the method designated by the *RunId* of the action. The thread identifier is set to the reference to the class instance from which the initial frame is started. The new thread is added to the list of active threads.

Enter There are two cases to consider when a thread wants to acquire the mutual exclusion lock of a class instance; is the lock locked or not? If the lock is locked, we add the current class to the end of the list of threads waiting for the lock. On the other hand, if the lock is unlocked, then we modify the status of the lock and continue.

Exit When releasing the monitor of a class instance, the first thread waiting for the lock will be moved into the active set of threads, thus transferring the lock to that thread.

5.4 Discussion

The described model of the JVM is a simple model, but still it captures many essential aspects of the actual JVM, such as threads. The model also excludes many important features and many instructions. In this section, a description and rationale for the exclusion made, as well as the inclusions done, is given. Some suggestions for improvements and extensions are also presented.

5.4.1 The Chosen Model

The model we have chosen for the developed semantics is primarily explicit in the actual semantics. This includes choices made about such things as scheduling policy, instruction execution, and system organisation.

The chosen model is a straightforward adaptation of the suggested memory-layout of the JVM in [LY99]. The adaptation is done to transform the information into a format that is natural for representation as a first-order term. For example, the information present in the constant pool, such as complete symbolic type-names and symbolic references to handle dynamic loading is dropped, and replaced by simple numeric identifiers.

The choice to use the suggested memory layout leads to an imperative style of semantics, where a natural imperative reading of the premises in the semantics gives a read-modify-update style. This approach is in some sense not the ideal structural operational semantics of Plotkin[Plo81]; the structural deconstruction is relegated to formulae in the associated logic. These formulae could, in principle, be incorporated into a more complex TSS, since they all are simple least fixpoint formulae. But this would be of dubious values, since the new TSS would be very hard to understand when there are many such predicates. For an example of a TSS for a language with a "program counter", that does not use the `at`-formula, see Section 2.1.3. The program counter of that example is implicit in the positions of the two stacks.

A close connection to the semantics given in the JVM specification has the benefit of making it easier to argue for the correctness of the semantics. We cannot hope to prove the correctness of our semantics, since this would require the semantics to also be specified formally.[1]

References When a new class-instance is created, the reference to that instance is the index into the heap where it is allocated. This is a simple idea, with some serious drawbacks. The benefits of the idea is that it is very simple and "natural". There are two main drawbacks, namely the proliferation of garbage and symmetry removal.

If we had a symbolically addressed heap such as a list of ⟨reference,instance⟩-pairs, instead of the current scheme, we could re-order this heap without affecting

[1] "One can't proceed from the informal to the formal by formal means", Alan Perlis.

the semantics. Thus, we could move elements that are known to be garbage to the end of the heap, and then abstract this tail of the list away.

The symmetry removal problem leads to an increase in the search space. To illustrate the problem, consider the fact that two threads poised to allocate a new instance could do this in any order. Intuitively, the chosen order of allocation does not matter, and we should be able to merge two different interleavings of the allocation. In the developed semantics, the resulting systems will be different.

A possible solution to these problems would be to introduce a congruence relation $\equiv$ on the systems, in the style used in the π-calculus process algebra (see [Mil99] for a definition of the π-calculus and for usage of a system congruence). This congruence relation would equate systems with the same static meaning, but with different syntactic representations. The old concrete transition relation could then be taken "up to" the congruence. This would mean that a transition $t_1 \equiv\xrightarrow{\alpha}\equiv t_2$ in the new transition relation would be permissible, iff a transition $t'_1 \xrightarrow{\alpha} t'_2$ exists, such that the congruences $t_1 \equiv t'_1$ and $t_2 \equiv t'_2$ are true. On the other hand, such a system might be much harder to automate and to reason about.

Native methods A common problem for formalisations of the semantics of the JVM is that a large part of the actual semantics is buried in native methods in the API of the standard library for the Java language[Jav]. These methods can not be described in pure Java, and must be modeled more directly in the style of the fictive start-instruction which models the start-method of the Thread class. It is not possible to give a complete set of extra instructions, since the native methods are continuously revised and more are added when the Java API is updated.

Which native methods to model, and which to omit is dependent on the programs of interest. The presented model includes the semantics of the *start*-method in the `java.lang.Thread`-class that can be viewed as an example of how such modelling can be done.

Scheduling The JVM standard does not specify any specific scheduling, and in fact, it does not even specify if the scheduler should time-slice. The choice of an unconstrained interleaving semantics represents all possible execution traces under the assumption that the effect of executing one JVM instruction is atomic (which is true, for our chosen subset, see Section 5.1). This means that the semantics include all the possible scheduling strategies of the JVM.

If we want to assume that a system has some fairness-property, we could encode this assumption in a μ-calculus formulae, in the style of Example 2.2.9. However, there is a problem since all transitions in our model have the same action. In order to adapt the formula, we would have to check which specific thread gets executed, using some predicates that examine the structure of the system before and after the transition.

It is not as easy to restrict attention to only the non time-sliced schedulings. The easiest way is probably to alter the semantics for this to work.

Fair mutexes? The mutual exclusion locks described in the JVM specification are not fair locks; when multiple threads are waiting to acquire a lock, any one of them may be the next to get it. In the described semantics for the locks, a thread trying to acquire a lock that is previously locked, will be put in a queue waiting for the lock to be released. A queue is a fair data structure (first come - first serve), where no element can be left indefinitely, passed by others. However, this does not mean that the locks themselves are fair, since a thread may wait indefinitely before actually entering the queue (assuming that we do not enforce fairness).

An alternative to the chosen scheme for locks, would be to let the threads stay in the thread-list while they are waiting for a lock, and that the execution of a waiting thread while the lock is locked would be a no-op, as is done in [M5]. This could be problematic, since it introduces non-progressing runs of the system, where only no-op actions are taken.

Instructions chosen With 20 instructions modeled, as opposed to the over 200 instructions found in the actual JVM, the model may at first glance seem very simplistic. But one has to consider the fact that many of the instructions found in the JVM are designed for code-size and speed, instead of orthogonality of design. For example, there is a general instruction `iload` n with n being an index into the local variables, the argument given in the code-stream There are also the specialised instructions `iload_0` up to `iload_5` that load the local variable at the addresses zero to five. Many different comparison-operators are available, but given two suitable ones, the rest may be implemented with some additional code. The instructions are also type-tagged, so that there are different instructions to load an integer and a reference. These factors must be weighed in when considering how much of the virtual machine that has been modeled.

5.4.2 Missing features

The Java Virtual Machine is a large and complex system; a real world design with all its implications and intricacies. This means that for a work of this size, we cannot hope but to omit many features. In this section, some comments about the missing features are given.

Class-hierarchies A major missing part of our formalisation is the class hierarchies of the type system of Java. This removes the object-oriented part of the JVM, but retains the object-based part. The addition of real Java class-hierarchies would mainly complicate the Get, Put, and Invoke rules for the system-transitions and force a slightly different representation of classes. There would also be a need to split the abstract `invoke` instruction that we use, into the actual instructions for invoking static members, special functions, and normal methods.

Furthermore, the `instanceof` instruction would also have to be modelled. The instruction is used to test the dynamic type of a reference at run-time. A method transition for the semantics of `instanceof` in this setting, would probably be akin to the handling of `getfield`. The result of the comparison would be an unconstrained boolean at the top of the stack, and the system transition would do the actual comparison.

Exceptions With the representation of frames chosen, the addition of exceptions is not very difficult. To add exceptions to the model, an action $raised(Ref)$ should be added, indicating that the frame has thrown the exception Ref (a reference). For each frame, the exception-table should be added, and the appropriate logic implemented.

This addition would also need some more instructions to be modelled. One such instruction pair are the instructions `jsr` (Jump SubRoutine) and `ret` (RETurn from subroutine). These instructions are necessary to implement exception handling correctly. Furthermore, the techniques used for implementing the `instanceof` instruction, as described in the previous paragraph, are needed for determining the type of exception that has been thrown.

Dynamic class loading Inherent in the JVM platform is the possibility for dynamic loading of classes. A class is loaded when needed, not necessarily when the program is started. There are also functions in the standard API that allow a program to interface with the class loader of the JVM. One use of such functions enables an application to generate new code that should be added to a running system[2].

How to structure dynamic loading of components in a verification context is not clear. One possibility is to keep the assumption about a statically known context of classes (that is, the same constant pool as in the presented model), but to allow a program to add a class at runtime using the interface to the class loader. This would allow the use of dynamically created classes, where the program creates the new classes. How to verify a property of a program that uses this feature, however, is not at all clear.

Thread support The model has support for starting threads and for locking the mutexes associated with a class instance. However, there is no support for waiting on the condition-variable associated with each instance. A model of these variables would be very much related to the queues used by the mutual exclusion locks for the waiting threads.

Furthermore, the API for the Thread class specifies many methods that have effect on the state of the system, or that return information about the state of the system[Jav]. This must be modeled if verification of programs that use these features is to be done.

[2] The `defineClass(...)` methods in the `java.lang.ClassLoader` class. See [Jav] for documentation.

Type information Dropping type-information is a rather drastic decision for a formalisation. Even though a real JVM would probably do the same, we do not retain all knowledge that is available to us that could be useful in verification. One practical result is that both references and unsigned integers will have the same type, a natural number.

