UNIVERSITY
of
GLASGOW

Department of
Computing Science

A Programming Logic for Java Bytecode Programs

Claire Louise Quigley

Submitted for the degree of
Doctor of Philosophy
in the University of Glasgow

June 2004

Abstract

One significant disadvantage of interpreted bytecode languages, such as Java, is their low execution speed in comparison to compiled languages like C. The mobile nature of bytecode adds to the problem, as many checks are necessary to ensure that downloaded code from untrusted sources is rendered as safe as possible. But there do exist ways of speeding up such systems.

One approach is to carry out static type checking at load time, as in the case of the Java Bytecode Verifier. This reduces the number of runtime checks that must be done and also allows certain instructions to be replaced by faster versions. Another approach is the use of a Just In Time (JIT) Compiler, which takes the bytecode and produces corresponding native code at runtime. Some JIT compilers also carry out some code optimization.

There are, however, limits to the amount of optimization that can safely be done by the Verifier and JITs; some operations simply cannot be carried out safely without a certain amount of runtime checking. But what if it were possible to prove that the conditions the runtime checks guard against would never arise in a particular piece of code? In this case it might well be possible to dispense with these checks altogether, allowing optimizations not feasible at present. In addition to this, because of time constraints, current JIT compilers tend to produce acceptable code as quickly as possible, rather than producing the *best* code possible. By removing the burden of analysis from them it may be possible to change this.

We demonstrate that it is possible to define a programming logic for bytecode programs that allows the proof of bytecode programs containing loops. The instructions available to use in the programs are currently limited, but the basis is in place to extend these. The development of this logic is non-trivial and addresses several difficult problems engendered by the unstructured nature of bytecode programs.

Acknowledgements

I would like to thank the following people:

- EPSRC for providing funding.
- Dr Alice Miller, for proof reading the dissertation.
- Dr Lewis MacKenzie, for allowing me to run Isabelle on one of his fast machines.
- Huw Evans, for advice on obscure Java features.
- Wolf Früh, Julie McAnulty, Jane Stuart-Smith and Jamie Thom, for fugue alerts and the Joke of the Week.
- Alison Stewart, for providing encouragement and (fairly traded) chocolate.
- My supervisors, Dr Simon Gay and Dr Tom Melham.
- Jonathan Paisley and Lyndell St Ville, for letting me turn G101 into a book-infested greenhouse.
- Daniel Velleman, for explaining "How To Prove It".
- Stuart Blair, Graham Collins, John Gormley, Dr Alison Haggith, Jonathan Hogg, Katie McGlew, Alasdair McLean, Morag Stark, and the members of the University Chapel Choir, for their friendship and support.
- My brothers Tony and Adrian, for comic relief, lifts in the pollen-mobile, and looking after Wimsey.
- My parents, for always being there to provide love and support.
- And finally to Michael, for love and pizza.

Contents

List of Definitions

List of Propositions

List of Lemmas

List of Figures

Chapter 1

Introduction

One significant disadvantage of interpreted bytecode languages, such as Java [3, 24, 18, 28], is their low execution speed in comparison to compiled languages like C. The mobile nature of bytecode adds to the problem, as many checks are necessary to ensure that downloaded code from untrusted sources is rendered as safe as possible. But there do exist ways of speeding up such systems.

One approach is to carry out static type checking at load time, as in the case of the Java Bytecode Verifier [29]. This reduces the number of runtime checks that must be done and also allows certain instructions to be replaced by faster versions. Another approach is the use of a Just In Time (JIT) Compiler, which takes the bytecode and produces corresponding native code at runtime. Some JIT compilers also carry out some code optimization [23].

There are, however, limits to the amount of optimization that can safely be done by the Verifier and JITs; some operations such as array bounds checking and type casting simply cannot be carried out safely without a certain amount of runtime checking [29]. But what if it were possible to *prove* that the conditions the runtime checks guard against would never arise in a particular piece of code?

In this case it might well be possible to dispense with these checks altogether, allowing optimizations not feasible at present. In addition to this, because of time constraints, current JIT compilers tend to produce acceptable code as quickly as possible, rather than producing the *best* code possible. By removing the burden of analysis from them it may be possible to improve on the current situation.

1.1 Java

Java is a concurrent object-oriented programming language developed by Sun Microsytems [2]. It is syntactically similar to C and C++, but imposes a ‘safer’ programming style than these languages. This is achieved by the use of stricter runtime type-checking, not allowing the user to manipulate pointers directly, and using automatic garbage collection as opposed to users explicitly allocating and deallocating memory.

Java was initially intended to be used in the construction of software systems running on networks of machines with varied architecture. This meant that it was important that the code produced should be portable—able to run on any machine in the network regardless of differences in architecture, and that a machine receiving some Java code across the network should be able to assure itself that the code was, to some degree of certainty, safe to run.

The issue of portability is addressed by a Java program being compiled into a *class file* containing architecture neutral *bytecodes*, which are then run on the Java Virtual Machine (JVM) [29, 55], an emulator running on a ‘real’ machine. This means that the same Java program can be run on any machine for which there exists an implementation of the JVM, without the rewriting/recompilation needed in traditional systems. The class files are designed to be downloaded from the Internet, which further simplifies the matter of obtaining new software.

Classes are loaded by the JVM using a class loader. The ‘primordial’ class loader, shipped with the JVM, loads both the trusted core classes shipped with the JVM and any classes that can be found on the CLASSPATH—a designated area of filespace [29, 30]. These classes will be assumed not to be malicious and are not subject to bytecode verification. If a class cannot be found on the CLASSPATH, a specialised class loader object will be instantiated to download it from a web server. These classes *are* subject to bytecode verification.

Once loaded, a class will be linked and initialized. During linking, classes obtained from outside the system the current JVM is running on will be verified by the JVM’s Bytecode Verifier. The Verifier ensures that the classfile meets certain criteria of type safety and well-formedness that mean it will not cause certain catastrophic problems at runtime, thereby dealing with the problem of the safety of code received across a network.

The verifier ensures, amongst other properties, that bytecode instructions receive the right number of arguments, and that the arguments are of the correct type and in the right order, that the operand stack will not overflow or underflow, the program counter is never pointing to somewhere outside the range of a method's code, and that objects are initialized before use. This ensures the type safety of any downloaded classes, i.e. that instructions are passed arguments of the correct type. It is assumed that classes written by a user and located on the CLASSPATH will have been created by the `javac` compiler and will therefore also be correctly formed. A full description of the verifier can be found in [29].

In addition to checking classes conform to certain rules, the Bytecode Verifier carries out some optimization of the bytecode by substituting faster versions of certain bytecodes (signified by the suffix `quick`). These instructions are more efficient because they do not contain checks that are redundant after bytecode verification.

Before it can be initialized, a class must be loaded and linked. It is up to an implementation of the JVM to decide whether it will load and link classes 'early', but a class must be loaded on its first active use. The initialization of a class will also trigger the initialization (preceded if necessary by loading and linking) of all its superclasses. A diagram outlining these operations can be seen in Figure 1.1. Note that the shaded parts of the diagram indicate the additional elements of the JVM necessary to run untrusted code from outside the local file system.

1.2 JIT Compilers

As mentioned, one of the biggest drawbacks of interpreted bytecode languages like Java is their slow execution speed. One solution to this might be to compile Java programs to native machine code rather than bytecode. But, unlike bytecode, the native code will be specific to a particular machine and, if downloaded from an untrusted source, cannot be verified by the JVM's bytecode verifier. The execution speedup will therefore be offset by a severe deterioration in the code's mobility—the main selling point of interpreted bytecode systems.

Systems using a Just In Time (JIT) compiler attempt to provide as much speedup as possible while still keeping the advantages of bytecode. This is done by downloading bytecode files and verifying them as usual, but then also calling the JIT compiler to translate the bytecode to native code at

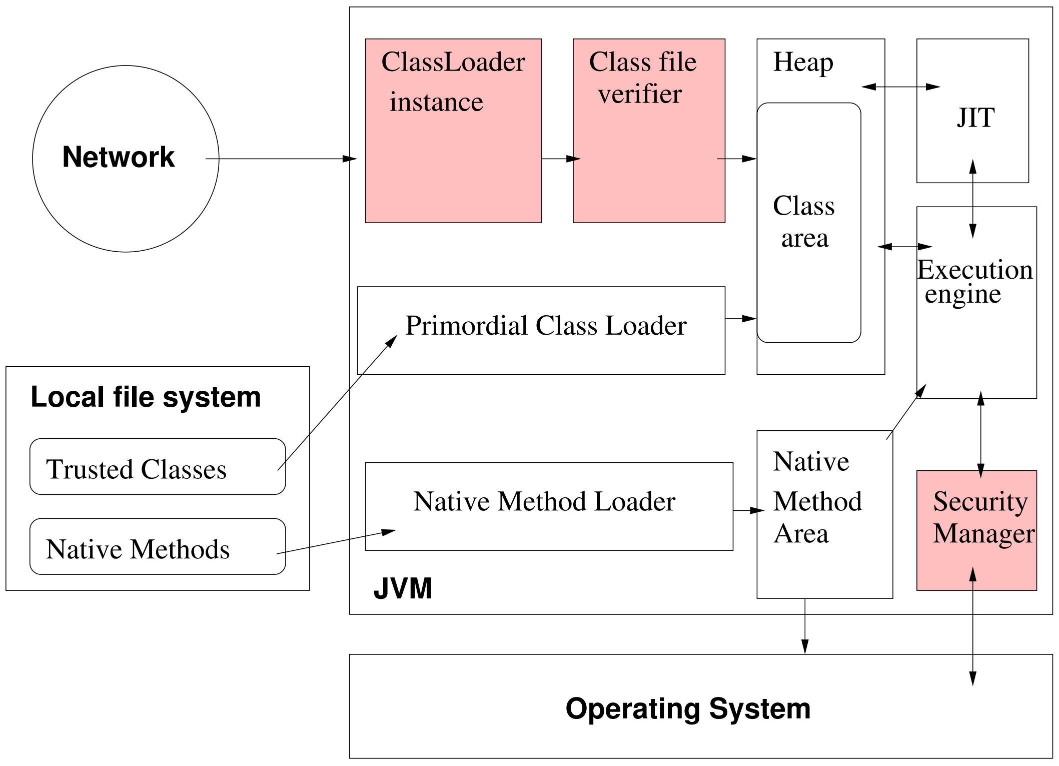

Figure 1.1: The Java Virtual Machine

runtime, producing the native code just before it is needed. JIT compilation is carried out on a method only at the point at which it is called so that unnecessary translation is not done. Some JIT compilers also profile code in order to determine whether it is worth compiling a method [23].

The native code produced by the JIT for a particular program is not stored after termination of the program but, during execution, the pointer to a method's code is replaced by a pointer to the compiled code for that method.

The most basic JIT compilers carry out a process known as *inlining*. The usual implementation of the JVM interpreter is a large switch statement with cases corresponding to the various instructions of the virtual machine; execution of a JVM program consists of repeated execution of this switch statement. This means that a signicant amount of time is spent executing the various jumps and comparisons involved in the switch statement itself, rather than in executing the instruction of the program running on the JVM.

Inlining the code means taking the corresponding native code instructions for each virtual machine instruction in a method's code and concatenating them into a single stream of machine code. This not only removes the overhead inherent in the interpreter, but means that it is no longer necessary to maintain the program counter and stack of the virtual machine; the method has effectively been detached from the virtual machine paradigm and can be treated just like any other program for the concrete machine it is running on.

For example, the chunk of code in one interpreter implementation, [19], corresponding to the JVM instruction `dup` is

```
ld [%l1 - 4], %o0  //load val on top of JVM stack into reg o0
st %o0, [%l1]      //store val in reg o0 at new top of JVM stack
add %l0, 1, %l0    //increment JVM program counter
add %l1, 4, %l1    //increment stack pointer
b .LL16            //branch
nop                //do nothing
```

Of these instructions, only half are actually concerned with carrying out the `dup` instruction; the others are there only to implement the interpreter. By inlining the code for `dup` we can reduce it to the following three instructions

```
ld [%l1 - 4], %o0  //load val on top of JVM stack into reg o0
st %o0, [%l1]      //store val in reg o0 at new top of JVM stack
add %l1, 4, %l1    //increment stack pointer
```

It is also possible to inline method calls, whereby the call to a method is replaced by the code of the method itself. This is usually only possible for very short methods.

In addition to this basic translation technique, more complex JIT compilers carry out optimization of the code. This is usually carried out on some sort of *intermediate code* which is at a lower level than that of the bytecode, while not actually native code. Methods used may include

Copy Propagation Bytecode is stack based whereas most 'real' machines are register based. The translation from bytecode to native code often causes unnecessary `mov` instructions, which move values from one register to another, to be generated. For example, `mov r1 -> r2` followed by `mov r2 -> r3` is equivalent to `mov r1 -> r3`. Copy propagation is concerned with eliminating these unnecessary instructions and can also be carried out backwards.

Assertion Merging When a bytecode instruction is broken down into simpler intermediate instructions it can become apparent that a particular assertion is being repeated, e.g. a reference is non-null or a number is non-zero. Analysis of the code may allow removal of these duplicate assertions.

Live Variable Analysis A variable is *live* if it holds a value that may be needed in the future. Therefore, if two variables in a program are never live at the same time, the same register may be used to store their values.

Dead Code Elimination This attempts to identify and eliminate instructions which carry out redundant operations.

Strength Reduction This attempts to replace an operation with an equivalent one that executes faster, e.g. use shift to divide and multiply by powers of two.

Common Subexpression Elimination Removes redundant calculations.

Loop Unrolling In cases where it is possible to calculate n, where n is the number of times the loop will be executed, remove the loop structure and replace with n copies of the loop body.

More information on code optimization can be found in [5] and [6].

One other way in which Java enforces type safety is by checking all array references at runtime to ensure that the array reference is non-null and that the index is not out of bounds. This avoids the potentially catastrophic results of writing to an area of memory outside the array bounds, a situation all too possible in programs written in C or C++. But it also means that bytecodes for array operations cannot be replaced by the more efficient `quick` bytecodes mentioned in Section 1.1. The following ix86 assembly code was produced by the GCJ program [44] and corresponds to the Java statement `testarray [i] := 2`

```
53:   movl   0xfffffff8(%ebp),%ebx     //bounds check
56:   movl   0xfffffffc(%ebp),%esi     //bounds check
59:   cmpl   0x8(%ebx),%esi            //bounds check
5c:   jb     70 <main__5ArrayPt6JArray1ZPQ34java4lang6String+0x50>
5e:   addl   $0xfffffff4,%esp
61:   pushl  %esi
62:   call   63 <main__5ArrayPt6JArray1ZPQ34java4lang6String+0x43>
67:   addl   $0x10,%esp
6a:   movl   %eax,%eax
6c:   testl  %eax,%eax
6e:   je     70 <main__5ArrayPt6JArray1ZPQ34java4lang6String+0x50>
70:   leal   0xc(%ebx),%eax
73:   leal   0x0(,%esi,4),%edx
7a:   addl   %edx,%eax
7c:   movl   $0x2,(%eax)
82:   leal   0xffffffe8(%ebp),%esp
85:   popl   %ebx
86:   popl   %esi
87:   movl   %ebp,%esp
89:   popl   %ebp
```

Only the instructions

```
70:   leal   0xc(%ebx),%eax
73:   leal   0x0(,%esi,4),%edx
7a:   addl   %edx,%eax
7c:   movl   $0x2,(%eax)
```

actually update the array, and so if it was possible to prove that the array bounds check for this operation was unnecessary it would be possible to eliminate the extra instructions. As it is often the case that an instruction such as `testarray [i] := 2` appears in the body of a loop in order to carry out an operation on the entire array, the number of instructions eliminated could potentially be quite large.

1.3 Reasoning about Programs

In order to reason about programs it is necessary first to build a logical model of the world of the programs: the language used to write them and its semantics, and the environment in which they run. This section provides a brief background to some techniques used to do this which are of particular relevance to the work described in the rest of this report. A detailed discussion of all the topics in this section can be found in [56].

1.3.1 Assigning Meaning to Programs

There are three main approaches to formalising the meaning of a program: operational semantics, denotational semantics, and axiomatic semantics. The ideas behind axiomatic semantics will be dealt with in the following section 1.3.2.

An operational semantics for a language is defined in terms of the operations carried out by an abstract machine, where a rule is stated defining the result of execution for each type of command in the language. Commands are identified syntactically, e.g. a rule would exist for an assignment statement $x := expr$. Application of the rules leads to evaluation of an expression in the language in relation to a particular *state* which encapsulates the environment in which the program is being executed. Evaluation can be described either as one complete operation, i.e. a boolean expression evaluating to a boolean value, or as a series of smaller transformations on a state leading eventually to a value. The former is known as a 'big-step' semantics, the latter a 'small-step' semantics [56].

A denotational semantics formulates the meaning of a program more abstractly as a partial function from states to states rather than rules of execution for particular syntactic constructs. This has the advantage of making it possible to compare the equivalence of two programs written in two different languages.

1.3.2 Reasoning about Program Properties

In his seminal paper *An Axiomatic Basis for Computer Programming*, [20], C. A. R. Hoare describes a set of rules (or axioms) that can be used to reason about what a program *does*. Rules of the form described in the paper are often referred to as a programming logic or *Hoare logic*, and allow us to go one step further than an operational or denotational semantics in terms of reasoning about programs. Rather than just allowing reasoning about the *value* of initial and final states with regard to a program, a Hoare logic allows us to make more fine-grained statements about states. We are able to say whether if we execute a command C in any state satisfying the predicate P—and execution terminates— we will end up in some state satisfying Q.

A Hoare logic specification takes the form

$$\vdash \{P\}\ C\ \{Q\}$$

where

- C is a statement in the programming language
- P is a *precondition*
- Q is a *postcondition*

Partial correctness specifications (signified by the curly brackets around pre- and post-conditions) do not require proof of termination.

Although in both operational semantics and Hoare logic predicates are used to assist reasoning about programs, they are essentially quite different:

- **Operational Semantics** describe what the operating environment of the program 'looks like' after the execution of each instruction. This description of the environment is known as the *state* and includes information such as the types of values held on the stack and in local variables.
- **Predicates in Hoare Logic** describe properties *which are true* at a particular point in the execution of a program (i.e. in a particular state). For example 'the value at the top of the stack is greater than 10' or 'variable x has the value 7'.

In situations where an operational semantics for a language exist, it is possible to formulate the Hoare triple in terms of the logic in which the semantics is described. In higher order logic the relationship between the Hoare rules and the operational semantics can be defined as follows:

$$\{P\}\ C\ \{Q\}\ \equiv\ \forall\ \sigma\ \sigma'.\ P(\sigma)\ \wedge\ \mathsf{eval}(C,\sigma) = \sigma'\ \Longrightarrow\ Q(\sigma') \qquad (1.1)$$

This states that a Hoare Logic specification $\{P\}\ C\ \{Q\}$ is equivalent to the statement that for all states, σ, σ', if P holds in the state σ, and executing C in the state σ results in the state σ', then Q will hold in the state σ'.

Although the rules of a Hoare logic are often stated as axioms—hence the alternative description of such rules as an *axiomatic semantics*—it is also possible to derive them from the operational semantics using 1.1. Derivations from denotational semantics are equally possible, as described by Gordon in [16].

Derivation from an operational or denotational semantics results in a programming logic in which we may have more confidence than one in which the rules are merely arbitrarily stated. This is particularly true if we are dealing with a language which is sufficiently different from the simple imperative language described by Hoare as to make it unclear exactly what form the rules should take. This is the situation described in Chapter 4, in which we describe the derivation of a programming logic for bytecode programs, which are certainly very different from the programs dealt with by Hoare. Our derivation is based on the operational semantics for the JVM developed by Pusch [45], and this is described in some detail in Section 2.1.2.

1.3.3 Inductively Defined Relations

It is often the case that operational semantics and other execution relations are defined in terms of *inductively defined sets*. And while Pusch's semantics are not defined in this way, two of the execution relations for bytecode described in Chapter 3 are. Consequently we give a brief outline of the concepts involved in such a definition and the related technique of *rule induction*. This technique is described fully by Winskel in his book *The Formal Semantics of Programming Languages* [56].

We can inductively define a set by a collection of rules. For example, the set of odd numbers is defined by the rules

$$\frac{}{\mathit{Odd}\ 1}$$

$$\frac{\mathit{Odd}\ x}{\mathit{Odd}\ (x\ +\ 2)}$$

In order to show that a property P is true of all members of such a set, we use the principle of rule induction. This is based on the idea that if a property is preserved by application of all the rules defining the set, it is true for all members of the set. The principle can be stated as: if for all axioms

$$\frac{}{x}$$

$P(x)$ is true, and for all rules of the form

$$\frac{x_1,\ x_2, \ldots, x_n}{x}$$

the statement $\forall i.\ 1 \le i \le n \implies P\ x_i \implies P\ x$ holds, then $P(n)$ holds for any n in the set.

1.4 Mechanized Reasoning

Much of the work that has been done recently on proving properties of the Java language has involved the use of some form of mechanical proof assistant [38, 45, 26, 46]. As the semantics for even subsets of Java, or simplified versions that disregard aspects of the language such as exception handling, are very large and complex it is easy for mistakes to creep into a paper and pencil proof. Indeed one of the few substantial pieces of work undertaken in the area without the aid of a proof tool, that of Drossopoulou and Eisenbach [14], was found to contain ‘one major error and one noteworthy omission’ when checked by Syme using his proof tool Declare [52].

This does not imply that all proofs carried out using a mechanised proof tool are completely flawless. All results depend, at the bottom line, on the definitions provided by the user. But if such definitions are correct, a proof tool can be relied upon to provide a greater degree of reassurance that results produced are also correct, leaving critics only the smaller task of examining the definitions. Obviously the degree of confidence in the proofs depends to a great extent on the proof tool used and its implementation.

1.4.1 Isabelle

Our work, and that of Pusch on which it is based, uses the Isabelle system [42]. Most proof systems support one particular logic from among the many used by computer scientists. For example, the HOL system [33] supports reasoning in higher order logic, whereas Larch [51] supports proofs in multisorted first order logic. Isabelle, designed by Paulson, is a *generic* prover, meaning it supports a variety of logics (known as object logics). Isabelle has a meta logic which is used to formulate object logics. The meta logic is the subset of higher order logic containing implication, universal quantification, and equality. A full description of the system can be found in [42], and discussion related to implementation and development issues in [40, 41, 43].

Like HOL, Isabelle is based on the LCF prover designed by Milner and his colleagues in the 1970s [32, 17]. In LCF, terms and formulae are values in ML, the meta-language used to implement the system, and can be composed and decomposed by ML functions. Theorems are values of type *thm*. Rather than constructing theorems arbitrarily, *inference rules* are used to map existing theorems to new theorems—starting with a small set of built in theorems, known as axioms. New theorems can be proved in two ways: by working forwards, using the inference rules to map already proved theorems to new theorems; or working backwards, splitting the original goal into smaller goals which can be proved trivially using existing theorems and inference rules. The process of backwards proof is managed by functions known as *tactics*.

Isabelle is an interactive prover, meaning that while it supports a variety of potentially complex logics, the user is expected to find the proof. Isabelle does, however, provide several powerful automatic tactics in the form of decision procedures based on non-logic-specific tableaux methods. Isabelle is a procedural prover, but recently the Isar interface has been developed [1], providing a more declarative interface.

The definitions and datatypes that make up each logic are stored in a *theory* file (denoted by the suffix `.thy`). Proof scripts for derivation of new theorems from these definitions are stored separately in files with the suffix `.ml` (for

ML).

1.5 Contribution

We demonstrate that it is possible to define a programming logic for bytecode programs that allows the proof of bytecode programs containing loops. The instructions available for use in the programs are currently limited, but the basis is in place to extend these.

The development of this logic was not by any means straightforward. It required the definition of several execution relations for bytecode programs, each necessary for proofs of different aspects of execution. In addition, the flat, unstructured nature of bytecode programs presents a number of difficulties, particularly when reasoning about loops. But there are, as we demonstrate, some quite elegant solutions to these problems.

1.6 Outline of Thesis

Chapter 2 examines work done in four main areas, all of which relate to our own work, namely: proving properties of the Java language itself, proving properties of programs written in Java, improving the performance of JIT compilers, and incorporating proof into 'real world' systems. In each case two or three papers are discussed in some detail, followed by a brief description of examples of other notable work in the area, concluding with a discussion of the relevance of the approaches taken to the work described in this report.

Chapter 3 describes the development of three execution relations for bytecode programs and the alterations and extensions of Pusch's semantics necessary for this. All these relations are necessary for the development of the programming logic for bytecode programs described in Chapter 4.

Chapter 4 describes the derivation of the rules of the bytecode programming logic and the difficulties encountered in its development, particularly with respect to the unstructured nature of bytecode.

Chapter 5 outlines in some detail the proof of soundness of the rule for loops in bytecode programs. Although the rule itself does not differ greatly from the rule for while statements in the conventional Hoare logic, the proof of its soundness is a great deal more complex. In this chapter we describe the soundness proof and the three main results necessary to achieve it.

Chapter 6 discusses the use of the bytecode programming logic to prove properties of example programs.

Chapter 7 discusses the results of the work and the lessons learnt from it, particularly with regard to two topics: the practicalities of proof at the bytecode level, and the role of mechanized reasoning in such proofs. The chapter concludes by suggesting areas in which the work could be developed further.

1.7 Related Publications

Early results relating to this project were published in [47], and a more complete summary of the work in [48].

Chapter 2

Related Work

In this chapter we review work related to our own in four general categories, namely: proving properties of the Java language itself, proving properties of programs written in Java, improving the performance of JIT compilers, and incorporating proof into 'real world' systems. In each case two or three papers are discussed in some detail, followed by a brief description of examples of other notable work in the area, concluding with a discussion of the relevance of the approaches taken to the work described in this report.

2.1 Proofs about the Java Language and JVM

A large amount of work has been done in recent years on formalizing aspects of the Java language and the JVM with the aim of proving that they possess certain desirable properties. The starting point for such projects is the English language specifications of the Java language [18] and the JVM [29] published by Sun Microsystems.

2.1.1 The Java Language

As one of Java's main attractions for users is its claim [18] that its strong type system makes downloading and running programs across a network safe, much work has been done on formally proving the type soundness of the language. The aim is to show that the static checks done on a Java program at compile time really do lead to type safe execution at runtime, i.e. the program will not carry out operations that violate the typing rules, such as attempting to add a value of type *String* to one of type *Integer*.

In *Java is type-safe—probably* [14], Drossopoulou and Eisenbach describe an operational semantics for a subset of Java that they call Java$_S$. Java$_S$ includes primitive types, classes with inheritance, instance variables and instance methods, interfaces, shadowing of instance variables, dynamic method binding, statically resolvable overloading of methods, object creation, null pointers, arrays and a minimal treatment of exceptions. The authors also define the notion of a well-formed environment and go on to prove that indeed a well-typed Java program run in a well-formed environment will not give rise to typing violations.

The work of Drossopoulou and Eisenbach is unusual in that it does not utilize any form of mechanized proof tool. In *Proving Java Type Soundness* [52], Syme uses their work as a basis for a proof of the type soundness of a very similar subset of the Java language using the prover DECLARE. Syme goes on to validate much of the work of Drossopoulou and Eisenbach, but does identify 'one major error and a noteworthy omission' in their work, highlighting the difficulties inherent in dealing with such large proofs without the aid of a proof tool.

In *Java$_{light}$ is Type Safe—Definitely* [38], Nipkow and von Oheimb have carried out a similar proof of type soundness, this time for a subset of Java they call Java$_{light}$, using the prover Isabelle/HOL. Though independent of the work of Drossopoulou and Eisenbach, and differing in certain aspects (such as the use of a big-step rather than small-step semantics), there are similarities between the two projects, which are discussed.

2.1.2 The JVM

The work described above is all concerned with the Java language as a high-level, object oriented language. But while the Java language and the JVM are obviously closely related and their type-systems very similar, they are separate entities and as such, a proof of type soundness of one does not necessarily imply the same property holds of the other.

In her paper *Formalizing the Java Virtual Machine in Isabelle/HOL* [45], Pusch details a formalization of the JVM—which she describes as preliminary —in the theorem prover Isabelle (using the HOL object logic). We describe this paper in more detail than the others mentioned in this chapter as it is the work on which our own is based.

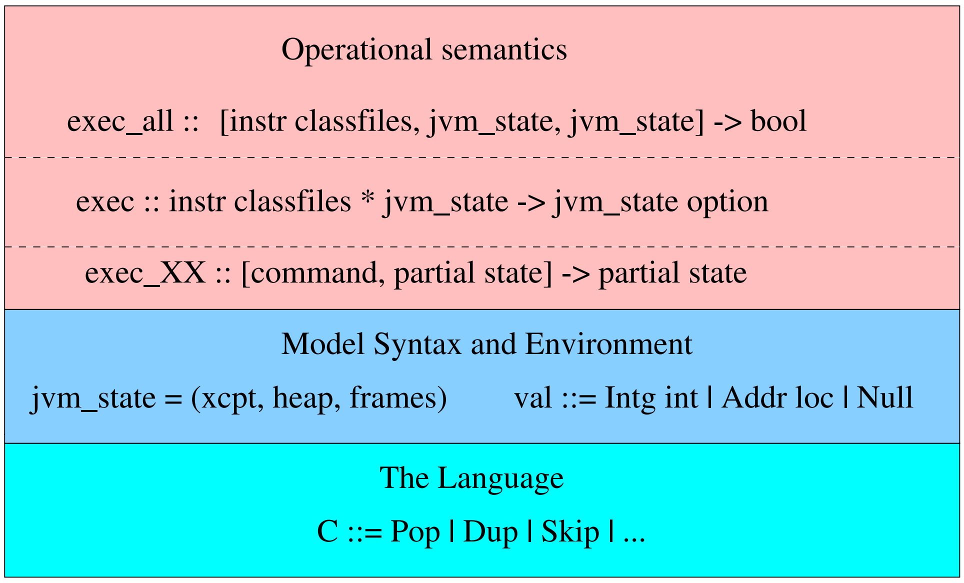
Operational semantics
exec_all :: [instr classfiles, jvm_state, jvm_state] -> bool
exec :: instr classfiles * jvm_state -> jvm_state option
exec_XX :: [command, partial state] -> partial state
Model Syntax and Environment
jvm_state = (xcpt, heap, frames)
val ::= Intg int | Addr loc | Null
The Language
C ::= Pop | Dup | Skip | ...

Figure 2.1: representation of the JVM operational semantics

Pusch's aim is to provide a formal version of the Java Virtual Machine Specification [29] that is not prey to the ambiguities and inconsistencies which tend to invade informal specifications (and indeed do in the case of the JVM Specification). As this is a considerable undertaking, the theorem prover Isabelle is used to ensure a degree of reliability not likely to be achieved in a proof by hand. Although a large subset of the Java language is formalised, there are areas not treated in this implementation; these include exception handling and dynamic class loading.

The paper outlines the formalization of both static aspects of Java programs, e.g. well-formedness of classfiles and relations between classes, and properties of the Java run-time system including object initialisation and the JVM heap. The author also describes an operational semantics for the subset of the JVM instruction set considered.

As Pusch's semantics are the basis for our work, we give a brief description of some of the main features of her formalization here. An outline of the form of the semantics can be seen in Figure 2.1.

The Language The commands in the language are Java bytecode instructions,

$$\mathsf{C} ::= \mathsf{Pop} \mid \mathsf{Dup} \mid \mathsf{Swap} \mid \ldots$$

The Environment Values are modelled by the datatype

$$val ::= \mathsf{Intg}\ int \mid \mathsf{Addr}\ loc \mid \mathsf{Null}$$

and a state is a triple (*xcpt Option*, *heap*, *frame list*) where *xcpt Option* is an exception option (this is None if no exception has been thrown at this point in execution), *heap* is the object heap of the JVM, and *frame list* is the frame stack for the program. The frame stack is a list of frames, each frame relating to the invocation of a particular method. A frame consists of variables of the following types:

- *opstack*—operand stack for the current method, modelled as a list of type *val*
- *locvars*—list of local variables for the current method, each of which can hold a value of type *val*
- *cname*—name of the class the current method belongs to,
- *method_loc*—method locator for the current method, and
- *p_count*—current value of the program counter.

Operational Semantics of Commands Unlike the operational semantics mentioned in Section 1.3 and described in detail in [56], Pusch's semantics for the JVM are not presented as a set of rules. Instead she takes the approach more common to denotational semantics of defining execution in terms of a partial function on states. Partial functions deal with the situation where, for some values, the result of the function can be undefined. They can be represented in Isabelle by the *Option* type

```
datatype 'a option = None | Some 'a
```

where an undefined result returns the value None.

Pusch's semantics can be viewed as having three layers:

execution of a single class of bytecode operations Pusch divides the bytecode instructions of the JVM into several categories:

- load and store
- create object
- manipulate object
- manipulate array
- check object

- method invocation
- method return
- operand stack
- conditional branch
- unconditional branch

The effect on the state of executing the commands in each category is defined in a separate Isabelle theory file for each category of command, for example

$$\mathsf{exec\_os} \;::[op\_stack,\ opstack,\ p\_count] \Rightarrow (opstack\ *\ p\_count)$$

$$\mathsf{exec\_las} \;::\ [load\_and\_store,\ opstack,\ locvars,\ p\_count] \Rightarrow (opstack\ *\ locvars\ *\ p\_count)$$

At this level the operational semantics are similar to conventional operational semantics, in that the command itself is passed to the function exec_XX along with a state (in this case actually only the part of the state affected by the execution) and the updated state returned.

execution of a bytecode instruction The next layer defines the execution of one bytecode instruction of *any* category by the function

$$\mathsf{exec}\ ::\ instr\ classfiles\ *\ jvm\_state \Rightarrow jvm\_state\ Option$$

It is at this point that the semantics become noticeably different to the usual style. Here the arguments to the function exec are a state and *instr classfiles*, so the function is not passed a single command and state, but a complete environment from which it must extract the relevant instruction and state. Once we have obtained the correct classfile from the set of classfiles passed to the function, we must look at the program counter of the current stack frame (contained in *jvm_state*) in order to determine the current instruction. The result *jvm_state Option* reflects the fact that the result of execution may not be defined.

execution of a whole program The execution of an entire program is given by the function

$$\mathsf{exec\_all} ::[bytecode, jvm\_state, jvm\_state] \Rightarrow bool$$
$$CFS \vdash s \longrightarrow^{*} t \equiv (s,t):\ \{(s,t).\ \mathsf{exec}\ (CFS, s)\ =\ \mathsf{Some}\ t\}^{*}$$

where *CFS* denotes a set of classfiles.

This corresponds to `eval`, mentioned in Section 1.3, in that it returns `true` only if executing the given code in the initial state s results in the final state t. Once again, the code to be executed is not explicit in the arguments to the function, but must be extracted from the state and relevant classfile.

In *Proving the Soundness of a Java Bytecode Verifier in Isabelle/HOL* [46], Pusch uses the operational semantics described here to prove the type-soundness of the Java bytecode verifier, i.e., that any program bytecode program passed by the verifier will have the properties described in Section 1.1.

2.1.3 Comments

The work described in this section differs from our own aims in that we wish to prove properties of specific bytecode programs compiled from Java source code and run on the JVM, rather than properties of the Java language and JVM themselves. But since it is pointless to prove 'extra' desirable properties of a program not believed to possess the more basic property of type-soundness, the proofs of Java's type-soundness can be considered fundamental to our own work. In addition to this, Pusch's formalization of the semantics of the JVM is the basis of our work.

2.2 Proving Properties of Java Programs

In this section we describe a number of projects whose aim is to enable the proof of properties of individual Java programs.

2.2.1 Extended Static Checking

The *Extended Static Checking* system (ESC) [13] aims to statically determine simple errors in programs, e.g. array out of bounds errors or simple deadlocks and race conditions in concurrent programs. The user annotates programs with simple specifications, and these are passed to a verification condition generator, which produces a logical formula encapsulating the desired property. This formula is then passed to ESC's dedicated automated proof system, **Simplify** which either proves the validity of the formula, or returns an instance in which it is false to the user.

ESC differs from the traditional approach to program verification, in that it does not attempt to prove that a program is *correct*, merely that it does not suffer from certain specific problems. The authors refer to this as 'lightweight' verification, but note that the comparative simplicity of the properties proved is offset by the complexities of the Java and Modula-3 languages and environment. The paper also draws attention to the fact that the information produced by ESC for an incorrect program is similar to that returned by a debugger. Furthermore, it is suggested that this additional verification of simple properties might be viewed by users in the future in a similar light to typechecking. To this end, any additional burden on users is reduced by ensuring that annotations are simple and proof of the required logical properties is both fast and automatic.

The authors also describe how in designing ESC's integral theorem prover, they faced the challenge of achieving the correct balance between interaction and automation. Interactive provers are often very powerful, but require a great deal of user input and knowledge—a feature the authors felt would be likely to discourage a great many programmers from using the system. Automatic provers, on the other hand, require little user interaction but are often unable to deal with the decision procedures the authors felt were essential to the system—e.g. those for linear arithmetic. The resultant prover is described as having two parts: a set of co-operating decision procedures, and a search procedure that manages the search for a proof.

The system has been used to verify properties of several programs, including an interface for Modula-3 that implements a dynamically expandable array, the IO streams package of Modula-3; and parts of the ESC system itself.

2.2.2 The LOOP Project

The aim of the *Logic of Object-Oriented Programming* (LOOP) Project [25] is to specify and verify properties of classes in object oriented languages, with the aid of proof tools such as Isabelle and PVS [4]. The main focus of the project is the verification of programs written in JavaCard—a subset of Java used to program SmartCards. In [54], van den Berg and Jacobs note that reasoning about "real world" programming languages, such as Java, which may not be mathematically clean has always been extremely challenging. However with improvements in theorem proving technology and increased computing power, it is now becoming a realistic goal. The authors also mention the high level of interaction and feedback between the theoretical basis of their work and the practical aspects involved in verifying actual programs.

The LOOP tool accepts Java programs, CCSL specifications [49], and JML programs (Java programs annotated with Java Modelling Language (JML) specifications). JML is a behavioural interface specification language, designed specifically for Java, which enables pre- and post-conditions to be written in a Java-like manner. It is designed with ease of use for those with little experience of logic in mind.