In [SBS01] it is shown that a type-error will never occur in a compiled Java program. This means that an offensive semantics (such as ours) and a defensive semantics (that check all types) agrees on all valid class-files.

Missing assumptions In order to prove a statement about a program, we sometimes may need all the guarantees present in an actual JVM implementation in the proof. These guarantees are not available as it is now. An example of such information loss is the fact that the same type of data is used for both references and numbers.

A possible way to introduce such assumptions would be through a predicate that specifies the well-formed JVM systems. In the present semantics, we can construct nonsensical systems, such as systems which call non-existing methods, and methods with jumps outside the list of instructions.

Garbage collector We do not model the garbage collector in any way. It is not clear if this choice is correct (and the handling of potential garbage in a system should be relegated to the proof), or if the semantics should implement some sort of garbage-collection. If such a feature would be added, then the choice of using heap-addresses as references would have to be changed also.

5.4.3 Possible extensions

In this section, some suggestions for possible extensions to these semantics are listed.

Symmetry reduction When using an interleaving semantics, one problem is that of uninteresting interleavings. For example, consider two threads in state a_1 and b_1, both eligible for execution with the transition-sequences $a_1 \rightarrow a_2 \rightarrow a_3$ and $b_1 \rightarrow b_2 \rightarrow b_3$ respectively. Furthermore, if these actions all represent some internal computation that the processes perform, then the 20 possible different interleavings are equivalent. In a proof of some property, we would have to prove the property for all these possible interleavings, with the threat of an exponential state-space explosion to cope with.

This problem can be mitigated in the developed proof-system by using the copydischarge-rule, but this requires manual effort to apply as well as actually generating the interleavings before joining them. We would like to handle as much as possible of needless interleavings inside the semantics.

One possibility is to adapt the idea of basic blocks that is commonly used in compilers[ASU86]. A basic block is a piece of code with no jumps out from the middle and no jumps into the middle. These blocks of code can be handled more efficiently since the compiler has more static knowledge. In our semantics, we could use this idea to handle more instructions at a time. The key idea is that the method-local non-jumping instructions are insensitive to the possible interleavings of systems. Thus, we could let a transition of a thread execute all such instructions in one single transition.

As an example, for such a semantics, the system

$$\langle[\langle TId,[\langle[push(1),dup,ifeq(0)],0,[],[]\rangle]\rangle],HP,CP\rangle$$

would go, in a single transition, to

$$\langle[\langle TId,[\langle[push(1),dup,ifeq(0)],2,[1,1],[]\rangle]\rangle],HP,CP\rangle,$$

instead of the two transitions it would take in the presented semantics.

This extension would reduce the amount of needless parallelism significantly, since the loading and storing of method-local variables (which would be covered in this scheme) is a common part of Java programs, for example when new class-instances are created. It would also be a safe extension, since no other thread can possibly influence the values local to a frame, and thus do not participate in the computation.

A natural implementation of this scheme in the presented semantics, would be to use a rule with a negative transition-premise. A negative transition-premise is a premise of the form $t \overset{\alpha}{\not\to}$ for some t in $T(\Sigma,V)_S$ and α in $T(\Sigma,V)_A$. The intuitive meaning is that the premise is valid if and only if t has no possible α-transition. Assume a transition-relation $\to_{sc}$ for the simply computable instructions (i.e., the method-local instructions minus `goto`, `iceq`, and `iclt`), removing them from the transition-relation $\to_m$. First we add a new layer in the form of a new transition-relation, named $\to_{sc_{\max}}$. The name $sc_{\max}$ stands for the maximal sequences in sc. The TSS for the new layer is composed of the following two rules.

$$\text{Compute}\ \frac{T \to_{sc} T'' \qquad T'' \to_{sc_{\max}} T'}{T \to_{sc_{\max}} T'}$$

$$\text{Finish}\ \frac{T \to_{sc} T' \qquad T' \not\to_{sc}}{T \to_{sc_{\max}} T'}$$

The reason that we need the negative premise in the rule Finish, is that we want to force the maximal sequences of $\to_{sc}$ actions. Without it, we would get all possible sequences of $\to_{sc}$ actions. This would increase the search-space, defeating the purpose.

To add these execution sequences to the method-transitions, We could add the following rule to the TSS specifying $\to_m$.

$$\text{Execute}\ \frac{T \rightarrow_{sc_{\max}} T'}{T \rightarrow_m T'}$$

This implementation would intuitively implement our desired semantics for compound execution of basic blocks. All this in a rather non-intrusive manner. Unfortunately, the meaning of a negative premise in a TSS is not as simple as the intuitive feel is. For example, in [Gla04] Glabbeek presents eleven *different* answers to the question of what a TSS with negative premises means. Thus, we must find some other way to specify these transitions.

Distributed computation The Java platform is not limited to single-site parallelism, it also support distributed computation via Remote Method Invocation (RMI). RMI works by allowing a JVM to send method-invocations to other JVM instances at some other location, to initiate some computation. A result is then returned to the invoker. This allows the construction of distributed cooperating applications in the Java environment.

An interesting further development would be to handle distributed computation in the semantics. One possibility to do this would be a parallel composition of the individual JVMs, in the style of Example 2.1.5. A remote method call would then be represented as a label on the system doing the call, matched by some accepting label at the receiving end. A complication is that the RMI system has extensive support for the Java type-system and polymorphism, which could have some interesting and unforeseen effects on a formalisation, especially in combination with dynamic loading of classes.

Modelling Java RMI has not been present in any of the related models found by the author.

Process algebra In the process algebra community, much research into operational semantics of concurrent processes exists. An interesting line of research would be to adapt the model of the JVM to the style of systems used in process algebra. This would mean a reduction of the imperative style used here, in favour of a more structural style, with evolving processes.

An example of such a different style would be to model the heap as one (or possibly several) processes, capable of sending and receiving messages containing the values of some variable. For an example of such an encoding in the context of Concurrent ML (an extension of SML that adds concurrency), see [Rep91]. As for the distributed computation-suggestion, it would be appropriate to adopt a parallel composition operator in the style of Example 2.1.5.

As an example of how this approach could potentially be a better alternative, consider a system that consists of a series of filter processes, $P_1, P_2, \ldots, P_n$. These processes are composed in a linear chain, with input into one end (at P_1), and output at the other end (at P_n). We are interested in the complete system, the actual processes used in composing the chain are not as interesting.

A probable situation is that two processes P_i and P_{i+1} communicate through a shared variable on the heap called H_i. In the presented semantics it is hard to "cut out" two processes and prove correctness for them individually, using that result in verifying the larger system. One problem for such a scheme is the global nature of the heap. If the heap was modelled as processes however, we could potentially cut out a process term $P_i \parallel H_i \parallel P_{i+1}$ and replace it with an abstract process with the interesting abstract properties of cut term. A benefit of such an approach would be that the replacement of one implementation of a process for another would only have local effects in the proof.

Bisimulation In a more compositional style of semantics, as in the two suggestions above, it would be interesting to examine the possibilities of defining a bisimulation relation on the components. As described in Section 2.4.1, bisimulation is a common method to determine if two processes behave in the same way. At the moment, it is not straight-forward to define an interesting bisimulation relation for the threads in the current semantics, since all transitions have the same label, and labels are what a bisimulation relation uses to define behaviour.

Since a significant amount of the computation still would have the silent action as label, it would probably be more interesting to use a weak bisimulation relation. Such a relation abstracts away the number of silent internal actions that occur in a system, and compares only the "interesting" labels.

Using a bisimulation relation, we could employ the techniques of the process algebra community, in proving properties about programs. It could, for example, be used to compare abstract specifications encoded as a program with the actual implementation. If we have verified a component using this method, we could potentially replace it with its simpler specification when verifying a larger system.

Chapter 6

Embedding the semantics in VCPT

The semantics described in Chapter 5 has been embedded as a theory in VCPT, and tactics have been developed that raise the level of abstraction so that the transitions of concrete programs are automatically found. In this chapter the embedding will be described.

There are different possibilities for embedding an operational semantics into VCPT. Either the transition-relation can be describe formally in the logic, or else the rules of the TSS can be coded as functions in SML. This thesis uses the former approach, while the latter approach is used for the JavaCard Applet Interaction model. See the discussion in Section 6.4 for more about this choice.

Describing a TSS in VCPT is relatively straightforward. A proof of a transition is a tree (Definition 2.1.4), which is a finite structure, and finite structures are naturally described as least fixed points.

6.1 Datatypes for systems

The datatypes for the system follow the datatypes in the formalization closely. Most tuples present are however of a distinct type instead of just a tuple, so that instead of the tuple $\langle CS, PC, VS, LS \rangle$ for a single frame, the construction $frame(CS, PC, VS, LS)$ is used. This is done for systems, method-constants, and so on.