The tool translates the input programs into higher order logic descriptions of their semantics acceptable to a theorem prover. Currently these descriptions are produced for the Isabelle and PVS provers, but the authors claim that this could be extended to any other prover that uses higher order logic. On the basis of these descriptions, the provers can be used to prove that the programs meet their specifications and other properties. These proofs are often carried out using a specialised Hoare logic.

Unlike the ESC, which requires no input from the user, but is limited in the properties it can check, the LOOP tool can be used to provide a basis for an unlimited variety of properties, but requires a great deal of user interaction. As both approaches are valid for different users and problems, and especially as both tools use the JML language, the authors suggest they may provide complementary approaches to proofs of 'real world' programs.

2.2.3 The TJVM

In [34] Moore describes the development of a simplified or 'toy' JVM (TJVM) in order to explore verification issues for object-oriented bytecode. The TJVM was formalized in ACL2 (A Computational Logic for Applicative Common Lisp) [27], and is based on Cohen's defensive JVM [12]. Moore employs the standard method of formalizing machines in ACL2, whereby the state of the TJVM is represented as a LISP object and an interpreter for TJVM bytecode as a LISP function. The TJVM differs from the JVM in that it does not deal with resource limitations, exceptions, or access types (e.g. a `load` instruction loads a value of any type).

The interpreter for the TJVM is defined as an iterated step function: `tjvm` s n where the function that evaluates a single step of execution is applied n times to the initial state s. This definition of the execution of a bytecode program in terms of a concrete number of steps contrasts with Pusch's definition in terms of the reflexive transitive closure of pairs of states in a successful execution path. This reflects the differing aims of the authors: Pusch's proof objectives are of abstract properties of the JVM and bytecode programs in general; Moore's involve proving that specific programs are "correct" in the more widely understood sense of the word, i.e. they result in a particular value being produced.

After setting up the ACL2 prover by proving several lemmas about simple arithmetic and single steps of execution, Moore describes how proofs can be obtained for the total correctness of several small programs (e.g. `factorial`) by instructing the prover to inductively "unwind" the code. Each method that is proved increases the capacity of the prover, as any future occurences of such a method will be treated as a primitive operation.

2.2.4 Comments

While all three projects described above involve proving properties of actual Java programs, ESC and LOOP are probably most closely related in that they attempt to model the Java world with as much accuracy as possible, and carry out proofs at the level of the Java source language. Moore's work, in addition to dealing with proofs at the bytecode level, is concerned with a restricted subset of the language and does not model many of the features that make reasoning about Java and other 'real world' languages particularly difficult.

Despite this, van den Berg and Jacobs claim that thanks to improved proof technology and advances in computing power such 'real world' proofs are becoming a realistic goal. Certainly ESC and LOOP demonstrate that this is true to a degree. However it seems likely that the aim of projects nowadays to prove only that programs possess certain, comparatively simple, desirable properties, or do not possess other undesirable ones plays a part. Due to the sheer complexity and scale of real world systems, it seems likely that the distinction between 'toy' systems, like Moore's, in which it can be proved that a very simple program is *correct*, and systems like ESC and LOOP which can prove that a program of some considerable complexity does not have certain clearly defined classes of error, will be preserved. But as the proof of a few specific properties is frequently all that is needed in practice, perhaps the tendency towards this 'lightweight' verification is an advance in itself.

With regard to our own work, despite the fact that Moore's work carries out proof at the level of bytecode, it seems likely that ESC and LOOP are more immediately relevant as we are aiming at real world applicability and proof of certain properties rather than correctness per se.

2.3 Improving Performance of JIT Compilers

In this section we describe two projects whose aim is to improve the performance of JIT compilers. One is specifically aimed at Java JIT compilers, the other is not specifically concerned with Java or JITs, but the techniques used to improve the efficiency of a bytecode-like assembly language may well have applications to JIT compilers.

In *Annotating the Java Bytecodes in Support of Optimization*,[22], Hummel, Azevedo, Kolson, and Nicolau observe that while Java provides a portable, platform-independent stack machine, it does so at the expense of execution speed, as stack machines do not map well onto modern CPUs, which rely heavily on the use of register and caches for speed. In addition to having no concept of registers, Java bytecodes are also unable to express optimizations like instruction scheduling, elimination of runtime checks, and automatic reclamation of memory.

With the goal of achieving C-like performance while retaining the portability of bytecode and preserving compatibility with existing JVMs, the authors propose an **annotating compiler**. This behaves initially like a traditional optimizing compiler, analysing the code and performing optimizations before emitting bytecode. But rather than discarding the information produced by the analysis, the compiler attaches the relevant information to each emitted bytecode in the form of an annotation.

The annotations contain information useful for

- register allocation
- memory disambiguation
- memory reclamation
- run-time checking

This information would normally have to be recomputed from the bytecode by the JVM, which in some instances may not be possible as too much

CODE	**src**	**inter**	**dest**	**last use**	**r-t check**	**memory ref tag**
aload a			v0			/stack/objref/a
iload i			v1			/stack/int/i
iconst 2						
aload a	v0		v0			/stack/objref/a
iload i	v1		v1			/stack/int/i
iaload	v0,v1	v2	v3		111	/heap/array/int/*
imul	v2,v3		v3			
aload b			v0			/stack/objref/b
iload i	v1		v1			/stack/int/i
iaload	v4,v1	v2	v4		101	/heap/array/int/*
iadd	v3,v4		v3	v4		
iastore	v0,v1,v3	v2	v3	v3	000	/heap/array/int/*

Figure 2.2: Example of annotated bytecode

information may have been lost. Annotations are stored separately from the bytecode in a classfile in order not to interfere with the running of the program on standard JVMs. A JVM with an annotation aware JIT compiler, however, can use the annotations to produce more efficient code more quickly.

The table in Figure 2.2 (taken from [22]) shows the annotated bytecode for the Java expression

```
a[i] = (2*a[i]) + b[i]
```

The *src*, *inter*, *dest*, and *last use* columns denote virtual register allocation performed during the original source to bytecode translation. Virtual register *v0* is mapped to physical register *R0*, *v1* to *R1*, etc. until all available registers are used up, after which the virtual registers are mapped to memory locations.

The *inter* column tells a JIT compiler to save intermediate values in the specified register(s) if possible. The *last use* column denotes when a register ceases to be used for a particular variable and so can potentially be used for another one. The *r-t check* column specifies which run-time checks should be performed. For array accesses, at most three possible checks are required:

1. Is the array reference equal to Null?

2. Is the array index less than 0?

3. Is the array index greater than or equal to the length of the array?

Each check is assigned a bit in *r-t check*; if the bit is 1, then the code must be generated to do the check. The *memory ref tag* column provides memory reference information suitable for performing disambiguation.

The authors report substantial improvements in performance when using the Annotated JIT. For one benchmark the performance is almost three times faster than with a standard JIT, for another, almost twice as fast.

In *A Dependently Typed Assembly Language* [57] Hongwei Xi and Robert Harper describe an assembly language with a restricted form of dependent types. Dependent types are types which depend on terms, e.g. `List(n)` is the type of lists of length n—as opposed to the more usual type of `List` which includes no information about its length. More information can be found in [58].

In an overly complex type system, type checking can be infeasible (or actually undecidable). In order to avoid this situation most type systems are very simple, which means that only very elementary properties can be expressed and hence checked: it is usually not possible for a type checker to identify an attempt to remove an item from an empty list, for example. By restricting the dependent types in the language to those needed to ensure certain specific properties, the authors attempt to strike a balance between these two extremes.

In *Dependent Types in Practical Programming* [58] the authors describe a method for eliminating array bounds checks in functional programs. They define **dependent type constructors** of the form

```
{n:nat}
```

Constructors may also contain conditions such as

```
{n:nat }{i:nat| i < n}
```

A function `nth`, which returns the nth item of a list, therefore has the type

```
{n:nat }{i:nat| i < n} 'a list(l) * int(n)  -> 'a
```

which eliminates the need for runtime checks on the length of the list. This approach is now applied to assembly language, producing a **dependently**

typed assembly language (DTAL) that supports a limited form of dependent type system which captures both type safety and memory safety. The paper describes an operation semantics and a set of typing rules for DTAL, from which type-soundness is proved.

2.3.1 Comments

The work described in this section represents two very different approaches to the optimization of assembly language programs. The AJIT project starts by gathering information from a high-level Java program, applying it to the 'middle' stage of a bytecode program, resulting in a more efficient assembly language program. The techniques involve no formal methods—although the authors mention the benefits in increased user confidence of applying such methods—but demonstrate a measurable improvement in the efficiency of the JIT.

The DTAL project, on the other hand, is not concerned with high level languages or stack-based languages, but purely with assembly language. It takes a very formal approach to the problem, but ultimately also results in more efficient code.

With regard to our work, it is clear that the AJIT project has more relevance. It demonstrates the value to JIT performance of the knowledge of the existence of certain properties in a bytecode program, and provides a system by which they can be conveyed to a user. It seems likely that a proof element could be incorporated into such a system. While not disregarding the applications of a JIT producing dependently typed assembly code, it appears to have less immediate relevance to our work.

2.4 Incorporating Proof in Systems

While users may be keen to have the added reassurance of code that has been proved to have desirable properties such as type-soundness, they may well be put off using such systems if they are presented with a great deal of extra work and complexity to achieve such a result. In this section we discuss three projects aimed at incorporating proof into real world systems while retaining a 'user-friendly' interface.

2.4.1 Eiffel

Eiffel [53] is an object oriented language that implements proof annotations (known as **assertions**) that allow programmers to express formal properties of classes. Assertions are boolean expressions and can have the following forms:

- **Routine Preconditions**: these express requirements a client must satisfy before they **call** a routine.
- **Routine Postconditions**: these express conditions the supplier (i.e. the routine being called) guarantees on **return**, if its preconditions were satisfied on entry.
- **Class Invariant**: this must be satisfied by every instance of the class, whenever an instance is externally accessible. It characterises the **semantics** of the class.

The assertion mechanism can be used to implement what the developers of Eiffel refer to as **programming by contract**. This means that every routine in the code has a **client-supplier contract** specifying how it should be used by calling procedures (clients) and what it does itself (as a supplier). The aim is that all routines used in a system conform to their client-supplier contracts. Unlike type-constraints which can be checked statically, Eiffel contracts may rely on data values and so can only be checked at runtime. The handling of such errors is dealt with by Eiffel's exception mechanism, and Eiffel provides a **history table** in order to make debugging of such situations easier.

2.4.2 Ada Spark

Spark [9] is a subset of the Ada programming language enriched with **annotations**. Spark uses a tool known as the **Examiner**, which has two basic functions:

- checking that the code conforms to the rules of the kernel language.
- checking consistency between the code and the embedded annotations by control, data and information flow analysis.

In order to ensure correct dynamic behaviour of the code, certain proof annotations can be inserted that allow analysis of a program's dynamic behaviour prior to execution. The annotations allow the Examiner to generate theorems; proving these theorems verifies that the program is correct with respect to the annotations. The proof annotations comprise

- pre and postconditions of subprograms
- assertions such as loop invariants
- declarations of proof functions

The generated theorems are known as **verification conditions** and can be verified by hand or by using other Spark tools such as the **Simplifier** and the **Proof Checker**. The Examiner is also able to generate path functions which show the effect of traversing the various paths in a subprogram.

The Examiner provides three different levels of analysis, according to how critical the safety of the code is

- The lowest level of analysis is **Data Flow Analysis**. This involves checking that the usage of parameters and global variables corresponds to their modes; that values are not overwritten without being used; that all imported variables are used somewhere. The interdependencies between variables as expressed in the **derives** annotation are not checked.

- The next level is **Information Flow Analysis**. This requires **derives** annotations, and in addition to carrying out data flow analysis it checks that the modes of parameters and global variables and their usage in the code of the body correctly match the interdependencies given in the **derives** annotation. This level of analysis is known as **shallow verification** as it checks the static semantic dependencies, though not dynamic ones.

- The highest level of analysis involves generating **verification conditions** in addition to performing flow analysis and requires **proof annotations**.

Verification conditions are obtained through a series of operations on the stated conditions which annotate the code. These operations correspond closely to those involved in Hoare Logic proofs, e.g. the Assignment Axiom.

2.4.3 Proof Carrying Code

Proof Carrying Code (PCC) [36, 35, 11, 50] is a technique developed by George Necula and Peter Lee to attempt to address the problem of safe execution of untrusted code. In an instance of PCC a code receiver establishes a set of safety rules that guarantee safe behaviour of programs (or at any rate, what the receiver is defining safe behaviour to be); the code producer then creates a formal **safety proof** that proves the untrusted code's adherence to the safety rules. The receiver is then able to use a simple and fast **proof validator** to check that the proof is valid and hence that it is safe to execute the untrusted code.

A PCC implementation contains the following elements:

- A **formal specification language**—first order predicate logic— is used to express the safety policy.
- **A formal semantics of the language used by the untrusted code**. This is usually in the form of a logic relating programs to specifications. A form of Floyd's verification-condition generator [15] is used to extract the safety properties of a program as a predicate. This predicate must then be proved by the code producer using the axioms and inference rules supplied by the code consumer as part of the safety policy.
- The **language used to express the proofs** is a variant of Edinburgh Logical Framework (LF) [8] (a typed lambda calculus).
- **An algorithm for validating the proofs**. This involves type-checking the LF expression that represents a proof according to a set of typing rules agreed on by the code producer and receiver.
- **A method for generating the safety proofs**. This element is used only by the code producer and the implementation involves a theorem prover that emits the required proofs. In [37] the use of a **certifying compiler**, which translates programs written in a typesafe subset of C into machine code and a certifier which checks the type and memory safety of any program produced by the compiler is investigated. The use of a certifying compiler has the added advantage that it is much simpler to verify the output of the compiler than to verify the compiler itself.

2.4.4 Comments

Both PCC and Java are designed to ensure the safety of untrusted executable code, downloaded from another system, without access to source code. Eiffel and Spark try to ensure the *production* of safe executable code within a system; and they depend on being able to examine the source code (in order to deal with annotations).

PCC employs a combination of First Order Logic, a language semantics, a theorem prover and a form of the lambda calculus to produce and encode its proofs of program correctness. It can therefore be viewed as the most 'heavy duty' in terms of formal methods. Spark involves data and information flow analysis combined with a form of Hoare logic in order to carry out both static and runtime checks. Whereas Java uses dataflow analysis with additional runtime checks to ensure the safety of programs, Eiffel uses a rather weaker form of the annotations used in Spark.

In terms of our own work, the notion of annotated code as used in Spark and Eiffel, could be used in some form in the proof of bytecode programs. It is unreasonable to imagine that users would be willing to annotate bytecode programs themselves. However as many of the properties involved are very low level (e.g. just prior to execution of an array store application, checking whether the array reference is non-null), an automatic annotating mechanism might be feasible.

The concept of Proof Carrying Code is very relevant to our work, as it provides a method of providing the user with the added reassurance of verified code in a relatively painless manner. As the vast majority of Java users are not well-versed in theorem proving techniques, they are unlikely to welcome a system that demands skill in such techniques.

2.5 Conclusions

A large amount of work has been carried out on proving that the Java language and JVM are type safe. Most of this work has been carried out with the aid of mechanized proof tools. In addition to this there are several ongoing projects that aim to develop proof systems for real world Java programs. These tend towards lightweight verification in which the aim is to prove particular properties of programs, rather than program correctness. We feel that these techniques are applicable to our own work on proving certain properties of Java bytecode programs with a view to improving the performance of JIT compilers.

In addition there exist several projects on improving the performance of JIT compilers, and incorporating proof into systems, which would be of relevance to developing a technique to proving properties of bytecode programs into a working system.

Chapter 3

Bytecode Execution Relations

The decision to prove properties of the bytecode programs themselves, rather than the corresponding Java source was made based on two main factors:

1. Java programs are downloaded by consumers as bytecode, not source
2. It is perfectly feasible (albeit not common in practice) to produce Java bytecode from another high level language, e.g. C, ML.

In order to reason about properties of bytecode programs it is necessary to develop a logical framework that supports this.

The fact that bytecode is 'flat' and contains goto instructions presents difficulties not encountered in the standard logic, which deals with a structured programming language. The standard Hoare logic has three main components, however, which can be applied to bytecode programs, namely:

1. The notion of evaluation of a section of code in the language (which can be based on the operational semantics)
2. Definition of a pre- and post-condition relation on execution of code.
3. Higher level rules for combining patterns of code

The development of some logical relations corresponding to the first item in this list—the evaluation of bytecode—is discussed in the rest of this chapter. There are three execution relations for bytecode instructions. The block execution relation (Section 3.3) describes the complete execution of a block of bytecode. The sequence execution relation (Section 3.2) describes the complete execution of a block of bytecode of a very restricted class of instructions. Finally, the execution path relation (Section 3.4) is concerned with the relationship between intermediate states in the execution of a block of bytecode and the initial and final states. These relations are all necessary for the development of points 2 and 3 on the list, which will be discussed fully in Chapters 4 and 5.

3.1 Extending the Semantics

While Pusch's formalization of the JVM is fairly comprehensive, the objective of the work differs from ours, and some alterations to the model are necessary in order to allow the definition of the bytecode logic. We describe these alterations briefly before commencing discussion of any bytecode execution relations.

3.1.1 Arithmetic Instructions

As Pusch's work formalises a subset of the JVM, certain instructions are omitted. These include all arithmetic instructions, such as iadd, isub. In order to prove properties of real programs, however trivial, Pusch's model must be extended to include this class of instructions.

Each of the load and store instructions in Pusch's instruction subset are represented by a value in the datatype *load_and_store* whose components are the name of a load and store instruction and its arguments. The following function is then defined for each element of the *load_and_store* datatype:

$$\begin{aligned}\mathsf{exec\_las} :: [&\mathit{load\_and\_store},\ \mathit{opstack},\ \mathit{locvars},\ \mathit{p\_count}] \Rightarrow \\ &(\mathit{opstack}\ *\ \mathit{locvars}\ *\ \mathit{p\_count})\end{aligned}$$

This takes a load and store instruction, an operand stack, a set of local variables and a program counter and returns the updated values of the operand stack, local variables and program counter.

It is simple to add the instructions

- iadd— add the two integers at the top of the stack

- iinc *var val*— increment local variable *var* by integer value *val*

to the existing load and store instructions of the model. The corresponding extensions to exec_las are also straightforward, and the resultant Isabelle theory file can be found in Appendix B of this report.

3.1.2 Branching Instructions

Problems are also encountered with the representation of branching instructions. In Java bytecode, branching instructions are absolute jumps to a label, but in Pusch's model they are represented by relative branches, where the new value of the program counter is obtained by adding an offset to the current value. This offset is positive for a branch forward, negative for a branch backwards.

While this convention appears suitable for Pusch's higher-level proofs, difficulties arise when using it to reason about lower-level properties. In particular, problems are encountered with proofs involving branches backwards where a negative integer is added to the program counter (a natural number cast to an integer) and the result is then cast to a natural. This repeated type-casting makes the proofs in Isabelle very awkward.

Consequently, in place of the two varieties of branching instructions in Pusch's model (*cond_branch* and *uncond_branch*) we have four types of branching instruction:

$$\begin{aligned} cond\_branch\_fwd \;=\; & \mathsf{Ifnull\_fwd}\ nat \\ & \mid \mathsf{Ifiacmpeq\_fwd}\ ins\_type\ nat \\ & \mid \mathsf{Ificmplt\_fwd}\ nat \end{aligned} \tag{3.1}$$

$$\begin{aligned} cond\_branch\_bwd \;=\; & \mathsf{Ifnull\_bwd}\ nat \\ & \mid \mathsf{Ifiacmpeq\_bwd}\ ins\_type\ nat \\ & \mid \mathsf{Ificmplt\_bwd}\ nat \end{aligned} \tag{3.2}$$

$$uncond\_branch\_fwd \;=\; \mathsf{Goto\_fwd}\ nat \tag{3.3}$$

$$uncond\_branch\_bwd \;=\; \mathsf{Goto\_bwd}\ nat \tag{3.4}$$

All of these take a natural number as offset, which is either added or subtracted to the current program counter depending on whether the branch is forwards or backwards. This keeps all branching proofs in the realm of natural number arithmetic, greatly simplifying the Isabelle proofs.

3.2 Execution of a Sequence of Bytecode Instructions

The conventional Hoare logic is based on an operational semantics where execution begins at the start of the sequence of commands and finishes at the end (assuming the program terminates). But with bytecode there is the possibility of jumping *into* the code at some point after the start and *out* at a point before the end. How, then, should execution of a sequence of bytecode be defined?

Consideration of a straightforward recursive definition of the form

$$\mathsf{exec}^*\ [\,]\ \sigma_0 = \sigma \tag{3.5}$$

$$\mathsf{exec}^*\ (x : xs)\ \sigma_0\ =\ \mathsf{exec}^*\ xs\ (\mathsf{exec}\ x\ \sigma_0) \tag{3.6}$$

where x is a bytecode instruction and xs a list of bytecode instructions, immediately reveals it to be unsuitable as the execution of xs would not necessarily be linear: execution might well jump back to the beginning of xs after a few instructions. Pusch's formalisation recognises this by defining execution of several bytecode instructions as the reflexive, transitive closure of a series of single execution steps:

$$\begin{array}{l} \mathsf{exec\_all} :: [\mathit{bytecode},\ \mathit{jvm\_state},\ \mathit{jvm\_state}] \Rightarrow \mathit{bool} \\ \mathit{CFS} \vdash s \longrightarrow^* t \equiv (s,t) :\ \{(s,t).\ \mathsf{exec}\ (\mathit{CFS}, s)\ =\ \mathsf{Some}\ t\}^* \end{array} \tag{3.7}$$

where *CFS* denotes a set of classfiles.

To define a partial correctness relation it is necessary to know that, for a sequence of bytecode instructions, if we start executing in state σ_0 we will finish execution in state σ_n. But the above relation does not have anything to say about a state's *position* in the sequence of states produced by executing a number of bytecode instructions, only whether or not it is *in* the sequence. We must therefore define what it means to 'finish' execution of a sequence of bytecode instructions.

One possibility is to state that execution of a sequence of instructions has finished *when the program counter is no longer pointing into the sequence.* This results in the definition of a relation describing the execution of a list of bytecode instructions in which if execution begins in state σ_0 inside a sequence, it results in state σ_n, where the program counter of σ_n is outside the section.

3.3 The Block Execution Relation

Suppose that

- *CFS* is a set of Classfiles
- The *start* of the bytecode sequence is signified by s, and the *finish* by f, where s and f are triples of the form *(classname, method locator, program counter)* each allowing identification of a single instruction in *CFS*.
- σ_0 and σ_n are states, each consisting of *(exception option, heap, frame stack list).*

We write $\langle \mathit{CFS}, \sigma_0 \rangle \xrightarrow[f]{s} \sigma_n$ to mean that executing the sequence of instructions in *CFS* that begins at the instruction indentified by s and ends at the instruction identified by f, starting in the state σ_0, results in the state σ_n, where the instruction identified by σ_n is not contained in the sequence of instructions in *CFS* bounded by s and f.

The program counter of s must be less than or equal to the program counter of f, i.e., the block consists of at least one instruction. The program counter of state σ_0, should be greater than or equal to the program counter of s and less than or equal to the program counter of f. The program counter of f should be less than the length of the code of the current method (measured from the start of the method code) to ensure that we are not referring to non-existent pieces of code. This condition also ensures that Isabelle's standard lemmas about lists can be used, as many require that an indexing value, e.g. $\mathsf{pc}(f)$, be less than the length of the list being operated on.

The program counter of the final state σ_n should be either less than the program counter of s or greater than the program counter of f. Finally, s and f should identify instructions in the same method of the same class.

We introduce the following definitions

Definition 1 (Program counter inside block)

$$\mathsf{inside}\ \mathsf{pc}(\sigma_0)\ s\ f\ \equiv\ \mathsf{pc}(s)\ \leq\ \mathsf{pc}(\sigma_0)\ \wedge\ \mathsf{pc}(\sigma_0) \leq\ \mathsf{pc}(f)$$

Definition 2 (Program counter outside block)

$$\mathsf{outside}\ \mathsf{pc}(\sigma_0)\ s\ f\ \equiv\ \mathsf{pc}(\sigma_0)\ <\ \mathsf{pc}(s)\ \vee\ \mathsf{pc}(f)\ <\ \mathsf{pc}(\sigma_0)$$

Definition 3 (States in same method)

$$\begin{aligned}\mathsf{same\_method}\ s\ \sigma_0\ \sigma_n\ f\ \equiv\ & (\mathsf{class}(s)\ =\ \mathsf{class}(\sigma_0)\ =\\ & \mathsf{class}(\sigma_n)\ =\ \mathsf{class}(f))\ \wedge\\ & (\mathsf{method}(s)\ =\ \mathsf{method}(\sigma_0)\ =\\ & \mathsf{method}(\sigma_n)\ =\ \mathsf{method}(f))\end{aligned}$$

And the block execution relation is defined inductively by the rules

Definition 4 (Block Execution Relation)

$$\frac{\begin{array}{l}\mathsf{exec}(CFS, \sigma_0) = \mathsf{Some}\ \sigma_n;\\ \mathsf{inside}\ \mathsf{pc}(\sigma_0)\ s\ f;\\ \mathsf{same\_method}\ s\ \sigma_0\ \sigma_n\ f;\\ \mathsf{pc}(f) < \mathsf{length}(\mathsf{get\_code}\ CFS\ s);\\ \mathsf{outside}\ \mathsf{pc}(\sigma_n)\ s\ f\end{array}}{\langle CFS, \sigma_0\rangle \xrightarrow[f]{s} \sigma_n} \quad \textbf{(Stop)}$$

$$\frac{\begin{array}{l}\mathsf{exec}(CFS, \sigma_0) = \mathsf{Some}\ \sigma_1;\\ \mathsf{inside}\ \mathsf{pc}(\sigma_0)\ s\ f;\\ \mathsf{same\_method}\ s\ \sigma_0\ \sigma_1\ f;\\ \mathsf{pc}(f) < \mathsf{length}(\mathsf{get\_code}\ CFS\ s);\\ \langle CFS, \sigma_1\rangle \xrightarrow[f]{s} \sigma_n\end{array}}{\langle CFS, \sigma_0\rangle \xrightarrow[f]{s} \sigma_n} \quad \textbf{(Continue)}$$

Rule **Stop** refers to the case in which one step of execution results in the program counter being outside the sequence of instructions under consideration. At this point execution of the block would be considered *finished*, giving us the final state required by the programming logic.

Rule **Continue** is the case where, after one step of execution, the program counter is still within the block of code delimited by s and f. This the inductive part of the definition, as the relation is defined in terms of itself. Execution of the block would now carry on, with continued application of the rules until the **Stop** rule applies and we obtain a final state.

As can be seen from the definition of the block execution relation, all states in the relation must refer to instructions within a single method. Obviously it would be desirable to extend the relation in the future to allow method invocation and possible ways of doing this are discussed in Section 7.2.

In addition to this, it is assumed either implicitly or, where necessary, explicitly, that execution of the instructions between the boundaries of an instance of the block execution relation do not throw exceptions. This is to simplify the definition of the bytecode programming logic described in Chapter 4. Again, further work could be done leading to a programming logic for bytecode which allows for abrupt termination, such as that described by Huisman and Jacobs in [21] for Java source code. The $\mathsf{exception}-\mathsf{free}$ property is defined in Section 3.4, Definition 9.

The case split and induction rules for the block execution relation are shown in Figure 3.1 as they appear in Isabelle. It will be apparent, however, that this format is not very readable and so all proofs will be described in the text using a less proof-tool specific notation.

As many of our proofs involve retrieving the code of a particular method in a set of classfiles, and then taking a smaller 'slice' from it, the following definitions are given:

Definition 5 (Isolate sequence of instructions from classfile)

$$\mathit{CFS}\ [s \ldots f] \ \equiv\ \mathsf{slice}\ \mathsf{pc}(s)\ \mathsf{pc}(f)\ (\mathsf{get\_code}\ \mathit{CFS}\ s)$$

Definition 6 ('Slice' instructions from longer list)

$$\mathsf{slice}\ a\ b\ xs \ =\ \mathsf{take}\ (\mathsf{Suc}(b-a))\ (\mathsf{drop}\ a\ xs)$$

```
val mycases =
  "[| ?CFS ?s ?f |- ?a -block-> ?b;
      [| exec (?CFS, ?a) = Some ?b; inside (third_of ?a) ?s ?f;
         pc_of ?f < length (get_code ?CFS (cn_of ?s) (ml_of ?s));
         same_method_frs ?s (hd (snd (snd ?a))) (hd (snd (snd ?b))) ?f;
         outside (third_of ?b) ?s ?f |] ==> ?P;
      !!c.
         [| exec (?CFS, ?a) = Some c; inside (third_of ?a) ?s ?f;
            pc_of ?f < length (get_code ?CFS (cn_of ?s) (ml_of ?s));
            same_method_frs ?s (hd (snd (snd ?a))) (hd (snd (snd c))) ?f;
            ?CFS ?s ?f |- c -block-> ?b |] ==> ?P |] ==> ?P" : thm

val exec_block3.induct =
  "[| ?xo (?xn, (?xm, ?xl),
          ?xk) (?xj, (?xi, ?xh), ?xg) |- (?xf, ?xe, ?xd) -block-> ?xc, ?xb,
      ?xa;
      !!CFS a aa b ab ac ba ad ae bb bc af ag bd be.
         [| exec (CFS, a, aa, b) = Some (ab, ac, ba);
            inside (third_of (a, aa, b)) (af, (ag, bd), be) (ad, (ae, bb), bc);
            pc_of (ad, (ae, bb), bc)
            < length
               (get_code CFS (cn_of (af, (ag, bd), be))
                 (ml_of (af, (ag, bd), be)));
            same_method_frs (af, (ag, bd), be) (hd (snd (snd (a, aa, b))))
             (hd (snd (snd (ab, ac, ba)))) (ad, (ae, bb), bc);
            outside (third_of (ab, ac, ba)) (af, (ag, bd), be)
             (ad, (ae, bb), bc) |]
         ==> ?P CFS af ag bd be ad ae bb bc a aa b ab ac ba;
      !!CFS a aa b ab ac ba ad ae bb af ag bc bd ah ai be bf.
         [| exec (CFS, a, aa, b) = Some (ad, ae, bb);
            inside (third_of (a, aa, b)) (ah, (ai, be), bf) (af, (ag, bc), bd);
            pc_of (af, (ag, bc), bd)
            < length
               (get_code CFS (cn_of (ah, (ai, be), bf))
                 (ml_of (ah, (ai, be), bf)));
            same_method_frs (ah, (ai, be), bf) (hd (snd (snd (a, aa, b))))
             (hd (snd (snd (ad, ae, bb)))) (af, (ag, bc), bd);
            CFS (ah, (ai, be),
                 bf) (af, (ag, bc), bd) |- (ad, ae, bb) -block-> ab, ac, ba;
            ?P CFS ah ai be bf af ag bc bd ad ae bb ab ac ba |]
         ==> ?P CFS ah ai be bf af ag bc bd a aa b ab ac ba |]
   ==> ?P ?xo ?xn ?xm ?xl ?xk ?xj ?xi ?xh ?xg ?xf ?xe ?xd ?xc ?xb ?xa" : thm
```

Figure 3.1: Case split and induction rule as they appear in Isabelle

3.3.1 Lemmas for the Block Execution Relation

Working with the rules defined above, we obtain proofs of the following properties as 'sanity checks' on the inductive definition of the block execution relation.

Lemma 1 (Initial state in block execution relation is inside the block)

$$\forall\ \mathit{CFS}\ s\ f\ \sigma_0\ \sigma_n.\ \langle \mathit{CFS}, \sigma_0 \rangle \xrightarrow[f]{s} \sigma_n \longrightarrow \mathsf{pc}(s) \leq \mathsf{pc}(\sigma_0) \leq \mathsf{pc}(f)$$

Proof This follows from the rules for the block execution relation (Definition 4) and the definition of inside (Definition 1). ■

Lemma 2 (Final state in relation is outside the block)

$$\forall \mathit{CFS}\ s\ f\ \sigma_0\ \sigma_n.\ \langle \mathit{CFS}, \sigma_0 \rangle \xrightarrow[f]{s} \sigma_n \longrightarrow \mathsf{pc}(\sigma_n) < \mathsf{pc}(s) \vee \mathsf{pc}(f) < \mathsf{pc}(\sigma_n)$$

Proof This follows from the rules for the block execution relation (Definition 4) and the definition of outside (Definition 2). ■

Lemma 3 (Initial and final states in relation not equal)

$$\forall\ \mathit{CFS}\ s\ f\ \sigma_0\ \sigma_n.\ \langle \mathit{CFS}, \sigma_0 \rangle \xrightarrow[f]{s} \sigma_n \longrightarrow \sigma_0 \neq \sigma_n$$

Proof This follows from the rules for the block execution relation (Definition 4) and the definitions of inside (Definition 1) and outside (Definition 2). The initial state is inside the block, the final state is outside and, as a state cannot be both inside and outside a block, the states cannot be equal. ■

Lemma 4 (List of frames in initial state not empty)

$$\forall\ \mathit{CFS}\ s\ f\ \sigma_0\ \sigma_n.\ \langle \mathit{CFS}, \sigma_0 \rangle \xrightarrow[f]{s} \sigma_n \longrightarrow \mathsf{frames}(\sigma_0) \neq [\ \ ]$$

Proof This follows from the rules for the block execution relation (Definition 4) and Pusch's definition of the partial function exec which defines execution of a state with an empty list of frames as evaluating to undefined.■

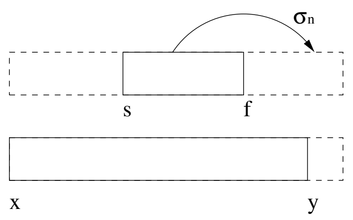

Figure 3.2: Extension of block, final program counter on right

As in imperative programs, a bytecode program can be viewed as a block composed of several smaller blocks of code. It is therefore useful to prove some lemmas relating to the extension and combining of blocks of code.

We begin by describing the extension of blocks. Given a block in the block execution relation and the position of the program counter on exit, $\mathsf{pc}(\sigma_n)$, it is possible to extend the relation to include all instructions in the method on the opposite side of the block from $\mathsf{pc}(\sigma_n)$, and all instructions up to it on the same side.

Lemma 5 (Extension of block with final program counter on right)

$$\begin{aligned}&\forall\ \mathit{CFS}\ s\ f\ x\ y\ \sigma_0\ \sigma_n.\ \langle \mathit{CFS}, \sigma_0\rangle \xrightarrow[f]{s} \sigma_n \longrightarrow\\ &\quad \mathsf{pc}(y) < \mathsf{pc}(\sigma_n)\ \wedge\ \mathsf{pc}(x) < \mathsf{pc}(s)\ \wedge\ \mathsf{pc}(f) < \mathsf{pc}(y)\ \wedge\\ &\quad \mathsf{pc}(y) < \mathsf{length}(\mathsf{get\_code}\ \mathit{CFS}\ s) \longrightarrow \langle \mathit{CFS}, \sigma_0\rangle \xrightarrow[y]{x} \sigma_n\end{aligned}$$

Proof By induction on the rules for the block execution relation (Definition 4). The base case then follows from construction rule **Stop** and the definitions for inside (Definition 1), and outside (Definition 2). The inductive step can be proved by the inductive hypothesis, construction rule **Continue**, and the definition of inside (Definition 1). This is shown in Figure 3.2. ■

Lemma 6 (Extension of block with final program counter on left)

$$\begin{aligned}&\forall\ \mathit{CFS}\ s\ f\ x\ y\ \sigma_0\ \sigma_n.\ \langle \mathit{CFS}, \sigma_0\rangle \xrightarrow[f]{s} \sigma_n \longrightarrow\\ &\quad \mathsf{pc}(\sigma_n) < \mathsf{pc}(x)\ \wedge\ \mathsf{pc}(x) < \mathsf{pc}(s)\ \wedge\ \mathsf{pc}(f) < \mathsf{pc}(y)\ \wedge\\ &\quad \mathsf{pc}(y) < \mathsf{length}(\mathsf{get\_code}\ \mathit{CFS}\ s) \longrightarrow \langle \mathit{CFS}, \sigma_0\rangle \xrightarrow[y]{x} \sigma_n\end{aligned}$$

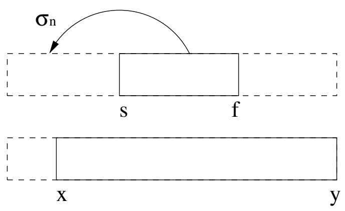

Figure 3.3: Extension of block, program counter on left

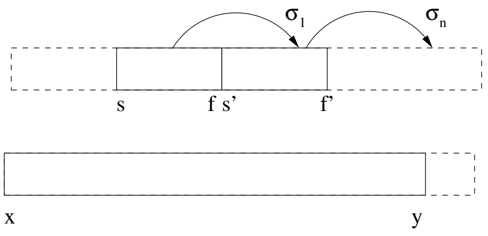

Figure 3.4: Add block on right, finish on right, blocks adjacent

Proof As for Lemma 5. See Figure 3.3. ■

We now consider the problem of combining two blocks in the block execution. Given two blocks in the relation where execution of the first block finishes inside the second, and execution of the second finishes to the right or left of both blocks, the two can be combined to form one larger block. This larger block can itself be extended to include all instructions in the method on the opposite side of the block from $\mathsf{pc}(\sigma_n)$—the position of the program counter on exit from the second block—and all instructions up to it on the same side.

The two original blocks may be adjacent to each other, as in Figure 3.4, be separated by a gap, as in Figure 3.5, or overlap to the extent that one block is exactly 'on top' of the other. The only situation not permitted is that the second block should be *contained within* the first, as it is necessary from the definition of the relation for the program counter of the final state to be outside the block. These lemmas refer to situations where there is no looping *between* blocks (although either or both of the individual blocks may contain a loop); loops will be dealt with in detail in Chapters 4 and 5.

There are four cases that we will consider, defined by the position of the program counter in the final stage of each block: add a block to the right of the initial block, finish on the right of the two blocks; add a block to the right of the initial block, finish on the left of the two blocks; add a block to the left of the initial block, finish on the right of the two blocks; and add a block to the left of the initial block, finish on the left of the two blocks.