The datatype used for the basic data-elements of the JVM is the natural numbers of VCPT. This is used both for references and for the actual data that is used in computations. The choice of naturals is not an essential one, but if it were to be changed, say into some representation of signed integers, then we would have to add a differentiation between references and data. However, we would not have to add any more instructions for the non-computational use of data, since we would still have a single sort for the elements of data.

```
iaddTrans : instruction → frame → action → frame → prop =
  λInst : instruction . λF1 : frame . λA : action . λF2 : frame .
    cases (Inst, F1, A, F2) of
      (iadd,
       frame(CS, PC, [N1, N2 | VS], LS),
       τ,
       frame(CS, PC+1, [N | VS], LS)
      ) ⇒
        natPlus N1 N2 N
    end /* cases */
END; /* iaddTrans */
```

Figure 6.1: The predicate implementing the semantics of `iadd` instructions.

6.2 The transition relations

We separate the semantics in the same way as the formalization does, with a transition relation for systems and one for single frames. The system-transitions are the transitions that are visible to the user of the embedded formalization, in that they are used for defining the modalities.

In the following description, we will use an imperative reading of the predicates defined when explaining them. This means, that for a conjunct $\phi \wedge \psi$, the "execution" of ϕ is said to happen before that of ψ. This order is enforced in the tactics, see Section 6.3.1 for more discussion about how this is achieved.

The method transition relation The method transition relation is a simple encoding of the corresponding TSS. The current instruction for the originating frame is found using the `at`-predicate. A suitable predicate is then called to match the originating frame and the resulting frame. There is one predicate per modelled instruction.

As an example, consider the `iadd` instruction. The definition of the semantics for this instruction in Table 5.1 on page 60, specifies that the two top elements of the stack should be replaced with the sum of them. In Figure 6.1 the predicate implementing this is shown.

As can be seen from this example, the structural parts of a rule in the TSS can be handled using the cases-construction defined in Section 2.3.1. If there are any more conditions (such as the addition constraint on the values N_1, N_2, and N), these are specified inside the case (and thus in the scope of the variables it needs to refer to).

For the instructions with two rules implementing them, we still use a single predicate to handle the instruction. Inside the cases of the predicate, an if-then-else formula is used to decide between the two possibilities. This technique is only suitable for mutually exclusive rules (as all the multiple rules in the method-transition are).

The system transition relation The system transition relation is more complicated to define than the method transition relation. For any frame, the next state is uniquely determined (apart from the values "returned" by the environment, such as the value in the `get`-action). For a system, on the other hand, there may be multiple possible next states that are very different, due to the unconstrained choice of next thread to execute. In the following paragraphs, an outline of the implementation of the system transition relation is given.

We equate the unconstrained choice of the next thread to execute with a disjunction of all the possibilities. The implementation manages this using a least fixed point that iterates through all the threads that are eligible for execution, starting with the highest index for a thread and continuing with decreasing indexes. As an example, assume that we have a system s with three threads. Let the systems s_1, s_2, and s_3 be the resulting systems when thread one, two, or three is executed. Then the formula $\mathit{transRel}\ s\ \tau\ s'$ will result in the disjunction $s' = s_3 \vee s' = s_2 \vee s' = s_1 \vee \bot$. The last disjunct is the ground case, i.e., when there are no more possible threads to execute. This choice of ground case does not interfere when there are threads to execute. If the list of threads is empty, then it means that there is no possibility of execution (since it would be the only disjunct in the disjunction).

When a thread is chosen for execution, the method-transition for the top frame is found with an unconstrained label for the action. Using the label on the transition, the correct formula implementing the rule is found. Each rule is implemented as an existential quantification over the free variables of the rule, followed by a conjunction of the premises in left-to-right, top-to-bottom order.

For the labels with more than one possible matching rule (i.e., the labels `enter` and `exit`), a single formula is responsible for both the rules, using if-then-else to distinguish the different rules, in the same manner as is used in the method-transition relation.

6.3 Automating reasoning

The derivation of a single transition using the formulae described above is extremely tedious. As an example, take the derivation of a single `dup`-transition when we have one thread. The following transition is a valid transition in the semantics.

$$\begin{array}{l}\langle[\langle \mathit{TId}, [\langle[\mathit{push}(1), \mathit{dup}, \mathit{iceq}(0)], 1, [1], []\rangle]\rangle], \mathit{HP}, \mathit{CP}\rangle \rightarrow\\ \langle[\langle \mathit{TId}, [\langle[\mathit{push}(1), \mathit{dup}, \mathit{iceq}(0)], 2, [1,1], []\rangle]\rangle], \mathit{HP}, \mathit{CP}\rangle\end{array}$$

When proving the transition in VCPT, it takes 52 tactic invocations and 2 navigation commands. This is using the tactics developed in Section 3.2 for automatically deriving the result of the list-handling formulae. To mitigate this problem, tactics for deriving the next transition for concrete systems have been developed, and are presented in this section.

6.3.1 General approach

The main tactics available to the user are `transrel_l` and `transrel_r`. Both tactics are defined to handle a transition relation at a specific place on the left or right hand side. The tactics use a depth-first approach to order the proofs of the conjuncts and disjuncts encountered during the proof-search, the same as in Prolog systems. This means that the leftmost conjunct or disjunct will be investigated first, and after it is completed or evaluated as far as possible, the next will be handled. The tactics are implemented with the exact structure of the transition relation in mind, which aids in making the predicates as fast as possible.

The focus of the tactics, is for the case of a sufficiently concrete system for the source of the transition. A sufficiently concrete system is a system where no parts of the system that are needed for the derivation of the next step are undetermined. For the result of a transition, the argument may be either a variable, or a concrete system itself.

The tactics are defined to handle the two levels of the semantics separately, the system-level tactics invoking the method-level tactics when needed.

6.3.2 Specific tactics

The specific tactics that are available are the following.

transrel_l pos This tactic will derive the possible next states for a transition-relation at position `pos` on the left-hand side.

If the result of the transition is a variable (i.e., not a concrete system), there will be one new goal for every thread that can be executed, where the resulting system is substituted for the variable. If the result of the transition is a concrete system, then the formula will be removed if the transition is valid, or the goal resolved if the transition is invalid.

transrel_r pos This tactic works like the above, but for a transition-formula to the right of the turnstile.

This tactic stops when an equation $t = t'$ is derived, for t the new state and t' the third argument to the transition-relation. This behaviour was chosen, since forcing the equality imposes a choice of which thread was executed, a choice that should be left to the user.[1]

[1] The combination of the depth-first approach and the reversed order of the threads in the disjunct in the transition-relation, would make the last thread the thread chosen if equality was forced and t' was a meta-variable.

methodtrans_l pos This tactic derives the next frame for a method transition relation at position `pos` on the left-hand side.

The tactic needs a sufficiently concrete first argument, the source frame, to work properly. The two subsequent argument, the label and the next frame, can be either variables or concrete terms. If they are variables, the next state will be derived, and the label properly instantiated with the available knowledge.

methodtrans_r pos This is essentially the same tactic as the above tactic, but for transitions to the right of the turnstile. If a new frame is to be calculated into the third argument, the argument needs to be a meta-variable.

6.3.3 Implementation details

The tactics are defined in the same structure as the predicates they operate on are defined. For example, the first level of the system-level tactics handles the scheduling scaffolding, and for each choice of thread, it defers the handling to a separate tactic.

The implementation of the tactics described above is, for most parts, straightforward for handling the actual formulae. The tactics make heavy use of the composition-tacticals to achieve a sequential composition of the applications needed to derive a new state.

For the method transition, the individual transition handling predicates used (cf. Figure 6.1) take four arguments, the first being the instruction in the code that the program counter designates. The reason to add this argument, which is redundant since the information is readily available anyway, is that the argument is examined in the tactics, to determine which predicate is called. The normal way of determining this in VCPT, using the accessor functions generated, does not work, since these predicates are defined to be local to the method-transition,. Therefore, there are no accessor-functions available. An added benefit is that the formula are slightly faster to prove, since we only look up the correct instruction once.

6.4 Discussion

Embedding the operational semantics in VCPT is, in almost every case, a straightforward task. Some scaffolding for deconstructing terms is written, followed by a list of the results of the different rules. If the semantics are deterministic, then the embedding follows easily. However, the scaffolding is in some places interesting since it handles the non-deterministic parts of the semantics.

Below are some discussion on various topics relating to the embedding of the semantic.

Scheduling Choosing the next thread to execute is the single most interesting part of the embedding. There are many possible variations to do this, with various properties for the choices. One problem with our chosen method, is that it assumes complete knowledge of all the threads in a system. If we want to reason about systems with a potentially unbounded amount of threads, this choice of a scheduler will become a problem.

As an alternative to the chosen method, we could recurse through the list of available threads and either execute the thread, or leave it and continue, in the manner in which the TSS for Example 2.1.5 is defined. However, there are many issues to handle for this scheme to work, since we want a disjunction of all the possible *single* choices of threads to execute.

Another possibility, is to use an oracle that chooses the next thread to execute. This is somewhat akin to the way in which the M5 model works, where theorems are stated about terms of the form `(run sched p)`, which represents running the program `p` according to a scheduler `sched`. For a general theorem, they let the scheduler be unconstrained. A scheduler in their implementation is simply a list of the threads to run. In our framework, it would be natural to model a scheduler as a greatest fixed point that generates an infinite amount of unconstrained integers.

Specialised tactics vs. Simplification A possible alternative to using these specialised tactics is to use a more general simplification tactic, since we already know that the system is concrete and therefore we can derive the result. For example, to derive the `dup`-transition from Section 6.3, we could use the command

```
Simplify_tacs [array_at_l, array_set_l, unfapp_l, eq_subst] [];
```

which finds the sought new state. If we could localise the simplification process to only one formula and its descendants, we could use this scheme instead.