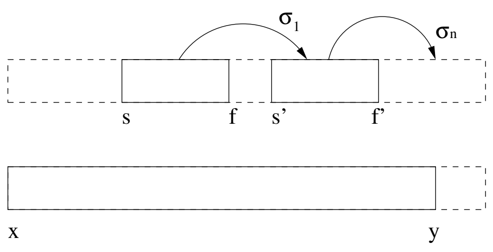

Figure 3.5: Add block on right, finish on right, gap between blocks

As the proofs are all very similar, we will only describe in detail the first case: execution of first block finishes on the right of the first block, but within the second; execution of the second block finishes to the right of both blocks. This is depicted in Figure 3.3.1.

We note briefly that one other case does exist, in which execution of the first block finishes in the second block, and execution of the second block finishes in the gap between the blocks. It is likely that proofs of this case would necessitate a different approach to those described below, but as it is not fundamental to any work later in the dissertation consideration of this case is omitted.

Lemma 7 (Add a block on right, finishing on right)

$$\begin{aligned}
&\forall\ \mathit{CFS}\ s\ f\ x\ y\ \ s'\ f'\ \sigma_0\ \sigma_n\ \sigma'_n.\ \langle \mathit{CFS}, \sigma_0\rangle \xrightarrow[f]{s} \sigma_n \ \longrightarrow\\
&\qquad \langle \mathit{CFS}, \sigma_n\rangle \xrightarrow[f']{s'} \sigma'_n \ \longrightarrow\\
&\qquad \mathsf{pc}(s) \le \mathsf{pc}(s') \ \wedge\ \mathsf{pc}(f) \le \mathsf{pc}(f') \ \longrightarrow\\
&\qquad \mathsf{pc}(x) < \mathsf{pc}(s) \ \wedge\ \mathsf{pc}(f') < \mathsf{pc}(y) \ \longrightarrow\\
&\qquad \mathsf{pc}(y) < \mathsf{length}(\mathsf{get\_code}\ \mathit{CFS}\ s) \ \longrightarrow\\
&\qquad \mathsf{pc}(y) < \mathsf{pc}(\sigma'_n)\\
&\qquad \longrightarrow \langle \mathit{CFS}, \sigma_0\rangle \xrightarrow[y]{x} \sigma'_n
\end{aligned}$$

Proof The proof proceeds by rule induction (Section 1.3.3) on the first assumption $\langle \mathit{CFS}, \sigma_0\rangle \xrightarrow[f]{s} \sigma_n$.

Basis

The base case deals with the case in which our initial assumption was produced by one application of the **Stop** rule, giving us the assumption

$$\begin{aligned}&\mathsf{exec}\ (\mathit{CFS},\ \sigma_0) = \mathsf{Some}\ \sigma_n\ \wedge\\ &\mathsf{inside}\ \sigma_0\ s\ f\ \wedge\\ &\mathsf{outside}\ \sigma_n\ s\ f\end{aligned} \tag{3.8}$$

We now consider the two possible positions of the program counter in σ_n. From the definition of outside (Definition 2) we know that

$$\mathsf{pc}(\sigma_n)\ <\ s\ \vee\ f\ <\ \mathsf{pc}(\sigma_n) \tag{3.9}$$

The left hand disjunct, $\mathsf{pc}(\sigma_n) < \mathsf{pc}(s)$ gives a contradiction as we know from Lemma 1 that

$$\mathsf{inside}\ \mathsf{pc}(\sigma_n)\ \mathsf{pc}(s')\ \mathsf{pc}(f') \tag{3.10}$$

and from 3.8 that

$$\mathsf{pc}(s) \leq \mathsf{pc}(s') \tag{3.11}$$

Thus $\mathsf{pc}(f)\ <\ \mathsf{pc}(\sigma_n)$ and the basis can then be proved using the **Continue** rule and Lemma 5.

Inductive Step

The inductive step deals with the case in which our initial assumption was produced by at least one application of the **Continue** rule. By rule induction, we have the assumption

$$\begin{aligned}&\mathsf{exec}\ (\mathit{CFS},\ \sigma_0) = \mathsf{Some}\ \sigma_1\ \wedge\\ &\mathsf{inside}\ \sigma_0\ s\ f\ \wedge\\ &\langle \mathit{CFS}, \sigma_1\rangle \xrightarrow[f]{s}\ \sigma_n\end{aligned} \tag{3.12}$$

And from the inductive hypothesis after simplification we have

$$\langle \mathit{CFS}, \sigma_1\rangle \xrightarrow[y]{x}\ \sigma_n' \tag{3.13}$$

which by application of the **Continue** rule, gives the desired result. ∎

A block of size one is equivalent to execution of that instruction, providing the instruction is not a 'degenerate' branch (one which branches back to itself, or branches outside the bounds of the method), or an instruction which results

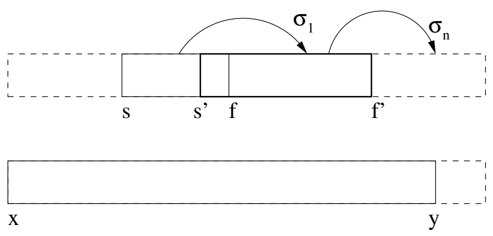

Figure 3.6: Add a block to the right, finish on the right

in a frame being pushed or popped from the stack, i.e. method invocation or return. The latter constraint is due to the current, simplified, definition of the block execution relation which demands that all states in the relation be in the same method.

The constraint on branching instructions is due to the fact that it would be possible to create a bytecode program (though probably not via a conventional compiler) which pushed the value Null onto the stack, followed by some other values, then had a "branch if not null" instruction which looped back to itself. Initially the top value on the stack would not be Null , but every time the branch instruction was evaluated, the top value on the stack would be popped, leading eventually to a state where the top value *was* Null and the loop was exited. This would mean that this instruction was in the block execution relation, but that this instance of the relation was *not* equivalent to a single-step execution of the initial state.

Lemma 8 (Single instruction block execution equivalence)

$$\begin{aligned}\forall \mathit{CFS}\ s\ x\ \sigma_0\ \sigma_n.\ & (\mathsf{get\_code}\ \mathit{CFS}\ s)!\mathsf{pc}(s) \ = \ x \ \wedge \\ & \mathsf{not\_degenerate\_branch}\ x \ \wedge \\ & \mathsf{not\_shift\_frame}\ x \ \wedge \\ & \mathsf{pc}(s) < \mathsf{length}\ (\mathsf{get\_code}\ \mathit{CFS}\ s) \longrightarrow \\ & \langle \mathit{CFS},\ \sigma_0 \rangle \xrightarrow[s]{s} \sigma_n \equiv \mathsf{exec}(\mathit{CFS},\ \sigma_0) = \mathsf{Some}\ \sigma_n\end{aligned}$$

Proof The result follows from case analysis of the instruction x and the operational semantics of the JVM. ∎

3.4 The Execution Path Relation

The block execution relation can be used to reason about an intermediate state in the execution of a block and the final state. It does not, however,

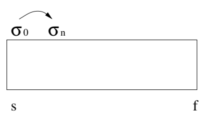

Figure 3.7: One step of execution

allow discussion of the connection between an intermediate state and the initial state, or between two intermediate states, as both the states in question are inside the block. Since it is clearly useful to be able to do this, a relation that enables us to reason about two states, at least one and possibly both of which are within a particular sequence of bytecode, is needed.

This relation is particularly useful in the proofs of the sequencing rule, where we have to prove the existence of a 'crossover' state in the execution of a block which is the result of joining two smaller blocks together. Also, in the proof of soundness of the while rule (Chapter 5), we must reason about the relationship between the initial state and various intermediate states in the execution of the loop.

The execution path relation is defined as the set of pairs of states obtained by a successful execution step, where the program counter of the first member of the pair is inside the block in question, see Figure 3.7. We write $\langle CFS, \sigma_0\rangle \overset{s}{\underset{f}{\Longrightarrow}} \sigma_1$ to mean that σ_0 is inside the block from s to f, and executing the instruction in CFS identified by σ_0 results in the state σ_1.

Definition 7 (Execution step in a block)

$$\langle CFS, \sigma_0\rangle \overset{s}{\underset{f}{\Longrightarrow}} \sigma_1 \;\equiv\; (\sigma_0, \sigma_1) \in \{(\sigma_0, \sigma_1).\ \mathsf{exec}\ (CFS, \sigma_0) \;=\; \mathsf{Some}\ \sigma_1 \;\wedge\; \mathsf{same\_method}\ s\ \sigma_0\ \sigma_1\ f \;\wedge\; \mathsf{inside}\ \sigma_0\ s\ f\}$$

The execution path relation is the transitive closure of this set, and we write $\langle CFS,\ \sigma_0\rangle \overset{s}{\underset{f}{\Longrightarrow}}{}^{+} \sigma_n$ to mean that the pair $(\sigma_0,\ \sigma_n)$ is an element of the transitive closure of the set of pairs of states represented by the relationship $\langle CFS, \sigma_0\rangle \overset{s}{\underset{f}{\Longrightarrow}} \sigma_1$ Figure 3.8.

Definition 8 (Execution Path Relation)

$$\langle CFS,\ \sigma_0\rangle \overset{s}{\underset{f}{\Longrightarrow}}{}^{+} \sigma_n \;\equiv\; (\sigma_0, \sigma_n) \in \{(\sigma_0, \sigma_n).\ \mathsf{exec}\ (CFS, \sigma_0) \;=\; \mathsf{Some}\ \sigma_n \;\wedge\; \mathsf{same\_method}\ s\ \sigma_0\ \sigma_n\ f \;\wedge\; \mathsf{inside}\ \sigma_0\ s\ f\}^{+}$$

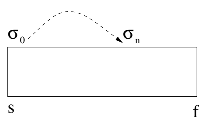

Figure 3.8: Execution path relation

Many of the proofs involving the execution path relation rely on lemmas about transitive closure which come as part of the standard Isabelle distribution.

The following lemmas show the correspondence between the execution path relation and the block execution relation.

Lemma 9 (Block execution relation implies execution path relation)

$$\forall\ CFS\ s\ f\ \sigma_0\ \sigma_n.\ \langle CFS, \sigma_0\rangle \xrightarrow[f]{s} \sigma_n \longrightarrow \langle CFS,\ \sigma_0\rangle \overset{s}{\underset{f}{\Longrightarrow}}{}^{+} \sigma_n$$

Proof This follows from induction on the construction rules for the block execution relation (Definition 4) and the definition of the execution path relation (Definition 8). ■

Lemma 10 (Execution path implies block execution)

$$\forall\ CFS\ s\ f\ \sigma_0\ \sigma_n.\ \langle CFS,\ \sigma_0\rangle \overset{s}{\underset{f}{\Longrightarrow}}{}^{+} \sigma_n \wedge \mathsf{outside}\ \sigma_n\ s\ f$$
$$\longrightarrow \langle CFS, \sigma_0\rangle \xrightarrow[f]{s} \sigma_n$$

Proof By induction on the execution path relation and the construction rules for the block execution (Definition 4). ■

Lemma 11 (Unrolling the relation from the start)

$$\forall\ CFS\ s\ f\ \sigma_0\ \sigma_n.\ \langle CFS,\ \sigma_0\rangle \overset{s}{\underset{f}{\Longrightarrow}}{}^{+} \sigma_n$$
$$\longrightarrow \langle CFS, \sigma_0\rangle \overset{s}{\underset{f}{\Longrightarrow}} \sigma_n \vee$$
$$\exists\ \sigma_1.\ \langle CFS,\ \sigma_0\rangle \overset{s}{\underset{f}{\Longrightarrow}} \sigma_1 \wedge \langle CFS,\ \sigma_1\rangle \overset{s}{\underset{f}{\Longrightarrow}}{}^{+} \sigma_n$$

Proof This follows from the definition of the execution path relation (Definition 8) and the standard lemmas for transitive closure in Isabelle. ■

Lemma 12 (Unrolling the relation from the end)

$$\begin{array}{l} \forall\ \mathit{CFS}\ s\ f\ \sigma_0\ \sigma_n.\ \langle \mathit{CFS},\ \sigma_0 \rangle \overset{s}{\underset{f}{\Longrightarrow}}{}^{+}\ \sigma_n \\ \quad \longrightarrow\ \langle \mathit{CFS},\ \sigma_0 \rangle \overset{s}{\underset{f}{\Longrightarrow}}\ \sigma_n\ \vee \\ \qquad\qquad \exists\ \sigma_{n-1}.\ \langle \mathit{CFS},\ \sigma_0 \rangle \overset{s}{\underset{f}{\Longrightarrow}}{}^{+}\ \sigma_{n-1}\ \wedge\ \langle \mathit{CFS},\ \sigma_{n-1} \rangle \overset{s}{\underset{f}{\Longrightarrow}}\ \sigma_n \end{array}$$

PROOF This follows from the definition of the execution path relation (Definition 8) and the standard lemmas for transitive closure in Isabelle. ∎

The execution path relation is also used to define the concept of a list of instructions being free of exceptions

Definition 9 (Exception free instructions)

$$\begin{array}{l} \mathsf{excep\_free}\ ys\ \equiv\ \forall\ \mathit{CFS}\ xp\ hp\ frs\ xp'\ hp'\ frs'\ s\ f. \\ \qquad ys\ =\ \mathit{CFS}\ [s \ldots f]\ \wedge \\ \qquad \langle \mathit{CFS},\ (xp, hp, frs) \rangle \overset{s}{\underset{f}{\Longrightarrow}}{}^{+}\ (xp', hp', frs') \\ \qquad \longrightarrow xp\ =\ \mathsf{None}\ \wedge\ xp'\ =\ \mathsf{None} \end{array}$$

3.5 Determinism Theorems

Lemma 13 (Execution of a single instruction is deterministic)

$$\begin{array}{l} \forall \mathit{CFS}\ s\ f\ \sigma_0\ \sigma_n\ \sigma'_n.\ \mathsf{exec}\ (\mathit{CFS},\ \sigma_0)\ =\ \mathsf{Some}\ \sigma_n\ \ \wedge \\ \qquad \mathsf{exec}\ (\mathit{CFS},\ \sigma_0)\ =\ \mathsf{Some}\ \sigma'_n \longrightarrow \\ \qquad \sigma_n\ =\ \sigma'_n \end{array}$$

PROOF By case analysis of the instruction identified by σ_0, followed by automatic simplification with the rules for exec. ∎

We now show that this determinism is preserved by the block execution relation, and discuss the determinism of execution of a series of instructions not defined in relation to any particular class file.

Theorem 1 (Block execution relation is deterministic)

$$\begin{aligned}\forall \mathit{CFS}\ s\ f\ \sigma_0\ \sigma_n\ \sigma'_n.\ &\langle \mathit{CFS}, \sigma_0\rangle \xrightarrow[f]{s} \sigma_n \longrightarrow\\ &\langle \mathit{CFS}, \sigma_0\rangle \xrightarrow[f]{s} \sigma'_n \longrightarrow\\ &\sigma_n = \sigma'_n\end{aligned}$$

Proof The proof proceeds by rule induction (Section 1.3.3) on the first assumption, $\langle \mathit{CFS}, \sigma_0\rangle \xrightarrow[f]{s} \sigma_n$.

Basis

The base case deals with the case in which our initial assumption was produced by one application of the **Stop** rule, giving us the assumption

$$\begin{aligned}&\mathsf{exec}\ (\mathit{CFS},\ \sigma_0) = \sigma_n\ \wedge\\ &\mathsf{inside}\ \sigma_0\ s\ f\ \wedge\\ &\mathsf{outside}\ \sigma_n\ s\ f\end{aligned} \tag{3.14}$$

We now consider the two possible cases of derivation of the second assumption, $\langle \mathit{CFS}, \sigma_0\rangle \xrightarrow[f]{s} \sigma'_n$.

1. The block was formed by one application of the **Stop** rule, giving us the assumption

$$\begin{aligned}&\mathsf{exec}\ (\mathit{CFS},\ \sigma_0) = \sigma'_n\ \wedge\\ &\mathsf{inside}\ \sigma_0\ s\ f\ \wedge\\ &\mathsf{outside}\ \sigma'_n\ s\ f\end{aligned} \tag{3.15}$$

   From this and (3.14) we are able to show the desired conclusion by Lemma 13.

2. The block was formed by at least one application of the **Continue** rule, giving us the assumption

$$\begin{aligned}&\mathsf{exec}\ (\mathit{CFS},\ \sigma_0) = \sigma'_1\ \wedge\\ &\mathsf{inside}\ \sigma_0\ s\ f\ \wedge\\ &\langle \mathit{CFS}, \sigma'_1\rangle \xrightarrow[f]{s} \sigma'_n\end{aligned} \tag{3.16}$$

From this assumption and Lemma 13 it follows that $\sigma_n = \sigma'_1$, but we also have from (3.14) $\mathsf{outside}\ \sigma_n\ s\ f$ and, from Lemma 1, $\mathsf{inside}\ \sigma'_1\ s\ f$. From the definitions of inside (Definition 1) and outside (Definition 2) the program counter of a state cannot be both inside and outside any block, and so we are able to show a contradiction.

Inductive Step

The inductive step deals with the case in which our initial assumption was produced by at least one application of the **Continue** rule. The Isabelle output for this step is shown in Figure 3.9 for comparison. By rule induction, we have the assumption

$$\begin{aligned}&\mathsf{exec}\ (CFS,\ \sigma_0) = \sigma_1\ \wedge\\ &\mathsf{inside}\ \sigma_0\ s\ f\ \wedge\\ &\langle CFS, \sigma_1\rangle \xrightarrow[f]{s} \sigma_n\end{aligned} \tag{3.17}$$

The inductive hypothesis is

$$\langle CFS, \sigma_1\rangle \xrightarrow[f]{s} \sigma'_n \longrightarrow \sigma_n\ =\ \sigma'_n \tag{3.18}$$

Again we consider the two possible cases of the second assumption,

1. The block was formed by one application of the **Stop** rule, giving us the assumption

$$\begin{aligned}&\mathsf{exec}\ (CFS,\ \sigma_0) = \sigma'_n\ \wedge\\ &\mathsf{inside}\ \sigma_0\ s\ f\ \wedge\\ &\mathsf{outside}\ \sigma'_n\ s\ f\end{aligned} \tag{3.19}$$

   From this, Lemma 13, and (3.17) we can show that $\sigma_1 = \sigma'_n$. It follows from Lemma 1 that $\mathsf{inside}\ \sigma_1\ s\ f$, and from 3.19 that $\mathsf{outside}\ \sigma_1\ s\ f$. Once again, from the definitions of inside (Definition 1) and outside (Definition 2) we are able to show a contradiction.

2. The block was formed by at least one application of the **Continue** rule, giving us the assumption

$$\begin{aligned}&\mathsf{exec}\ (CFS,\ \sigma_0) = \sigma'_1\ \wedge\\ &\mathsf{inside}\ \sigma_0\ s\ f\ \wedge\\ &\langle CFS, \sigma'_1\rangle \xrightarrow[f]{s} \sigma'_n\end{aligned} \tag{3.20}$$

```
CFS (cn1, (mn1, pd1), PCm) (cn2, (mn2, pd2), PCj)
|- (xp1, hp1, frs1) -block-> xp2, hp2, frs2 -->
CFS (cn1, (mn1, pd1), PCm) (cn2, (mn2, pd2), PCj)
|- (xp1, hp1, frs1) -block-> xp3, hp3, frs3 -->
frs2 ~= [] --> xp1 = None --> (xp2, hp2, frs2) = (xp3, hp3, frs3)
 1. !!CFS a aa b ab ac ba ad ae bb af ag bc bd ah ai be bf.
  [| exec (CFS, a, aa, b) = Some (ad, ae, bb);
 inside (third_of (a, aa, b)) (ah, (ai, be), bf) (af, (ag, bc), bd);
  pc_of (af, (ag, bc), bd)
 < length (get_code CFS (cn_of (ah, (ai, be), bf))
(ml_of (ah, (ai, be), bf)));
 same_method_frs (ah, (ai, be), bf)
(hd (snd (snd (a, aa, b))))
  (hd (snd (snd (ad, ae, bb))))
          (af, (ag, bc), bd);
CFS (ah, (ai, be), bf) (af, (ag, bc), bd)
|- (ad, ae, bb) -block-> ab, ac, ba;

CFS (ah, (ai, be), bf) (af, (ag, bc), bd)
|- (ad, ae, bb) -block-> xp3, hp3, frs3 -->
   ba ~= [] --> ad = None --> (ab, ac, ba) = (xp3, hp3, frs3);

CFS (ah, (ai, be), bf) (af, (ag, bc), bd)
   |- (a, aa, b) -block-> xp3, hp3, frs3; ba ~= [];a = None|]
==>  (ab, ac, ba) = (xp3, hp3, frs3)
```

Figure 3.9: Induction step in Isabelle for Theorem 1

By this, Lemma 13, and (3.17) we have $\sigma_1 = \sigma_1'$ which by the inductive hypothesis gives the required result. ■

3.5.1 Determinism of Two Identical Instruction Sequences

It would seem likely that executing the sequence of instructions xs starting in state σ_0 should result in state σ_n regardless of whether xs is found in one set of class files CFS, or a different set CFS'.

But this is not the case as, in both the block execution relation and the operational semantics on which it is based, instructions are not independent entities which can be described separately from the state in which they are being executed. We might consider that a state contains two separate sets of information: traditional environmental data such as the stack, local variables and program counter; and contextual information in the form of a class name and method locator. This means that two states which are environmentally equal—and would consequently be equal in the traditional sense—may have

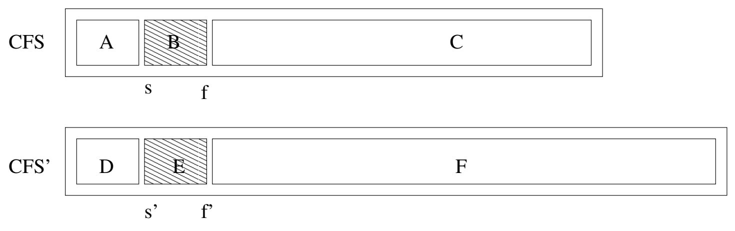

Figure 3.10: Two 'coinciding' class files

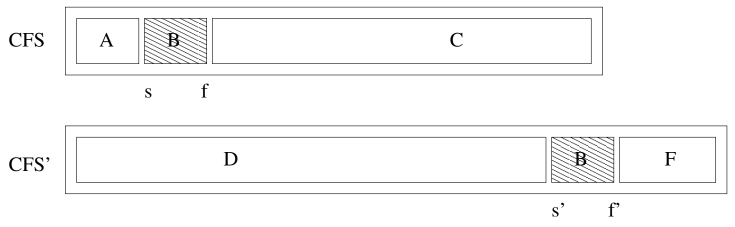

Figure 3.11: Classfiles containing same sequence at different points

different contextual information and therefore not be equal at all in the JVM setting.

Thus it *is* possible in some cases to prove that the execution of a sequence of instructions is deterministic even if it appears in two, non-identical sets of classfiles. However it is only possible to prove this in the—somewhat unlikely—situation in which the two sets of classfiles containing the same instructions, at the same point, in identically named methods inside identically named classes (Figure 3.10). This is a very limited result and leaves us unable to prove determinism for the more realistic scenarios of Figures 3.11 and 3.12, where *xs* appears in two disparate sets of classfiles, or in two different classes in the same set of classfiles.

3.5.2 Data-equality of States

As we have seen in the previous section, it is not possible to talk meaningfully about deterministic execution in terms of an entire JVM state. Pusch's for-

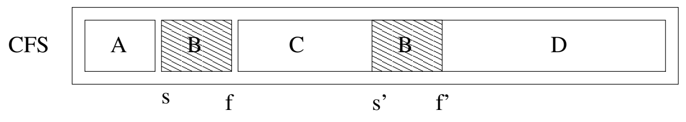

Figure 3.12: Two instances of a sequence in one set of classfiles

malization of the JVM aims to mirror as closely as possible the 'real world' in which Java bytecode programs are executed. Consequently, in Pusch's model of the JVM world instructions are not viewed in isolation as independent entities, but as part of the state itself. Therefore, in order to discuss determinism in the accepted sense of the word, we must define a different type of 'equality' for states.

Two states are said to be dataequal ($\cong$), if their exception options and the values of the stack and local variables in the top frame of their frame stacks are equal:

Definition 10 (Data-equality of states)

$$\begin{aligned}(xp, hp, frs) \;\cong\; (xp', hp', frs') \;\equiv\;& xp = xp' \;\wedge \\ & \mathsf{stk}\, frs \;=\; \mathsf{stk}\, frs' \;\wedge \\ & \mathsf{loc}\, frs \;=\; \mathsf{loc}\, frs'\end{aligned}$$

Of course, this definition of data-equality is not the only possible one. In a situation where we wish to compare two states within the execution of a single method—as will be discussed in Chapters 4 and 5 in relation to while loops—it would be necessary only to exclude the current value of the program counter from the comparison. Equally, if we wished to talk about a situation involving the execution of instructions which reference the heap, this would have to become part of the definition of data-equality. And so it is apparent that there may be several equalities of this nature.

For the purposes of this document, however, there are only two situations in which we will need to use the idea of data-equality. First, in the proof of the determinism (in terms of data-equality) of two identical instruction sequences appearing in different places; and second, in the comparison of two states in the execution of a single method.

Both cases are involved either directly, or indirectly, with the calculation of the weakest precondition with respect to a sequence of instructions and a condition Q. As will be discussed in some detail in Chapters 4 and 5, the instructions will be restricted to classfile independent, non-branching instructions, and all instances of Q will be concerned only with the stack and local variables of the topmost frame. As there will be no alteration to the object fields or heap, the notion of data-equality given in Definition 10 is therefore sufficient for the work described here.

3.5.3 The Sequence Execution Relation

In order to prove that if the initial states of two instances of the block execution relation are data-equal, then the final states will also be data-equal, it would seem reasonable to proceed by induction on the block execution relation. But while this approach succeeds for the base case, the step case requires us to prove that if the initial cases of each block are data-equal, then the initial state of the second block and the state reached after one step of execution of the first block are also data-equal.

Unfortunately, this is not necessarily the case. For example, consider the blocks $\langle CFS, \sigma_0 \rangle \xrightarrow[f]{s} \sigma_n$ and $\langle CFS', \sigma'_0 \rangle \xrightarrow[f']{s'} \sigma'_n$ where $CFS[s \ldots f] = CFS'[s' \ldots f'] = xs$, $\sigma_0 \cong \sigma'_0$ and $\mathsf{exec}\ (CFS,\ \sigma_0) = \sigma_1$. If the first instruction in xs pushes a value onto the stack, the length of the stack in state σ_1, i.e. the state reached after one step of execution of the first block, will be greater than the length of the stack in σ'_0, and so the states are not data-equal. We must therefore find another solution to the problem.

One possibility is to again abstract away from the contextual aspects of the state within a set of classfiles. This results in an inductively defined relation operating only on the elements of state involved in the definition of data-equality, the relevant instructions as a sequence in its own right, and a pointer into the current position within this sequence.

One further restriction necessary to produce a truly ‘context-independent’ relation is to limit the instructions involved to only *classfile independent* instructions. These are instructions that do not reference additional information in the relevant classfile, such as whether or not it contains a particular class or method instance, and comprise the load and store instructions, op-stack instructions, and all branching instructions.

Suppose that

- xs is a list of bytecode instructions
- σ_0 and σ_n are of type $\mathsf{minstate}$, a tuple consisting of an exception option, an operand stack, a list of local variables, and a program counter relative to the start of xs.

We write $xs \vdash \sigma_0 \rightsquigarrow \sigma_n$ if executing the sequence of instructions in xs in the state σ_0 results in the state σ_n, where the instruction identified by σ_n is not contained in xs (where xs is non-empty).

The function exec_indep is defined as a partial function using Option types. It returns the updated state according to the operational semantics for all opstack instructions, load and store instructions, and branching instructions, and the value None for all other instructions. It follows as closely as possible the definition of exec in the underlying semantics, but with the omission of the 'exception handling' method of returning an empty list of frames if a state containing an exception is executed. Here the result is simply None—the result reached by exec on the next step of execution in any case.

Definition 11 (Execute CFS independent instruction)

```
exec_indep ($[\,], (xp, frs)$) = None
exec_indep ($xs$, (None, ($stk, loc, pc$))) = case $xs!pc$ of
    |LAS $ins$ ⇒ let ($stk', loc', pc'$) = exec_las $ins$ $stk$ $loc$ $pc$ in
                 Some (None,($stk', loc', pc'$))
    |CO $ins$  ⇒ None
    |MO $ins$  ⇒ None
    |MA $ins$  ⇒ None
    |CH $ins$  ⇒ None
    |MI $ins$  ⇒ None
    |MR $ins$  ⇒ None
    |OS $ins$  ⇒ let ($stk', pc'$) = exec_os $ins$ $stk$ $pc$ in
                 Some (None, ($stk', loc, pc'$))
    |CBF $ins$ ⇒ let ($stk', pc'$) = exec_cb_fwd $ins$ $stk$ $pc$ in
                 Some (None, ($stk', loc, pc'$))
    |CBB $ins$ ⇒ let ($stk', pc'$) = exec_cb_bwd $ins$ $stk$ $pc$ in
                 Some (None, ($stk', loc, pc'$))
    |UBF $ins$ ⇒ let ($pc'$) = exec_ub_fwd $ins$ $pc$ in
                 Some (None, ($stk, loc, pc'$))
    |UBB $ins$ ⇒ let ($pc'$) = exec_ub_bwd $ins$ $pc$ in
                 Some (None, ($stk, loc, pc'$))
exec_indep ($xs$, (Some $xp, frs$)) = None
```

We also define

Definition 12 (Inlist)

inlist xs pc $\equiv$ $0 \leq pc \ \wedge \ pc \leq ((\text{length } xs) - 1)$

and

Definition 13 (Outlist)

outlist xs pc $\equiv$ $(\text{length } xs) \leq pc$

The sequence execution relation is then described by the following rules

Definition 14 (Sequence Execution Relation)

$$\frac{\begin{array}{l}\mathsf{exec\_indep}(xs,\ \tau_0) \ = \ \tau_n; \\ xs \ \neq \ [\,]; \\ \mathsf{inlist}\ xs\ \mathsf{pc}(\tau_0); \\ \mathsf{outlist}\ xs\ \mathsf{pc}(\tau_n)\end{array}}{xs \ \vdash \ \tau_0 \ \rightsquigarrow \ \tau_n} \qquad (\textbf{Seq-Stop})$$

$$\frac{\begin{array}{l}\mathsf{exec\_indep}(xs,\ \tau_0) \ = \ \tau_1; \\ xs \ \neq \ [\,]; \\ \mathsf{inlist}\ xs\ \mathsf{pc}(\tau_0); \\ xs \ \vdash \ \tau_1 \ \rightsquigarrow \ \tau_n\end{array}}{xs \ \vdash \ \tau_0 \ \rightsquigarrow \ \tau_n} \qquad (\textbf{Seq-Continue})$$

Using these definitions, we are able to prove determinacy for exec_indep and, by induction, for the sequence execution relation.

Lemma 14 (Execute CFS independent function is deterministic)

$$\begin{array}{ll}\forall\ xs\ s\ f\ \tau_0\ \tau_n\ \tau_n'. & \mathsf{exec\_indep}(xs,\ \tau_0) \ = \ \mathsf{Some}\ \tau_n \ \wedge \\ & \mathsf{exec\_indep}(xs,\ \tau_0) \ = \ \mathsf{Some}\ \tau_n' \\ & \longrightarrow \tau_n \ = \ \tau_n'\end{array}$$

Proof By exhaustion on the instruction at $\mathsf{pc}(\tau_0)$ and simplification using the definition of exec_indep (Definition 11). ■

Theorem 2 (Sequence execution relation is deterministic)

$$\begin{array}{ll}\forall\ xs\ s\ f\ \tau_0\ \tau_n\ \tau_n'. & xs \ \vdash \ \tau_0 \ \rightsquigarrow \ \tau_n \\ & xs \ \vdash \ \tau_0 \ \rightsquigarrow \ \tau_n' \\ & \longrightarrow \tau_n \ = \ \tau_n'\end{array}$$

Proof By induction on the rules for the sequence execution relation (**Seq-Stop**, **Seq-Continue**). ■

3.5.4 Determinism of Instruction Sequences with Data-equality

Using the sequence execution relation described above, we are now able to prove the determinacy of a block of bytecode instructions, regardless of their location within a set of classfiles. We begin by establishing the relationship between the block execution and sequence execution relations.

Although in the above definition and the subsequent lemmas involving the sequence execution relation values of type *minstate* are denoted by the symbol τ for ease of reading, such variables of course represent a 4-tuple of the form (xp, stk, loc, pc). Despite being stated in terms of a single state in the Isabelle proof scripts, Isabelle produces an induction theorem in terms of the fully expanded *minstate*. This is useful as it makes it possible to prove by induction lemmas concerned with the value of individual elements of a *minstate* (Figure 3.13).

The definition of the block execution relation is similarly stated in terms of a single variable of type *state* of the form $(xp, hp, frame\ stack\ list)$. In this case, however, Isabelle produces an induction theorem where the frame stack list is a single, unexpanded variable (Figure 3.13). This means that, unlike the case of the sequence execution relation, we are unable to use the induction theorem to prove statements involving individual parts of a frame stack, e.g., Lemma 21.

This makes it necessary to introduce a definition that is equivalent to the block execution definition but which explicitly mentions the individual parts of a frame stack. This is done by defining the frame stack list of any state as two separate variables: the topmost frame on the stack and a list containing the lower frames. The production rules for this and the original block execution relation are shown in Figure 3.14 and the induction rule for the expanded version in Figure 3.15.

It was proved that if two states are in the block execution relation with expanded syntax then they are also in the standard syntax block execution, allowing lemmas involving both the block execution relation and the sequence execution relation to be obtained. But as the expanded relation is essentially an artefact of the proof tool and not of any interest in itself no further lemmas explicitly involving it will be discussed in this document. Interested readers can examine the proofs on the attached CD.