There are, however, some problems with this approach. One problem is that it is inherently slower, since the specialised tactics know what to do at every state while the simplification must search through many possibilities instead. Despite the knowledge about the systems, the specialised tactics are already slower than desired; deriving the `dup`-transition takes about one second on a modern computer. Furthermore, the use of simplification is not always a good idea. We may equate some things that are not necessary for the successful derivation of a new state when we add, for example, the `eq_subst` tactic to the simplification process.

Describing vs. Implementing Another approach to model an operational semantic in VCPT is the one taken for the JavaCard Applet Interaction model in [BGH02]. While they define their semantics explicitly in the model, the actual definition is not needed, nor is it used. The definitions may, in fact, be safely omitted, leaving the transition-predicate as an empty definition. The implementation of their semantics that is used, is done through SML functions that manipulate the term-structures explicitly. There is one function for each rule in the semantics. This is in contrast to our model of the semantics, where we have a description of the semantics in the logic of the tool itself and this definitions is used to derive the new states.

In contrast to our embedding, implementing a semantics in SML is more vulnerable to errors, since the SML functions are not constrained by a sound logic. The upside is that their system is much more efficient in deriving new states, which is very useful for, for example, model checking small parts of systems. Tactics for model checking has been used with success for Erlang in the EVT tool[ACD$^+$03].

Chapter 7

Examples

In this chapter, some examples of using the system for proving properties of programs are presented. The examples are chosen to show some interesting properties of both VCPT and the model.

The first example shows a property of a single threaded program. The subsequent examples all centre around the same system, which consists of two identical threads that compete for entry into a critical section. All the examples presented in this chapter use a common set of formula abbreviations, which will be presented first.

7.1 Common Properties

The examples in this chapter use some common abbreviations for μ-calculus formulae. These abbreviations encode some standard branching time temporal logic operators.

The abbreviations are specified in the tool as described in Section 2.3.1. A list of the names of the abbreviations and their associated meaning is shown in Table 7.1. The properties are defined using the fact that the only action a system can perform is the τ-action.

The `Eventually` property expresses the fact that the argument property ϕ will necessarily sometime in the future be true. This is the case, if either ϕ is true, or else there is some transition from the current state, and for every reachable state, ϕ will eventually be true. We cannot follow the unfoldings of the fixpoint definition indefinitely, since it is a least fixed point. Thus, we are guaranteed to reach a state that satisfies ϕ. The clause $\langle\tau\rangle\top$ is needed, since without it a terminating state would satisfy the formula, regardless of whether ϕ is true at that state or not.

The property `Sometime` states that it is possible to reach a state that satisfies the argument propertyϕ. This is expressed as a disjunction where either ϕ is true, or for some possible next state, we can reach a state satisfying ϕ. As with `Eventually`, we can not loop indefinitely, since the fixpoint definition is a least fixed point.

Name	Definition
$\mathtt{Eventually}(\phi)$	$\mu X : s \to prop.\phi \vee (\langle\tau\rangle\top \wedge [\tau]X)$
$\mathtt{Sometime}(\phi)$	$\mu X : s \to prop.\phi \vee \langle\tau\rangle X$
$\mathtt{Always}(\phi)$	$\nu X : S \to prop.\phi \wedge [\tau]X$

Table 7.1: Common formula abbreviations

As described in Section 2.3.1, the `Always` property states that for every possible path through the system, the argument property ϕ will be true at all the reachable states.

The key difference between the `Sometime` and the `Eventually` properties is possibility versus necessity. Either the argument property *will* hold, or it *can* hold. For a discussion about the differences between these two, see [Lam80].

7.2 A Loop That Eventually Ends

In this section, an example illustrating the proof of a property of a small single-threaded system is given. The property that we are interested in, is that the program will eventually end.

The program we study is composed of a loop that decrements a counter on the stack, the counter having an initial value of X. The value X can be any natural number. We will let the heap and the constant-pool be empty, since they are not needed in the example. A program-fragment in Java that corresponds to the example, could be the following code.

```
int i = X;
while(i != 0)
  i = i - 1;
```

However, the compiled program for this snippet of code would use a local variable to store the value of `i`, which we will not use.

In Figure 7.1, the theorem that we prove is shown as it is expressed in VCPT. The program is initially at instruction 0, with a natural number X on the stack. First the program checks if the number is equal to zero, and if so it jumps to the end of the program. Otherwise, the number is larger than zero, and it is decremented by one. After this, the program jumps back to the beginning again.

The property that the program will end, is specified as `Eventually`(*spec*), where *spec* is a property that checks if the program counter is equal to 7, indicating that the program has ended.

The proof of the theorem is essentially an induction on the value of X, which is a natural number. Using the command `add_type_info` with the term X as argument, we get a predicate describing the value (or rather, the structure) of X.

```
PREDICATES
/* Specification that the program has stopped.
 */
spec : system → prop =
  λS : system.
    cases S of
    system([[frame(CS, PC, VS, LS)]], H, CP) ⇒
      PC = 7
    end
end /* spec */
END /* PREDICATES */

THEOREM eventually_ends
declare X:nat in
⊢ system([t(0, [frame([dup(), push(0), iceq(7),
                       push(1), swap(), isub(),
                       goto(0), nop()],
                    0, [X], [] : locals)])],
                    []:heap,
                    []:constpool
         ) : Eventually(spec)
END
```

Figure 7.1: The theorem that is proven.

This predicate is added as an assumption in our sequent. Since X is a natural number, we get the predicate that describes the natural numbers, which is the following least fixed point.

$$\begin{array}{l} \mu U : nat \rightarrow prop.\lambda N : nat. \\ \qquad N = 0 \\ \quad \vee \quad \exists N' : nat. \\ \qquad \left(\begin{array}{ll} & N = N' + 1 \\ \wedge & U\ N' \end{array} \right) \end{array}$$

This predicate is approximated, and then unfolded, resulting in two goals. The goals are for the cases X=0 and X=$X_1 + 1$, where X_1 a new variable introduced via existential quantification. The case of X=0 is handled by simply executing the program until it stops, which takes three steps.

To actually execute the program, we use the modalities of the `Eventually` definition. First, the formula is unfolded. Then a check is made to see if the current state fulfills the property that we are interested in. If it does, then we are satisfied. Otherwise, we follow the modalities, the box modality generating a new goal with the same property. These steps are easily programmed as a couple of small tactics.

To prove the case when $X = X_1+1$, we proceed by executing the program. Since we have the term $X_1 + 1$ on the stack, subtracting 1 is possible without problems using the developed tactics. When the program loops back, we have reached a state that has been encountered previously. The difference is that we have the value X_1 on the stack instead of X. By calling `discharge1`, the companion node is found, and the proof is completed. The successful use of the rule of discharge is based on the fact that we have unfolded a least fixed point to the left of the turnstile. The induction shows that we can decrease the value of X, resulting in new numbers $X_1, X_2, \ldots$, until we finally will reach a ground state, with the value 0.

This small example illustrates the fact that we can use the developed tactics together with a structural decomposition of abstract data, in order to prove a property of a program. When we have enough information about our environment (which in this example comes from the type-information about the natural number X), the derivation of the next step can be performed automatically.

7.3 Competing for a Critical Section

In this section we will present proofs of some properties of a multi-threaded program that has a critical section. First the program will be presented, then the different properties and a description of how they can be proved using VCPT.

7.3.1 The Program

Consider the program in Figure 7.2. This Java program is a multi-threaded program where two threads compete for entry into a critical section. The program can be seen as a typical example of two threads working together. The Container could, for example, be some sort of buffer or aggregator of results for the two threads.

The properties that we are interested in proving, are related to the critical section in the `run()`-method of the Worker class. Thus, we can without loss of generality assume that the program has reached a state where both worker threads are started, and the initial thread has stopped. If we instead prove the properties from the beginning of the `main(...)` method, we will simply get a larger proof.

The state of the system after the main thread has exited, consists of two threads and four instances on the heap. The heap the instance for the initial object, the allocated Container object, and the two threads, in that order. The two threads are the worker threads. They have identifiers 2 and 3, since those are the indexes into the heap for the corresponding class instances. None of the threads have been executed yet.

The code implementing the `run()`-method of the Worker class is presented in Figure 7.3, together with a description of the data referenced. The code is translated directly from the compiled program, as compiled by the Java SDK 1.4 compiler and

```
class Container { public int value; }

class Worker extends Thread {
      Container cont;

      public Worker(Container cont) {
          this.cont = cont;
      }

      public void run() {
          while(true) {
              synchronized(cont) {
                  // do something
              }
          }
      }
}

class Fair {
      public static void main(String [] args) {
            Container cont = new Container();
            Worker w2 = new Worker(cont);
            Worker w3 = new Worker(cont);
            w2.start();
            w3.start();
      }
}
```

Figure 7.2: A Java program with threads competing for a critical section.

Code		Data referenced
PC	Instruction	
0	goto(1)	
1	load(0)	Local variables:
2	getfield(0)	0: Reference to class instance.
3	dup()	1: Stored Container reference.
4	store(1)	
5	monitorenter()	Class variables:
6	load(1)	0: Reference to Container instance.
7	monitorexit()	
8	goto(1)	

Figure 7.3: The code for the `run` method of the Worker class.

disassembler[JDK]. The only real difference is that the (in our model unreachable) exceptions handling code has been removed. The initial jump from the first instruction to the next is an artifact of the specific compiler. One interesting fact is that the reference used to unlock the monitor is stored in a local variable, instead of being retrieved from the class instance. This is not an optimization, it is done because the compiler does not know that the reference in the class instance will not change.

7.3.2 Basic Predicates and Proof Structure

To inspect the state of the system in our formulae, we need to define some predicates. The predicates we define are called $t2inCS$ and $t3inCS$, both of type $system \rightarrow prop$. These predicates are true if the thread with identifier 2 and 3, respectively, is in the critical section of the code. A thread is in the critical section, if it is in the list of active threads and its program counter has the value 6.

The actual definition of the predicates use the `contains` formula from Chapter 3.2 to find the thread. If the thread is unavailable (it resides in the queue of the container), the formula will be false, even though the program counter has the value 6. To check the value of the program counter, the case-construction is used.

Each of the following proofs are structured in the same way. Assume that the initial state of the system is sys, and that the property that we want to prove is prp, then the initial proof goal will be of the form $\vdash sysy : prp$.

7.3.3 No Global Fairness

In Section 5.4.1 on page 68, it is argued that the mutual exclusion locks in the developed model are not fair, even though they are modelled using a queue to store the waiting threads. The reason is that a thread has no guarantee of ever entering the queue. In this section, we will prove that this is the case for the two worker threads.

The Property The property that we prove is that the thread with identifier 2 is not guaranteed to enter the critical section. The choice of which thread to prove this property for is arbitrary, since the system is symmetric. The property is expressed formally in VCPT as $\neg$`Eventually`$(t2inCS)$.

The Proof Removing the negation from the property moves the system to the left hand side. This gives us a goal with a least fixed point to the left. Approximating the formula gives us a basis for induction. The strategy for the proof is to execute thread 3 until the induction goes through.

One unfolding of the formula results in a disjunction with $t2inCS$ and a clause for advancing the system. The first disjunct is easy to handle, thread 2 is still at the start, and the formula is false. Advancing the system is done through a conjunction of a diamond modality, expressing the existence of at least one next state, and a box modality for recursive validity of the property. Since we always have some next state, we can disregard from the diamond modality (by weakening out the formula). Following the box modality results in a choice which thread to execute, where we always chose to execute thread 3.

The above strategy for unfolding the formula one time can be iterated, until it is possible to invoke the rule of discharge successfully (which is possible when the system has looped once).

7.3.4 Eventual Exit from a Queue

When a thread in our system has entered the queue waiting for entry into the critical section, no other thread not in the queue may pass it over. In this section, we will prove a property that depends on this fact.

The Property The property that we prove is that at some point in time in the possible executions of the system, it is guaranteed that thread 2 will get to enter the critical section, even though it is not there yet. Formally, this is expressed as $\texttt{Sometime}(\neg t2inCS \wedge \texttt{Eventually}(t2inCS))$.

The Proof To prove this property, we have to demonstrate that such a sought state exists. The `Sometime` definition allows us to follow any path that we like through the programs execution, and then prove the validity of the argument property from there. Doing this, we can quickly reach a state where thread 3 has entered the critical section. Now we can advance thread 2 until it enters the queue of the container. This is the sought point in time for the `Sometime` definition.

Proving the nested property is equally easy. The first conjunct of the argument property is true (since thread 2 is not among the threads eligible for execution). To prove the `Eventually` formula, we can do a guided simulation until all possible paths through the system have been investigated. Since there is no branching possible of the execution, we can follow the program along until thread 3 wants to exit the critical section. When it does so, thread 2 is activated again, and is in the critical section. Now the argument property of the `Eventually` formula is true, and all the goals are taken care of.

7.3.5 Mutual Exclusion

In the comments to the program, we have stated that the program we examine has a critical section. A critical section is implemented correctly if no more than one thread may be in the critical section at any one time. In this section we will prove this mutual exclusion property holds for our system.

The Property The property that we prove is that thread 2 and 3 will never both be in the list of threads eligible for execution and have a program counter that is equal to 6, i.e., they are not in their critical section at the same time. To express this formally, we use the formula `Always`$(\neg(t2inCS \wedge t3inCS))$. This formulation uses the fact that "never" can be expressed as "always not".

The Proof This property is more complicated to prove than the previous two properties. Following modalities in this case means that we must explore all the possibilities in the branching structure of the system. For each of these points in the execution, we must validate the truth of the argument formula. The problem is that, although the system is small, there are more cases to consider than is practical to do for hand. Thus, we must automate the proof. First we will describe the basic tactic used and the proof-search employed. Then we will present some statistics of the proof generated.

To advance on goal into the next possible states, we unfold the `Always` definition. After the conjunction is split, the validity of the argument property is handled using the simplification routines from Section 3.3, augmented with the tactic for handling a `contains`-formula and the unfold-apply tactic. The box modality is handled by applying the box tactic and the transition relation tactic.

We initiate our proof by approximating the goal formula. The proof search is a breadth first search through the proof tree. At each goal in the current level, we try to discharge it using `copydischarge1` and `discharge1`. If this is not possible, we apply the tactic described in the previous paragraph to generate one or two new goals, depending on the current state of the system.

Using the above proof search on a 600 MHz computer, the proof was completed after 17 minutes of CPU time. The size of the proof graph is 376 proof nodes, where 186 of these nodes are leaf nodes. Of these leaf nodes, 19 were discharged using the global rule of discharge and the rest were subsumed by some other node.

Chapter 8

Conclusions and Further Work

This thesis has presented an operational semantics of a subset of the Java Virtual Machine (JVM) and an embedding of it into the VeriCode Proof Tool (VCPT). In this chapter, we will give some conclusions about the semantics and its embedding, and present some topics where further work is needed.

8.1 Conclusions

The semantics of the JVM is described in a book by Lindholm and Yellin[LY99] in an informal style. In this thesis, a formal operational semantics for a subset of the JVM is given, and the semantics is mechanised in a proof assistant.

There are many different descriptions of the semantics of the JVM available, for example [M5, Bal, SBS01]. The semantics in this thesis have the following distinguishing features.

- It contains a treatment of the semantics of JVM threads of computation.
- The semantics are presented as an operational semantics.
- The semantics are embedded in an advanced proof tool, supporting lazy discovery of induction schema.

Modelling of JVM level threads is not common in the available semantics (the exception being the M5 model). The usage of multiple threads of computation in programs is increasing, stressing the need for a semantical investigation into their definition and behaviour. The semantics used in this thesis present a clear view of the possible behaviour of parallel threads.

The development of semantics for the JVM has mainly taken place in the context of some proof assistant (as is the case of this thesis). The definition of the semantics is often tightly coupled to the system at hand, and presented as the input to the tool used. In contrast, this thesis tries to develop a semantics in a format that is readable without the need to understand VCPT in detail.

The proof assistant used for the development of the semantics, VCPT, is a powerful and interesting tool. The support for lazy discovery of induction schemas is a distinguishing feature, that makes the construction of proofs using abstract interpretation possible. This makes the environment of VCPT very interesting for examining properties of systems defined by operational semantics.

The example proofs in Chapter 7 shows that properties of small programs may be proven using the semantics. Proving properties of larger, more realistic, systems is a topic for further work.

The work presented in Appendix A on improving the user interface of VCPT, shows that using standard libraries, many of the interesting features of more complex user interfaces for proof assistants can be implemented easily. The extensions made VCPT much more user friendly, and removed many of the small annoyances of the interface to the tool that existed previously.

8.2 Further work

In the discussions in Chapters 5 and 6, some specific remarks and suggestions for improvements and extensions are given for the work presented there. In this section, a more general view is taken, and some possible directions for future work are outlined.

8.2.1 Improving and extending VCPT

VCPT implements a proof system that has many interesting and powerful properties. However, the support for actually defining a complex structure and managing larger proofs could be improved.

One specific issue that was encountered, was the problem of specifying general operations on lists. While the list construct is available for any type, it is not possible to construct predicates that work for any list. This problem was solved in this thesis by using macro-expansion on the list predicates (as described in Section 3.2). While serviceable, it is not a good solution. A more advanced typing system for the logic where predicates could have some simple polymorphism in the definition, would make the use of lists and tuples more interesting and generally usable for defining interesting operations. This is a suggestion based on the perceived need of this feature, and no investigations on the impact of this for the logic has been done.

The simplifier presented in Section 3.3 is a first approach at simplification. There is no “real” simplification done at the conceptual level. It would be very interesting to expand these tactics, for example with rewriting techniques for simplifying complex predicate expressions about abstract data. This would help in using the system for reasoning about unbounded systems, where data naturally must be given in an abstract form.

8.2.2 The Semantics of the JVM

Obviously, the semantics of the JVM would be more interesting if it was a more complete model of the JVM. The addition of class hierarchies and exceptions would make the model more true to the spirit of the JVM. On the other hand, the addition of more complex data-manipulation instructions is not very interesting from a conceptual level, although they would increase the tool's applicability for the verification of a real programs.