We now prove a number of lemmas involving the sequence execution relation, several of which use the following predicate on lists as defined in the standard

```
val exec_instrs.induct =
  "[| ?xi |- (?xh, ?xg, ?xf, ?xe) -instrs-> ?xd, ?xc, ?xb, ?xa;
      !!a aa ab b ac ad ae ba xs.
         [| execCFSindep (xs, a, aa, ab, b) = Some (ac, ad, ae, ba); xs ~= [];
            inlist xs (snd (a, aa, ab, b));
            outlist xs (snd (ac, ad, ae, ba)) |]
         ==> ?P xs a aa ab b ac ad ae ba;
      !!a aa ab b ac ad ae ba af ag ah bb xs.
         [| execCFSindep (xs, a, aa, ab, b) = Some (af, ag, ah, bb); xs ~= [];
            inlist xs (snd (a, aa, ab, b));
            xs |- (af, ag, ah, bb) -instrs-> ac, ad, ae, ba;
            ?P xs af ag ah bb ac ad ae ba |] ==> ?P xs a aa ab b ac ad ae ba |]
   ==> ?P ?xi ?xh ?xg ?xf ?xe ?xd ?xc ?xb ?xa" : thm

val exec_block3.induct =
  "[| ?xo (?xn, (?xm, ?xl),
           ?xk) (?xj, (?xi, ?xh), ?xg) |- (?xf, ?xe, ?xd) -block-> ?xc, ?xb,
      ?xa;
      !!CFS a aa b ab ac ba ad ae bb bc af ag bd be.
         [| exec (CFS, a, aa, b) = Some (ab, ac, ba);
            inside (third_of (a, aa, b)) (af, (ag, bd), be) (ad, (ae, bb), bc);
            pc_of (ad, (ae, bb), bc)
            < length
               (get_code CFS (cn_of (af, (ag, bd), be))
                 (ml_of (af, (ag, bd), be)));
            same_method_frs (af, (ag, bd), be) (hd (snd (snd (a, aa, b))))
             (hd (snd (snd (ab, ac, ba)))) (ad, (ae, bb), bc);
            outside (third_of (ab, ac, ba)) (af, (ag, bd), be)
             (ad, (ae, bb), bc) |]
         ==> ?P CFS af ag bd be ad ae bb bc a aa b ab ac ba;
      !!CFS a aa b ab ac ba ad ae bb af ag bc bd ah ai be bf.
         [| exec (CFS, a, aa, b) = Some (ad, ae, bb);
            inside (third_of (a, aa, b)) (ah, (ai, be), bf) (af, (ag, bc), bd);
            pc_of (af, (ag, bc), bd)
            < length
               (get_code CFS (cn_of (ah, (ai, be), bf))
                 (ml_of (ah, (ai, be), bf)));
            same_method_frs (ah, (ai, be), bf) (hd (snd (snd (a, aa, b))))
             (hd (snd (snd (ad, ae, bb)))) (af, (ag, bc), bd);
            CFS (ah, (ai, be),
                 bf) (af, (ag, bc), bd) |- (ad, ae, bb) -block-> ab, ac, ba;
            ?P CFS ah ai be bf af ag bc bd ad ae bb ab ac ba |]
         ==> ?P CFS ah ai be bf af ag bc bd a aa b ab ac ba |]
   ==> ?P ?xo ?xn ?xm ?xl ?xk ?xj ?xi ?xh ?xg ?xf ?xe ?xd ?xc ?xb ?xa" : thm
```

Figure 3.13: Induction theorems for sequence and block execution relations

```
val exec_block3.Stop =
  "[| exec (?CFS, ?a) = Some ?b; inside (third_of ?a) ?s ?f;
      pc_of ?f < length (get_code ?CFS (cn_of ?s) (ml_of ?s));
      same_method_frs ?s (hd (snd (snd ?a))) (hd (snd (snd ?b))) ?f;
      outside (third_of ?b) ?s ?f |] ==> ?CFS ?s ?f |- ?a -block-> ?b" : thm

val exec_block3.Continue =
  "[| exec (?CFS, ?a) = Some ?c; inside (third_of ?a) ?s ?f;
      pc_of ?f < length (get_code ?CFS (cn_of ?s) (ml_of ?s));
      same_method_frs ?s (hd (snd (snd ?a))) (hd (snd (snd ?c))) ?f;
      ?CFS ?s ?f |- ?c -block-> ?b |] ==> ?CFS ?s ?f |- ?a -block-> ?b" : thm

val exec_block0.Stop =
  "[| exec (?CFS, ?xp, ?hp, (?stk, ?loc, ?cn, (?mn, ?pd), ?pc) # ?frs) =
      Some (?xp', ?hp', (?stk', ?loc', ?cn', (?mn', ?pd'), ?pc') # ?frs');
      inside ((?stk, ?loc, ?cn, (?mn, ?pd), ?pc) # ?frs)
       (?cnS, (?mnS, ?pdS), ?pcS) (?cnF, (?mnF, ?pdF), ?pcF);
      ?pcF < length (get_code ?CFS ?cnS (?mnS, ?pdS));
      same_method_frs (?cnS, (?mnS, ?pdS), ?pcS)
       (?stk, ?loc, ?cn, (?mn, ?pd), ?pc)
       (?stk', ?loc', ?cn', (?mn', ?pd'), ?pc') (?cnF, (?mnF, ?pdF), ?pcF);
      outside ((?stk', ?loc', ?cn', (?mn', ?pd'), ?pc') # ?frs')
       (?cnS, (?mnS, ?pdS), ?pcS) (?cnF, (?mnF, ?pdF), ?pcF) |]
   ==> (?CFS, (?cnS, (?mnS, ?pdS), ?pcS), (?cnF, (?mnF, ?pdF), ?pcF),
        (?xp, ?hp, ?stk, ?loc, ?cn, (?mn, ?pd), ?pc),
        (?xp', ?hp', ?stk', ?loc', ?cn', (?mn', ?pd'), ?pc'), ?frs, ?frs')
       : exec_block0" : thm

val exec_block0.Continue =
  "[| exec (?CFS, ?xp, ?hp, (?stk, ?loc, ?cn, (?mn, ?pd), ?pc) # ?frs) =
      Some
       (?xp'', ?hp'', (?stk'', ?loc'', ?cn'', (?mn'', ?pd''), ?pc'') # ?frs'');
      inside ((?stk, ?loc, ?cn, (?mn, ?pd), ?pc) # ?frs)
       (?cnS, (?mnS, ?pdS), ?pcS) (?cnF, (?mnF, ?pdF), ?pcF);
      ?pcF < length (get_code ?CFS ?cnS (?mnS, ?pdS));
      same_method_frs (?cnS, (?mnS, ?pdS), ?pcS)
       (?stk, ?loc, ?cn, (?mn, ?pd), ?pc)
       (?stk'', ?loc'', ?cn'', (?mn'', ?pd''), ?pc'')
       (?cnF, (?mnF, ?pdF), ?pcF);
      (?CFS, (?cnS, (?mnS, ?pdS), ?pcS), (?cnF, (?mnF, ?pdF), ?pcF),
       (?xp'', ?hp'', ?stk'', ?loc'', ?cn'', (?mn'', ?pd''), ?pc''),
       (?xp', ?hp', ?stk', ?loc', ?cn', (?mn', ?pd'), ?pc'), ?frs'', ?frs')
      : exec_block0 |]
   ==> (?CFS, (?cnS, (?mnS, ?pdS), ?pcS), (?cnF, (?mnF, ?pdF), ?pcF),
        (?xp, ?hp, ?stk, ?loc, ?cn, (?mn, ?pd), ?pc),
        (?xp', ?hp', ?stk', ?loc', ?cn', (?mn', ?pd'), ?pc'), ?frs, ?frs')
       : exec_block0" : thm
```

Figure 3.14: Production rules for block execution relations

```
val exec_block0.induct =
"[| (?ya, (?xz, (?xy, ?xx), ?xw), (?xv, (?xu, ?xt), ?xs),
     (?xr, ?xq, ?xp, ?xo, ?xn, (?xm, ?xl), ?xk),
     (?xj, ?xi, ?xh, ?xg, ?xf, (?xe, ?xd), ?xc), ?xb, ?xa) : exec_block0;
    !!CFS cn cn' cnF cnS frs frs' hp hp' loc loc' mn mn' mnF mnS pc pc' pcF
       pcS pd pd' pdF pdS stk stk' xp xp'.
       [| exec (CFS, xp, hp, (stk, loc, cn, (mn, pd), pc) # frs) =
          Some (xp', hp', (stk', loc', cn', (mn', pd'), pc') # frs');
          inside ((stk, loc, cn, (mn, pd), pc) # frs) (cnS, (mnS, pdS), pcS)
           (cnF, (mnF, pdF), pcF);
          pcF < length (get_code CFS cnS (mnS, pdS));
          same_method_frs (cnS, (mnS, pdS), pcS) (stk, loc, cn, (mn, pd), pc)
           (stk', loc', cn', (mn', pd'), pc') (cnF, (mnF, pdF), pcF);
          outside ((stk', loc', cn', (mn', pd'), pc') # frs')
           (cnS, (mnS, pdS), pcS) (cnF, (mnF, pdF), pcF) |]
       ==> ?P CFS cnS mnS pdS pcS cnF mnF pdF pcF xp hp stk loc cn mn pd pc
             xp' hp' stk' loc' cn' mn' pd' pc' frs frs';
    !!CFS cn cn' cn'' cnF cnS frs frs' frs'' hp hp' hp'' loc loc' loc'' mn
       mn' mn'' mnF mnS pc pc' pc'' pcF pcS pd pd' pd'' pdF pdS stk stk'
       stk'' xp xp' xp''.
       [| exec (CFS, xp, hp, (stk, loc, cn, (mn, pd), pc) # frs) =
          Some
           (xp'', hp'', (stk'', loc'', cn'', (mn'', pd''), pc'') # frs'');
          inside ((stk, loc, cn, (mn, pd), pc) # frs) (cnS, (mnS, pdS), pcS)
           (cnF, (mnF, pdF), pcF);
          pcF < length (get_code CFS cnS (mnS, pdS));
          same_method_frs (cnS, (mnS, pdS), pcS) (stk, loc, cn, (mn, pd), pc)
           (stk'', loc'', cn'', (mn'', pd''), pc'') (cnF, (mnF, pdF), pcF);
          (CFS, (cnS, (mnS, pdS), pcS), (cnF, (mnF, pdF), pcF),
           (xp'', hp'', stk'', loc'', cn'', (mn'', pd''), pc''),
           (xp', hp', stk', loc', cn', (mn', pd'), pc'), frs'', frs')
          : exec_block0;
          ?P CFS cnS mnS pdS pcS cnF mnF pdF pcF xp'' hp'' stk'' loc'' cn''
           mn'' pd'' pc'' xp' hp' stk' loc' cn' mn' pd' pc' frs'' frs' |]
       ==> ?P CFS cnS mnS pdS pcS cnF mnF pdF pcF xp hp stk loc cn mn pd pc
            xp' hp' stk' loc' cn' mn' pd' pc' frs frs' |]
 ==> ?P ?ya ?xz ?xy ?xx ?xw ?xv ?xu ?xt ?xs ?xr ?xq ?xp ?xo ?xn ?xm ?xl ?xk
      ?xj ?xi ?xh ?xg ?xf ?xe ?xd ?xc ?xb ?xa" : thm
```

Figure 3.15: Induction theorem for expanded syntax block execution relation

Isabelle distribution

Definition 15 (All elements in list have property *P*)

$$\mathsf{list\_all}\ P\ xs\ \equiv\ \forall x.\ x \in (set\ xs) \longrightarrow\ Px$$

In the following lemmas involving the sequence execution relation full states are abbreviated by σ and minstates by τ in the statement of the lemma, followed by expanded versions signified by ‘where’.

Lemma 15 (Full state execution implies minstate execution)

$$\begin{aligned}
\forall\ \mathit{CFS}\ s\ f\ \sigma_0\ \sigma_n \tau_0\ \tau_n\ xs.\ & \mathsf{exec}(\mathit{CFS}, \sigma_0)\ =\ \mathsf{Some}\ \sigma_n\ \wedge \\
& \mathsf{list\_all\ CFS\_indep}\ xs\ \wedge \\
& \mathsf{list\_all\ not\_branch}\ xs\ \wedge \\
& xp_0\ =\ \mathsf{None}\ \wedge \\
& \mathit{CFS}[s \ldots f]\ =\ xs\ \wedge \\
& \mathsf{pc}(s) \leq \mathsf{pc}(f)\ \wedge \\
& \mathsf{pc}(f) < \mathsf{length\ (get\_code)}\ \mathit{CFS}\ s) \\
& \longrightarrow \mathsf{exec\_indep}\ (xs,\ \tau_0)\ =\ \mathsf{Some}\ \tau_n
\end{aligned}$$

where

$$\begin{aligned}
\sigma_0 &= (xp_0,\ hp_0,\ (stk_0, loc_0, cn_0, ml_0, pc_0) : frs_0) \\
\sigma_n &= (xp_n,\ hp_n,\ (stk_n, loc_n, cn_n, ml_n, pc_n) : frs_n) \\
\tau_0 &= (xp_0, stk_0, loc_0, (pc_0 - \mathsf{pc}(s))) \\
\tau_n &= (xp_n, stk_n, loc_n, (pc_n - \mathsf{pc}(s)))
\end{aligned}$$

PROOF By case analysis of the instruction at $(\mathsf{pc}(s) - \mathsf{pc}(s))$, followed by simplification with the definition of exec_indep (Definition 11) and the operational semantics of JVM execution. Note that this instruction is the instruction at pc_0 in the original code, and $pc_0\ -\ \mathsf{pc}(s)$ in the block xs, e.g. if $pc_0\ =\ \mathsf{pc}(s)$, the instruction would be the first instruction in xs. ■

Lemma 16 (Minstate execution implies full state execution)

$$\begin{aligned}
\forall\ \mathit{CFS}\ s\ f\ \sigma_0\ \sigma_n \tau_0\ \tau_n\ xs.\ & \mathsf{exec\_indep}\ (xs,\ \tau_0)\ =\ \mathsf{Some}\ \tau_n\ \wedge \\
& \mathsf{list\_all\ CFS\_indep}\ xs\ \wedge \\
& \mathsf{list\_all\ not\_branch}\ xs\ \wedge \\
& xp_0\ =\ \mathsf{None}\ \wedge \\
& \mathit{CFS}[s \ldots f]\ =\ xs\ \wedge \\
& \mathsf{pc}(s) \leq \mathsf{pc}(f)\ \wedge \\
& \mathsf{pc}(f) < \mathsf{length\ (get\_code}\ \mathit{CFS}\ s)\ \longrightarrow \\
& \forall\ hp\ frs.\ \mathsf{exec}(\mathit{CFS}, \sigma_0)\ =\ \mathsf{Some}\ \sigma_n
\end{aligned}$$

where

$$\begin{aligned}
\sigma_0 &= (xp_0,\ hp_0,\ (stk_0, loc_0, cn_0, ml_0, \mathsf{pc}(s) + pc_0) : frs_0)\\
\sigma_n &= (xp_n,\ hp_0,\ (stk_n, loc_n, cn_0, ml_0, \mathsf{pc}(s) + pc_n) : frs_n)\\
\tau_0 &= (xp_0, stk_0, loc_0, pc_0)\\
\tau_n &= (xp_n, stk_n, loc_n, pc_n)
\end{aligned}$$

Proof By case analysis of the instruction at $\mathsf{pc}(s)$, followed by simplification with the definition of exec_indep (Definition 11) and the operational semantics of JVM execution. ■

Lemma 17 (Inlist implies inside)

$$\begin{aligned}
\forall\ CFS\ s\ f\ \sigma_0\ \sigma_n\tau_0\ \tau_n\ xs.\ & xs \neq [\,]\ \wedge\ xs = CFS[s \ldots f]\ \wedge\\
& \mathsf{inlist}\ xs\ pc_0\ \wedge\\
& \mathsf{pc}(f)\ <\ (\mathsf{length}\ (\mathsf{get\_code}\ CFS\ s))\\
& \longrightarrow \mathsf{inside}\ \mathsf{pc}(\sigma_0)\ s\ f
\end{aligned}$$

where

$$\begin{aligned}
\sigma_0 &= (xp_0, hp_0, (stk_0, loc_0, cn_0, ml_0, pc_0 + \mathsf{pc}(s)) : frs_0)\\
\tau_0 &= (xp_0, stk_0, loc_0, pc_0)
\end{aligned}$$

Proof From the definitions of inlist (Definition 12), inside (Definition 1), and $CFS[s \ldots f]$ (Definition 5). ■

Lemma 18 (Inside implies inlist)

$$\begin{aligned}
\forall\ CFS\ s\ f\ \sigma_0\ \sigma_n\tau_0\ \tau_n\ xs.\ & xs \neq [\,]\ \wedge\ xs = CFS[s \ldots f]\ \wedge\\
& \mathsf{inside}\ \mathsf{pc}(\sigma_0)\ s\ f\ \wedge\\
& \mathsf{pc}(f)\ <\ (\mathsf{length}\ (\mathsf{get\_code}\ CFS\ s))\\
& \longrightarrow \mathsf{inlist}\ xs\ \tau_0
\end{aligned}$$

where

$$\begin{aligned}
\sigma_0 &= (xp_0, hp_0, (stk_0, loc_0, cn_0, ml_0, pc_0) : frs_0)\\
\tau_0 &= (xp_0, stk_0, loc_0, pc_0 - \mathsf{pc}(s))
\end{aligned}$$

Proof From the definitions of inlist (Definition 12) , inside (Definition 1), and $CFS[s \ldots f]$ (Definition 5). ■

Lemma 19 (Outlist implies outside)

$$\begin{aligned}\forall\ \mathit{CFS}\ s\ f\ \sigma_0\ \sigma_n \tau_0\ \tau_n\ xs.\ & xs \neq [\,]\ \wedge\ xs = \mathit{CFS}[s \ldots f]\ \wedge \\ & \mathsf{outlist}\ xs\ \tau_0 \wedge \\ & \mathsf{pc}(f)\ <\ (\mathsf{length}\ (\mathsf{get\_code}\ \mathit{CFS}\ s)) \\ & \longrightarrow \mathsf{outside}\ \mathsf{pc}(\sigma_0)\ s\ f\end{aligned}$$

where

$$\begin{aligned}\sigma_0 &= (xp_0, hp_0, (stk_0, loc_0, cn_0, ml_0, pc_0 + \mathsf{pc}(s)) : frs_0) \\ \tau_0 &= (xp_0, stk_0, loc_0, pc_0)\end{aligned}$$

PROOF From the definitions of outlist (Definition 13), outside (Definition 2), and $\mathit{CFS}[s \ldots f]$ (Definition 5). ■

Lemma 20 (Outside implies outlist)

$$\begin{aligned}\forall\ \mathit{CFS}\ s\ f\ \sigma_0\ \sigma_n \tau_0\ \tau_n\ xs.\ & \langle \mathit{CFS}, \sigma_0 \rangle \underset{f}{\overset{s}{\longrightarrow}}\ \sigma_n\ \longrightarrow \\ & \mathsf{inside}\ \mathsf{pc}(\sigma_0)\ s\ f\ \wedge \\ & \mathsf{excep\_free}\ xs\ \wedge \\ & \mathit{CFS}[s \ldots f]\ = xs\ \wedge \\ & \mathsf{list\_all}\ \mathsf{CFS\_indep}\ xs\ \wedge \\ & \mathsf{list\_all}\ \mathsf{not\_branch}\ xs\ \wedge \\ & \mathsf{pc}(f)\ <\ (\mathsf{length}\ (\mathsf{get\_code}\ \mathit{CFS}\ s))\ \wedge \\ & \mathsf{outside}\ \mathsf{pc}(\sigma_n)\ s\ f \\ & \longrightarrow \mathsf{outlist}\ xs\ \tau_n\end{aligned}$$

where

$$\begin{aligned}\sigma_0 &= (xp_0, hp_0, (stk_0, loc_0, cn_0, ml_0, pc_0) : frs_0) \\ \sigma_n &= (xp_n, hp_n, (stk_n, loc_n, cn_n, ml_n, pc_n) : frs_0) \\ \tau_n &= (xp_n, stk_n, loc_n, pc_n - \mathsf{pc}(s))\end{aligned}$$

PROOF In addition to the definitions of outlist (Definition 13), outside (Definition 2), and $\mathit{CFS}[s \ldots f]$ (Definition 5), we need the additional assumptions $\langle \mathit{CFS}, \sigma_0 \rangle \underset{f}{\overset{s}{\longrightarrow}} \sigma_n$ and list_all not_branch xs, which allow us to show that $\mathsf{pc}(\sigma_n) = \mathsf{pc}(f) + 1$.

This is necessary since all program counters are defined as natural numbers and, while *xs* may have instructions on the left as part of a larger list of instructions in a classfile, we cannot refer to positions to the left of the head of the list *xs* in isolation. ■

Lemma 21 (Block execution implies sequence execution)

$$\begin{aligned}\forall\ CFS\ s\ f\ \sigma_0\ \sigma_n \tau_0\ \tau_n\ x\ xs.\ \langle CFS, \sigma_0\rangle \xrightarrow[f]{s} \sigma_n \quad \longrightarrow \\ CFS[s \ldots f]\ = xs\ \wedge \\ \mathsf{list\_all}\ \mathsf{CFS\_indep}\ xs\ \wedge \\ \mathsf{list\_all}\ \mathsf{not\_branch}\ xs\ \wedge \\ \mathsf{excep\_free}\ xs \quad \longrightarrow \\ xs\ \vdash\ \tau_0 \rightsquigarrow \tau_n\end{aligned}$$

where

$$\begin{aligned}\sigma_0 &= (xp_0,\ hp_0,\ (stk_0, loc_0, cn_0, ml_0, pc_0) : frs_0)\\ \sigma_n &= (xp_n,\ hp_n,\ (stk_n, loc_n, cn_n, ml_n, pc_n) : frs_n)\\ \tau_0 &= (xp_0, stk_0, loc_0, (pc_0 - \mathsf{pc}(s)))\\ \tau_n &= (xp_n, stk_n, loc_n, (pc_n - \mathsf{pc}(s)))\end{aligned}$$

PROOF Under the condition that the initial states in the block are data-equal and are pointing to the same instruction within the block of instructions, the proof proceeds by rule induction on the assumption $\langle CFS, \sigma_0\rangle \xrightarrow[f]{s} \sigma'$.

Base

The base case deals with the case in which our initial assumption was produced by one application of the **Stop** rule for the block execution relation, giving us the assumption

$$\begin{aligned}&\mathsf{exec}(CFS, \sigma_0) = \mathsf{Some}\ \sigma_n\ \wedge\\ &\mathsf{inside}\ \mathsf{pc}(\sigma_0)\ s\ f\ \wedge\\ &\mathsf{same\_method}\ s\ \sigma_0\ \sigma_n\ f\ \wedge\\ &\mathsf{pc}(f) < \mathsf{length}(\mathsf{get\_code}\ CFS\ s)\ \wedge\\ &\mathsf{outside}\ \mathsf{pc}(\sigma_n)\ s\ f\end{aligned} \tag{3.21}$$

from this assumption, the first production rule for the sequence execution relation **Seq-Stop**, and Lemmas 15, 18 and 20, we are able to show the desired conclusion.

Inductive step

The inductive step deals with the case in which our initial assumption was produced by one application of the second production rule for the block execution relation (**Continue**), giving us

$$\begin{aligned}
&\mathsf{exec}(\mathit{CFS}, \sigma_0) = \mathsf{Some}\ \sigma_1\ \wedge \\
&\mathsf{inside}\ \mathsf{pc}(\sigma_0)\ s\ f\ \wedge \\
&\mathsf{same\_method}\ s\ \sigma_0\ \sigma_1\ f\ \wedge \\
&\mathsf{pc}(f) < \mathsf{length}(\mathsf{get\_code}\ \mathit{CFS}\ s)\ \wedge \\
&\langle \mathit{CFS}, \sigma_1\rangle \xrightarrow[f]{s} \sigma_n
\end{aligned} \tag{3.22}$$

The inductive hypothesis is

$$\begin{aligned}
&\mathit{CFS}[s \ldots f]\ = xs\ \wedge \\
&\mathsf{list\_all}\ \mathsf{CFS\_indep}\ xs\ \wedge \\
&\mathsf{list\_all}\ \mathsf{not\_branch}\ xs\ \wedge \\
&\mathsf{excep\_free}\ xs \quad \longrightarrow \\
&xs\ \vdash\ \tau_1 \leadsto \tau_n
\end{aligned} \tag{3.23}$$

where

$$\begin{aligned}
\sigma_1 &= (xp_1,\ hp_1,\ (stk_1, loc_1, cn_1, ml_1, pc_1) : frs_1) \\
\tau_1 &= (xp_1, stk_1, loc_1, (pc_1 - \mathsf{pc}(s)))
\end{aligned}$$

Thus, from the rule **Seq-Continue** and Lemmas 15 and 18, we are able to show the desired conclusion. ■

Lemma 22 (Sequence execution implies block execution)

$$\begin{aligned}
&\mathit{wedge}\forall\ \mathit{CFS}\ s\ f\ \tau_0\ \tau_n\ x\ xs. \\
&\qquad xs \vdash\ \tau_0 \leadsto \tau_n \quad \longrightarrow \\
&\qquad \mathit{CFS}[s \ldots f]\ = xs\ \wedge \\
&\qquad \mathsf{pc}(f) < \mathsf{length}(\mathsf{get\_code}\ \mathit{CFS}\ s)\ \wedge \\
&\qquad \mathsf{pc}(s)\ \leq\ \mathsf{pc}(f)\ \wedge \\
&\qquad \mathsf{list\_all}\ \mathsf{CFS\_indep}\ xs\ \wedge \\
&\qquad \mathsf{list\_all}\ \mathsf{not\_branch}\ xs\ \wedge \\
&\qquad \mathsf{list\_all}\ \mathsf{excep\_free\_instr}\ xs \quad \longrightarrow \\
&\qquad \forall\ hp\ frs.\ \langle \mathit{CFS}, \sigma_0\rangle \xrightarrow[f]{s} \sigma_n
\end{aligned}$$

where

$$\begin{aligned}
\sigma_0 &= (xp_0,\ hp_0,\ (stk_0, loc_0, cn_0, ml_0, \mathsf{pc}(s) + pc_0) : frs_0) \\
\sigma_n &= (xp_n,\ hp_0,\ (stk_n, loc_n, cn_0, ml_0, \mathsf{pc}(s) + pc_n) : frs_0) \\
\tau_0 &= (xp_0, stk_0, loc_0, \mathsf{pc}(s)) \\
\tau_n &= (xp_n, stk_n, loc_n, pc_n)
\end{aligned}$$

Proof By rule induction on the assumption $xs \vdash \tau_0 \rightsquigarrow \tau_n$.

Base

The base case deals with the case in which our initial assumption was produced by one application of the first production rule for the sequence execution relation (**Seq-Stop**), giving us

$$\begin{aligned}
&\mathsf{exec\_indep}\ (xs, \tau_0) = \mathsf{Some}\ \tau_n \wedge \\
&xs \neq [\,] \wedge \\
&\mathsf{inlist}\ \mathsf{pc}(\tau_0) \wedge \\
&\mathsf{outlist}\ \mathsf{pc}(\tau_n)
\end{aligned} \tag{3.24}$$

Thus, from the **Stop** rule for the block execution relation, and Lemmas 16, 17 and 19 we are able to show the desired conclusion.

Inductive step

The inductive step deals with the case in which our initial assumption was produced by one application of the second production rule for the sequence execution relation (**Seq-Continue**), giving us the assumption

$$\begin{aligned}
&\mathsf{exec\_indep}\ (xs, \tau_0) = \mathsf{Some}\ \tau_1 \wedge \\
&xs \neq [\,] \wedge \\
&\mathsf{inlist}\ \mathsf{pc}(\tau_0) \wedge \\
&xs \vdash \tau_1 \rightsquigarrow \tau_n
\end{aligned} \tag{3.25}$$

The inductive hypothesis is

$$\begin{array}{l}
\mathit{CFS}[s \ldots f] \ = \mathit{xs} \ \wedge \\
\mathsf{pc}(f) < \mathsf{length}(\mathsf{get\_code}\ \mathit{CFS}\ s) \wedge \\
\mathsf{pc}(s) \ \leq \ \mathsf{pc}(f) \wedge \\
\mathsf{list\_all}\ \mathsf{CFS\_indep}\ \mathit{xs} \wedge \\
\mathsf{list\_all}\ \mathsf{not\_branch}\ \mathit{xs} \wedge \\
\mathsf{list\_all}\ \mathsf{excep\_free\_instr}\ \mathit{xs} \ \longrightarrow \\
\forall\ \mathit{hp}\ \mathit{frs}.\ \langle \mathit{CFS}, \sigma_1 \rangle \ \xrightarrow[f]{s}\ \sigma_n
\end{array} \tag{3.26}$$

where

$$\begin{array}{lcl}
\exists\ \mathit{hp_0}\ \mathit{frs_0}.\ \sigma_1 & = & (\mathit{xp_1},\ \mathit{hp_0},\ (\mathit{stk_1}, \mathit{loc_1}, \mathit{cn_0}, \mathit{ml_0}, \mathsf{pc}(s) + \mathit{pc_1}) : \mathit{frs_0}) \\
\exists\ \mathit{hp_0}\ \mathit{frs_0}.\ \sigma_n & = & (\mathit{xp_n},\ \mathit{hp_0},\ (\mathit{stk_n}, \mathit{loc_n}, \mathit{cn_0}, \mathit{ml_0}, \mathsf{pc}(s) + \mathit{pc_n}) : \mathit{frs_0})
\end{array}$$

We instantiate *hp* and *frs* in 3.26 to $\mathit{hp_0}$ and $\mathit{frs_0}$ (as in σ_0) and then, by the **Continue** rule for the sequence execution relation and Lemmas 16 and 17, we are able to show the desired conclusion. ■

Theorem 3 (Block execution deterministic for data-equal states)

$$\begin{array}{l}
\forall\ \mathit{CFS}\ \mathit{CFS}'\ s\ f\ s'\ f'\ \sigma_0\ \sigma_n\ \sigma_0'\ \sigma_n'\ \mathit{xs}. \\
\quad \mathit{CFS}[s \ldots f] \ = \mathit{xs} \ \wedge \ \mathit{CFS}'[s' \ldots f'] \ = \mathit{xs} \\
\quad \mathsf{list\_all}\ \mathsf{CFS\_indep}\ \mathit{xs} \wedge \\
\quad \mathsf{list\_all}\ \mathsf{not\_branch}\ \mathit{xs} \wedge \\
\quad \mathsf{excep\_free}\ \mathit{xs} \wedge \\
\quad \mathit{pc_0} - \mathsf{pc}(s) = \mathit{pc_0'} - \mathsf{pc}(s') \ \wedge \\
\quad \langle \mathit{CFS}, \sigma_0 \rangle \ \xrightarrow[f]{s}\ \sigma_n \ \wedge \\
\quad \langle \mathit{CFS}', \sigma_0' \rangle \ \xrightarrow[f']{s'}\ \sigma_n' \ \wedge \\
\quad \sigma_0 \ \cong \ \sigma_0' \\
\quad \longrightarrow \sigma_n \ \cong \ \sigma_n'
\end{array}$$

where

$$\begin{array}{lcl}
\sigma_0 & = & (\mathit{xp_0}, \mathit{hp_0}, (\mathit{stk_0}, \mathit{loc_0}, \mathit{cn_0}, \mathit{ml_0}, \mathit{pc_0}) : \mathit{frs_0}) \\
\sigma_n & = & (\mathit{xp_n}, \mathit{hp_n}, (\mathit{stk_n}, \mathit{loc_n}, \mathit{cn_n}, \mathit{ml_n}, \mathit{pc_n}) : \mathit{frs_n}) \\
\sigma_0' & = & (\mathit{xp_0'},\ \mathit{hp_0'},\ (\mathit{stk_0'}, \mathit{loc_0'}, \mathit{cn_0'}, \mathit{ml_0'}, \mathit{pc_0'}) : \mathit{frs_0'}) \\
\sigma_n' & = & (\mathit{xp_n'},\ \mathit{hp_n'},\ (\mathit{stk_n'}, \mathit{loc_n'}, \mathit{cn_n'}, \mathit{ml_n'}, \mathit{pc_n'}) : \mathit{frs_n'})
\end{array}$$

Proof From Lemma 21,

$$\langle CFS, \sigma_0 \rangle \xrightarrow[f]{s} \sigma_n \longrightarrow xs \ \vdash \ \tau_0 \rightsquigarrow \tau_n$$

where

$$\begin{aligned} \tau_0 &= (xp_0, stk_0, loc_0, (pc_0 - \mathsf{pc}(s))) \\ \tau_n &= (xp_n, stk_n, loc_n, (pc_n - \mathsf{pc}(s))) \end{aligned} \tag{3.27}$$

and

$$\langle CFS', \sigma'_0 \rangle \xrightarrow[f']{s'} \sigma'_n \longrightarrow xs \ \vdash \ \tau'_0 \rightsquigarrow \tau'_n$$

where

$$\begin{aligned} \tau'_0 &= (xp'_0, stk'_0, loc'_0, (pc'_0 - \mathsf{pc}(s'))) \\ \tau'_n &= (xp'_n, stk'_n, loc'_n, (pc'_n - \mathsf{pc}(s'))) \end{aligned} \tag{3.28}$$

By the fact that the states σ_0 and σ'_0 are data-equal (Definition 10) we now have from (3.27) and (3.28)

$$\tau_0 = \tau'_0 \tag{3.29}$$

This gives us

$$xs \ \vdash \ \tau_0 \rightsquigarrow \tau_n \wedge \ xs \ \vdash \ \tau_0 \rightsquigarrow \tau'_n \tag{3.30}$$

and so by Theorem 2

$$\tau_n = \tau'_n \tag{3.31}$$

which, from the definition of data-equal (10), gives us

$$\sigma_n \cong \sigma'_n \tag{3.32}$$

as required. ∎

Lemma 23 (Extend CFS independent execution to the right)

$$\begin{aligned} \forall\ xs\ ys\ \tau_0\ \tau_n.\ & \mathsf{exec\_indep}\ (xs,\ \tau_0) = \mathsf{Some}\ \tau_n \wedge \\ & \mathsf{inlist}\ xs\ \tau_0\ \wedge\ xs \neq [\,] \\ & \longrightarrow \mathsf{exec\_indep}\ (xs@ys,\ \tau_0) = \tau_n \end{aligned}$$

where

$$\begin{aligned}\tau_0 &= (xp_0, stk_0, loc_0, pc_0)\\ \tau_n &= (xp_n, stk_n, loc_n, pc_n)\end{aligned}$$

PROOF By case analysis of the instruction at pc_0, followed by simplification with the definition of exec_indep (Definition 11) and the operational semantics of JVM execution. ■

Lemma 24 (Extend CFS independent execution to the left)

$$\begin{aligned}\forall xs\ ys\ \tau_0\ \tau_n.\ &\mathsf{exec\_indep}\ (ys,\ \tau_0)\ =\ \mathsf{Some}\ \tau_n\ \wedge\\ &\mathsf{not\_branch}\ ys!pc_0\ \wedge\\ &\mathsf{inlist}\ xs\ \tau_0\ \wedge\ xs\ \neq\ [\,]\ \wedge\ ys\ \neq\ [\,]\\ &\longrightarrow \mathsf{exec\_indep}\ (xs@ys,\ \tau_0)\ =\ \mathsf{Some}\ \tau_n\end{aligned}$$

where

$$\begin{aligned}\tau_0 &= (xp_0, stk_0, loc_0, pc_0 + |xs|)\\ \tau_n &= (xp_n, stk_n, loc_n, pc_n + |xs|)\end{aligned}$$

PROOF By case analysis of the instruction at pc_0, followed by simplification with the definition of exec_indep (Definition 11) and the operational semantics of JVM execution. ■

Lemma 25 (Extend instance of sequence execution relation to left)

$$\begin{aligned}&xs\ \vdash\ (xp_0, stk_0, loc_0, pc_0) \leadsto (xp_n, stk_n, loc_n, pc_n)\ \longrightarrow\\ &\mathsf{list\_all\ CFS\_indep}\ xs\ \longrightarrow\\ &\mathsf{list\_all\ not\_branch}\ xs\ \longrightarrow\\ &ys@xs\ \vdash\ (xp_0, stk_0, loc_0, pc_0\ +\ |ys|) \leadsto (xp_n, stk_n, loc_n, pc_n\ +\ |ys|)\end{aligned}$$

PROOF By rule induction on the sequence execution relation and by the construction rules for the sequence execution relation (**Seq-Stop**, **Seq-Continue**), and Lemma 24. ■

Lemma 26 (Combine two instances of sequence execution relation)

$$\begin{aligned}
& xs \;\vdash\; (xp_0, stk_0, loc_0, pc_0) \leadsto (xp_1, stk_1, loc_1, pc_1) \;\longrightarrow \\
& xs \;\neq\; [\,] \longrightarrow \\
& \mathsf{list\_all\ CFS\_indep}\ xs \;\longrightarrow \\
& \mathsf{list\_all\ not\_branch}\ xs \;\longrightarrow \\
& ys \;\vdash\; (xp_1, stk_1, loc_1, 0) \leadsto (xp_n, stk_n, loc_n, pc_n) \;\longrightarrow \\
& xs@ys \;\vdash\; (xp_0, stk_0, loc_0, pc_0) \leadsto (xp_n, stk_n, loc_n, pc_n \;+\; |ys|)
\end{aligned}$$

PROOF By rule induction on the sequence execution relation, and by the construction rules for the sequence execution relation (**Seq-Stop**, **Seq-Continue**) and Lemma 25, using the fact that since none of the instructions in *xs* are branching instructions, $pc' = |ys|$. This means that in execution of the sequence *xs@ys*, execution of *xs* finishes and leaves the program counter pointing to the first instruction in *ys*. ■

3.6 Conclusions

Having extended the instruction set formalized by Pusch to include arithmetic instructions, and modified the model of branching instructions to facilitate proofs of concrete values, we now have sufficient relations describing execution to develop a programming logic for bytecode described in the next chapter. It has been demonstrated that while a simple execution relation defined by entry and exit from a block of code is enough to base much of our programming logic on, it does have several limiting features.

Firstly, it does not allow us to reason about intermediate states in the execution of a block. Often in the proofs outlined in Chapters 4 and 5 we wish to say something about the final state in the execution of a block. As the value of this state will usually depend on the result of execution of some state between the initial and final states, the execution path relation becomes necessary. Additionally, the idea of deterministic execution of instructions is complicated by the fact that bytecodes are seen not as separate entities, but as part of a classfile. Either we could define a concept of determinism that includes the context of the instruction, or as we have done here retain the accepted definition of determinism, but with a restricted class of instructions, as in the sequence execution relation.

These considerations, and others that will be addressed in the coming chapters, raise the question once more as to whether it is really desirable to reason about code at this level. This is open to debate, but the fact remains that as the current situation for both Java and new platforms such as Microsoft's .NET [31] is based on a stack machine model running bytecode or similar, considering these questions and finding solutions is of definite value.

Chapter 4

A Bytecode Programming Logic

Having defined execution relations for bytecode programs, we now define a pre- and post-condition relation for the execution of such programs. Traditionally, such a relation is defined in terms of the various syntactic patterns of the programming language in question. As bytecode programs are flat, no such patterns are explicit in the code and we must therefore determine what constitutes, for example, a loop or a conditional statement.

It is at this point that we must address again the question raised in Section 3.3: what do we mean by the execution of a sequence of bytecode? None of the relations discussed so far place any particular restraints on the position of the program counter of the initial state within the block. But, while one could potentially begin execution of a sequence of instructions at any of a number of points in the sequence, the final state would be dependent on which instructions in the sequence had actually been executed.

For example, execution of the instructions

```
bipush 5
bipush 4
iadd
bipush 2
```

could lead to a type-safe execution beginning at either the first instruction, or at the last. In the first case, the value of the stack in the final state would be $2 :: 9 :: \mathit{init\_stk}$, in the second it would be $2 :: \mathit{init\_stk}$. Consequently the weakest precondition with respect to a particular condition for any list of instructions will be entirely dependent on which of these instructions are actually executed.

Clearly then, a meaningful weakest precondition definition for bytecode must identify the initial position of the program counter, and it seems reasonable to decide that execution should start at the first instruction in the list. This problem does not arise in standard programming logics as one cannot start execution midway through a command in the language. Even in commands built up inductively from other commands, it is implicit that execution starts at the beginning of the topmost command, and not at one of the inner subcommands.

4.1 A Pre- and Post-Condition Relation for Execution of Bytecode

We write $\{P\}\ xs\ \{Q\}$ to mean that for all classfiles *CFS* containing the instruction sequence xs bounded by the instructions identified by s and f, if the condition P holds in state σ_0 and $\langle \mathit{CFS},\ \sigma_0\rangle \xrightarrow[f]{s} \sigma_n$, then condition Q holds in state σ_n. The definition is given by cases on whether or not xs is empty.

Definition 16 (Pre-/Post-condition relation for bytecode)

$$\{P\}\ [\ ]\ \{Q\} \equiv \forall\ \mathit{CFS}\ \sigma_0\ \sigma_n.\ P(\sigma_0) \longrightarrow Q(\sigma_n) \tag{4.1}$$

$$\begin{aligned}\{P\}\ x\!:\!xs\ \{Q\} = \ & \forall\ \mathit{CFS}\ \sigma_0\ \sigma_n\ s\ f.\ [\langle \mathit{CFS}, \sigma_0\rangle \xrightarrow[f]{s} \sigma_n\ \wedge \\ & \mathit{CFS}[s \ldots f]\ =\ x\!:\!xs\ \wedge\ \mathsf{pc}(\sigma_0) = \mathsf{pc}(s)\ \wedge \\ & P(\sigma_0)] \longrightarrow Q(\sigma_n)\end{aligned} \tag{4.2}$$

Note that logical operations are defined pointwise, i.e. the assertion $\{P\ \wedge\ S\}$ applied to state σ is taken to mean $P\ \sigma\ \wedge\ S\ \sigma$.

4.2 Rules

We present a collection of derived rules for simple bytecode patterns. Proofs for loops and conditional statements will be dealt with later in this chapter and in Chapter 5.