An interesting question is how to handle the problem of native methods. While it is possible to extend the semantics further and further with new instructions for every new native method that is found, a more general framework could be sought. One approach would be to model a native method as a predicate that acts upon the whole program, and couple each native method to a predicate.

The concept of proof carrying code[NL96] is very interesting, with the possibility of truly secure mobile code as the ultimate goal. A basic necessity for such a scheme is some way in which to present proofs about the code in question. The JVM is a natural candidate when implementing mobile code, since it is designed to be platform neutral. The combination of the μ-calculus for expressing complex interesting properties and the possibility of using the presented semantics as a base for constructing proofs, would be interesting to investigate. Issues that arise, apart from the usability of the semantics for proving the properties, are how to represent a proof in a suitable manner and which properties to prove.

8.2.3 Embedding the JVM semantics

Defining the semantics in VCPT, much routine work in translating the definitions between the conceptual level of a TSS and the formal description as a predicate had to be done. After a definition was written, tactics for automating the computation with it was written. Many of these steps would benefit from being simplified. One interesting area is computing the value of a deterministic formula, that has sufficiently concrete arguments for simple evaluation to work.

One possibility for evaluating the semantics during the design, would be to use a Prolog system to simulate them. For a semantics such as the one presented in this thesis, the Prolog implementation is a very natural translation of almost all constructs used. This would enable a more productive continuous evaluation.

The question of how a semantics should be embedded into VCPT is not clear; should it be described in the logic or should it be implemented as SML functions that manipulate the terms directly? The former has the advantage of being a formal definition, constrained by the logic it is written in. The latter has the advantage of being able to efficiently represent and implement complex concepts and side conditions; a pure logical formulation is inherently much slower than a programmed formulation is. The possibilities of the two approaches are both interesting, and worthy of examination. With the approach of using programmed rules, model checking small parts of the system, as described in [ACD$^+$03], could be a viable method for proving properties of some small programs.

8.2.4 Proving properties of JVM programs

The semantics presented in this thesis is not yet a practical tool for proving properties about JVM programs. There are many steps on the way for making a formal environment that is both suitable and practical.

The ideal environment for constructing proofs of program properties, will abstract away as much of the technical detail as possible. Advanced simplifiers, appropriate representations and abstractions, and the ability to construct hierarchical proofs are all necessary tools. The issue of simplifiers has been discussed above.

The representation of a program must be appropriate for constructing proofs. Perhaps the approach taken in this thesis with a strong connection to the specification is inappropriate in this regard. Constructing the semantics in a different style, perhaps by translating the systems into something that is close to process algebra instead, would be more productive.

The ability to construct proofs in a hierarchical fashion is necessary when tackling larger systems. However, it is not an easy task. In [Chu04], Chugunov investigated the possibilities of proving properties of side effect free Erlang expressions in the EVT environment. The approach taken there used pre- and post-conditions of expressions to express their behaviours. An interesting idea is to study the possibilities for proving such properties of JVM programs.

Appendix A

The VCPT User Interface

This appendix reports on work done improving the user interface of the VeriCode Proof Tool. VCPT is implemented in Standard ML (SML), using the top-level environment as the user interface. The top-level environment is a command-line where the user may evaluate expressions in SML. This is a fairly standard approach taken by many theorem-provers and proof-assistants, originating with the theorem-prover LCF[GMM+78], for which the ML language was created. For more information about Standard ML, a dialect in the ML language family, see [MCP93].

In addition to the above interface, there is a connection to the DaVinci graph visualisation system[uDr]. The connection enables the user to see a graphical view of the current proof. The connection is implemented as a program written in C, that filters the input and output of the SML system, invoking DaVinci when the user requests it.

A.1 Alternative User Interfaces

Even though many proof-assistants use the ML top-level environment as a user-interface, it is not the only style used.

In [Chu04], Chugunov presented a graphical user interface for the EVT system (on which VCPT is based). This GUI gives a more structured view of the current proof, with simplified navigation. It also supported proof-by-pointing, a technique where proofs are constructed by clicking on the next part of a goal that is to be handled. Furthermore, there is support for lemma-libraries and for recording proofs.

The ACL2 theorem prover is integrated into a Lisp-environment. This means that the tool works much in the same way as VCPT, but with Lisp as the command-language instead of SML.

The Proof General user interface is a generic user interface for theorem proving tools. It has support for Isabelle, Coq, PhoX, and LEGO, as well as experimental support for, for example, ACL2. Proof General is based on the Emacs editor, and works by filtering interaction with the base proof assistant, typically a top-level ML system. Among the features, it has proof-by-pointing, real mathematical symbols, remote execution of the proof assistant, and split buffer interaction (a command buffer, a buffer to display the current goals, and a buffer to show messages from the proof assistant). The purpose of Proof General, apart from being a better user interface than the base programs, is to be portable between different proof assistants.

The Coq proof-assistant has traditionally been used in conjunction with the Proof General user interface. In version 8 of Coq, there is also a new GUI called CoqIde. The new user interface is in the same style as Proof General, but is not Emacs-based.

A.2 Improving the VCPT User Interface

A proof-assistant used for proofs about computer systems, is a tool mainly targeted towards an expert audience that has a high degree of computer literacy. Furthermore, we can expect that the user is willing to invest some effort in learning to use the tool efficiently. Our assumption is that the purpose of the user interface is to allow the user to instruct the tool efficiently and to understand the effect the instructions has on the state.

The possibility to port the Proof General interface to VCPT was reviewed, as this would have been a good approach to get a full-featured user interface. Some experiments were done, but without much success. As a consequence, a simpler approach was taken.

A.2.1 Basic approach

The basic approach taken is to use the already existing filter of the interaction between the user and VCPT (a filter that handles the interaction between VCPT and DaVinci) for extending the user interface. This approach was chosen for the simplicity; the total time for adding the new features was two days.

A new level of control of the tool has been added to the tool. Any expression starting with a colon, is interpreted as a command instead of an SML expression. The commands added are presented in Table A.1. In some cases, the descriptions refer to the features presented below.

A.2.2 Instructing the tool

When the user instructs the tool, he writes Standard ML expressions at the prompt, thus submitting them for evaluation by the tool. To simplify the textual construction of these expressions, the GNU Readline Library[Rea] has been added to handle the interactive editing of the commands.

The readline library has support for editing a command-line using either Emacs-style key-bindings or vi-style key-bindings. The implementation currently uses the Emacs bindings, although this is easy to change. Features supported through this

Command	Description
`:help`	Prints information about the available commands.
`:exit`	Ends the VCPT session.
`:write` *file*	Writes the current proof-history to the file *file*.
`:read` *file*	Reads commands from the file *file*.
`:clear`	Clears the current proof-history.
`:!` *text*	Executes the *text* as a command in the current shell.
`:print_goals`	Toggles the printing of the current goal on or off.

Table A.1: The new commands added to VCPT.

are, for example, advanced cut-and-paste commands for removal and insertion of words or whole lines of text.

Furthermore, there is also a history function with advanced search-capabilities. This works by placing every line of text that the user inputs into a history buffer. The user then has several possibilities to recall a particular line of text. Most notably, the user can either search through the history linearly (using the arrow keys), or interactively. To search interactively, the user presses Control-R and writes some text. The most recent possible match in the history is then shown at the prompt.

A.2.3 Recording proofs

An important task for a proof-assistant is to prove theorems. In the original version of VCPT, there is no concept of recording the proof. If the user wants to be able to save the steps necessary to perform the proof, it is his responsibility to do so (this applies, of course, to theorems proved interactively, not to programmed proofs as the one in [BGH02]).

When constructing a proof interactively, many of the interactions are not correct applications of the tactics. Instead, one tries to apply tactics, and see if they succeed. This is a difference between a proof and a simple history as in the previous section. To approximate the proof performed, we introduce the notion of *successful interaction*. A successful interaction with VCPT is any interaction that does not result in an exception. To find these interactions, we parse the output of VCPT looking for exceptions.

A proof is then any sequence of successful interactions. To handle these sequences, we use the commands `:clear`, `:write`, and `:read`, as described in Table A.1.

A.2.4 Presenting state

When successful execution of an expression is done, the current goal has often changed. Using the detection of successful interaction from recording proofs, we now know when there is a possibility that the current goal has changed. To present the new goal, we let the ML process evaluate `print_goal()` after each successful interaction.

This feature can be turned on or off using the command `:print_goals`. The purpose of turning the feature off, is that very large goals can take some time to print, and we may sometimes want to skip this when we are familiar with the goals and the tactics that we apply.

A.3 Usability of the new user interface

The usability of the new user interface has only been tested by the author. Despite this, some statements about the usability of the tool can be made. In total, the perceived ease of use of the tool increased much with the new UI. Below, some specific comments on the parts are given.

Instructing the tool The basic method of instructing the tool is still the same - entering commands at a command-line. To do this, we must either write a new command or recall an earlier command for the same thing.

The history of the command-line is very convenient for on-the-fly automation of repetitive steps thanks to the quick interactive search through the history that is available. When some steps have been performed once, they can be recalled from the history and executed without the burden of re-writing them. It is also useful when we want to change a previously entered expression in a small way.