Lemma 27 (Precondition strengthening)

$$(\forall \sigma_0.\ P\ \sigma_0 \longrightarrow R\ \sigma_0)\ \wedge\ \{R\}\ xs\ \{Q\} \Longrightarrow \{P\}\ xs\ \{Q\}$$

Proof From Definition 16 and induction on the list of instructions xs. ∎

Lemma 28 (Postcondition weakening)

$$(\forall \sigma_0.\ R\ \sigma_0 \longrightarrow Q\ \sigma_0)\ \wedge\ \{P\}\ xs\ \{R\} \Longrightarrow \{P\}\ xs\ \{Q\}$$

Proof From Definition 16 and induction on the list of instructions xs. ∎

We now define a block of instructions, execution of which always results in the program counter pointing to the instruction following that block. This describes the idea of sequential execution of blocks of instructions, e.g., x immediately followed by ys.

Definition 17 (Simple Block)

$$\begin{aligned}\mathsf{simple}\ xs\ \equiv\ \forall CFS\ \sigma_0\ \sigma_n\ s\ f.\ xs\ =\ & CFS[s \ldots f]\ \wedge\ \mathsf{inside}\ \sigma_0\ s\ f\ \wedge \\ & \mathsf{exec}(CFS,\ \sigma_0) = \mathsf{Some}\ \sigma_n \longrightarrow \\ & \mathsf{inside}\ \sigma_n\ s\ f\ \vee\ \mathsf{pc}(\sigma_n)\ =\ \mathsf{pc}(f) + 1\end{aligned}$$

Note that a simple block may contain an internal loop provided this does not prevent the block meeting the requirements of the **simple** definition.

Lemma 29 (Splitting a slice of instructions)

$$\begin{aligned}\forall\ a\ b\ xs\ ys\ zs.\ & xs \neq [\,]\ \wedge\ ys\ \neq\ [\,]\ \wedge\ a < b\ \wedge \\ & b < |zs|\ \wedge\ xs@ys\ =\ \mathsf{slice}\ a\ b\ zs \longrightarrow \\ & xs\ =\ \mathsf{slice}\ a\ ((a + |xs|) - 1)\ zs\ \wedge \\ & ys\ =\ \mathsf{slice}\ (a + |xs|)\ b\ zs\end{aligned}$$

PROOF From definition of slice (Definition 6) and standard lemmas for take and drop in the Isabelle distribution. ■

Lemma 30 (Existence of two simple blocks)

$$\begin{aligned}\forall CFS\ \sigma_0\ \sigma_1\ \sigma_n\ s\ f.\ \langle CFS, \sigma_0\rangle \xrightarrow[f]{s}\ \sigma_n \\ \wedge\ CFS\ [s \ldots f]\ =\ xs@ys\ \wedge \\ \mathsf{simple}\ xs\ \wedge\ \mathsf{simple}\ ys \longrightarrow \\ \exists \sigma_1 . \langle CFS, \sigma_0\rangle \xrightarrow[y]{s}\ \sigma_1\ \wedge\ \langle CFS, \sigma_1\rangle \xrightarrow[f]{w}\ \sigma_n\end{aligned}$$

where

$$\begin{aligned} m &= s\{\mathsf{pc} := \mathsf{pc}(s) + |xs| - 1\} \\ n &= s\{\mathsf{pc} := \mathsf{pc}(s) + |xs|\}\end{aligned}$$

PROOF From Lemma 9 we show the existence of an instance of the execution path relation corresponding to the relation $\langle CFS, \sigma_0\rangle \xrightarrow[f]{s} \sigma_n$. We then carry out induction on this relation using one of the standard Isabelle induction theorems converse_trancl_induct, which 'unrolls' the transitive closure relation from the start. This in conjunction with the definition of simple (Definition 17), gives us

$$\begin{aligned}\exists\ \sigma_1 .\ \langle CFS,\ \sigma_0\rangle \overset{s}{\underset{y}{\Longrightarrow}}{}^{+}\ \sigma_1\ \wedge\ \langle CFS,\ \sigma_1\rangle \overset{s}{\underset{f}{\Longrightarrow}}{}^{+}\ \sigma_n \\ \wedge\ \mathsf{pc}(\sigma_1)\ =\ \mathsf{pc}(s) + |xs|\end{aligned} \tag{4.3}$$

Similar treatment of the second conjunct in 4.3 then gives us

$$\langle CFS,\ \sigma_1\rangle \overset{w}{\underset{f}{\Longrightarrow}}{}^{+}\ \sigma_n\ \wedge\ \mathsf{pc}(\sigma_n)\ =\ \mathsf{pc}(f){+}1 \tag{4.4}$$

and from 4.3, 4.4 and Lemma 10 we have the desired result. ■

Theorem 4 (Sequencing Rule)

$$\frac{\begin{array}{c}\{P\}\ xs\ \{R\} \\ \{R\}\ ys\ \{Q\} \\ \mathsf{simple}\ xs \\ \mathsf{simple}\ ys\end{array}}{\{P\}\ xs@ys\ \{Q\}}$$

Proof This follows from Definition 16 and Lemmas 29 and 30. ∎

We now describe the lemmas necessary for the proof of a rule for unconditional branches forwards.

Lemma 31 (Narrowing of block containing a simple block)

$$\begin{aligned}
&\forall\ \sigma_0\ \sigma_n\ CFS\ s\ f.\ \langle CFS, \sigma_0\rangle \xrightarrow[f]{s} \sigma_n\ \wedge \\
&CFS[s \ldots f]\ =\ xs@ys\ \wedge \\
&xs\ \neq\ [\,]\ \wedge\ ys\ \neq\ [\,]\ \wedge \\
&\mathsf{simple}\ ys\ \wedge \\
&\mathsf{pc}(\sigma_0)\ = \mathsf{pc}(s)\ + |xs| + 1 \\
&\longrightarrow \langle CFS, \sigma_0\rangle \xrightarrow[f]{y} \sigma_n
\end{aligned}$$

where

$$y = s\{\mathsf{pc} := \mathsf{pc}(s) + |xs| + 1\}$$

Proof This follows from the fact that the instructions ys make up a simple block (Definition 17), the execution of which can only result in a state where the program counter is pointing inside the block or at the instruction immediately to the right of it. Under these circumstances, once execution reaches the beginning of ys (or if it begins there) the instructions in xs will not be executed again within this block.

By induction on the block execution relation followed by use of its construction rules, we are able to demonstrate the existence of the smaller block as the program counter will always be inside this smaller block during execution. ∎

Lemma 32 (Elimination of unconditional branch forward)

$$\begin{aligned}
&\forall\ \sigma_0\ \sigma_n\ CFS\ s\ f.\ \langle CFS, \sigma_0\rangle \xrightarrow[f]{s} \sigma_n\ \wedge \\
&CFS[s \ldots f]\ =\ [\mathsf{Goto\_fwd}\ (|xs|{+}1)]@xs@ys\ \wedge \\
&xs\ \neq\ [\,]\ \wedge\ ys\ \neq\ [\,]\wedge \\
&\mathsf{simple}\ xs\ \wedge\ \mathsf{simple}\ ys\ \longrightarrow \\
&\exists\ \sigma_1.\ \langle CFS, \sigma_0\rangle \xrightarrow[s]{s} \sigma_1\ \wedge\ \langle CFS, \sigma_1\rangle \xrightarrow[f]{y} \sigma_n
\end{aligned}$$

where

$$y = s\ \{\mathsf{pc} := \mathsf{pc}(s) + 1\}$$

PROOF By induction on the block execution relation, and the operational semantics of the JVM, we are able to show that

$$\begin{aligned} \exists\ \sigma_1.\ \mathsf{exec}(CFS, \sigma_0) &\ =\ \mathsf{Some}\ \sigma_1\ \wedge \\ pc(\sigma_1) &\ =\ \mathsf{pc}(s) + |xs| + 1\ \wedge \\ \langle CFS, \sigma_1\rangle &\ \xrightarrow[f]{s}\ \sigma_n \end{aligned} \tag{4.5}$$

Then by Lemma 31, we can show that this implies the existence of a smaller instance of the block execution relation involving σ_1 and σ_n.

Note that as xs and ys are simple, execution will never return to the unconditional branch forward instruction from the block $xs@ys$. ∎

Theorem 5 (Unconditional branch forward rule)

$$\frac{\begin{array}{l} \{P\}\ [\mathsf{Goto\_fwd}\ |xs| + 1]\ \{R\} \\ \{R\}\ ys\ \{Q\} \\ xs\ \neq\ [\,] \\ ys\ \neq\ [\,] \\ \mathsf{excep\_free}\ [\mathsf{Goto\_fwd}\ |xs| + 1]@(xs@ys) \\ \mathsf{simple}\ xs \\ \mathsf{simple}\ ys \end{array}}{\{P\}\ \ [\mathsf{Goto\_fwd}\ |xs| + 1]@(xs@ys)\ \ \{Q\}}$$

PROOF This follows from Definition 16 and Lemmas 32 and 31.

4.3 A Rule for Loops in Bytecode

Unlike the simple imperative language used in the standard Hoare logic which contains the `while` command, there are no explicit loop constructs in bytecode programs. In order to develop rules for programs containing loops it is therefore necessary to identify the patterns of bytecode instructions that are used to code them. Of course, Java programs may contain loops other than `while` loops—namely `for` loops and `repeat-until` loops. For the purposes of this work, however, we shall restrict our attention to while loops.

Consider the following very simple Java program that repeatedly increments a variable i

```
public class SimpleWhile {
public static void  main(String args[])
  {
    int i=0;
    while (i<5)
      { i++; }
  }
}
```

the corresponding bytecode for this program is

```
   0 bipush 0
   1 istore_1
   2 goto 8
   5 iinc 1 1
   8 iload_1
   9 iconst_5
  10 if_icmplt 5
  13 return
```

where the instructions

```
  0 bipush 0
  1 istore_1
```

initialize the variable i to 0, by pushing the constant value 0 onto the stack and then storing it in variable location 1. The `return` instruction simply returns at the end of the method, and so the loop itself consists of the instructions

```
   2 goto 8
```

```
 5 iinc 1 1
 8 iload_1
 9 iconst_5
10 if_icmplt 5
```

where

```
 5 iinc 1 1
```

is the *body* of the loop, incrementing the value in variable location 1, that is, i. The instructions

```
 8 iload_1
 9 iconst_5
```

load the value of i and the constant value 5 on to the stack, ready for the actual branching instruction

```
10 if_icmplt 5
```

Here, if the second value on the stack is less than 5, the program counter is set to the instruction labelled `5` ready to execute the body of the loop again, otherwise the loop exits. The instruction

```
 2 goto 8
```

is only executed once—at the beginning of the loop—and ensures that the guard condition is tested before the body is executed, thereby making this a `while` loop rather than a `repeat-until` loop.

A diagram showing the outline of this loop can be seen in Figure 4.1, where xs represents the instructions making up the body of the loop, and ys the instructions used to prepare the stack for the conditional branch. A general representation of such a loop may now be written

$$[(\mathsf{UBF}\ |xs|\ +\ 1)]@[xs]@[ys]@[(\mathsf{CBB}\ |xs@ys|)] \tag{4.6}$$

where $\mathsf{UBF}\ |xs|\ +\ 1$ is the unconditional branch forward instruction to the head of ys—a jump of one more than the length of xs, and $\mathsf{CBB}\ |xs@ys|$ is the conditional branch backwards to the start of xs—a jump of the length of $xs@ys$.

While other possible loop forms exist—one is shown in Figure 4.2–we shall discuss only the type described above in this work. Treatment of the alternative forms would, however, be similar.

The notation UBF n, UBB n, CBF n and CBB n for branching instructions is an abbreviation used for clarity of reading. As detailed in Chapter 3, there are a number of branching instructions:

$$\begin{aligned} \mathit{cond\_branch\_fwd} &= \mathsf{Ifnull\_fwd}\ \mathit{nat} \\ &\mid \mathsf{Ifiacmpeq\_fwd}\ \mathit{ins\_type}\ \mathit{nat} \\ &\mid \mathsf{Ificmplt\_fwd}\ \mathit{nat} \end{aligned}$$

$$\begin{aligned} \mathit{cond\_branch\_bwd} &= \mathsf{Ifnull\_bwd}\ \mathit{nat} \\ &\mid \mathsf{Ifiacmpeq\_bwd}\ \mathit{ins\_type}\ \mathit{nat} \\ &\mid \mathsf{Ificmplt\_bwd}\ \mathit{nat} \end{aligned}$$

$$\mathit{uncond\_branch\_fwd} = \mathsf{Goto\_fwd}\ \mathit{nat}$$

$$\mathit{uncond\_branch\_bwd} = \mathsf{Goto\_bwd}\ \mathit{nat}$$

In the case of a rule involving conditional branch instructions, such as the loop rule and the conditional branch forwards rule discussed in Section 4.5.2, it would be necessary to prove the rule for all three cases of the conditional branching instruction in question in order to obtain a general loop rule.

Due to the prohibitive size and complexity of such proofs (an issue discussed in some detail in Sections 6.2 and 7.1), while the loop rule and conditional branch forwards rule are stated generally, only the case relating to the Ificmplt variety has been proved. This refers to a branch taken if the value at the top of the stack is less than the value immediately below it on the stack. Proofs involving the other two conditions would be practically identical.

The while rule in the standard Hoare logic is

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

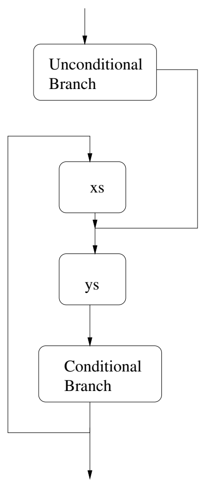

Figure 4.1: Loop structure

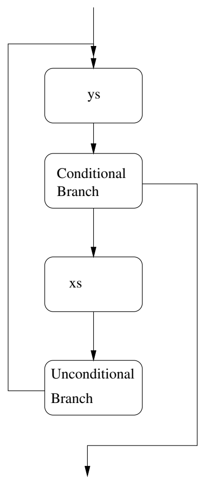

Figure 4.2: Loop structure

where P is an invariant of the loop and S is the loop guard. In a similar rule for the bytecode representation of a while loop it seems obvious that xs in Figure 4.1 corresponds to C (the body of the loop), and that the invariant P does not depend on the language we are dealing with. This leaves the question of what constitutes S, the loop guard, in the bytecode.

The loop guard is not explicit in the bytecode, as it is in the higher level language. But if we consider what role the loop guard plays in the imperative language, it becomes clearer. Evaluation of the loop guard determines whether or not the body of the loop will be executed for each iteration of the loop. In the bytecode, evaluation of the conditional branch determines whether or not the body of the loop is executed for a particular iteration. So this must mean that S is the condition—branch_cond—tested by the conditional branch, and our proposed rule looks something like this

$$\frac{\begin{array}{l}\{P \ \wedge \ \mathsf{branch\_cond}\} \\ \quad xs \\ \{P\}\end{array}}{\begin{array}{l}\{P\} \\ \quad [\mathsf{UBF}\ |xs| + \mathit{1}]@[xs]@[ys]@[\mathsf{CBB}\ |xs@ys|] \\ \{P \ \wedge \ \neg\mathsf{branch\_cond}\}\end{array}}$$

But this is not quite accurate. The conditional branch instructions test certain properties of the value or values at the top of the stack, e.g. ‘Jump if the value at top of the stack is equal to Null’, or ‘Jump if the value at the top of the stack is not equal to zero’. A side effect of the comparison is to pop the values involved in this comparison off the stack, with the result that any predicate involving the top of the stack is meaningful *immediately before execution of the branch instruction.* An example of this is shown in Figure 4.3 where the branching condition is ‘branch if the second value on the stack is less than the top value’. Immediately prior to execution of the branch instruction this condition can be evaluated, but when the branching instruction has been executed the relevant values are no longer on the stack and the statement is no longer sensible. In fact, in the case where popping the values empties the stack, it is undefined.

So a rule stating that a branch condition involving values on the stack holds anywhere other than just before the branch is executed is incorrect. This means that our rule is incorrect, as we require the branch condition to be

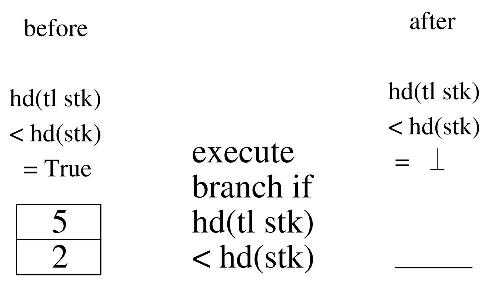

Figure 4.3: State of stack before and after conditional branch

true at the beginning of execution of the body, xs, and to be false at the end of the loop, immediately after execution of the branching instruction.

The solution to this particular problem is to 'wind back' the conditional being tested until we have a condition in terms of actual variables and values rather than items on the stack. We are, in effect, reconstructing the original guard condition present in the Java source code which is concealed in the bytecode instructions. If we look at the bytecode for the loop we can see that the sequence of instructions ys is executed prior to the conditional branch every time through the loop. These instructions 'set up' the stack so that the correct values are available for the comparison. By taking the *weakest precondition* of these instructions and the condition of the branch we are able to determine the actual guard S. Our loop rule is therefore as follows:

$$\frac{\begin{array}{l}\{P \ \wedge\ \mathsf{wp}\ ys\ \mathsf{branch\_cond}\}\\ \quad xs\\ \{P\}\end{array}}{\begin{array}{l}\{P\}\\ \quad [(\mathsf{UBF}\ |xs| + \mathit{1})]@[xs]@[ys]@[(\mathsf{CBB}\ |xs@ys|)]\\ \{P \wedge \neg\mathsf{wp}\ ys\ \mathsf{branch\_cond}\}\end{array}}$$

4.4 Weakest Precondition

The weakest precondition—actually called the weakest liberal precondition in cases of partial correctness—is the least condition necessary to hold at the start of execution of a command that will guarantee that a particular condition holds at the end of execution (assuming termination):

$$\mathsf{wp}\ C\ Q\ \equiv\ (\lambda\, \sigma_0.\, \forall\, \sigma_n.\ (\sigma_0, \sigma_n) \in \{(\sigma_0, \sigma_n).\ \mathsf{eval}(C,\ \sigma_0) = \sigma_n\ \longrightarrow\ Q\ \sigma_n\ \})$$

Our definition of weakest precondition for bytecode programs is:

Definition 18 (Weakest precondition for bytecode)

$$\begin{aligned}\mathsf{wp}\ \ xs\ Q\ \equiv\ (\lambda\ \sigma_0.\ \forall\ \sigma_n.\ (\sigma_0, \sigma_n) \in \{(\sigma_0, \sigma_n).\langle CFS, \sigma_0\rangle \xrightarrow[f]{s} \sigma_n\ \wedge \\ \mathsf{pc}(\sigma_0) = \mathsf{pc}(s)\ \wedge \\ CFS[s \ldots f]\ =\ xs\ \longrightarrow\ Q\ \sigma'\})\end{aligned}$$

Using this, the two defining properties of the weakest precondition can be proved, namely:

Lemma 33 (Weakest precondition is a precondition)

$$xs \neq [\,] \Longrightarrow \{\mathsf{wp}\ xs\ Q\}\ \ xs\ \ \{Q\}$$

PROOF From the definition of weakest precondition for bytecode (Definition 18). ∎

Lemma 34 (Weakest precondition is the weakest possible precondition)

$$\forall P.\ \{P\}\ xs\ \{Q\}\ \ \longrightarrow \forall \sigma_0.\ P\ \sigma_0 \supset\ (\mathsf{wp}\ \ xs\ Q)\ \sigma_0$$

PROOF From the definition of weakest precondition for bytecode (Definition 18). ∎

Several other useful properties may also be shown:

Lemma 35 (Postcondition false implies weakest precondition false)

$$\begin{aligned}\forall\ \sigma_0\ \sigma_n.\ \langle CFS, \sigma_0\rangle \xrightarrow[f]{s} \sigma_n\ \wedge\ xs\ =\ CFS[s \ldots f]\ \wedge\ \neg Q\ \sigma_n \\ \longrightarrow\ \neg(wp\ xs\ Q)\ \sigma_0\end{aligned}$$

PROOF Suppose it is possible to execute xs starting in state σ_0 and finishing in state σ_n, where the condition Q is false in σ_n. Then the weakest precondition did not hold in σ_0. This is, of course, the contrapositive of Lemma 34. ∎

4.4.1 Calculating the Weakest Precondition

As is the case with conventional Hoare logics, to calculate the value of the weakest precondition for a sequence of instructions we start with the desired postcondition and work backwards. In this way we can determine what condition must have held *before* an instruction is executed in order to ensure the postcondition holds after execution. This condition is in fact the weakest precondition of that instruction with respect to the postcondition. This operation is carried out incrementally over each instruction in the list.

To illustrate this, we might calculate the weakest precondition with respect to the branching condition of part of our example loop program in Section 4.3:

```
public class SimpleWhile {
public static void  main(String args[])
  {
    int i=0;
    while (i<5)
      { i++; }
  }
}
```

with corresponding bytecode

```
   0 bipush 0
   1 istore 1
   2 goto 8
   5 iinc 1 1
   8 iload 1
   9 bipush 5
  10 if_icmplt 5
  13 return
```

We wish to calculate the weakest precondition of the instructions

```
  8 iload 1
  9 bipush 5
```

with respect to the branch condition, which is ‘is the second value on the stack less than the value at the top of the stack?’, i.e.

$$\mathsf{hd}\ (\mathsf{tl}\ stk) < \mathsf{hd}\ stk$$

the calculation of which, using the hoisting technique, is as follows. Taking the desired postcondition

$$\mathsf{hd}\ (\mathsf{tl}\ stk) < \mathsf{hd}\ stk$$

we calculate the weakest precondition with respect to the second instruction in our list

```
bipush 5
```

which pushes the value 5 onto the stack, giving us the condition

$$\mathsf{hd}\ (\mathsf{tl}\ (5 : stk)) < \mathsf{hd}\ (5 : stk)$$

we now calculate the weakest precondition with respect to this new postcondition of the first instruction in the list

```
iload 1
```

which pushes the value stored in local variable 1 onto the stack, resulting in the condition

$$\mathsf{hd}\ (\mathsf{tl}\ (5 : (lv1 : stk))) < \mathsf{hd}(5 : (lv1 : stk))$$

When simplified, the weakest precondition is therefore

$$lv1\ < 5$$

As the value of i is stored in local variable 1, as can be seen from the initialisation instructions at the start of the program

```
0 bipush 0
1 istore 1
```

the weakest precondition is $i < 5$: the loop guard of the original Java program.

In order to carry out this operation we need to know the weakest precondition of any individual instruction. Below we give proofs of two instructions: Bipush i is a classfile independent instruction that pushes the integer value i onto the stack, *ins_type* IAastore i is a classfile dependent instruction that stores the value i in an array.

Definition 19 (Non-terminating state)

$$\mathsf{not\_term\_state} \;=\; \lambda\ (xp, hp, frs).\ xp \;=\; \mathsf{None} \wedge frs \neq [\,]$$

Definition 20 (Insert load and store arguments)

$$\begin{aligned}\mathsf{putLASargs} \;=\; & \lambda(xp, hp, frs)\ (stk', loc', pc').\ \mathsf{let}\ (stk, loc, cn, ml, pc) \;=\\ & \mathsf{hd}\ frs\ \mathsf{in}\ (xp, hp, (stk', loc', cn, ml, pc') : (\mathsf{tl}\ frs))\end{aligned}$$

Definition 21 (Extract first load and store argument)

$$\mathsf{getlASarg1} \;=\; \lambda(xp, hp, frs).\ \mathsf{let}\ (stk, loc, cn, ml, pc) \;=\; \mathsf{hd}\ frs\ \mathsf{in}\ stk$$

Definition 22 (Extract second load and store argument)

$$\mathsf{getLASarg2} \;=\; \lambda(xp, hp, frs).\ \mathsf{let}\ (stk, loc, cn, ml, pc) \;=\; \mathsf{hd}\ frs\ \mathsf{in}\ loc$$

Definition 23 (Extract third load and store argument)

$$\mathsf{getLASarg3} \;=\; \lambda(xp, hp, frs).\ \mathsf{let}\ (stk, loc, cn, ml, pc) \;=\; \mathsf{hd}\ frs\ \mathsf{in}\ pc$$

Lemma 36 (Block execution of Bipush implies exec_las Bipush)

$$\begin{aligned}\forall CFS\ s\ \sigma_0\ \sigma_n\ i.\ CFS!\, s \;=\; & (\mathsf{Bipush}\ i) \wedge \mathsf{xp}(\sigma_0) \;=\; \mathsf{None} \wedge\\ & \langle CFS, \sigma_0 \rangle \xrightarrow[s]{s} \sigma_n \longrightarrow\\ & \sigma_n = (\mathsf{putLASargs}\ \sigma_0\ (\mathsf{exec\_las}\ (\mathsf{Bipush}\ i)\\ & \quad (\mathsf{getLASarg1}\ \sigma_0)\ (\mathsf{getLASarg2}\ \sigma_0)\ (\mathsf{getLASarg3}\ \sigma_0)))\end{aligned}$$

Proof By Lemma 8 and the operational semantics of the JVM. ■

Lemma 37 (exec_las Bipush implies block execution of Bipush)

$$\begin{aligned}\forall CFS\ s\ \sigma_0\ \sigma_n\ i.\ CFS!\,s\ =\ & (\mathsf{Bipush}\ i)\ \wedge\ \mathsf{xp}\ (\sigma_0)\ =\ \mathsf{None}\ \wedge\\ & \mathsf{pc}\ (s) < \mathsf{length}(\mathsf{get\_code}\ \ CFS\ \ s)\ \wedge\\ & \sigma_n = (\mathsf{putLASargs}\ \sigma_0\ (\mathsf{exec\_las}\ (\mathsf{Bipush}\ i)\\ & \qquad (\mathsf{getLASarg1}\ \sigma_0)\ (\mathsf{getLASarg2}\ \sigma_0)\ (\mathsf{getLASarg3}\ \sigma_0)))\\ & \longrightarrow \langle CFS, \sigma_0\rangle \xrightarrow[s]{s}\ \sigma_n\end{aligned}$$

Proof By the rules for construction of the block execution relation and the operational semantics of the JVM. ■

Lemma 38 (Existence of set of classfiles containing Bipush)

$$\exists\ CFS.\ CFS[s\ldots s]\ =\ [\mathsf{Bipush}\ i]$$

Proof We show that such a set of classfiles exists by constructing an instance of Pusch's formalisation of a classfile, substituting this for the existentially quantified variable, and simplifying with Isabelle's automatic tactics. ■

Lemma 39 (Weakest precondition of bipush instruction)

$$\begin{aligned}\mathsf{wp}\ [\mathsf{Bipush}\ i]\ Q = \ & (\lambda\ \sigma_0.\ (\mathsf{not\_term\_state}\ \sigma_0)\ \longrightarrow\\ & Q\ (\mathsf{putLASargs}\ \sigma_0\ (\mathsf{exec\_las}\ (\mathsf{Bipush}\ i)\\ & \quad (\mathsf{getLASarg1}\ \sigma_0)\ (\mathsf{getLASarg2}\ \sigma_0)\ (\mathsf{getLASarg3}\ \sigma_0))))\end{aligned}$$

Proof In order to prove equality, we prove implication in both directions. This involves Lemmas 36 and 37. ■

Definition 24 (Insert manipulate array arguments)

$$\begin{aligned}\mathsf{putMAargs}\ =\ & \lambda(xp, hp, frs)\ (stk', loc', pc').\ \mathsf{let}\ (stk, loc, cn, ml, pc)\ =\\ & \mathsf{hd}\ frs\ \mathsf{in}\ (xp, hp, (stk', loc', cn, ml, pc') : (\mathsf{tl}\ frs))\end{aligned}$$

Definition 25 (Extract first manipulate array argument)

$$\mathsf{getMAarg1}\ =\ \lambda(xp, hp, frs).\ hp$$

Definition 26 (Extract second manipulate array argument)

$$\mathsf{getMAarg2} \;=\; \lambda(xp, hp, frs).\ \mathsf{let}\ (stk, loc, cn, ml, pc) \;=\; \mathsf{hd}\ frs\ \mathsf{in}\ stk$$

Definition 27 (Extract third manipulate array argument)

$$\mathsf{getMAarg3} \;=\; \lambda(xp, hp, frs).\ \mathsf{let}\ (stk, loc, cn, ml, pc) \;=\; \mathsf{hd}\ frs\ \mathsf{in}\ pc$$

Lemma 40 (Weakest precondition of IAastore)

$$\begin{array}{l}
\mathsf{excep\_free}\ [\ (\mathit{ins\_type}\ \mathsf{IAastore}\ i)]\ \Longrightarrow \\
\mathsf{wp}\ [(\mathit{ins\_type}\ \mathsf{IAastore}\ i)]\ = \\
(\lambda\ \sigma_0.\ \forall\ \mathit{CFS}.\ (\mathsf{not\_term\_state}\ \sigma_0)\ \longrightarrow \mathit{CFS}!\,\sigma_0\ =\ \mathsf{IAastore} \\
\ \longrightarrow Q\ (\mathsf{putMAargs}\ \ \sigma_0\ (\mathsf{exec\_ma}\ (\mathit{ins\_type}\ \mathsf{IAastore}\ i)\ \mathit{CFS} \\
\qquad\qquad (\mathsf{getMAarg1}\ \ \sigma_0)\ (\mathsf{getMAarg2}\ \sigma_0)\ (\mathsf{getLASarg3}\ \sigma_0))))
\end{array}$$

PROOF Similar to Lemma 39. ∎

We now return to the original aim of this section: a method of calculating the weakest precondition of a sequence of instructions. The technique described at the beginning of this section is based on the equality: $\mathsf{wp}\ (x{:}xs)\ (\mathsf{assert}\ Q)\ =\ (\mathsf{wp}\ [x]\ (\mathsf{wp}\ xs\ (\mathsf{assert}\ Q)))$.

To prove this we rely on the assumption—correct in the case of most standard high level languages—that execution proceeds in a linear manner, moving through the code from left to right one command at a time. In the case of bytecode instructions this may not be the case, and so we must attach another predicate, linear, to instruction sequences the weakest precondition of which we wish to be able to calculate. This is defined as

Definition 28 (Linear code)

$$\mathsf{linear}\ xs\ \equiv\ \mathsf{list\_all\ not\_branch}\ xs\ \wedge\ \mathsf{list\_all\ CFS\_independent}\ xs \qquad (4.7)$$

It is necessary that all the instructions are classfile independent and non-branching. This is due to the fact that the proof hinges on the idea of the deterministic execution of a list of instructions. As discussed in Section 3.5.3, in order to obtain proofs involving the standard idea of determinism (i.e. for a list of instructions independent of context) we must restrict the instructions involved to those which are classfile independent.

The function (assert) provides a way of applying a predicate that is concerned solely with the operand stack, *stk*, and local variables, *loc*, of a method frame to a complete state consisting of an exception option, heap, and frame stack. This is necessary as we want to restrict the discussion to data-equal states, and our present definition of data-equality involves only the stack and local variables.

Definition 29 (Assert)

$$\mathsf{assert}\ Q \ \equiv\ (\lambda(xp, hp, frs).\ (\mathsf{let}\ (stk, loc, cn, ml, pc)\ =\ (\mathsf{hd}\ frs)\ \mathsf{in}\ Q(stk, loc)))$$

Lemma 41 (Decompose block)

$$\begin{aligned}\forall CFS\ \sigma_0\ \sigma_n\ s\ f\ m\ x\ xs.\ &\langle CFS, \sigma_0\rangle \xrightarrow[f]{s} \sigma_n\ \wedge\\ &CFS\ [s \ldots f]\ =\ x : xs\ \wedge\\ &\mathsf{linear}\ (x : xs)\ \wedge\\ &\mathsf{pc}(\sigma_0) = \mathsf{pc}(s) \longrightarrow\\ &\exists \sigma_1. \langle CFS, \sigma_0\rangle \xrightarrow[s]{s} \sigma_1\ \wedge\ \langle CFS, \sigma_1\rangle \xrightarrow[f]{y} \sigma_n\end{aligned}$$

where

$$y = s\{\mathsf{pc} := \mathsf{pc}(s){+}1\}$$

PROOF This follows by case analysis of block execution relation. We know from the definition of linear that the instruction at $\mathsf{pc}(\sigma_0)$ is not a branching instruction. This means that execution of the instruction will result in a state whose program counter is equal to $\mathsf{pc}(s)+1$. As none of the other instructions in the block are branches, execution of the instructions from $\mathsf{pc}(s)\ +\ 1$ to $\mathsf{pc}(f)$ will continue without returning to the instruction at $\mathsf{pc}(s)$. Hence, by application of the rules for the block execution relation we can show the existence of the two blocks as required. ∎

As we wish to prove that assert Q holds in the state resulting from execution of the instructions $x\!:\!xs$, we must show that the execution of x in one classfile, followed by xs from another classfile is equivalent in effect to executing $x\!:\!xs$ from a single classfile.

Lemma 42 (Two adjoining block executions imply a sequence)

$$\begin{aligned}
\forall \mathit{CFS}\ \sigma_0\ \sigma_1\ \sigma_n\ s\ f\ m\ x\ xs.\ & \textsf{list\_all not\_branch}\ (x\!:\!xs)\ \wedge \\
& \textsf{list\_all CFS\_independent}\ (x\!:\!xs)\ \wedge \\
& \textsf{excep\_free}\ (x\!:\!xs)\ \wedge \\
& \langle \mathit{CFS}, \sigma_0\rangle \xrightarrow[f]{s} \sigma_1\ \wedge\ \mathit{CFS}\ [s \ldots f]\ =\ [x]\ \wedge \\
& \langle \mathit{CFS}', \sigma_1\rangle \xrightarrow[f']{s'} \sigma_n\ \wedge\ \mathit{CFS}'\ [s' \ldots f']\ =\ xs \\
& \longrightarrow \tau_0\ \tau_n.\ (x\!:\!xs)\ \vdash\ \tau_0 \rightsquigarrow \tau_n
\end{aligned} \tag{4.8}$$

where

$$\begin{aligned}
\sigma_0 &= (xp_0,\ hp_0,\ (stk_0, loc_0, cn_0, ml_0, pc_0) : frs_0) \\
\sigma_1 &= (xp_1,\ hp_0,\ (stk_1, loc_1, cn_0, ml_0, pc_1) : frs_0) \\
\sigma_n &= (xp_n,\ hp_0,\ (stk_n, loc_n, cn_0, ml_0, pc_n) : frs_0) \\
\tau_0 &= (xp_0, stk_0, loc_0, pc_0 - \textsf{pc}(s)) \\
\tau_n &= (xp_n, stk_n, loc_n, pc_n - \textsf{pc}(s))
\end{aligned}$$

PROOF We show that the two instances of the block execution relation for x and xs and σ_0 and σ_n admit an instance of the sequence execution relation involving $x\!:\!xs$ and the corresponding minstates τ and τ'. This follows from Lemma 21 and Lemma 26, together with the fact that there exists a classfile containing the sequence $x\!:\!xs$ (proof similar to that of Lemma 38). ■

Lemma 43 (Weakest precondition of a list of instructions)

$$\begin{aligned}
& \textsf{excep\_free}\ [x]\ \wedge \\
& \textsf{list\_all not\_branch}\ (x\!:\!xs)\ \wedge \\
& \textsf{list\_all CFS\_independent}\ (x\!:\!xs)\ \wedge \\
& \textsf{excep\_free}\ (x\!:\!xs)\ \wedge \\
xs \neq\ [\,] & \\
& \Longrightarrow\ \textsf{wp}\ (x\!:\!xs)\ (\textsf{assert}\ Q)\ =\ (\textsf{wp}\ [x]\ (\textsf{wp}\ xs\ (\textsf{assert}\ Q)))
\end{aligned}$$

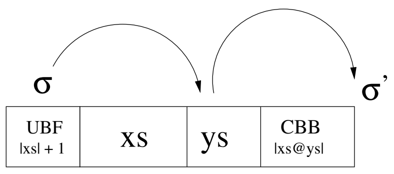

Figure 4.4: States before and after evaluation of guard to false

PROOF Once again, we show equality by proving the implication in each direction. After rewriting with the definition of weakest precondition (Definition 18), the right-to-left version of the equality can be proved using Lemma 42, followed by Lemma 22 to show that the existence of this instance of the sequence execution relation implies the existence of an equivalent instance of the block execution relation. The left-to-right part of the equality can be shown using Lemma 41. ■

4.5 Data-equality and Loops

One major difference between the execution of a while loop in an imperative language and a loop sequence in the bytecode is the effect of executing the 'structure' of the loop. As discussed in Section 9 a general pattern for loops in bytecode is

$$[(\mathsf{UBF}\ |xs|\ +\ 1)]@[xs]@[ys]@[(\mathsf{CBB}\ |xs@ys|)] \tag{4.9}$$

where the instructions xs represent the loop body, and the instructions $(\mathsf{UBF}\ |xs|\ +\ 1)$, ys, and $(\mathsf{CBB}\ |xs@ys|)$ constitute the structual parts of the loop.

In an imperative language, the rules for execution state that executing a while statement in an initial state in which the loop guard is false results in an unchanged state. In the bytecode, execution of a loop sequence in which the guard is false in the initial state results in a different state, as evaluating the sequences which constitute the structure of the loop means the value of the program counter will have changed. This is illustrated in Figure 4.4.

Similarly, with the situation where the body of the loop *is* executed, if a while statement is executed in state σ_0 in which the loop guard is true, we can talk of executing the body of the loop in the same state—evaluation of the loop guard does not affect the state. Again, this is not the case in the bytecode sequence. This is illustrated in Figure 4.5.

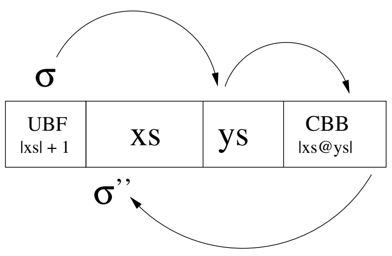

Figure 4.5: States before and after evaluation of guard to true

At first glance, this may seem likely to add to the complexity of a proof of the bytecode rule. But closer inspection of the effect of executing the structure of a bytecode loop sequence, shows that the only element of the state affected (assuming the instructions concerned satisfy certain constraints) is the program counter. This means that once again we can use the idea of *data-equality*, discussed in the previous chapter. As the branch condition does not mention the program counter, and assuming that the loop invariant does not either, data-equal states in the bytecode execution can take the place of equal states in the source code execution.

4.5.1 Data-equality and the Weakest Precondition

Returning to our postulated While Rule

$$\frac{\begin{array}{l}\{P \ \wedge \ \mathsf{wp}\ ys\ \mathsf{branch\_cond}\}\\ \quad xs\\ \{P\}\end{array}}{\begin{array}{l}\{P\}\\ \quad [(\mathsf{UBF}\ |xs| + \mathit{1})]@[xs]@[ys]@[(\mathsf{CBB}\ |xs@ys|)]\\ \{P \wedge \neg\mathsf{wp}\ ys\ \mathsf{branch\_cond}\}\end{array}}$$

we see that we need $\mathsf{wp}\ ys\ \mathsf{branch\_cond}$ to be false in state σ_n, one step **after** the execution of ys is complete. But wp is a relation which deals specifically with the state σ_0 immediately **prior** to execution of ys. In fact, even if the two states in question are data-equal—as they will be in a well-formed loop—it is not the case that

$$\mathsf{wp}\ \ ys\ \mathsf{branch\_cond}\ \sigma_0 \quad \longrightarrow \quad \mathsf{wp}\ \ ys\ \mathsf{branch\_cond}\ \sigma_n \tag{4.10}$$

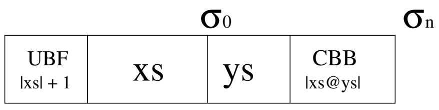

Figure 4.6: States for which weakest precondition must evaluate to true or false

This is because the definition of wp depends on the block execution relation which is expressed in terms of the position of the program counter in a state relative to the class file. Unlike higher level languages, a state is inextricably bound to the instruction indicated by its program counter. This means that despite every other element of the two states σ_0 and σ_n being equal, wp ys branch_cond does not hold in σ_n as its program counter value is outside the block containing ys. The position of states σ_0 and σ_n with respect to ys is illustrated in Figure 4.6. Equally, wp ys branch_cond will not hold in the state in which execution of the loop begins, because its program counter will be pointing to the unconditional branch instruction.

The above definition of weakest precondition for bytecode is clearly not suitable for the purposes of our While Rule, and we must modify it to remove the dependence on the position of the program counter. One solution is to define the weakest precondition in terms of the sequence execution relation (Section 3.5.3) rather than the block execution relation. This effectively transforms the definition into that of the conventional definition of weakest precondition, where only the 'non-positional' parts of a state are relevant. We define the **sequence weakest precondition** as:

Definition 30 (Sequence weakest precondition)

$$\mathsf{sequence\_wp}\ \ xs\ Q\ \equiv\ \lambda\,\sigma_0.\ \forall\,\tau_n.\ \mathsf{not\_term\_state}\ \sigma_0\ \wedge$$
$$xs\ \vdash\ \tau_0\ \leadsto\ \tau_n \longrightarrow\ (\mathsf{assert}\ Q\ \sigma_n)$$

where

$$\begin{aligned}
\sigma_0 &= (xp_0, hp_0, ((stk_0, loc_0, cn_0, ml_0, pc_0) : frs_0))\\
\tau_0 &= (xp_0, stk_0, loc_0, 0)\\
\tau_n &= (xp_n, stk_n, loc_n, pc_n)\\
\sigma_n &= (xp_n, hp_0, ((stk_n, loc_n, cn_0, ml_0, pc_0\ +\ pc_n) : frs_0))
\end{aligned}$$

This new definition does not replace our previous definition (Definition 18) since the sequence weakest precondition can only be used to reason about classfile independent instructions (from the definition of the sequence execution relation, 3.5.3). In order to reason about programs including non-classfile independent instructions we will need the original definition. However, since the instructions contained in a well-formed while loop are assumed to be classfile independent, we are able to use the sequence weakest precondition.

Once again, the two defining properties of the weakest precondition were proved for sequence weakest precondition:

Lemma 44 (swp is a precondition)

$$\begin{array}{l} xs \neq [\,] \wedge \text{ list\_all not\_branch } xs \wedge \\ \text{list\_all CFS\_independent } xs \wedge \\ \text{excep\_free } xs \\ \Longrightarrow \{\text{sequence\_wp } xs\ Q\}\ \ xs\ \ \{\text{assert } Q\} \end{array}$$

Proof From the definition of sequence weakest precondition (Definition 30) and Lemma 21. ■

Lemma 45 (swp is the weakest possible precondition)

$$\begin{array}{l} CFS[s \ldots f] = xs \wedge \text{pc}(f) < \text{length}(\text{get\_code } CFS\ s) \wedge \\ \text{pc}(\ s) \leq \text{pc}(f) \wedge xs \neq [\,] \wedge \text{ list\_all not\_branch } xs \wedge \\ \text{list\_all CFS\_independent } xs \wedge \\ \text{list\_all excep\_free\_instr } xs \wedge \\ \{\text{assert } P\}\ \ xs\ \ \{\text{assert } Q\} \\ \Longrightarrow \forall \sigma_0.\ (\text{assert } P\ \sigma_0) \longrightarrow (\text{sequence\_wp } xs\ Q) \end{array}$$

Proof From the definition of sequence weakest precondition (Definition 30) and Lemma 22. As the pre- and post-condition relation (Definition 16) is defined in terms of the block execution relation, it is necessary to include more information in the assumptions of the theorems involving both the pre- and post-condition relation and sequence_wp than in those concerned only with only sequence_wp.

As for the block execution definition of weakest precondition, the following lemmas were proved for the sequence weakest precondition:

Lemma 46 (Condition false implies swp false)

$$\begin{aligned}\forall\ \sigma_0\ \sigma_n\ s\ f\ \mathit{CFS}\ xs\ Q.\ & xs \neq [\,] \wedge \textsf{list\_all not\_branch}\ xs \wedge \\ & \textsf{list\_all CFS\_independent}\ xs\ \wedge \\ & \textsf{excep\_free}\ xs\ \wedge\ \langle \mathit{CFS}, \sigma_0\rangle \xrightarrow[f]{s} \sigma_n\ \wedge \\ & \neg(\textsf{assert}\ Q)\sigma_n \\ & \longrightarrow\ \neg\ (\textsf{sequence\_wp}\ xs\ Q)\ \sigma_0\end{aligned}$$

PROOF If it is possible to execute xs starting in state σ_0 and finishing in state σ_n, where the condition Q is false in σ_n, the sequence weakest precondition does not hold in σ_0. This is the contrapositive of Lemma 44 and can be understood intuitively as the consequence of Lemma 45. ■

Lemma 47 (swp false implies condition false)

$$xs\ \vdash\ \tau_0 \leadsto \tau_n\ \wedge\ \neg(\textsf{sequence\_wp}\ xs\ Q)\ \sigma_0 \Longrightarrow\ \neg(\textsf{assert}\ Q)\ \sigma_n$$

where

$$\begin{aligned}\sigma_0 &= (xp_0, hp_0, ((stk_0, loc_0, cn_0, ml_0, pc_0) : frs_0))\\ \tau_0 &= (xp_0, stk_0, loc_0, 0)\\ \tau_n &= (xp_n, stk_n, loc_n, pc_n)\\ \sigma_n &= (xp_n, hp_0, ((stk_n, loc_n, cn_0, ml_0, pc_0\ +\ pc_n) : frs_0))\end{aligned}$$

PROOF If the relevant parts of states σ_0 and σ_n are in the sequence execution relation, and sequence_wp xs Q does not hold in σ_0, then assert Q will not hold in σ_n. The result follows from the definition of sequence weakest precondition (Definition 30). ■

Lemma 48 (Condition holds implies swp held)

$$xs\ \vdash\ \tau_0 \leadsto \tau_n\ \wedge\ (\textsf{assert}\ Q)\ \sigma_n \Longrightarrow (\textsf{sequence\_wp}\ xs\ Q)\ \sigma_0$$

where

$$\begin{aligned}\sigma_0 &= (xp_0, hp_0, ((stk_0, loc_0, cn_0, ml_0, pc_0) : frs_0))\\ \tau_0 &= (xp_0, stk_0, loc_0, 0)\\ \tau_n &= (xp_n, stk_n, loc_n, pc_n)\\ \sigma_n &= (xp_n, hp_0, ((stk_n, loc_n, cn_0, ml_0, pc_0\ +\ pc_n) : frs_0))\end{aligned}$$

Proof If the relevant parts of states σ_0 and σ_n are in the sequence execution relation, and assert Q holds in σ_n, then sequence_wp xs Q must have held in σ_0. The result follows from the definition of sequence weakest precondition (Definition 30). ■

Lemma 49 (Relation of wp to swp)

$$\begin{aligned}&\mathsf{linear}\ xs\ \wedge\ \mathsf{excep\_free}\ xs\ \wedge\\ &\mathsf{list\_all\ excep\_free\_instr}\ xs\ \wedge\ xs\ \neq\ [\,]\\ &\Longrightarrow \mathsf{wp}\ xs\ (\mathsf{assert}\ Q)\ =\ \mathsf{sequence\_wp}\ xs\ Q\end{aligned}$$

Proof This is similar to the the proof of Lemma 43, and again we show equality by proving implication in both directions. In this case, however, the result follows simply from Lemma 21 and Lemma 22, along with a proof that there exists a classfile containing the sequence xs (as in Lemma 41). ■

Lemma 50 (swp preserved by data-equality)

$$(\mathsf{sequence\_wp}\ xs\ Q)\ \sigma_0\ \wedge\ \sigma_0\ \cong\ \sigma_n \Longrightarrow\ (\mathsf{sequence\_wp}\ xs\ Q)\ \sigma_n$$

Proof This follows from the definition of sequence weakest precondition (Definition 30), assert (Definition 29), and data-equality (Definition 10). ■

Lemma 51 (Assert preserved by data-equality)

$$(\mathsf{assert}\ Q)\ \sigma_0\ \wedge\ \sigma_0\ \cong\ \sigma_n \Longrightarrow\ (\mathsf{assert}\ Q)\ \sigma_n$$

Proof From the definitions of assert (Definition 29) and data-equality (Definition 10). ■

We are now able to state the final version of the While Rule for bytecode. As we have seen, the role in the conventional Hoare logic of the loop guard S is played here by the sequence weakest precondition of the instructions ys with respect to the branch condition. From its definition, (Definition 30), we know that the sequence weakest precondition is only meaningful in the context of a non-terminating initial state. So, in order to ensure that the actual value of sequence_wp ys branch_cond can be calculated—essential when actually using the logic to prove properties of programs—we add the predicate not_term_state to the postcondition of the while rule.

The predicate well_formed_loop is defined as

Definition 31 (Well-formed loop)

$$\begin{aligned}\mathsf{well\_formed\_loop}\ zs \equiv\ & \exists xs\ ys.\ CFS[s \ldots f]\ =\ zs\ \wedge \\ & zs\ =\ [(UBF|xs| + 1)]@[xs]@[ys]@[(CBB\ |xs@ys|)]\ \wedge \\ & xs \neq [\,]\ \wedge\ ys \neq [\,]\wedge \\ & \mathsf{simple}\ xs\ \wedge\ \mathsf{simple}\ ys\ \wedge \\ & \mathsf{ref\_trans\_2}\ ys\ \wedge\ \mathsf{linear}\ ys\end{aligned}$$

Note that the above definition includes the requirement that the instructions *ys* should be *referentially transparent*, denoted by the term ref_trans_2 *ys*. The execution of a sequence of referentially transparent instructions affects only the stack and the program counter. For a list of instructions *ys*, the execution of which places two values on the stack, the definition is

Definition 32 (Referential transparency of a list of instructions)

$$\begin{aligned}\mathsf{ref\_trans\_2}\ ys\ \equiv\ & \forall\ CFS\ \sigma_0\ \sigma_n\ s\ f.\ ys =\ CFS[s \ldots f]\ \wedge \\ & \mathsf{linear}\ ys\ \wedge\ \mathsf{pc}(\sigma_0)\ =\ \mathsf{pc}(s)\ \wedge \\ & \langle CFS, \sigma_0\rangle \xrightarrow[f]{s}\ \sigma_n\ \longrightarrow \\ & xp_0\ =\ xp_n\ \wedge\ hp_0\ =\ hp_n\ \wedge \\ & \exists\ a\ b.stk_n\ =\ a : (b : (stk_0))\ \wedge\ loc_0\ =\ loc_n\ \wedge \\ & cn_0\ =\ cn_n\ \wedge\ ml_0\ = ml_n\ \wedge\ frs_0\ = frs_n\end{aligned}$$

where

$$\begin{aligned}\sigma_0 &= (xp_0, hp_0, ((stk_0, loc_0, cn_0, ml_0, pc_0) : frs_0)) \\ \sigma_n &= (xp_n, hp_n, ((stk_n, loc_n, cn_n, ml_n, pc_n) : frs_n))\end{aligned}$$

This, of course, states that execution of the instructions *ys* results in two values being placed on the stack. If, for example, *ys* = [Pop, Pop] this would not be the case. Certainly, any well-formed program involving an Ificmplt or Ifiacmpeq branch *should* place two values to be compared on the stack immediately before the branch. However, this would not be the case with an Ifnull branch (where only one value would be placed on the stack).

Thus the definitions of referential transparency and of a well-formed loop would have to be modified to reflect these situations in any future version of the programming logic. For the purposes of the proofs described in this dissertation, however, the current definitions are adequate.

Our final version of the While Rule is as follows, and a proof of the soundness of this rule appears in Chapter 5.

$$\frac{\begin{array}{l}\{P\ \wedge\ \mathsf{sequence\_wp}\ ys\ \mathsf{branch\_cond}\}\\ \quad xs\\ \{P\}\\ \mathsf{well\_formed\_loop}[(\mathsf{UBF}\ |xs| + \mathit{1})]@[xs]@[ys]@[(\mathsf{CBB}\ |xs@ys|)]\end{array}}{\begin{array}{l}\{P\}\\ \quad [(\mathsf{UBF}\ |xs| + \mathit{1})]@[xs]@[ys]@[(\mathsf{CBB}\ |xs@ys|)]\\ \{P \wedge \neg\mathsf{sequence\_wp}\ ys\ \mathsf{branch\_cond}\}\ \wedge\ \mathsf{not\_term\_state}\end{array}}$$

4.5.2 Conditional Branch Forward Rule

We now describe a rule for conditional branches forward which, like the while rule, depends on the lemmas for sequence weakest precondition and the fact that the instructions *zs* immediately prior to the conditional branch instruction are referentially transparent.

Lemma 52 (Execute up to conditional branch)

$$\begin{array}{l}\forall\ \mathit{CFS}\ \sigma_0\ \sigma_n\ s\ f\ m\ n\ xs\ ys\ zs.\ \langle \mathit{CFS}, \sigma_0\rangle \xrightarrow[f]{s} \sigma_n\ \wedge\\ \qquad \mathit{CFS}\ [s \ldots f]\ =\ zs@[\mathit{CBF}\ (|xs|\ +\ 1)]xs@ys\ \wedge\\ \qquad \mathsf{pc}(\sigma_0)\ =\ \mathsf{pc}(s)\ \wedge\\ \qquad \mathsf{simple}\ zs\ \wedge\\ \qquad \mathsf{simple}\ [\mathit{CBF}\ (|xs|\ +\ 1)]@xs@ys\\ \qquad \longrightarrow \exists\sigma_1.\ \langle \mathit{CFS}, \sigma_0\rangle \xrightarrow[y]{s} \sigma_1\ \wedge\\ \qquad\qquad \langle \mathit{CFS}, \sigma_1\rangle \xrightarrow[f]{z} \sigma_n\end{array}$$

where

$$y = s\{\mathsf{pc} := \mathsf{pc}(s) + |zs| - 1\}$$
$$z = s\{\mathsf{pc} := \mathsf{pc}(s) + |zs|)\}$$

PROOF From the definition of simple (Definition 17), and Lemma 30. ∎

Theorem 6 (Conditional branch rule)

$$\frac{\begin{array}{l} \{P\}\ zs@[CBF\ (|xs|\ +\ 1)]\ \{R\} \\ \{R\ \wedge\ (\mathsf{sequence\_wp}\ zs\ \mathsf{branch\_cond})\}\ ys\ \{Q\} \\ \{R\ \wedge\ \neg(\mathsf{sequence\_wp}\ zs\ \mathsf{branch\_cond})\}\ xs@ys\ \{Q\} \\ xs\ \neq\ [\,];\ ys\ \neq\ [\,];\ zs\ \neq\ [\,] \\ \mathsf{excep\_free}\ zs@[CBF\ (|xs|\ +\ 1)]@xs@ys \\ \mathsf{simple}\ xs;\ \ \mathsf{simple}\ ys \\ \mathsf{ref\_trans\_2}\ zs;\ \mathsf{simple}\ [CBF\ (|xs|\ +\ 1)]@xs@ys \end{array}}{\{P\}\ \ zs@[CBF\ (|\ xs|\ +\ 1))]@xs@ys\ \{Q\}}$$

PROOF From the pre- and post-condition relation for bytecode (Definition 16) we know that there is a block

$$\langle CFS, \sigma_0\rangle \xrightarrow[f]{s} \sigma_1 \tag{4.11}$$

such that

$$CFS[s \dots f]\ =\ zs@[CBF\ (|xs|\ +\ 1)]@xs@ys \tag{4.12}$$

and P holds in state σ_0.

From Lemma 52 we know that

$$\exists \sigma_1.\ \langle CFS, \sigma_0\rangle \xrightarrow[y]{s} \sigma_1\ \wedge\ \langle CFS, \sigma_1\rangle \xrightarrow[f]{z} \sigma_n \tag{4.13}$$

where

$$y = s\,\{\mathsf{pc}\ :=\ \mathsf{pc}(s) + |zs|) - 1\}$$
$$z = s\,\{\mathsf{pc}\ :=\ \mathsf{pc}(s) + |zs|\}$$

where the first block, from s to y, corresponds to execution of the instructions zs prior to the conditional branch instruction. The program counter in state σ_1 points to the conditional branch instruction.

Execution of the second block can take one of two forms depending on whether the branching condition is true in state σ_1 or not. In the case where the condition is true, the branch is taken, branch_cond holds in state σ_1 and only the instructions ys are executed. In the case where the condition is false, $\mathsf{branch\_cond}$ does not hold in state σ_1 and the instructions $xs@ys$ are executed.

We now apply case analysis to the second block, noting that the case involving the **Stop** rule cannot apply as it would require the program counter to finish outside the block $xs@ys$ as a result of executing a jump of length $xs{+}1$ where both xs and ys are non-empty. This gives us

$$\exists \sigma_2.\ \mathsf{exec}(CFS, \sigma_1) \ = \ \mathsf{Some}\ \sigma_2 \ \wedge \ \langle CFS, \sigma_2\rangle \xrightarrow[f]{z} \ \sigma_n \tag{4.14}$$

A case split on the value of the branch condition in state σ_1, followed by narrowing of the blocks as in Lemma 31, along with Lemma 52, gives us the two possible situations

$$\begin{aligned} &\exists \sigma_1\ \sigma_2.\mathsf{branch\_cond}(\sigma_1) \ \wedge \ \langle CFS, \sigma_0\rangle \xrightarrow[y]{s} \ \sigma_1 \ \wedge \\ &\qquad \mathsf{exec}(CFS, \sigma_1) \ = \ \mathsf{Some}\ \sigma_2 \ \wedge \ \langle CFS, \sigma_2\rangle \xrightarrow[f]{z} \ \sigma_n \end{aligned} \tag{4.15}$$

and

$$\begin{aligned} &\exists \sigma_1\ \sigma_2.\ \neg\mathsf{branch\_cond}(\sigma_1) \ \wedge \ \langle CFS, \sigma_0\rangle \xrightarrow[y]{s} \ \sigma_1 \ \wedge \\ &\qquad \mathsf{exec}(CFS, \sigma_1) \ = \ \mathsf{Some}\ \sigma_2 \ \wedge \ \langle CFS, \sigma_2\rangle \xrightarrow[f]{w} \ \sigma' \end{aligned} \tag{4.16}$$

where

$$w = s\{\mathsf{pc} := \mathsf{pc}(s) + |zs| + |xs| {+} 1\}$$

Using Lemma 7 we are now able to obtain a block covering execution of the instructions $zs@[CBF\ (|xs|\ +\ 1)]$

$$\langle CFS, \sigma_0\rangle \xrightarrow[z]{s} \ \sigma_1 \tag{4.17}$$

And from the assumption

$$\{P\}\ zs@[CBF\ (|xs|\ +\ 1)]\ \{R\} \tag{4.18}$$

we know that R holds in state σ_2.

From Lemma 21 and Lemma 48 we know that

$$\begin{aligned} &\mathsf{branch\_cond}\ (\sigma_1) \ \wedge \ \langle CFS, \sigma_0\rangle \xrightarrow[y]{s} \ \sigma_1 \ \wedge \\ &\mathsf{exec}(CFS, \sigma_1) \ = \ \mathsf{Some}\ \sigma_2 \\ &\Longrightarrow (\mathsf{sequence\_wp}\ zs\ \mathsf{branch\_cond})\ \sigma_0 \end{aligned} \tag{4.19}$$

and from Lemma 21 and Lemma 47

$$\begin{array}{l}\neg\mathsf{branch\_cond}\,(\sigma_1)\ \wedge\ \langle CFS, \sigma_0\rangle \xrightarrow[y]{s} \sigma_1\ \wedge\\ \mathsf{exec}(CFS, \sigma_1)\ =\ \mathsf{Some}\ \sigma_2\\ \Longrightarrow \neg\,(\mathsf{sequence\_wp}\ zs\ \mathsf{branch\_cond})\ \sigma_0\end{array} \tag{4.20}$$

We must now show that in the case of 4.19 that

$$(\mathsf{sequence\_wp}\ zs\ \mathsf{branch\_cond})\ \sigma_2 \tag{4.21}$$

and in the case of 4.20 that

$$\neg(\mathsf{sequence\_wp}\ zs\ \mathsf{branch\_cond})\ \sigma_2 \tag{4.22}$$

Since the instructions zs are referentially transparent it follows that when zs is of length 2, in state σ_0 the stack is equal to the stack in state σ_1 with the addition of two extra values at the top meaning that

$$stk_0 = \mathsf{tl}(\mathsf{tl}(stk_1)) \tag{4.23}$$

and all other values in σ_1—excluding the program counter—are equal to those of σ_0. Execution of the conditional branch instruction in state σ_1 pops the two topmost values from the stack (after comparing them). This means that

$$stk_0 = stk_2 \tag{4.24}$$

and, as all other elements of the states apart from the program counters are equal

$$\sigma_0\ \cong\ \sigma_2 \tag{4.25}$$

This situation is illustrated in Figure 4.7. We can now use this result with 4.15, 4.16 and the assumptions $\{R\ \wedge\ (\mathsf{sequence\_wp}\ zs\ \mathsf{branch\_cond})\}\ ys\ \{Q\}$ and $\{R\ \wedge\ \neg(\mathsf{sequence\_wp}\ zs\ \mathsf{branch\_cond})\}\ xs@ys\ \{Q\}$
to show that Q holds in state σ_n. ■

4.6 Conclusions

We have described a programming logic containing rules for sufficient patterns of bytecode to prove bytecode programs including loops and branching

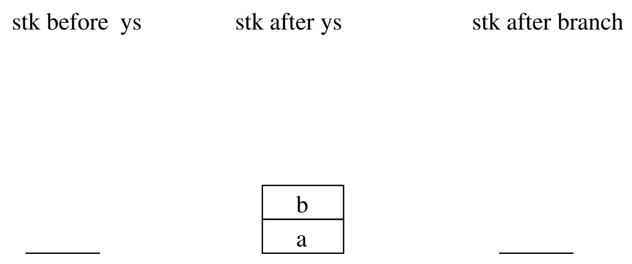

Figure 4.7: Stack unchanged by zs followed by branch

structures. Although the rules described are clearly closely related to the rules in the conventional Hoare logic, they differ in a number of important ways. In particular, the necessity of treating the execution of the instructions defining a program structure explicitly. In the following chapter we describe in detail the difficulties this presents in the case of a proof of soundness of the rule for loops in the bytecode logic, and the ways in which these problems can be overcome.

Chapter 5

Soundness of the Loop Rule

In this chapter we will discuss a proof of the soundness of the rule for loops in bytecode programs that was proposed in Chapter 4. We begin by describing the proof of the while rule in the conventional Hoare logic and how it relates to our proof of the loop rule for bytecode programs. Both of these proofs depend on proofs of two subsidiary properties: w

1. The loop guard condition will be *false* on exit from the loop

2. If a property is invariant for one execution of the loop body it is also invariant for the loop itself

In a conventional axiomatic semantics these properties are very simple to prove as they follow almost immediately from the execution rules for the language. In the bytecode world, however, they are considerably more difficult to prove and we discuss the methods used to achieve these proofs in some detail.

5.1 Outline of Proof Method

The standard Hoare logic while rule for a simple imperative programming language (4.3) states that if P is an invariant for one execution of C whenever S holds then it is also an invariant for the execution of the statement *while* S *do* C, and that S will be false on termination of the loop.

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

Example proofs of its soundness can be found in [10] and [56], and they can be broken down into proofs of each of the two properties mentioned in the introduction to this chapter. In [10] these are represented by the following lemmas:

Guard false on exit from loop—conventional Hoare logic

$$\begin{aligned}&\forall\ C\ s1\ s2.\ \mathsf{eval}\ C\ s1\ =\ s2\ \longrightarrow (\forall\ S'\ C'.\ (C\ =\ \mathsf{while}\ S'\ C')\\ &\qquad\qquad\qquad\qquad\qquad\qquad\longrightarrow \neg(S'\ s2))\end{aligned}$$

Preservation of invariant—conventional Hoare logic

$$\begin{aligned}&\forall\ C\ s1\ s2.\\ &\ \ \mathsf{eval}\ C\ s1\ =\ s2\ \longrightarrow\\ &\qquad (\forall\ S'\ C'.\ (C\ =\ \mathsf{while}\ S'\ C')\ \longrightarrow\\ &\qquad\qquad (\forall\ s1\ s2.\ P\ s1\ \wedge\ S'\ s1\ \wedge\ C'\ s1\ s2\ \longrightarrow\ P\ s2)\ \longrightarrow\\ &\qquad\qquad\ \ (P\ s1\ \longrightarrow\ P\ s2))\end{aligned}$$

Both lemmas can be proved using the operational semantics of the languages they are concerned with, either immediately or by strong rule induction.

For the Bytecode Programming Logic, recall that our postulated Loop Rule is

$$\frac{\begin{array}{l}\{P\ \wedge\ \mathsf{sequence\_wp}\ ys\ \mathsf{branch\_cond}\}\\ \quad xs\\ \{P\}\\ \mathsf{well\_formed\_loop}[(\mathsf{UBF}\ |xs| + 1)]@[xs]@[ys]@[(\mathsf{CBB}\ |xs@ys|)]\end{array}}{\begin{array}{l}\{P\}\\ \quad [(\mathsf{UBF}\ |xs| + 1)]@[xs]@[ys]@[(\mathsf{CBB}\ |xs@ys|)]\\ \{P \wedge \neg\mathsf{sequence\_wp}\ ys\ \mathsf{branch\_cond}\ \wedge\ \mathsf{not\_term\_state}\}\end{array}}$$

and we assert that equivalent statements can be defined for the bytecode:

Proposition 1 (Guard false on exit from loop—bytecode logic)

$$
\begin{aligned}
\forall\ \sigma_0\ \sigma_n\ \mathit{CFS}\ s\ f.\ \langle \mathit{CFS}, \sigma_0\rangle \xrightarrow[f]{s} \sigma_n\ \wedge\\
&\mathit{CFS}[s \ldots f]\ =\ [(\mathsf{UBF}\ |xs| + 1)]@[xs]@[ys]@[(\mathsf{CBB}\ |xs@ys|)]\ \wedge\\
&\mathsf{well\_formed\_loop}\ \mathit{CFS}[s \ldots f]\\
&\longrightarrow\ \neg\,(\mathsf{sequence\_wp}\ ys\ \mathsf{branch\_cond})\ \sigma_n
\end{aligned}
$$

Proposition 2 (Preservation of invariant—bytecode logic)

$$
\begin{aligned}
\forall\ \sigma_0\ \sigma_n\ \mathit{CFS}\ s\ f.\ \langle \mathit{CFS}, \sigma_0\rangle \xrightarrow[f]{s} \sigma_n \wedge\\
&\mathsf{well\_formed\_loop}\ \mathit{CFS}[s \ldots f]\ \wedge\\
&P\ \sigma_0\ \wedge\\
&[\forall\ \sigma'_0\ \sigma'_1\ \sigma'_n\ \mathit{CFS}'\ w\ y\ s'\ f'.\\
&\qquad \langle \mathit{CFS}', \sigma'_0\rangle \xrightarrow[y]{w} \sigma'_1\ \wedge\\
&\qquad \langle \mathit{CFS}', \sigma'_1\rangle \xrightarrow[f]{z} \sigma'_n\ \wedge\\
&\qquad \mathsf{well\_formed\_loop}\ \mathit{CFS}'[s' \ldots f']\ \wedge\\
&\qquad P\ \sigma'_0\ \wedge\\
&\qquad (\mathsf{sequence\_wp}\ ys\ \mathsf{branch\_cond})\ \sigma'_0\ \longrightarrow\\
&\qquad P\ \sigma'_n]\\
&\longrightarrow\ P\ \sigma_n
\end{aligned}
$$

where

$$
\begin{aligned}
w &= s'\ \{\mathsf{pc} := \mathsf{pc}(s') + 1\}\\
y &= s'\ \{\mathsf{pc} := \mathsf{pc}(s') + |xs|\}\\
z &= s'\ \{\mathsf{pc} := \mathsf{pc}(s') + |xs| + 1\}
\end{aligned}
$$

Here, the fact that the invariant is preserved by one execution of the body of the loop is represented by the statement

$$
\begin{aligned}
\forall\ \sigma'_0\ \sigma'_1\ \sigma'_n\ \mathit{CFS}'\ w\ y\ s'\ f'.\ \langle \mathit{CFS}', \sigma'_0\rangle \xrightarrow[y]{w} \sigma'_1\ \wedge\\
&\langle \mathit{CFS}', \sigma'_1\rangle \xrightarrow[f]{z} \sigma'_n\ \wedge\\
&\mathsf{well\_formed\_loop}\ \mathit{CFS}'[s' \ldots f']\ \wedge\\
&P\ \sigma'_0\ \wedge\\
&(\mathsf{sequence\_wp}\ ys\ \mathsf{branch\_cond})\ \sigma'_0\ \longrightarrow\\
&P\ \sigma'_n
\end{aligned}
\tag{5.1}
$$

which says that, in fact, the invariant is preserved by one execution of the body of the loop—the list of instructions xs—plus execution of the 'structural' instructions $ys@CBB$.

In the next three sections we present proofs of Propositions 1 and 2, followed by a proof of soundness of the Loop Rule.

5.2 Guard Condition False on Exit from Loop

Our aim here is a proof of Proposition 1

$$\begin{aligned}
&\forall\ \sigma_0\ \sigma_n\ CFS\ s\ f.\ \langle CFS, \sigma_0\rangle \xrightarrow[f]{s} \sigma_n\ \wedge \\
&\quad CFS[s\text{`}ldotsf] \ = \ [(\mathsf{UBF}\ |xs| + 1)]@[xs]@[ys]@[(\mathsf{CBB}\ |xs@ys|)] \\
&\quad \mathsf{well\_formed\_loop}\ CFS[s \ldots f] \\
&\quad \longrightarrow\ \neg\,(\mathsf{sequence\_wp}\ ys\ \mathsf{branch\_cond})\ \sigma_n
\end{aligned}$$

Our strategy involves showing that there exists a penultimate state σ_{n-1}, whose program counter points to the conditional branch instruction, and execution of which results in final state σ_n:

Proposition 3 (Penultimate state exists)

$$\begin{aligned}
&\forall\ \sigma_0\ \sigma_n\ CFS\ s\ f\ y\ z.\ \langle CFS, \sigma_0\rangle \xrightarrow[f]{s} \sigma_n\ \wedge \\
&\quad \mathsf{well\_formed\_loop}\ CFS[s \ldots f] \longrightarrow \\
&\quad \exists\ \sigma_{n-2}\ \sigma_{n-1}\ .\ \langle CFS, \sigma_{n-2}\rangle \xrightarrow[z]{y} \sigma_{n-1}\ \wedge \\
&\qquad\qquad \mathsf{exec}\ (CFS, \sigma_{n-1}) = \mathsf{Some}\ \sigma_n\ \wedge \\
&\qquad\qquad \mathsf{pc}(\sigma_{n-1}) = \mathsf{pc}(f)
\end{aligned}$$

where

$$\begin{aligned}
y &= s\ \{\mathsf{pc} := \mathsf{pc}(s) + |xs|\} \\
z &= f\ \{\mathsf{pc} := \mathsf{pc}(f) - 1\}
\end{aligned}$$

We then use the fact that the branching condition must be false in this penultimate state in order for us to exit the loop, the idea of data equality (Definition 10), and the lemmas for the sequence weakest precondition (Definition 30) to show $\neg\,(\mathsf{sequence\_wp}\ ys\ \mathsf{branch\_cond})\ \sigma_n$.

5.2.1 Lemmas for Guard Condition False

In this section we describe a series of lemmas necessary for the proof of Proposition 1: the guard condition is false on exit from the loop.

To allow us to focus attention on the main part of the loop—the section $xs@ys@[CBB\ |xs@ys|]$—we show that for a well-formed loop in the block execution relation, executing the initial unconditional branch forward results in another instance of the relation consisting only of the instructions $xs@ys@[CBB\ |xs@ys|]$.

Lemma 53 (Elimination of unconditional branch)

$$\begin{aligned}
&\forall\ \sigma_0\ \sigma_n\ CFS\ s\ f.\ \langle CFS, \sigma_0\rangle \xrightarrow[f]{s} \sigma_n\ \wedge \\
&\qquad \mathsf{well\_formed\_loop}\ CFS[s\dots f] \longrightarrow \\
&\qquad \exists\ \sigma_1.\ \mathsf{exec}(CFS, \sigma_0)\ =\ \mathsf{Some}\ \sigma_1\ \wedge \\
&\qquad\qquad \mathsf{pc}(\sigma_1)\ =\ \mathsf{pc}(s)\ +\ |xs|\ +\ 1\ \wedge \\
&\qquad\qquad \langle CFS, \sigma_1\rangle \xrightarrow[f]{w} \sigma_n
\end{aligned}$$

where

$$w = s\ \{\mathsf{pc} := \mathsf{pc}(s) + 1\}$$

<span style="font-variant:small-caps">Proof</span> This follows from the fact that the instructions $xs@ys@[CBB\ |xs@ys|]$ make up a simple block (Definition 17), the execution of which can only result in a state where the program counter is pointing inside the block or at the instruction immediately to the right of it. Under these circumstances, once the unconditional branch instruction at the beginning of the loop has been executed once, it will never be executed again within this block. We are therefore able to effectively discard it and concentrate our attention on the smaller resultant block. The situation is represented in Figure 5.1. ∎

We now consider our smaller loop, and show that there exists a state, σ_2, in which the conditional branch instruction is executed for the first time.

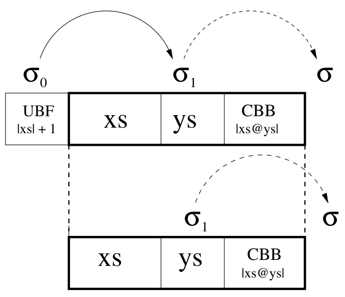

Figure 5.1: Elimination of unconditional branch

Lemma 54 (Conditional branch is executed at least once)

$$\begin{aligned}
&\forall CFS\ \sigma_1\ \sigma_n\ s\ f\ w\ y\ z.\langle CFS, \sigma_1\rangle \xrightarrow[f]{w} \sigma_n \wedge \\
&\qquad \mathsf{well\_formed\_loop}\ CFS[s \ldots f] \wedge \\
&\qquad \mathsf{simple}\ ys\ \wedge \\
&\qquad \mathsf{pc}(\sigma_1) = \mathsf{pc}(s) +\ |xs| + 1 \longrightarrow \\
&\qquad \exists \sigma_2.\langle CFS, \sigma_1\rangle \xrightarrow[z]{y} \sigma_2 \wedge \\
&\qquad\quad \mathsf{pc}(\sigma_2) = \mathsf{pc}(f) \wedge \\
&\qquad\quad \langle CFS, \sigma_2\rangle \xrightarrow[f]{w} \sigma_n
\end{aligned}$$

where

$$\begin{aligned}
w &= s\ \{\mathsf{pc} := \mathsf{pc}(s) + 1\} \\
y &= s\ \{\mathsf{pc} := \mathsf{pc}(s) + |xs| + 1\} \\
z &= f\ \{\mathsf{pc} := \mathsf{pc}(f) - 1\}
\end{aligned}$$

Proof We know that the program counter of σ_1 is pointing to the first instruction of ys. As we are assuming the existence of the larger block from σ_0 to σ_n, we know that execution must leave the block ys at some point in order to reach state σ_n.

But because ys is a simple block we cannot reach σ_n directly from any instruction in ys, and there must exist an intermediate state, σ_2, whose program counter is pointing to the instruction immediately to the right of ys, i.e. the conditional branch instruction. Execution of the conditional branch instruction in this state will then lead to σ_n, either immediately or following a number of executions of the loop—represented by the instance of the block execution relation $\langle CFS, \sigma_2\rangle \xrightarrow[f]{w} \sigma_n$ ■

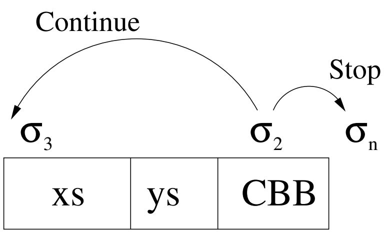

Figure 5.2: Possible outcomes of executing branch instruction

In order to prove the existence of a penultimate state (Proposition 3), we now consider the block $\langle CFS, \sigma_2 \rangle \xrightarrow[f]{w} \sigma_n$, with $\mathsf{pc}(\sigma_2) = \mathsf{pc}(f)$, and apply the block execution case rule. This presents the two possibilities shown in Figure 5.2

- the Stop rule applies:

$$\mathsf{exec}(CFS, \sigma_2) = \mathsf{Some}\ \sigma_n \ \wedge\ \mathsf{pc}(\sigma_2) = \mathsf{pc}(f) \tag{5.2}$$

- the Continue rule applies:

$$\exists \sigma_3.\ \mathsf{exec}(CFS, \sigma_2) = \mathsf{Some}\ \sigma_3 \wedge \langle CFS, \sigma_3 \rangle \xrightarrow[f]{w} \sigma_n \wedge \mathsf{pc}(\sigma_2) = \mathsf{pc}(f) \tag{5.3}$$

In the Stop case the proof of Proposition 3 is complete; in the second case we need to know whether or not the branch condition is true in order to determine the position of the program counter after execution.