Editing a line of text is much more convenient with the advanced commands from Emacs or vi. Examples of useful features are deleting words at the cursor and cutting and pasting lines of text. The familiarity of the commands from Emacs made the task of learning to use the interface effectively much easier.

Recording a proof The basic facilities for recording a proof implemented has, on occasion, been very useful. The main use has been when a new formula is tried out, and after a change, we want to reload the theory and perform the same steps to take us to the point in question. Recording the steps and running them from a file is a simple procedure, thus enabling the easy use of the feature.

The detection of failed tactics works. Unfortunately, there is no detection of aborted tactics. This should be corrected to make the system useful, since some tactics may need to be aborted.

Using the feature for recording actual proofs has not been done in any large scale (proofs with more than fifty applications of tactics), since not many proofs have been done. Thus, there is not enough experience to comment on the usefulness of these features for recording proofs.

Understanding the output Most successful applications of tactics transform the current goal, and possibly generates additional goals. The simple printing of the current goal that has been added is very useful for the class of tactics where the current goal is the only one affected. The tool gives immediate feed-back, which for many cases is sufficient to correctly interpret the new state.

On the other hand, the author has often missed that more than one goal is produced from some tactics. It could be argued that this is just a matter of training, and that the user should be able to recognise the tactics where more things happen. But signalling these cases is just the kind of automation that we seek from a user-interface; relieving the user of tedious tasks.

A further drawback is that printing a large goal consumes a non-negligible amount of time, making the interactive use of the assistant less attractive. When very large goals are handled, it is useful to turn this feature off, and only print goals when necessary.

A.4 Further Improvements

There are some further improvements that could be interesting and useful to implement. In this section, we present some further ideas that are not (yet) implemented in VCPT.

The Readline Library has advanced features for tab-expansion of values. At the present, this is only used for finding file-names in the current directory (which, for example, can be used when reading and writing files with proofs in them). The most interesting use of tab-expansion would be to expand tactics and commands. To make the tab-expansion really useful, not only the pre-existing commands (e.g., And_l) should be expanded, also commands that are defined interactively or from files read. These commands could be found by parsing the output looking for lines such as

```
val Simplify_tacs_l = fn : (int -> tactic) list -> unit
```

which should lead to the addition of `Simplify_tacs_l` in the list of possible expansions.

Sometimes, as described earlier, it is not clear that the application of a tactic generates many goals. This problem could be alleviated by adding some sort of output that indicates either the current number of goals, or the change in number of goals. To add the number of goals, we could use the function `num_goals()` available in VCPT, but it is probably more interesting to make a clear statement about the changes in number of goals.

Sometimes one expression is entered over several lines (for example, when pasting an ML function definition from another application). It would be very useful to detect this and to concatenate the lines into one line for the history.

A more advanced form of collecting repeated sequences of interaction can also be envisioned. The idea is that the user could perform an interactive recording of the interaction anytime, and then recall this recording easily. Furthermore, there could be multiple buffers of recorded interaction. Appropriate key-bindings could be Control-Fn to initiate and stop a recording, and Alt-Fn to recall a recording.[1]

These ideas for further development of the presented extensions to the user interface could all be very useful.

[1] This feature was suggested by Gunnar Kreitz.

Appendix B

Formulae for managing lists

This appendix contains a listing of all the formulae described in Chapter 3, with their definitions in the μ-calculus. The formulae are shown with a generic sort for elements, called *elem*.

```
create : nat → elem → elem list → prop ⇐
  λSize:nat. λInitval:elem. λRes:elem list.
    cases (Size, Res) of
      (0, [])                  ⇒
        ⊤
    | (S'+1, [Initval | Rest]) ⇒
        create S' Initval Rest
    end
end;
```

```
at : elem list → nat → elem → prop ⇐
  λA:elem list. λN:nat. λE:elem.
    cases A of
    [Head | Tail] ⇒
      cases N of
        0    ⇒ Head = E
      | N'+1 ⇒ at Tail N' E
      end
    end
end;
```

```
 set : elem list → nat → elem → elem list → prop ⇐
  λA:elem list. λN:nat. λE:elem. λA':elem list.
    cases (A, A') of
    ([H | T], [H' | T']) ⇒
      cases N of
        0    ⇒
          E=H' ∧
          T=T'
      | N'+1 ⇒
          H=H' ∧
          set T N' E T'
      end
    end
end;
```

```
size : elem list → nat → prop ⇐
  λA:elem list. λN:nat.
    cases A of
    [Head | Tail] ⇒
      if Tail = [] then
        N = 1
      else
        cases N of
        N'+1 ⇒ size Tail N'
        end
    end
end;
```

```
extend : elem list → elem → elem list → prop ⇐
  λA:elem list. λE:elem. λA':elem list.
    cases A of
      []       ⇒
        [E] = A'
    | [H | T]  ⇒
        cases A' of
        [H | T'] ⇒ extend T E T'
        end
    end
end;
```

```
 remove : elem list → nat → elem list → prop ⇐
  λA1:elem list. λN:nat. λA2:elem list.
    cases (A1, N) of
      ([H1 | T1 ], 0)    ⇒
        A2 = T1
    | ([H1 | T1 ], N'+1) ⇒
        cases A2 of
        [H1 | T2] ⇒ remove T1 N' T2
        end
    end
end;


LET
reverse : elem list → elem list → prop =
  λA1:elem list. λA2:elem list.
    impl A1 [] A2
WHERE
  impl : elem list → elem list → elem list → prop ⇐
    λA1:elem list. λA2:elem list. λRes:elem list.
      cases A1 of
        []       ⇒
          A2 = Res
      | [H | T]  ⇒
          ∃ A2':elem list.
            A2' = [H | A2] ∧
            impl T A2' Res
      end
  end
END;
```

```
LET
set_from : elem list → nat → elem list → elem list → prop ⇐
  λA1:elem list. λN:nat. λA2:elem list. λA3:elem list.
    cases (N, A1, A3) of
      (0, X, Y)                ⇒
        impl A1 A2 A3
    | (N'+1, [H | T1], [H | T3]) ⇒
        set_from T1 N' A2 T3
    end
WHERE
  impl : elem list → elem list → elem list → prop ⇐
    λA1:elem list. λA2:elem list. λA3:elem list.
      cases A2 of
        []       ⇒
          A1 = A3
      | [H2 | T2] ⇒
          cases (A1, A3) of
          ([H1 | T1], [H2 | T3]) ⇒
            impl T1 T2 T3
          end
      end
  end
END;


split : elem list → nat → elem list → elem list → prop ⇐
  λA1:elem list. λN:nat. λA2:elem list. λA3:elem list.
    cases (A1, N, A2) of
      (X, 0, [])               ⇒
        A1 = A3
    | ([H | T1], N'+1, [H | T2]) ⇒
        split T1 N' T2 A3
    end
end;
```

```
LET
split_rev : elem list → nat → elem list → elem list → prop =
  λA1:elem list. λN:nat. λA2:elem list. λA3:elem list.
    impl A1 N [] A2 A3
WHERE
  impl : elem list → nat       → elem list
      → elem list → elem list → prop ⇐
  λA:elem list. λN:nat. λAcc:elem list.
  λTop:elem list. λRest:elem list.
    cases (A, N) of
      (X, 0)          ⇒
        To' = Acc ∧
        A   = Rest
    | ([H | T], N'+1) ⇒
        impl  T N' [H | Acc] Top Rest
    end
  end
END;


contains : elem list → key → val → prop ⇐
  λA : elem list . λK : key . λV : val .
    cases A of
    [Head | Tail] ⇒
      cases Head of
      constructor(Key, Val) ⇒
        if K = Key then
          V = Val
        else
          contains Tail K V
      end
    end
end
```

Bibliography

[ACD$^+$03] Thomas Arts, Gennady Chugunov, Mads Dam, Lars-Åke Fredlund, Dilian Gurov, and Thomas Noll. A Tool for Verifying Software Written in Erlang. *International Journal on Software Tools for Technology Transfer (STTT)*, 4(4):405–420, August 2003.

[ACL] ACL2. http://www.cs.utexas.edu/users/moore/acl2/. A theorem prover for Applicative Common Lisp.

[AFV01] Luca Aceto, Willem Jan Fokkink, and Chris Verhoef. *Structural Operational Semantics*, chapter 3. In Bergstra et al. [BPS01], 2001.

[AGM92a] Samson Abramsky, Dov M. Gabbay, and Thomas S. E. Maibaum, editors. *Handbook of Logic in Computer Science, Background: Computational Structures*, volume 2. Oxford University Press, 1992. ISBN 0-19-853761-1.

[AGM92b] Samson Abramsky, Dov M. Gabbay, and Thomas S. E. Maibaum, editors. *Handbook of Logic in Computer Science, Background: Mathematical Structures*, volume 1. Oxford University Press, 1992. ISBN 0-19-853735-2.

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: principles, techniques, and tools.* Addison-Wesley Longman Publishing Co., Inc., 1986. ISBN 0-201-10088-6.

[Bal] Project Bali. http://isabelle.in.tum.de/bali/. A formalization of various aspects of the Java programming language in Isabelle[Isa].

[BCMS01] Olaf Burkart, Didier Caucal, Faron Moller, and Bernhard Steffen. *Verification on Infinite Structures*, chapter 5. In Bergstra et al. [BPS01], 2001.

[BGH02] Gilles Barthe, Dilian Gurov, and Marieke Huisman. Compositional verification of secure applet interactions. In *Procceedings of FASE'02*, volume 2306 of *Lecture Notes in Computer Science*, pages 15–32. Springer-Verlag, 2002.

[BPS01] Jan A. Bergstra, Alban Ponse, and Scott A. Smolka, editors. *Handbook of Process Algebra.* Elsevier, 2001. ISBN 0-444-82830-3.

[BS01] Julian C. Bradfield and Colin Stirling. *Modal logics and mu-calculi: an introduction*, chapter 4. In Bergstra et al. [BPS01], 2001.

[CDH$^+$00] James C. Corbett, Matthew B. Dwyer, John Hatcliff, Shawn Laubach, Corina S. Păsăreanu, Robby, and Hongjun Zheng. Bandera: extracting finite-state models from java source code. In *International Conference on Software Engineering*, pages 439–448. ACM Press, June 2000.

[Chu04] Gennady Chugunov. *Reasoning about Side-Effect Free Erlang Code in a Modal μ-calculus Based Framework.* Licentiate thesis, Royal Institute of Technology, Stockholm, Sweden, 2004. ISSN 1652-4076.

[Dam92] Mads Dam. CTL* and ECTL* as fragments of the modal μ-calculus. In *CAAP '92: Proceedings of the 17th Colloquium on Trees in Algebra and Programming*, pages 145–164. Springer-Verlag, 1992. ISBN 3-540-55251-0.

[EH83] E. Allen Emerson and Joseph Y. Halpern. "Sometimes" and "not never" revisited: on branching versus linear time (preliminary report). In *POPL '83: Proceedings of the 10th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 127–140, New York, NY, USA, 1983. ACM Press. ISBN 0-89791-090-7.

[ESC] Extended Static Checker for Java. http://www.sos.cs.ru.nl/research/escjava/main.html. A program for finding common errors in Java programs, using formal modelling and automatic theroem proving.

[Fre01] Lars-Åke Fredlund. *A Framework for Reasoning about Erlang Code.* PhD thesis, Royal Institute of Technology, Stockholm, Sweden, 2001. ISSN 1403-5286.

[Gla01] Robert J. van Glabbeek. *The Linear Time – Branching Time Spectrum I; The Semantics of Concrete, Sequential Processes*, chapter 1. In Bergstra et al. [BPS01], 2001.

[Gla04] Robert J. van Glabbeek. The meaning of negative premises in transition system specifications II. *Journal of Logic and Algebraic Programming*, 60–61:229–258, 2004.

[GMM$^+$78] Michael J. C. Gordon, Robin Milner, L. Morris, Malcolm C. Newey, and Cristopher P. Wadsworth. A metalanguage for interactive proof in lcf. In *POPL '78: Proceedings of the 5th ACM SIGACT-SIGPLAN symposium on Principles Of Programming Languages*, pages 119–130. ACM Press, 1978.

[GTL89] Jean-Yves Girard, Paul Taylor, and Yves Lafont. *Proofs and types*. Cambridge University Press, New York, NY, USA, 1989. ISBN 0-521-37181-3. Available from http://www.cs.man.ac.uk/~pt/stable/Proofs+Types.html.

[HM80] Matthew Hennessy and Robin Milner. On observing nondeterminism and concurrency. In *Proceedings of the 7th Colloquium on Automata, Languages and Programming*, pages 299–309, London, UK, 1980. Springer-Verlag. ISBN 3-540-10003-2.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969. ISSN 0001-0782.

[Hoa74] C. A. R. Hoare. Monitors: an operating system structuring concept. *Commun. ACM*, 17(10):549–557, 1974. ISSN 0001-0782.

[HP00] Klaus Havelund and Thomas Pressburger. Model checking Java programs using Java PathFinder. *International Journal on Software Tools for Technology Transfer (STTT)*, 2(4), April 2000.

[IPW01] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight Java: a minimal core calculus for Java and GJ. *ACM Trans. Program. Lang. Syst.*, 23(3):396–450, 2001. ISSN 0164-0925.

[Isa] Isabelle. http://isabelle.in.tum.de/. Isabelle is a generic theorem prover.

[Jav] The Java 2 Standard Edition Application Programming Interface, specification and documentation, version 1.5.0. http://java.sun.com/j2se/1.5.0/docs/api.

[JDK] Java 2 enterprise edition software development kit 1.4. Available from http://java.sun.com/j2ee/1.4/download.html.

[KeY] The key project. ww.key-project.org. An integration of a dynamic logic for Java into a CASE application.

[Koz82] Dexter Kozen. Results on the propositional μ-calculus. In *Proceedings of the 9th Colloquium on Automata, Languages and Programming*, pages 348–359. Springer-Verlag, 1982.

[Lam79] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Computers*, 28(9):690–691, 1979.

[Lam80] Leslie Lamport. "Sometime" is sometimes "not never": on the temporal logic of programs. In *POPL '80: Proceedings of the 7th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 174–185, New York, NY, USA, 1980. ACM Press. ISBN 0-89791-011-7.

[LY99] Tim Lindholm and Frank Yellin. *Java Virtual Machine Specification, 2nd edition.* Addison-Wesley Longman Publishing Co., Inc., 1999. ISBN 0-201-43294-3. Available at http://java.sun.com/docs/books/vmspec/.

[M5] M5. http://www.cs.utexas.edu/users/moore/publications/m5/. A model of the JVM for the ACL2[ACL] theorem prover.

[MCP93] Colin Meyers, Chris Clack, and Ellen Poon. *Programming with Standard ML.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1993.

[Mil89] Robin Milner. *Communication and Concurrency.* Prentice-Hall, Inc., 1989. ISBN 0-131-14984-9.

[Mil99] Robin Milner. *Communicating and mobile systems: the* π*-calculus.* Cambridge University Press, 1999. ISBN 0-521-65869-1.

[MPA05] Jeremy Manson, William Pugh, and Sarita V. Adve. The Java memory model. In *POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 378–391, New York, NY, USA, 2005. ACM Press. ISBN 1-58113-830-X.

[MT92] Karl Meinke and John V. Tucker. *Universal Algebra*, chapter 1. Volume 1 of Abramsky et al. [AGM92b], 1992.

[NL96] George C. Necula and Peter Lee. Proof-carrying code. Technical Report CMU-CS-96-165, Computer Science Department, Carnegie Mellon University, Sept 1996.

[NPW02] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL: a proof assistant for higher-order logic*, volume 2238 of *Leture Notes in Computer Science.* Springer-Verlag, 2002. ISBN 3-540-43376-7.

[Pau92] Lawrence C. Paulson. *Designing a Theorem Prover*, chapter 4. Volume 2 of Abramsky et al. [AGM92a], 1992.

[Pel01] Doron A. Peled. *Software Reliability Methods.* Texts in Computer Science. Springer, 2001. ISBN 0-38795-106-7.

[Plo81] Gordon D. Plotkin. A Structural Approach to Operational Semantics. Technical Report DAIMI FN-19, University of Aarhus, 1981.

[Rea] The GNU Readline Library. http://directory.fsf.org/readline.html.

[Rep91] John H. Reppy. An operationl semantics of first-class synchrounous operations. Technical Report TR 91-1232, Department of COmputer Science, Cornell University, 1991.

[RS92] Mark Ryan and Martin Sadler. *Valuation Systems and Conseqence Relations*, chapter 1. Volume 1 of Abramsky et al. [AGM92b], 1992.

[SBS01] Robert F. Stärk, Egon Börger, and Joachim Schmid. *Java and the Java Virtual Machine – Definition, Verification, Validation*. Springer-Verlag New York, Inc., 2001. ISBN 3-540-42088-6.

[SD03] Christoph Sprenger and Mads Dam. On the structure of inductive reasoning: Circular and tree-shaped proofs in the μ-calculus. In *Proc. FOSSACS'03*, Lecture Notes in Computer Science 2620, pages 425–440. Springer-Verlag, 2003.

[SPI] The SPIN model checker. http://www.spinroot.com/.

[SS94] Leon Sterling and Ehud Shapiro. *The art of Prolog (2nd ed.): advanced programming techniques*. MIT Press, 1994. ISBN 0-262-19338-8.

[Sti92] Colin Stirling. *Modal and Temporal Logics*, chapter 5. Volume 2 of Abramsky et al. [AGM92a], 1992.

[uDr] The uDraw graph visualization system (formerly know as DaVinci). http://www.informatik.uni-bremen.de/uDrawGraph/. uDraw is a program for visualizing graphs with automatic layout of the nodes. It is possible to embed it into programs as a visualization component.

[VCP] VeriCode Proof Tool. http://www.sics.se/fdt/projects/vericode/vcpt.html. A proof assistant for μ-calculus with support for defining operationl semantics.

[VCP02] *Reference Manual for the VeriCode Proof Tool for version 1.00*, September 2002. Available at ftp://ftp.sics.se/pub/fdt/vericode/vcpt/refman.pdf.

[Win93] Glynn Winskell. *The Formal Semantics of Computer Programming Languages: An Introduction*. Foundations of Computing Series. The MIT Press, 1993. ISBN 0-262-73103-7.