In the case where the condition is *false* we have a contradiction: executing the branch instruction in this situation results in the program counter of σ_3 being outside the block, but the existence of the block $\langle CFS, \sigma_3 \rangle \xrightarrow[f]{w} \sigma_n$ means that, from Lemma 1 it must be inside the block.

This leaves the third case in which the branching condition is *true*. Again we consider statement 5.3, relating to the situation where the branching condition is true. From the definition of a well formed loop (Definition 31) we know that the program counter of state σ_3 points to the first instruction in *xs* as a result of the conditional branch backwards in a state where the condition is *true*. We now reason 'backwards' from the final state, σ_n.

We show that if, starting in a state whose program counter is equal to $\mathsf{pc}(s)+1$, i.e. the state produced by executing the conditional branch when the condition is true, we have reached σ_n—a fact implicit in the definition of the block execution relation (**Stop**, **Continue**)—then there must be some state σ_{n-1} in the execution path relation for the block that evaluates to σ_n.

Lemma 55 (Penultimate state exists in execution of the loop)

$$\begin{aligned}
&\forall \mathit{CFS}\ \sigma_3\ s\ f\ w.\ \langle \mathit{CFS}, \sigma_3\rangle \xrightarrow[f]{w} \sigma_n \ \wedge \\
&\qquad \mathsf{well\_formed\_loop}\ \mathit{CFS}[s \ldots f] \longrightarrow \\
&\qquad \exists \sigma_{n-1}.\ \langle \mathit{CFS},\ \sigma_3\rangle \overset{w}{\underset{w}{\Longrightarrow}}{}^{+}\ f\sigma_{n-1} \ \wedge \\
&\qquad\qquad \mathsf{pc}(\sigma_{n-1}) = \mathsf{pc}(f) \\
&\qquad\qquad \wedge\ \mathsf{exec}(\mathit{CFS}, \sigma_{n-1}) = \mathsf{Some}\ \sigma_n
\end{aligned}$$

where

$$w = s\,\{\mathsf{pc} := \mathsf{pc}(s) + 1\}$$

Proof From Lemma 9 we know that as σ_3 and σ_{n-1} are in the block execution relation, they are also in the execution path relation. We then apply transitive closure elimination to 'unwind' the final execution step from the relation which, along with the definition of the execution path relation (Definition 8), implies that $\exists \sigma_{n-1}.\ \mathsf{exec}(\mathit{CFS}, \sigma_{n-1}) = \mathsf{Some}\ \sigma_n$. As xs and ys are simple, we can show that $\mathsf{pc}(\sigma_{n-1}) = \mathsf{pc}(f)$, since from the definition of simple, (Definition 17), it is impossible to reach the final state outside the block from either xs or ys. ■

We show that if we have reached this penultimate state, σ_{n-1}, starting from a position σ_3 inside xs there must be a sequence of states spanning the final execution of ys that starts in a state whose program counter points to the first instruction in ys and culminates in σ_{n-1}.

Lemma 56 (Ante-penultimate state exists in loop execution)

$$\begin{aligned}
&\forall \mathit{CFS}\ \sigma_3\ s\ f\ w.\ \langle \mathit{CFS}, \sigma_3\rangle \xrightarrow[f]{w} \sigma_n \ \wedge\ \mathsf{well\_formed\_loop}\ \mathit{CFS}[s \ldots f] \longrightarrow \\
&\qquad \exists \sigma_{n-1}\ \sigma_{n-2}.\ \langle \mathit{CFS},\ \sigma_3\rangle \overset{w}{\underset{f}{\Longrightarrow}}{}^{+}\ \sigma_{n-2} \ \wedge \\
&\qquad\qquad \langle \mathit{CFS},\ \sigma_{n-2}\rangle \overset{y}{\underset{z}{\Longrightarrow}}{}^{+}\ \sigma_{n-1} \ \wedge \\
&\qquad\qquad \mathsf{pc}(\sigma_{n-2}) = \mathsf{pc}(m) \\
&\qquad\qquad \mathsf{pc}(\sigma_{n-1}) = \mathsf{pc}(f) \\
&\qquad\qquad \mathsf{exec}(\mathit{CFS}, \sigma_{n-1}) = \sigma_n
\end{aligned}$$

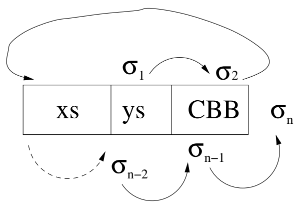

Figure 5.3: Final steps of execution

where

$$
\begin{aligned}
w &= s\,\{\mathsf{pc} := \mathsf{pc}(s) + \mathit{1}\}\\
y &= s\,\{\mathsf{pc} := \mathsf{pc}(s) + |xs| + \mathit{1}\}\\
z &= f\,\{\mathsf{pc} := \mathsf{pc}(f) - 1\}
\end{aligned}
$$

Proof From the definition of simple, (Definition 17), the only section of the block from which it is possible to reach the conditional branch instruction is *ys*, and in order to reach *ys* we must have come from *xs* (as it is not reachable from the conditional branch). Furthermore, the only instruction outside *xs* reachable from within it is the first instruction in *ys*. The path of execution is shown in Figure 5.3 ■

This allows us to prove Proposition 3

Lemma 57 (Final step preceded by execution of *ys* exists)

$$
\begin{aligned}
&\forall\ \sigma_{\mathit{0}}\ \sigma_n\ \mathit{CFS}\ s\ f.\ \langle \mathit{CFS}, \sigma_{\mathit{0}}\rangle \xrightarrow[f]{s} \sigma_n\ \wedge\\
&\qquad \mathsf{well\_formed\_loop}\ \mathit{CFS}[s\ldots f] \longrightarrow\\
&\qquad \exists\ \sigma_{n-\mathit{1}}\ \sigma_{n-\mathit{2}}\ .\ \langle \mathit{CFS}, \sigma_{n-\mathit{2}}\rangle \xrightarrow[z]{y} \sigma_{n-\mathit{1}}\ \wedge\\
&\qquad\qquad \mathsf{exec}\ (\mathit{CFS}, \sigma_{n-\mathit{1}}) = \mathsf{Some}\ \sigma_n\ \wedge\\
&\qquad\qquad \mathsf{pc}(\sigma_{n-\mathit{2}}) = \mathsf{pc}(m)\\
&\qquad\qquad \mathsf{pc}(\sigma_{n-\mathit{1}}) = \mathsf{pc}(f)
\end{aligned}
$$

where

$$
\begin{aligned}
w &= s\,\{\mathsf{pc} := \mathsf{pc}(s) + \mathit{1}\}\\
y &= s\,\{\mathsf{pc} := \mathsf{pc}(s) + |xs| + \mathit{1}\}\\
z &= f\,\{\mathsf{pc} := \mathsf{pc}(f) - 1\}
\end{aligned}
$$

Proof From Lemma 55 and Lemma 56. ∎

We now prove some more preliminary lemmas in order to prove prove Proposition 1.

We know from Lemma 57 that

$$\exists \sigma_{n-1}.\ \mathsf{exec}\ (CFS, \sigma_{n-1}) = \mathsf{Some}\ \sigma_n \tag{5.4}$$

where the instruction at σ_{n-1} is the conditional branch backwards to the start of xs. The fact that the branch is not taken—execution results in state σ_n which is outside the block—implies that the branch condition is false in state σ_{n-1} (from the operational semantics of branching instructions). This gives us

Lemma 58 (Branch condition false implies swp false)

$$\begin{aligned}\forall\ \sigma_{n-1}\ \sigma_{n-2}\ .\ \langle CFS, \sigma_{n-2}\rangle \xrightarrow[z]{y} \sigma_{n-1}\ \wedge \\ \mathsf{well\_formed\_loop}\ CFS[s \ldots f]\ \wedge \\ \neg(\mathsf{branch\_cond}\ \sigma_{n-1}) \longrightarrow \\ \neg\ (\mathsf{sequence\_wp}\ ys\ \mathsf{branch\_cond})\ \sigma_{n-2}\end{aligned}$$

where

$$y = s\,\{\mathsf{pc} := \mathsf{pc}(s) + |xs| + 1\}$$
$$z = f\,\{\mathsf{pc} := \mathsf{pc}(f) - 1\}$$

Proof From Lemma 56 and Lemma 46. ∎

We now show that states σ_{n-2} and σ_n are data-equal:

Lemma 59 (Antepenultimate and final states are data-equal)

$$\begin{aligned}\forall\ \sigma_{n-1}\ \sigma_{n-2}\ .\ \langle CFS, \sigma_{n-2}\rangle \xrightarrow[z]{y} \sigma_{n-1}\ \wedge \\ \mathsf{well\_formed\_loop}\ CFS[s \ldots f]\ \wedge \\ \mathsf{exec}(CFS, \sigma_{n-1})\ =\ \mathsf{Some}\ \sigma_n \longrightarrow \\ \sigma_{n-2}\ \cong\ \sigma_n\end{aligned}$$

where

$$y = s\,\{\mathsf{pc} := \mathsf{pc}(s) + |xs| + 1\}$$
$$z = f\,\{\mathsf{pc} := \mathsf{pc}(f) - 1\}$$

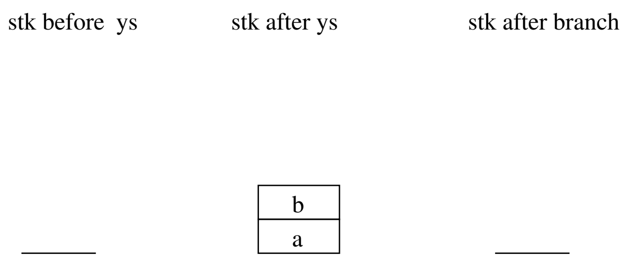

Figure 5.4: Stack unchanged by ys followed by branch

Proof From the definition of a well formed loop (Definition 31) we know that *ys* is referentially transparent. From Definition 32 we see that the execution of a sequence of referentially transparent instructions affects only the stack and the program counter.

So in state σ_{n-1} the stack is equal to the stack in state σ_{n-2} with the addition of two extra values at the top implying that

$$stk_{n-2} = \mathsf{tl}(\mathsf{tl}(stk_{n-1}) \tag{5.5}$$

and all other values in σ_{n-1}—excluding the program counter—are equal to those of σ_{n-2}.

Execution of the conditional branch instruction in state σ_{n-1} pops the two topmost values from the stack (after comparing them). This means that

$$stk_{n-2} = stk_n \tag{5.6}$$

and, as all other elements of the states apart from the program counters are data-equal

$$\sigma_{n-2} \cong \sigma_n \tag{5.7}$$

This situation is illustrated in Figure 5.4. ∎

Using these lemmas and the lemmas for the sequence weakest precondition we can now prove Proposition 1:

Theorem 7 (Loop guard false on exit—bytecode logic)

$$\begin{aligned} \forall\ \sigma_0\ \sigma_n\ CFS\ s\ f.\ \langle CFS, \sigma_0\rangle \xrightarrow[f]{s} \sigma_n\ \wedge \\ CFS[s \ldots f]\ =\ [(\mathsf{UBF}\ |xs| + 1)]@[xs]@[ys]@[(\mathsf{CBB}\ |xs@ys|)] \\ \mathsf{well\_formed\_loop}\ CFS[s \ldots f] \\ \longrightarrow\ \neg\,(\mathsf{sequence\_wp}\ ys\ \mathsf{branch\_cond})\ \sigma_n \end{aligned}$$

Proof From Lemma 56, Lemma 57, and Lemma 59. ∎

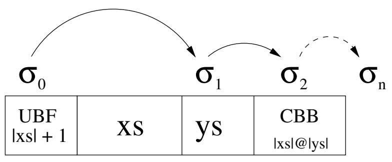

Figure 5.5: Preliminary execution of the loop

5.3 Preservation of the Invariant

Assuming that execution of the loop begins in state σ_0, in which the invariant assert P holds and whose program counter points to the unconditional branch instruction, we show that the conditional branch instruction is reached for the first time via the execution of the instructions ys. This is illustrated in Figure 5.5.

Lemma 60 (Initial pattern of loop execution)

$$\begin{aligned}
\forall\ \sigma_0\ \sigma_n\ CFS\ s\ f\ y\ z.\ & \langle CFS, \sigma_0\rangle \xrightarrow[f]{s} \sigma_n \ \wedge \\
& \mathsf{pc}(\sigma_0) = \mathsf{pc}(s) \ \wedge \\
& \mathsf{well\_formed\_loop}\ CFS[s \ldots f] \ \longrightarrow \\
& \exists \sigma_1\ \sigma_2.\ \mathsf{exec}\ (CFS, \sigma_0) \ = \ \mathsf{Some}\ \sigma_1 \ \wedge \\
& \quad \mathsf{pc}(\sigma_1) = \mathsf{pc}(m) \ \wedge \\
& \quad \langle CFS, \sigma_1\rangle \xrightarrow[z]{y} \sigma_2 \ \wedge \\
& \quad \mathsf{pc}(\sigma_2) = \mathsf{pc}(f) \ \wedge \\
& \quad \langle CFS, \sigma_2\rangle \xrightarrow[f]{s} \sigma_n
\end{aligned}$$

where

$$\begin{aligned}
y &= s\,\{\mathsf{pc} := \mathsf{pc}(s) + |xs| + 1\} \\
z &= f\,\{\mathsf{pc} := \mathsf{pc}(f) - 1\}
\end{aligned}$$

PROOF From Lemma 53 and Lemma 54. ■

As in the proof described in Section 5.2, we now consider the block $\langle CFS, \sigma_2\rangle \xrightarrow[f]{w} \sigma_n$, where $w = s\,\{\mathsf{pc} := \mathsf{pc}(s) + 1\}$ and $\mathsf{pc}(\sigma_2) = \mathsf{pc}(f)$, and apply the block execution case rule. Once again this presents the two possibilities shown in Figure 5.2

- the Stop rule applies:

$$\mathsf{exec}\ (CFS,\ \sigma_2) = \mathsf{Some}\ \sigma_n\ \wedge\ \mathsf{pc}(\sigma_2) = \mathsf{pc}(f) \tag{5.8}$$

- the Continue rule applies:

$$\exists\sigma_3.\ \mathsf{exec}\ (CFS,\ \sigma_2) = \mathsf{Some}\ \sigma_3 \wedge \langle CFS, \sigma_3\rangle \xrightarrow[f]{w} \sigma_n \wedge \mathsf{pc}(\sigma_2) = \mathsf{pc}(f) \tag{5.9}$$

In the Stop case, we prove the following lemma:

Lemma 61 (Initial, second and final states data-equal)

$$\begin{aligned}\forall\ \sigma_0\ \sigma_1\ \sigma_2\ \sigma_n\ CFS\ s\ f\ y\ z.\ & \mathsf{well\_formed\_loop}\ CFS[s\ldots f] \wedge \\ & \mathsf{pc}(\sigma_0) = \mathsf{pc}(s) \wedge \\ & \mathsf{exec}\ (CFS, \sigma_0)\ =\ \mathsf{Some}\ \sigma_1\ \wedge \\ & \langle CFS, \sigma_1\rangle \xrightarrow[z]{y} \sigma_2 \wedge \\ & \mathsf{exec}\ (CFS,\ \sigma_2) = \mathsf{Some}\ \sigma_n \wedge \\ & \mathsf{pc}(\sigma_2) = \mathsf{pc}(f) \longrightarrow \\ & \sigma_0\ \cong\ \sigma_1\ \wedge\ \sigma_1\ \cong\ \sigma_n\end{aligned}$$

where

$$\begin{aligned} y &= s\ \{\mathsf{pc} := \mathsf{pc}(s) + |xs| + 1\} \\ z &= f\ \{\mathsf{pc} := \mathsf{pc}(f) - 1\}\end{aligned}$$

PROOF From the operational semantics of the unconditional branch instruction, we know that only the program counters of states σ_0 and σ_1 differ, so $\sigma_0\ \cong\ \sigma_1$.

From the definition of a well formed loop (Definition 31) we know that ys is referentially transparent and so, as in Lemma 59, $\sigma_1\ \cong\ \sigma_n$. ∎

This then allows us to show that the invariant is preserved for the **Stop** case:

Lemma 62 (Invariant preserved—Stop case)

$$\begin{aligned}\forall\ \sigma_0\ \sigma_n\ CFS\ s\ f\ y\ z.\ & \langle CFS, \sigma_0\rangle \xrightarrow[f]{s} \sigma_n\ \wedge \\ & \mathsf{exec}(CFS, \sigma_0)\ =\ \sigma_n \\ & \mathsf{pc}(\sigma_0) = \mathsf{pc}(s) \wedge \\ & \mathsf{well\_formed\_loop}\ CFS[s\ldots f]\ \longrightarrow \\ & (\mathsf{assert}\ P)\ \sigma_n\end{aligned}$$

Proof From Lemma 60, Lemma 61 and Lemma 51. ∎

We now turn to the case relating to the **Continue** rule. As in the proof of Theorem 7, we reason by cases on the branch condition, which produces a contradiction in the case where the condition is false, leaving us to consider the case in which the branch is taken.

We again use the fact that the unconditional branch instruction will not be reached again during the execution of this block (Lemma 53). This leaves us to concentrate on the preservation of the invariant, assert P, by the execution of the instructions $xs@ys@[CBB]$, assuming we start in a state pointing to the start of xs and in which assert P holds.

It is obvious that, in order to prove that the preservation of assert P by one execution of the loop body implies its preservation by multiple executions, it will be necessary to use some form of inductive argument. This is the approach used in the proofs of soundness of the while rule for more traditional Hoare logics [56, 10]. These proofs are reasonably straightforward as in the inductive definition of the language, one step of execution corresponds to one execution of the body of the loop.

In the bytecode programming logic, however, this is not the case. The block execution relation works at a much finer grain, i.e. that of individual bytecode instructions, several of which may be needed to represent a single 'higher level' instruction like array assignment. In addition, although the invariant holds at the beginning and end of the body of the loop, it may or may not hold between these points.

In standard inductive definitions of execution like those mentioned above, this is not a problem as the body, C, of a loop many be inductively built up from several commands $C_1, ..., C_n$, but is viewed as a single command in its own right. In this way we can abstract away from the finer detail and view its execution to be viewed as a single step. In the block execution relation however, a single step of execution is that of a single bytecode instruction, and so we cannot use the block execution relation directly to reason about the preservation of assert P across a loop body consisting of several bytecode instructions.

It is clearly necessary to find a relation describing a 'big step' of execution in the bytecode world. If the execution of the loop can be described in such a manner we can then carry out a successful induction leading to the proof of preservation of the invariant. Of course, this relation must also take into account the fact that we must explicitly execute the 'structure' of the loop,

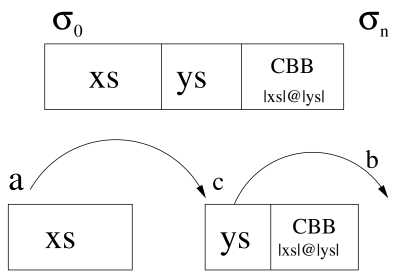

Figure 5.6: Decomposition of loop

represented by the instructions $ys@[CBB]$.

5.3.1 A 'Big Step' Execution Relation for Loops

If we consider the section $xs@ys@[CBB\ |xs@ys|]$ of a well-formed loop for which the relation $\langle CFS, \sigma_0\rangle \xrightarrow[f]{w} \sigma_n$ holds, where $w = s\,\{\mathsf{pc} := \mathsf{pc}(s) + 1)/\mathsf{pc}(s)]$, we see that it could be viewed as two separate blocks: xs and $ys@[CBB]$. The states σ_0 and σ_n can then be seen to be members of a set of states representing 'big steps' of execution:

Definition 33 (Loop as series of big steps)

$$\mathsf{big\_step\_loop} \;\equiv\; \{(a,b) :\ \exists\ c.\ \langle CFS, a\rangle \xrightarrow[x]{w}\ c\ \wedge\ \langle CFS, c\rangle \xrightarrow[f]{y}\ b\}^{+}$$

where

$$w = s\,\{\mathsf{pc} := \mathsf{pc}(s) + \mathit{1})/\mathsf{pc}(s)]$$
$$x = s\,\{\mathsf{pc} := \mathsf{pc}(s) + |xs|)/\mathsf{pc}(s)]$$
$$y = s\,\{\mathsf{pc} := \mathsf{pc}(s) + |xs| + \mathit{1})/\mathsf{pc}(s)]$$

Even if the branch is taken back to the start of xs, the state b is outside the block $ys@[CBB\ |xs@ys|]$ and so the relation holds. This is shown in Figure 5.6 and Figure 5.7.

As the pairs of states in this relation span the whole of the block $xs@ys@[CBB\ |xs@ys|]$ we now have a relation upon which we can perform induction. But first we must show that if two states are in the block execution relation they are also in the 'big step' relation, (Definition 33). Once again

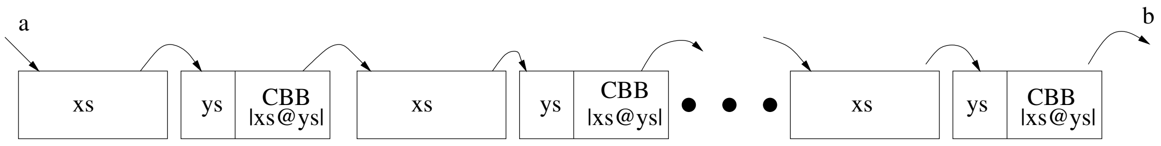

Figure 5.7: Loop as transitive closure of blocks

the mismatch of step size means the proof cannot be obtained directly by induction on $\langle CFS, \sigma_0 \rangle \xrightarrow[f]{w} \sigma_n$, and we must first address this issue.

The following lemma defines the notion of 'big' steps of execution that start anywhere in the instructions $xs@ys$ and finish by executing the conditional branch instruction at $\mathsf{pc}(f)$, as shown in Figure 5.8. The conditional branch can be viewed as a 'pivot' instruction: after each execution of the body of the loop, the conditional branch must be executed, resulting either in termination of the loop, or at least one subsequent execution of the loop body.

Lemma 63 (Loop execution 'pivots' on conditional branch)

$$\begin{aligned}
&\forall CFS\ s\ f\ w\ z\ \sigma_0\ \sigma_n.\ \langle CFS, \sigma_0 \rangle \xrightarrow[f]{w} \sigma_n\ \wedge \\
&\qquad \mathsf{well\_formed\_loop}\ CFS[s \ldots f] \longrightarrow \\
&\qquad (\sigma_0,\ \sigma_n) \in \\
&\qquad\quad \{(a, b) :\ \mathsf{exec}(CFS, a) = b\ \wedge\ \mathsf{pc}(a)\ =\ \mathsf{pc}(f)\ \vee \\
&\qquad\qquad (\exists\ c.\ \mathsf{exec}(CFS, c)\ =\ b\ \wedge \\
&\qquad\qquad\quad \mathsf{pc}(c)\ =\ \mathsf{pc}(f)\ \wedge \\
&\qquad\qquad\quad \langle CFS, a \rangle \xrightarrow[z]{w} c)\ \}^{+}
\end{aligned}$$

where

$$\begin{aligned}
w &= s\ \{\mathsf{pc} := \mathsf{pc}(s) + 1\} \\
z &= f\ \{\mathsf{pc} := \mathsf{pc}(f) - 1\}
\end{aligned}$$

Proof By rule induction on $\langle CFS, \sigma_0 \rangle \xrightarrow[f]{w} \sigma_n$. In both the Stop and Continue cases we now proceed by cases on the condition $\mathsf{pc}(\sigma_0)\ =\ \mathsf{pc}(f)$.

In the Stop case we have

$$\mathsf{exec}(CFS, \sigma_0)\ =\ \mathsf{Some}\ \sigma_n \tag{5.10}$$

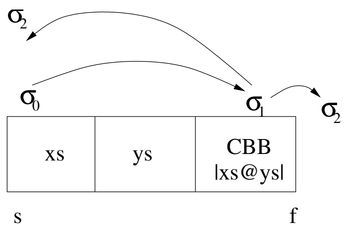

Figure 5.8: Relation “pivoting” on instruction at f

If $\mathsf{pc}(\sigma_0) = \mathsf{pc}(f)$, the lefthand disjunct of our goal holds. In the case where $\mathsf{pc}(\sigma_0) \neq \mathsf{pc}(f)$, we have a contradiction as $\mathsf{pc}(\sigma_n)$ is outside the block and is only reachable from the instruction at f.

For both truth values of $\mathsf{pc}(\sigma_0) = \mathsf{pc}(f)$ in the Continue case, we use the induction hypothesis, along with the standard Isabelle theorems for “unrolling” transitive closure relations and the construction rules for the block execution relation to show that the required result holds. ■

Lemma 64 (States in block relation imply states in big step relation)

$$\begin{aligned} &\forall CFS\ \sigma_0\ \sigma_n\ s\ f\ w\ x\ y.\ \langle CFS, \sigma_0\rangle \xrightarrow[f]{w} \sigma_n \quad \wedge \\ &\qquad \mathsf{well\_formed\_loop}\ CFS[s \ldots f] \longrightarrow \\ &\qquad (\sigma_0,\ \sigma_n) \in\ \{(a, b) :\ \exists\ c.\ \langle CFS, a\rangle \xrightarrow[l]{w}\ c\ \wedge \\ &\qquad\qquad \langle CFS, c\rangle \xrightarrow[f]{y}\ b\}^{+} \end{aligned}$$

where

$$\begin{aligned} w &= s\ \{\mathsf{pc} := \mathsf{pc}(s) + 1\} \\ x &= s\ \{\mathsf{pc} := \mathsf{pc}(s) + |xs|\} \\ y &= s\ \{\mathsf{pc} := \mathsf{pc}(s) + |xs| + 1\} \end{aligned}$$

PROOF From Lemma 63 ■

This result allows us to carry out the induction over the body of the loop (plus structure instructions), leading us to a proof of Proposition 2:

Theorem 8 (Invariant preserved by loop)

$$\begin{aligned}
&\forall\ \sigma_0\ \sigma_n\ CFS\ s\ f.\ \langle CFS, \sigma_0\rangle \xrightarrow[f]{s} \sigma_n \wedge \\
&\quad \mathsf{well\_formed\_loop}\ CFS[s \ldots f]\ \wedge \\
&\quad P\ \sigma_0\ \wedge \\
&\quad [\forall\ \sigma'_0\ \sigma'_1\ \sigma'_n\ CFS'\ w\ y\ z\ s'\ f'.\ \langle CFS', \sigma'_0\rangle \xrightarrow[x]{w} \sigma'_1\ \wedge \\
&\qquad \langle CFS', \sigma'_1\rangle \xrightarrow[f]{y} \sigma'_n\ \wedge \\
&\qquad \mathsf{well\_formed\_loop}\ CFS'[s' \ldots f']\ \wedge \\
&\qquad P\ \sigma'_0\ \wedge \\
&\qquad (\mathsf{sequence\_wp}\ ys\ \mathsf{branch\_cond})\ \sigma'_0\ \longrightarrow \\
&\qquad P\ \sigma'_n] \\
&\quad \longrightarrow\ P\ \sigma_n
\end{aligned}$$

where

$$\begin{aligned}
w &= s'\ \{\mathsf{pc} := \mathsf{pc}(s') + 1\} \\
x &= s'\ \{\mathsf{pc} := \mathsf{pc}(s') + |xs|\} \\
y &= s'\ \{\mathsf{pc} := \mathsf{pc}(s') + |xs| + 1\}
\end{aligned}$$

Proof By induction on the Bigstep relation (Definition 33), and from Lemma 60, Lemma 61, Lemma 62, Lemma 63, and Lemma 64. ■

5.4 Proof of Soundness of Loop Rule

In this section we describe the proof of soundness of the Loop Rule itself, beginning with the proof of some necessary lemmas.

Lemma 65 (First exception is None in exception free block)

$$\begin{aligned}
&\forall\ \sigma_0\ \sigma_n\ CFS\ s\ f.\ \langle CFS, \sigma_0\rangle \xrightarrow[f]{s} \sigma_n\ \wedge \\
&\quad CFS[s \ldots f]\ =\ xs\ \wedge \\
&\quad \mathsf{excep\_free}\ xs \\
&\quad \longrightarrow xp_0\ =\ \mathsf{None}
\end{aligned}$$

where

$$\sigma_0 \;=\; (xp_0,\; hp_0,\; frs_0)$$

Proof By case analysis of the block execution relation plus the definition of excep_free. ∎

Lemma 66 (Exception free implies final state non-terminating)

$$\begin{aligned}\forall\ \sigma_0\ \sigma_n\ \mathit{CFS}\ s\ f.\ \langle \mathit{CFS}, \sigma_0\rangle \xrightarrow[f]{s}\ \sigma_n\ \wedge \\ \mathit{CFS}[s \ldots f] \;=\; xs\ \wedge \\ \mathit{list\_all\_not\_shift\_frame}\ xs\ \wedge \\ \mathsf{excep\_free}\ xs \\ \longrightarrow \mathsf{not\_term\_state}\ \sigma_n\end{aligned}$$

Proof By induction on the block execution relation, and Lemmas 4 and 65. ∎

When the Loop Rule is rewritten with the definition of the pre- and post-condition relation described in Chapter 4 and simplified, it becomes a proof of the statement

$$(\mathsf{assert}\ P\ \wedge\ \neg\ \mathsf{sequence\_wp}\ ys\ \mathsf{branch\_cond}\ \wedge\ \mathsf{not\_term\_state})\ \sigma_n \tag{5.11}$$

under the assumptions

$$\begin{aligned}[\forall\ \mathit{CFS}\ \sigma_0\ \sigma_n\ s\ f.\ (\langle \mathit{CFS}, \sigma_0\rangle \xrightarrow[f]{s}\ \sigma_n\ \wedge \\ \mathit{CFS}[s \ldots\ f] \;=\; xs\ \wedge\ (\mathsf{assert}\ P)\ \sigma_0 \\ \longrightarrow (\mathsf{assert}\ P)\ \sigma_n]\end{aligned} \tag{5.12}$$

$$\langle \mathit{CFS}, \sigma_0\rangle \xrightarrow[f]{s}\ \sigma_n \tag{5.13}$$

$$\mathit{CFS}[s \ldots f] \;=\; [\mathit{UBF}\ |xs| + 1]@[xs]@[ys]@[\mathsf{CBB}\ |xs@ys|)] \tag{5.14}$$

$$\mathsf{well\_formed\_loop}\ \ [\mathsf{UBF}\ |xs| + 1]@[xs]@[ys]@[\mathsf{CBB}\ |xs@ys|)] \tag{5.15}$$

$$(\mathsf{assert}\ P)\ \sigma_0 \tag{5.16}$$

A proof of ($\neg$ $\mathsf{sequence\_wp}$ ys $\mathsf{branch\_cond}$) σ_n, can be obtained immediately by Theorem 7 and Assumptions 5.13, 5.14 and 5.15. Similarly, a proof of the third conjunct, $\mathsf{not\_term\_state}$ σ_n, can be achieved by Lemma 66 and Assumptions 5.13, 5.14 and 5.15.

The proof of (assert P) σ_n is, however, slightly more complex. We begin by resolving with Theorem 8 (after implication introduction), as its conclusion matches that of our current goal. After simplification, we require to prove

$$\begin{aligned}\forall\ \sigma'_0\ \sigma'_1\ \sigma'_n\ CFS'\ s'\ f'\ w\ x\ y\ y.\ & \langle CFS', \sigma'_0\rangle \xrightarrow[x]{w} \sigma'_1 \ \wedge \\ & \langle CFS', \sigma'_1\rangle \xrightarrow[f']{y} \sigma'_n \ \wedge \\ & \mathsf{well\_formed\_loop}\ CFS'[s' \ldots f'] \ \wedge \\ & (\mathsf{assert}\ P)\ \sigma'_0 \ \wedge \\ & (\mathsf{sequence\_wp}\ ys\ \mathsf{branch\_cond})\sigma'_0 \ \longrightarrow \\ & (\mathsf{assert}\ P)\ \sigma'_n\end{aligned} \tag{5.17}$$

where

$$\begin{aligned} w &= s'\,\{\mathsf{pc} := \mathsf{pc}(s') + 1\} \\ x &= s'\,\{\mathsf{pc} := \mathsf{pc}(s') + |xs|\} \\ y &= s'\,\{\mathsf{pc} := \mathsf{pc}(s') + |xs| + 1\} \end{aligned}$$

under Assumptions 5.12, 5.13, 5.14, 5.15 and 5.16. Further simplification transforms this into the problem of proving

$$(\mathsf{assert}\ P)\ \sigma'_n \tag{5.18}$$

under the assumptions

$$\begin{aligned}[\forall\ CFS\ \sigma_0\ \sigma_n\ s\ f.\ (\langle CFS, \sigma_0\rangle \xrightarrow[f]{s} \sigma_n \ \wedge & \\ & CFS[s \ldots\ f]\ =\ xs\ \wedge\ (\mathsf{assert}\ P)\ \sigma_0 \\ & \longrightarrow (\mathsf{assert}\ P)\ \sigma_n]\end{aligned} \tag{5.19}$$

$$\langle CFS, \sigma_0\rangle \xrightarrow[f]{s} \sigma_n \tag{5.20}$$

$$\mathsf{well\_formed\_loop}\ CFS[s \ldots f] \quad (\mathsf{assert}\ P)\ \sigma_0 \tag{5.22}$$

$$\langle CFS', \sigma'_0\rangle \xrightarrow[x]{w} \sigma'_1 \tag{5.23}$$

$$\langle CFS', \sigma'_0 \rangle \overset{y}{z}\xrightarrow[f]{} \sigma'_n \tag{5.24}$$

$$\mathsf{well\_formed\_loop}\ CFS'[s' \dots f'] \tag{5.25}$$

$$(\mathsf{assert}\ P)\ \sigma'_0 \tag{5.26}$$

$$(\mathsf{sequence\_wp}\ ys\ \mathsf{branch\_cond})\sigma'_0 \tag{5.27}$$

Working with these assumptions, we begin by proving that the invariant holds in state σ'_1, after one execution of the loop body:

Lemma 67 (Preservation of invariant across body of loop)

$$\begin{aligned}\forall\ CFS'\ \sigma'_0\ w\ x\ s'\ f'.\ [\forall\ CFS\ \sigma_0\ \sigma_n\ s\ f.\ & \langle CFS, \sigma_0 \rangle \xrightarrow[f]{s} \sigma_n\ \wedge \\ & CFS[s \dots\ f]\ =\ xs\ \wedge\ (\mathsf{assert}\ P)\ \sigma \\ & \longrightarrow (\mathsf{assert}\ P)\ \sigma_n]\ \wedge \\ & \mathsf{well\_formed\_loop}\ CFS'[s' \dots f']\ \wedge \\ & \langle CFS', \sigma'_0 \rangle \xrightarrow[x]{w} \sigma'_1\ \wedge \\ & (\mathsf{assert}\ P)\ \sigma'_0 \\ & \longrightarrow\ (\mathsf{assert}\ P)\ \sigma'_1\end{aligned}$$

where

$$w = s'\ \{\mathsf{pc} := \mathsf{pc}(s') + 1\}$$
$$x = s'\ \{\mathsf{pc} := \mathsf{pc}(s') + |xs|\}$$

PROOF From instantiation of the universally quantified variables CFS, σ_0, σ_n, s, f in Assumption 5.19 to CFS', σ'_0, σ'_1, k, l, followed by simplification.■

We now need to show that the invariant is preserved by the block $\langle CFS', \sigma'_1 \rangle \xrightarrow[f']{y} \sigma'_n$, representing execution of the instructions $ys@[(CBB\ |xs@ys|)]$.

The definition of a well-formed loop (Definition 31) tells us that ys is simple, and we begin by showing that this means that the block $\langle CFS, \sigma'_1 \rangle \xrightarrow[f']{y} \sigma'_n$ can be split into a smaller block representing the execution of ys, followed by the execution of the conditional branch.

Lemma 68 (Final state reached by execution of branch)

$$\begin{aligned}\forall\ \mathit{CFS}'\ \sigma'_1\ \sigma'_n\ s'\ x\ f'\ xs\ ys.\ \langle \mathit{CFS}', \sigma'_1\rangle \xrightarrow[f']{s'} \sigma'_n\ \wedge\\ \mathsf{inside}\ s'\ x\ \sigma'_1\ \wedge\\ xs\ \neq [\,]\ \wedge\ ys\ \neq [\,]\ \wedge\\ \mathit{CFS}'[s' \ldots\ f']\ =\ ys@[(\mathit{CBB}\ |xs@ys|)] \longrightarrow\\ \exists \sigma_2.\ \langle \mathit{CFS}', \sigma'_1\rangle \xrightarrow[x]{s'} \sigma'_2\ \wedge\\ \mathsf{exec}(\mathit{CFS}', \sigma'_2)\ =\ \mathsf{Some}\ \sigma'_n\ \wedge\\ \mathsf{pc}(\sigma'_2)\ =\ \mathsf{pc}(f')\end{aligned}$$

where

$$x = s'\ \{\mathsf{pc} := \mathsf{pc}(s') + |xs|\}$$

Proof From the definition of simple (Definition 17), the only position outside ys that it is possible to reach from within ys is the instruction at f. Therefore the final state, σ'_n, must be reached by executing the instruction at f. We know that σ'_2 is inside the instructions ys as it is reached by executing the simple block xs and so its program counter must be equal to $\mathsf{pc}(s)\ +\ \mathsf{length}\ xs\ + 1$. State σ'_1 is the state reached by executing the loop once (as mentioned in the previous lemma) and so is inside the block xs. ∎

This gives us (assert P) σ'_n from Lemma 51, and we are now able to prove the Loop Rule.

Theorem 9 (Loop Rule)

$$\frac{\begin{array}{l}\{P\ \wedge\ \mathsf{sequence\_wp}\ ys\ \mathsf{branch\_cond}\}\\ \quad xs\\ \{P\}\\ \mathsf{well\_formed\_loop}\ [(\mathit{UBF}\ |xs|\ +\ 1)]@[xs]@[ys]@[(\mathit{CBB}\ |xs@ys|)]\end{array}}{\begin{array}{l}\{P\}\\ \quad [(\mathit{UBF}\ |xs|\ +\ 1)]@[xs]@[ys]@[(\mathit{CBB}\ |xs@ys|)]\\ \{P \wedge \neg(\mathsf{sequence\_wp}\ ys\ \mathsf{branch\_cond}\ \wedge\ \mathsf{not\_term\_state}\}\end{array}}$$

Proof From Theorem 7, Theorem 8, Lemma 51, Lemma 66, Lemma 67, and Lemma 68. ∎

5.5 Conclusions

Despite the superficial similarity of the rule for loops in bytecode to the while rule of the conventional Hoare logic, the proof of soundness for the former is a great deal more complex than that of the latter. This is due mainly to two points: the need to impose a structure on a 'flat' bytecode program, and the granularity or step-size of our execution relations.

Conventional Hoare logics are normally based on an operational semantics defined in terms of a small number of *commands*. The semantics consist of rules for how these commands may be combined to form other commands, and these map easily onto the corresponding rules in the Hoare logic. Additionally, each command can be viewed as a separate entity, independent of its context. In the bytecode world, the commands involved relate to very small steps of execution, and there is no concept of combining commands to create other commands. A program has no structure, it is simply a list. This means that in order to prove more complex properties of bytecode programs, we are compelled to impose a structure on the bytecode which, to some extent, does not really exist. Also, in order to reflect the 'real world' nature of the JVM, instructions are not independent entities, but must be viewed within the context of a classfile.

When it comes to proving properties of our imposed higher level structures that involve induction, such as in the case of the preservation of an invariant by a loop, we again come up against the problem of the lack of any sort of 'combinatory' property of bytecode commands. In the conventional logic, the body of a loop is a command and so, using an inductively defined semantics based on the idea of the execution of commands, we are able to induct over the body of any loop regardless of how it is formed. In the bytecode world, the only situation in which we could induct over the body of a loop using the execution relations defined at the level of execution of a bytecode command, is if the body only contained one instruction. Again, there is no notion of two bytecode instructions combining to become another type of bytecode instruction, they will merely form a list of two instructions. Consequently, there is no way of proving that any property is preserved by one step of execution of the body, as there is no relation which describes this concept.

As we have demonstrated, these problems can be overcome and indeed quite elegant solutions found. The idea of data-equality of states deals with the problem of executing the structure of higher level patterns explicitly. This brings the proof more in line with that of the conventional logic, where such 'control' structure as `if` and `while` are explicitly present in the rules by nature of their formulation in terms of syntax. The definition of execution of a block of bytecode in terms of 'big-steps' again puts us in a situation parallel to the conventional world, where we are able to talk about the execution of a 'compound' command in one step.

We have shown that, despite the apparent mismatch between the structured language of the conventional Hoare logic and the flat unstructured world of bytecode, and the considerable challenges it represents, it is possible to reconcile the two. In addition, we have been able to produce a workable logic that will allow proofs of simple bytecode programs including loops.

Chapter 6

Verification Example

In this chapter we present a proof involving a simple bytecode program containing a loop, demonstrating the bytecode programming logic in use. We also discuss why, while is would be possible to prove that the array bounds checks for another small program could be eliminated, it is not currently practical to do so.

6.1 While Program

This section details a proof involving the small loop program from Chapter 4:

```
public class SimpleWhile {
public static void  main(String args[])
  {
    int i=0;
    while (i<5)
      { i++; }
  }
}
```

with corresponding bytecode

```
   0 bipush 0
   1 istore 1
   2 goto 8
```

```
 5 iinc 1 1
 8 iload 1
 9 bipush 5
10 if_icmplt 5
13 return
```

We wish to prove that, if execution terminates, the value stored in variable `i` is 5. In terms of the bytecode program, we want to show that the final value in local variable 1 (written *loc* ! 1) is equal to 5. We omit the `return` instruction as the bytecode logic currently does not allow method call or return.

Proposition 4 (Proof of a small program with loop)

$$\{\lambda\ \sigma.\ (\mathsf{let}\ (stk, loc, cn, ml, pc)\ =\ \mathsf{hd}\ (\mathsf{snd}(\mathsf{snd}\ \sigma))\ \mathsf{in}$$
$$1\ <\ \mathsf{length}\ loc\ \wedge\ xp\ =\ \mathsf{None}\ \wedge\ \mathsf{frs}\ \neq\ [\ ])\}$$

$$[\mathsf{LAS\ bipush}\ 0,\ \mathsf{LAS\ IAstore}\ 1,$$
$$\mathsf{UBF}\ (\mathsf{Goto\_fwd}\ 2),\ \mathsf{LAS}\ (\mathsf{Iinc}\ 1\ 1),\ \mathsf{LAS}\ (\mathsf{iload}\ 1),$$
$$\mathsf{LAS}\ (\mathsf{Bipush}\ 5),\ \mathsf{CBB}\ (\mathsf{Ificmplt\_bwd}\ 3)]$$

$$\{\lambda\ \sigma.\ (\mathsf{let}\ (stk, loc, cn, ml, pc)\ =\ \mathsf{hd}\ (\mathsf{snd}(\mathsf{snd}\ \sigma))\ \mathsf{in}$$
$$1\ <\ \mathsf{length}\ loc\ \wedge\ (loc\,!\,1)\ =\ 5)\}$$

We begin by considering the part of the program containing the loop:

```
 2 goto 8
 5 iinc 1 1
 8 iload 1
 9 bipush 5
10 if_icmplt 5
```

We take the loop invariant to be

$$\begin{aligned}\lambda\ \sigma.\ (\mathsf{let}\ (stk, loc, cn, ml, pc)\ =\ \mathsf{hd}(\mathsf{snd}(\mathsf{snd}\ \sigma)\ \mathsf{in}\\ \mathit{1}\ <\ \mathsf{length}\ loc\ \wedge\ (loc\,!\,1) \leq 5)\end{aligned} \tag{6.1}$$

and want to show that this is preserved across the body of the loop. The term $1\ <\ \mathsf{length}\ loc$ is included because the local variable are represented in the operational semantics as a list—with indices 0, 1, 2, ...—but the Java compiler starts indexing the local variables from 1. This means that, in order for us to be able to carry out proofs on the list of local variables in Isabelle, we must know that the length of the list is at least one greater than the highest indexed variable we refer to.

The loop guard is

$$\begin{aligned}&\mathsf{sequence\_wp}\ [\ \mathsf{LAS}\ (\mathsf{iload}\ 1),\ \mathsf{LAS}\ (\mathsf{Bipush}\ 5)]\\ &\qquad(\lambda\ \sigma.\ \mathsf{let}\ (stk, loc, cn, ml, pc)\ =\ \mathsf{hd}\ (\mathsf{snd}(\mathsf{snd}\ \sigma))\ \mathsf{in}\\ &\qquad\qquad(\mathsf{hd}(\mathsf{tl}\ stk)) < \mathsf{hd}\ stk))))\end{aligned} \tag{6.2}$$

and we want to show that this is false on exit from the loop.

We begin by calculating the sequence weakest precondition (Definition 30) of the instructions [LAS (iload 1), LAS (Bipush 5)] which set up the stack for the conditional branch.

Lemma 69 (Example calculation of swp of list)

$$\begin{aligned}&\forall\ \sigma.\ \mathsf{not\_term\_state}\ \sigma \longrightarrow\\ &[(\mathsf{sequence\_wp}\ [\ \mathsf{LAS}\ (\mathsf{iload}\ 1),\ \mathsf{LAS}\ (\mathsf{Bipush}\ 5)]\\ &\qquad(\lambda\ \sigma.\ \mathsf{let}\ (stk, loc, cn, ml, pc)\ =\ \mathsf{hd}\ (\mathsf{snd}(\mathsf{snd}\ \sigma))\ \mathsf{in}\\ &\qquad\qquad(\mathsf{hd}(\mathsf{tl}\ stk)) < \mathsf{hd}\ stk)))\ \sigma\\ &\equiv\\ &(\lambda\ \sigma.\ \mathsf{let}\ (stk, loc, cn, ml, pc)\ =\ \mathsf{hd}\ (\mathsf{snd}(\mathsf{snd}\ \sigma))\ \ \mathsf{in}\ (loc\ !\ \mathit{1}) < \mathit{5})\ \sigma]\end{aligned} \tag{6.3}$$

Proof By Lemma 43, Lemma 49, and lemmas for weakest precondition of iload and bipush. ■

We now show that execution of the body of the loop preserves the invariant, as required by the loop rule:

Lemma 70 (Loop body preserves invariant)

$$\begin{aligned}&\{\lambda\ \sigma.\ (\mathsf{let}\ (stk, loc, cn, ml, pc)\ =\ \mathsf{hd}\ (\mathsf{frames}\ (\sigma))\ \mathit{in}\\ &\qquad \mathit{1}\ <\ \mathsf{length}\ loc\ \wedge\ (loc!\mathit{1}) \le \mathit{5}\ \wedge\ (loc!\mathit{1}) < \mathit{5})\}\\ &\\ &[\mathsf{LAS}\ (\mathsf{Iinc}\ \mathit{1}\ \ \mathit{1})]\\ &\\ &\{\lambda\ \sigma.\ (\mathsf{let}\ (stk, loc, cn, ml, pc)\ =\ \mathsf{hd}\ (\mathsf{frames}\ (\sigma))\ \mathit{in}\\ &\qquad\qquad \mathit{1}\ <\ \mathsf{length}\ loc\ \wedge\ (loc!\mathit{1}) \le \mathit{5})\}\end{aligned} \tag{6.4}$$

PROOF By the weakest precondition of the Iinc instruction, and simplification. ■

These lemmas now allow us to prove the following lemma about the loop section of the program

Lemma 71 (Proof of correctness—loop section)

$$\{(\lambda\ \sigma.\ (\mathsf{let}\ (stk, loc, cn, ml, pc)\ =\ \mathsf{hd}\ (\mathsf{frames}\ (\sigma))\ \mathsf{in}$$
$$\mathit{1}\ <\ \mathsf{length}\ loc\ \wedge\ (loc!\ \mathit{1}) \leq \mathit{5}\ \wedge\ (loc\ !\ \mathit{1}) < \mathit{5}\}$$

$$[\mathsf{UBF}\ (\mathsf{Goto\_fwd}\ 2),\ \mathsf{LAS}\ (\mathsf{Iinc}\ 1\ 1),\ \mathsf{LAS}\ (\mathsf{iload}\ 1),$$
$$\mathsf{LAS}\ (\mathsf{Bipush}\ 5),\ \mathsf{CBB}\ (\mathsf{Ificmplt\_bwd}\ 3)]$$

$$\{(\lambda\ \sigma.\ (\mathsf{let}\ (stk, loc, cn, ml, pc)\ =\ \mathsf{hd}\ (\mathsf{frames}\ (\sigma))\ \mathsf{in}$$
$$\mathit{1}\ <\ \mathsf{length}\ loc\ \wedge\ (loc!\ \mathit{1}) \leq \mathit{5}\}$$

PROOF By Lemma 70 and the Theorem 9 (Loop Rule). ■

We are also able to prove that, on termination of the loop, the desired postcondition holds.

Lemma 72 (Invariant and negation of guard implies postcondition)

$$\forall\sigma.\ (\lambda\ \sigma.\ (\mathsf{let}\ (stk, loc, cn, ml, pc)\ =\ \mathsf{hd}\ (\mathsf{frames}\ (\sigma))\ \mathsf{in}$$
$$\mathit{1}\ <\ \mathsf{length}\ loc\ \wedge\ (loc!\ \mathit{1}) \leq \mathit{5}\ \wedge\ \neg\ (loc\ !\ \mathit{1}) < \mathit{5}$$
$$\wedge\ \mathsf{not\_term\_state}\ \sigma))\ \sigma$$
$$\Longrightarrow (\lambda\ \sigma.\ (\mathsf{let}\ (stk, loc, cn, ml, pc)\ =\ \mathsf{hd}\ (\mathsf{frames}\ (\sigma))\ \mathsf{in}$$
$$\mathit{1}\ <\ \mathsf{length}\ loc\ \wedge\ (loc!\mathit{1}) = \mathit{5}))\ \sigma$$

PROOF By Lemma 28 (Postcondition Weakening) and simplification. ■

We now return to the first section of the program:

```
0 bipush 0
1 istore 1
```

and show that, assuming the desired precondition, execution of these instructions results in the loop invariant

Lemma 73 (Precondition leads to invariant)

$\{\lambda\ \sigma.\ 1\ <\ \mathsf{length}\ loc\ \wedge\ xp\ =\ \mathsf{None}\ \wedge\ (\mathsf{frames}\ (\sigma))\ \neq\ [\,]\}$
$(\mathsf{let}\ (stk, loc, cn, ml, pc)\ =\ \mathsf{hd}\ (\mathsf{frames}\ (\sigma))\ \mathsf{in}$

$[\mathsf{LAS\ bipush}\ 0,\ \mathsf{LAS\ IAstore}\ 1]$

$\{(\lambda\ \sigma.\ (\mathsf{let}\ (stk, loc, cn, ml, pc)\ =\ \mathsf{hd}\ (\mathsf{frames}\ (\sigma))\ \mathsf{in}$
$\mathit{1}\ <\ \mathsf{length}\ loc\ \wedge\ (loc!\ \mathit{1}) \leq \mathit{5}))\}$

Proof By Theorem 4 (Sequencing Rule) and lemmas for the weakest preconditions of the instructions bipush and IAstore. ■

We are now able to prove Proposition 4:

Theorem 10 (Proof of simple incrementation while loop program)

$\{\lambda\ \sigma.\ (\mathsf{let}\ (stk, loc, cn, ml, pc)\ =\ \mathsf{hd}\ (\mathsf{frames}\ (\sigma))\ \mathsf{in}$
$1\ <\ \mathsf{length}\ loc\ \wedge\ xp\ =\ \mathsf{None}\ \wedge\ (\mathsf{frames}\ (\sigma))\ \neq\ [\,])$

$[\mathsf{LAS\ bipush}\ 0,\ \mathsf{LAS\ IAstore}\ 1, \mathsf{UBF}\ (\mathsf{Goto\_fwd}\ 2),\ \mathsf{LAS}\ (\mathsf{Iinc}\ 1\ 1),\ \mathsf{LAS}\ (\mathsf{iload}\ 1),$
$\mathsf{LAS}\ (\mathsf{Bipush}\ 5),\ \mathsf{CBB}\ (\mathsf{Ificmplt\_bwd}\ 3)]$

$\{\lambda\ \sigma.\ (\mathsf{let}\ (stk, loc, cn, ml, pc)\ =\ \mathsf{hd}\ (\mathsf{frames}\ (\sigma))\ \mathsf{in}$
$1\ <\ \mathsf{length}\ loc\ \wedge\ (loc\,!\,1)\ =\ 5)\}$

Proof By the Sequencing Rule (Theorem 4), Lemma 72 and Lemma 73. ■

6.2 Array Bounds Elimination

After the proof of a small program containing a loop, our intention was to prove that it was safe to eliminate the array bounds checks on a program containing a loop which updated an array:

```
public class Arraybounds {

       public static void main(String args[]) {
               int i=0;
               int myarray[] = new int[5];
               while (i< 5) {
                    myarray[i] = 2;
                    i++;
               }
       }
 }
```

with bytecode

```
   0 iconst_0
   1 istore_1
   2 iconst_5
   3 newarray int
   5 astore_2
   6 goto 16
   9 aload_2
  10 iload_1
  11 iconst_2
  12 iastore
  13 iinc 1 1
  16 iload_1
  17 iconst_5
  18 if_icmplt 9
  21 return
```

Our aim is to prove that, immediately before an array update operation the array address is non-null and the index is within bounds. However, the array update instruction is in the *middle* of the body of the loop. As we can only say with certainty that the loop invariant holds at the beginning and end of the loop body, we cannot ensure that a loop invariant that included our desired property will hold just before the array update instruction.

The ideal solution to this problem would be to introduce some kind of *assertion* statements to the logic. These would allow us to state that certain properties hold at various intermediate points in the code. But this would entail a fairly major addition to the logic, and is therefore included in the "Further Work" section of Chapter 7.

A more ad hoc solution would be to alter the bytecode of the program slightly so that the array update instruction appeared at the beginning of the loop body, where the loop invariant *would* hold.

```
 0 iconst_0
 1 istore_1
 2 iconst_5
 3 newarray int
 5 astore_2
 6 aload_2
 7 iload_1
 8 iconst_2
 9 goto 16
10 aload_2
11 iload_1
12 iconst_2
13 iastore
14 iinc 1 1
16 iload_1
17 iconst_5
18 if_icmplt 9
21 return
```

But while this would produce a program that our current programming logic could deal with, it seems to defeat the purpose of our aim of smaller, more efficient bytecode programs as the "provable" bytecode program is three instructions longer than the original program.

In addition to this, the proof involving the smaller loop program of seven instructions is in the region of 400 lines long. As the longer proof involves a program containing almost three times as many instructions, it is likely to be around 1,200 lines long.

Therefore, while it would be *feasible* to carry out an array bounds elimination with the logic in its current state, it can be argued that it is not really *reasonable*. Rendering such proofs managable in practice is likely to require two features. Firstly, the addition of assertion statements to the logic to avoid the necessity of reordering bytecode instructions, and secondly, a tactic capable of taking as arguments a list of instructions, a precondition, and a postcondition and carrying out the proof using the rules of the programming logic. A description of such a tactic can be found in [16].

6.3 Conclusions

Although we have not successfully managed to prove a property more related to JIT optimization, the above proof demonstrates that we have been successful in constructing a framework in which it is possible to prove properties of bytecode programs. With the addition of the features discussed in Section 7.2—most of them already documented in other work— it would be possible to prove many properties necessary for JIT optimization.

Chapter 7

Conclusions

Our aim was to develop a way of proving properties of bytecode programs that would allow JIT compilers to make optimizations not currently possible.

In chapter 3 we give the definitions of a number of relations for the execution of bytecode programs, and describe the proof of some related lemmas and theorems. The concept of data-equality is introduced as a method of comparing the parts of two JVM states not related to position within a particular classfile.

These formalized concepts form the basis for the subsequent development of several rules constituting a programming logic for bytecode programs. The derivation of these rules is presented in Chapter 4, and the idea of data-equality is used again, this time to overcome the problems inherent in having to execute the structure of loops and conditional sequences explicitly.

In chapter 5 we present a proof of the soundness of our proposed rule for loops, contrasting its complexity with that of the same proof in conventional Hoare logics.

Finally, in chapter 6 we present a proof of a simple bytecode program containing a loop, thereby demonstrating the use of our programming logic.

7.1 Bytecode Proof and Mechanized Reasoning

In this section we discuss our experiences of the difficulties inherent in carrying out proof at the level of bytecode instructions, along with the benefits and disadvantages of using a mechanized proof tool.

The use of a mechanized proof tool is central to our results. It has the benefit of enabling us to keep track of very complex proofs involving many definitions. Additionally, as mentioned before, it provides an additional degree of confidence in the validity of these proofs. The Isabelle system was considered particularly suitable for our work as it facilitates the definition of logics and subsequent proofs involving them. In the course of the work, however, we encountered several difficulties which offset these advantages.

7.1.1 Size and Complexity of Proofs

As previously discussed, bytecode programs lack the sort of syntactic structure present in the higher level languages for which Hoare logics are more usually defined. This means that rather than recognising, for example, the keyword `while` and applying the relevant rule, we must identify 'structural instructions' within a bytecode pattern, check that they conform to certain constraints, and explicitly execute them. This results in a great deal of proof in addition to that necessary in the conventional logics (c.f. Chapter 5).

Possibly the main difficulty we encountered in the course of this work was the sheer length and complexity of the proofs involved. Although the concepts behind the proofs can be communicated in a fairly high-level way to human beings—as we hope we have demonstrated in the preceding chapters—this approach cannot be applied to the Isabelle definitions and proof scripts. The JVM world is very detailed; it contains a great deal of information and the Isabelle model must reflect this. It means, however, that there can be no 'glossing over' of the details, and every inference—however small—must be spelled out.

The proofs of the various theorems in this report each run to several hundred lines of code, not including the necessary lemmas. The files related to the soundness of the while rule contain in the region of 10,000 lines of code. The complete count for the whole logic is around 22,000 lines. It is likely that this could be reduced to some extent by packaging repeated patterns of proof as tactics, or by more effective use of the automatic tactics. It gives an idea, however of the amount of detail involved in the proofs.

One feature that would be invaluable with proofs of this length would be the ability to save only the *successful* commands in a proof session. The basic Isabelle interface relies on the user remembering to note down every successful command—and remove every unsuccessful command—used to achieve a proof in a text editor. This is reasonable with smaller proofs, but in those running to several hundred lines it is all too easy to make mistakes. Unfortunately even one instruction missed out or left in unintentionally may necessitate running through the entire proof in small steps in order to identify the error.

Fortunately, Proof General [7], the most recently available interface for use with Isabelle, has a mechanism for "locking" commands in a text editor as they are successfully executed by the prover. It is our opinon that improved user interfaces to proof tools will become essential as proofs get larger and more complex in order to deal with real world systems (c.f. the ESC and LOOP projects). Certainly it is the case that no team of software engineers would attempt to carry out sizeable projects with only a compiler and a text editor.

7.1.2 Proofs Involving Lists

Another drawback to the structureless form of bytecode programs is the necessity of dealing with a large number of proofs involving lists. Normally this would not be a problem: list properties can usually be proved by induction, and the Isabelle distribution already contains many lemmas about lists.

Unfortunately the lists of instructions we are interested in are often not lists in their own right as such, but slices of a larger list (Definition 6). As we are, in a sense, coming at the list from both ends we cannot use induction: if we induct on either the start or end position we change the length of our list; if we try to induct on the slice itself we upset the relationship between the start and end points. This means that we must rely on rewriting with the various lemmas for take and drop, which can result in some quite tricky proofs.

7.1.3 Automatic Tactics

The fact that instructions are not viewed as independent entities, but rather must be extracted from a set of classfiles and a state, means that a lot of information is contained within the assumptions of each proof. The block execution relation involves two states each consisting of three elements, a set of classfiles, two class name identifiers, two method identifiers, and two program counter values. So an assumption or definition involving quantification over these variables requires thirteen instantiations. Often there are too many possibilities for Isabelle's resolution tactics to find these instantiations automatically, so each variable must be instantiated by hand.

The large number of assumptions in many of the proofs also frequently confuses the automatic tactics. It is often the case that the simplifier will get nowhere with a particular goal if it contains many assumptions not pertinent to the desired conclusion. But if the relevant assumptions are extracted and used in the goal of a separate lemma the tactics succeed almost immediately.

This may well be a problem that is solved in more recent versions of Isabelle; in particular the rewriting of `asm_full_simp_tac` so that results do not depend on the order in which the assumptions appear might well have a significant effect on this problem. Unfortunately, one of the more recent versions of Isabelle made significant changes which would have necessitated changes in Pusch's formalisation of the semantics, and consequently we made the decision to stay with our current version of the prover (Isabelle 99) despite the improved features of newer versions.

The size and complexity of our proofs pushed the computing power available to us to its limits. The memory requirements of our proofs often exceeded the 256Mb of RAM available to us, causing Isabelle to crash.

These difficulties call into question the wisdom of attempting proofs of bytecode programs. But, as mentioned before, despite its drawbacks, the stack based virtual machine appears to be here to stay, at least for the foreseeable future. Therefore the ability to carry out proof at this level has definite value.

7.2 Further Work

Having developed a simple programming logic for bytecode programs, there are a number of ways in which it could be extended. The most obvious first step is the extension of the operational semantics to include all bytecode instructions available in the JVM, rather than the subset currently treated. One possibility would be to transfer the underlying formalization on which our work is based to the μJava theories [39], which include a larger set of instructions and deal with exception handling.

In terms of the bytecode execution relations, the main drawback is the restriction that states must be all in the same method of a particular class. This means that it is impossible to work with bytecode programs that include method invocation or return, which is clearly not realistic. One possible way of lifting this restriction might be to in effect *inline* the code of the method being called (Section 1.2), which would result in a larger block that included the code of all methods called.

Alternatively, as the frame of the method in which we start a block remains on the frame stack when a new method is called, we could require that rather than all classes and methods being equal across a path of execution that there exists a chain through various method calls, returning to the calling method. That is

$$\begin{aligned}\mathsf{in\_frame\_stack}\ s\ (xp, hp, fr : frs) \ \equiv\ & \mathsf{same\_method}\ s\ (xp, hp, fr) \vee \\ & \mathsf{in\_frame\_stack}\ s\ (xp, hp, frs)\end{aligned}$$

$$\begin{aligned}\mathsf{same\_method\_path}\ s\ \sigma_0\ \sigma_n\ f \ \equiv\ & \mathsf{in\_frame\_stack}\ s\ \sigma_0 \wedge \\ & \mathsf{same\_method}\ s\ \sigma_n\end{aligned}$$

The fact that our programming logic relies on the code that is being executed never throwing exceptions is again unrealistic in a real world situation. Consequently, another useful extension would involve modelling Java's exception handling method in the operational semantics, and altering the programming logic in such a way that it allows us to reason about programs that terminate abruptly as a result of exceptions being raised. The logic described by Jacobs in [21] has this ability.

Finally, it would be useful to add *assertion* statements to the programming logic. This would allow the proof of assertions at intermediate points in a program, rather than just the start and finish. This would facilitate the proof of more complex loop programs, as discussed Section 6.2.

In terms of incorporating bytecode proof into a working system, it would obviously be unreasonable to invoke Isabelle at runtime. Also, any system requiring much extra work from users is unlikely to be popular, and so a model similar to that used by ESC where user input to the proof process is minimal and viewed as a kind of “advanced typechecker” would be preferable.

Another possibility would be to build on either the Annotated JIT project where bytecode programs arrive with optimizing annotations than can be used by an annotation aware JIT compiler. Obviously this raises the question of whether or not the annotations can be trusted. Therefore it seems likely that some sort of digital signature might be required here, or possibly a proof checker like that used in proof carrying code systems. Alternatively, it might be possible to prove that for certain patterns of bytecode particular optimizations are safe, e.g. a loop of *this* form including an array operation can have the array bounds check eliminated. These ’proved patterns’ could be stored as a library against which incoming programs could be compared and action taken by the JIT accordingly.

7.3 Contribution

We have demonstrated that it is possible to define a programming logic for bytecode programs that allows the proof of bytecode programs containing loops. The instructions available for use in the programs are currently limited, but the basis is in place for extension.

The development of this logic was not by any means straightforward. It required the definition of several execution relations for bytecode programs, each necessary for proofs of different aspects of execution. In addition, the flat, unstructured nature of bytecode programs presents a number of difficulties, particularly when reasoning about loops. But there are, as we demonstrate, some quite elegant solutions to these problems.

While it would be possible to use the logic in its current state to prove properties that would allow the speeding up of bytecode programs, we believe that such proofs would be of an unreasonable size and complexity to carry out in practice. The addition of assertion statements to the logic and the creation of an Isabelle tactic capable of automating such proofs would be necessary to render such proofs managable.

7.4 Concluding Remarks

In the course of this work we have demonstrated that it is possible to carry out proof at the level of Java bytecode instructions. In order to reach this state we have been forced to consider and find solutions for some hard problems, involving some long and complex proofs which pushed the bounds of what our mechanized proof tool was able to handle. The resulting programming logic, while not complete, provides a firm basis that with some extension should allow the proofs of bytecode programs necessary to allow several JIT optimizations.

Appendix A

Extension of LoadAndStore.thy

```
LoadAndStore = Runtime +

(** load and store instructions transfer values between local variables
    and operand stack **)

datatype load_and_store =
  IAload  ins_type nat  ("_ load _" 30)
(* load int/ref from local variable *)
| IAstore ins_type nat  ("_ store _" 30)
(* store int into/ref local variable *)
| Bipush int
(* push int *)
| Aconst_null
(* push null *)
| Iinc nat int
(*increment local var by int *)
| Iadd
(*add two integers at top of stack *)

consts
 exec_las :: "[load_and_store,opstack,locvars,p_count] =>
(opstack * locvars * p_count)"

primrec

 "exec_las (X load idx) stk vars pc =
 ((vars ! idx) # stk , vars , pc+1)"

 "exec_las (X store idx) stk vars pc =
 (tl stk , vars[idx:=hd stk] , pc+1)"
 "exec_las (Bipush ival) stk vars pc =
  (Intg ival # stk , vars , pc+1)"

 "exec_las Aconst_null stk vars pc =
  (Null # stk , vars , pc+1)"

 "exec_las (Iinc idx ival) stk vars pc =
 (stk, vars[idx:= (Intg(get_Intg(vars ! idx) + ival))], pc+1)"

 "exec_las Iadd stk vars pc =
  (Intg ((get_Intg (hd stk)) +
  (get_Intg (hd(tl stk)))) # (tl(tl stk)) , vars , pc+1)"

end
```

Appendix B

Isabelle .thy Files

B.1 ListSlice.thy

```
ListSlice = HOL + List + Nat +

(* mid m n xs  consists of n elements of xs from position m
onwards. Precondition is that m + n <= length xs. *)

constdefs mid :: "nat => nat => 'a list => 'a list"
"mid m n xs == take n (drop m xs)"

(* fromto s f xs  consists of the elements of xs from position s to
position f inclusive. Precondition is that s < length xs &
 f < length xs *)

constdefs fromto :: "nat => nat => 'a list => 'a list"
"fromto s f xs == mid s (Suc (f - s)) xs"

end
```

B.2 State_parts.thy

```
State_parts =HOL + Exec +
types

 boundary = "cname*(mname*param_desc)*p_count"

 minframe = "opstack *locvars *p_count"



minstate = "(xcpt option  *minframe)"

constdefs

  get_stk :: frame => opstack
 "get_stk ==  %(stk,loc,cn,ml,pc). stk"

  get_loc :: "frame => locvars"
 "get_loc == %(stk,loc,cn,ml,pc). loc"

  get_cn :: "boundary => cname"
 "get_cn == %(cn,ml,pc). cn"

 cn_of:: "boundary => cname"
 "cn_of strt  == let (cn,ml,pc) = strt in cn"


 st_cn_of:: "frame=> cname"
 "st_cn_of st  == let (stk, loc, cn,ml,pc) = st in cn"

  stk_of :: "frame => opstack"
  "stk_of st == let (stk, loc, cn,ml,pc) = st in stk"

  loc_of :: "frame => locvars"
"loc_of st == let (stk, loc, cn,ml,pc) = st in loc"

  st_pc_of :: "frame => p_count"
  "st_pc_of st == let (stk, loc, cn,ml,pc) = st in pc"


  get_ml :: "boundary => method_loc"
 "get_ml == %(cn,ml,pc). ml"

  ml_of :: "boundary => method_loc"
 "ml_of s  == let (cn,ml,pc) = s in ml"

  st_ml_of :: "frame => method_loc"
 "st_ml_of st  == let (stk,loc, cn,ml,pc) = st in ml"

  get_pc :: "frame => p_count"
 "get_pc == %(stk,loc,cn,ml,pc). pc"

  pc_equals :: "[frame, p_count] => bool"
  "pc_equals frs pca == let (stk,loc,cn,ml ,pc) = frs in
                        pc = pca"

  pc_of :: "boundary => p_count"
 "pc_of == %(cn,ml,pc). pc"

  same_method_bounds :: "[boundary, boundary] => bool"
  "same_method_bounds s f == (cn_of s = cn_of f) &
                             (ml_of s = ml_of f)"

same_method_frs :: "[boundary, frame,frame, boundary] => bool"
  "same_method_frs s a b f == ((st_cn_of a = st_cn_of b) &
         (st_cn_of a = cn_of s) & (st_cn_of a  = cn_of f)) &
         ((st_ml_of a = st_ml_of b) & (st_ml_of a = ml_of s)  &
         (st_ml_of a = ml_of f))"



inlist:: "[instr list, minframe] => bool"
"inlist xs a == let (stk,loc,pc) = a in
                    0 <= pc & pc <= ((length xs) - 1)"


outlist ::"[instr list, minframe] => bool"
"outlist xs a == let (stk,loc,pc) = a in
                   (length xs) <= pc"


  third_of :: "(xcpt option * heap *frame list) =>frame list"
  "third_of == % (xp, hp, frs). frs"



  inside :: "[frame list, boundary , boundary] => bool"
  "inside frs s f == let (stk,loc,cn,ml,pc) = hd frs in

                   (pc_of s <= pc & pc <= pc_of f)"

 outside :: "[frame list, boundary, boundary] => bool"
  "outside frs s f == let (stk,loc,cn,ml,pc) = hd frs  in

                   (pc < pc_of s | pc_of f < pc)"




constdefs

  getOSarg1 :: "frame  => opstack"
 "getOSarg1 == %(stk,loc,cn,ml,pc). stk"

  getOSarg2 :: "frame => p_count"
 "getOSarg2 == %(stk,loc,cn,ml,pc). pc"

  getOSarg3 :: "frame => cname"
 "getOSarg3 == %(stk,loc,cn,ml,pc). cn"

  getOSarg4 :: "frame => method_loc"
 "getOSarg4 == %(stk,loc,cn,ml,pc). ml"


  putOSargs ::"[frame,(opstack*p_count)] => frame"
 "putOSargs == %(stk', loc', cn',ml',pc') (stk, pc).
                 (stk, loc',cn',ml',pc)"



  getLASarg1 :: "frame => opstack"
 "getLASarg1 == %(stk,loc,cn,ml,pc). stk"

  getLASarg2 :: "frame => locvars"
 "getLASarg2 == %(stk,loc,cn,ml,pc). loc"

  getLASarg3 :: "frame => p_count"
 "getLASarg3 == %(stk,loc,cn,ml,pc). pc"

  putLASargs ::"[frame,(opstack*locvars*p_count)] => frame"
 "putLASargs == %(stk', loc', cn',ml',pc') (stk,loc, pc).
  (stk, loc,cn',ml',pc)"


  get_p_count :: "frame => p_count"
  "get_p_count == %(stk,loc,cn,ml,pc). pc"


consts
 execCFSindep::"( instr list*minstate) => (minstate option)"

recdef execCFSindep "{}"



"execCFSindep ([], (xp, frs)) = None"


"execCFSindep (xs, (None,(stk,loc,pc))) = (case (xs!pc) of
LAS ins => (let (stk',loc',pc') = exec_las ins stk loc pc
in
Some (None,(stk',loc',pc')))
|CO ins => None
|MO ins => None
|MA ins => None
|CH ins => None
|MI ins => None
|MR ins => None
|OS ins => (let (stk',pc') = exec_os ins stk  pc
in
Some (None,(stk',loc,pc')))
|CBF ins =>(let (stk',pc') = exec_cb_fwd ins stk  pc
in
         Some (None,(stk',loc,pc')))
|CBB ins =>(let (stk',pc') = exec_cb_bwd ins stk  pc
in
          Some (None,(stk',loc,pc')))
|UBF ins =>(let (pc') = exec_ub_fwd ins pc
in
     Some (None,(stk,loc,pc')))
|UBB ins =>(let (pc') = exec_ub_bwd ins  pc
in
      Some (None,(stk,loc,pc')))) "
 "execCFSindep (xs, (Some xp, f)) = None"



end
```

B.3 Exec_block0.thy

```
(***********************************************************)
(* Relation equivalent to the block execution relation, but with *)
(* separate listing of head and tail of frame list             *)
(* Used to produce big step relation for loops                 *)
(***********************************************************)

Exec_block0 = HOL + Set + List  + Exec+  State_parts + ListSlice +

consts exec_block0 ::"(bytecode * (cname*(mname*param_desc) *p_count) *
 (cname*(mname*param_desc)*p_count) * (xcpt option * heap *(opstack *
locvars *cname *(mname*param_desc)*p_count))*(xcpt option * heap *
(opstack *locvars *cname *(mname*param_desc)*p_count))*(opstack *
locvars *cname *(mname*param_desc) *p_count)list*(opstack *locvars *
cname *(mname*param_desc) *p_count)list) set"

inductive exec_block0

intrs

(*********** End of execution of a block - ************)
(*********** pc' outside block************)

Stop " [|(exec(CFS, (xp,hp,(stk,loc,cn,(mn,pd),pc)#frs)) = \
\       Some (xp',hp',(stk',loc',cn',(mn',pd'),pc')#frs'));\
\  inside ((stk,loc,cn,(mn,pd),pc)#frs) (cnS,(mnS,pdS),pcS)
(cnF, (mnF,pdF),pcF);\
\  pcF < (length(get_code CFS cnS (mnS, pdS))); \
\  same_method_frs (cnS,(mnS,pdS),pcS) ((stk,loc,cn,(mn,pd),pc))
((stk',loc',cn',(mn',pd'),pc'))  (cnF, (mnF,pdF),pcF);\
\  outside ((stk',loc',cn',(mn',pd'),pc')#frs')
(cnS,(mnS,pdS),pcS) (cnF, (mnF,pdF),pcF)|]\
\      ==> (CFS, (cnS,(mnS,pdS),pcS), (cnF, (mnF,pdF),pcF),
(xp, hp, (stk, loc, cn, (mn,pd), pc)),
(xp', hp',( stk', loc', cn', (mn',pd'), pc')), frs, frs'):exec_block0"




(*********** Continuation of execution of a block -`***********)
(*********** pc'' inside block ***********)

Continue "[|(exec(CFS, (xp,hp,(stk,loc,cn,(mn,pd),pc)#frs)) =
Some (xp'',hp'',(stk'',loc'',cn'',(mn'',pd''),pc'')#frs''));\
\inside ((stk,loc,cn,(mn,pd),pc)#frs) (cnS,(mnS,pdS),pcS)
(cnF, (mnF,pdF),pcF);\
\        pcF < (length(get_code CFS cnS (mnS, pdS)));\
\  same_method_frs (cnS,(mnS,pdS),pcS) ((stk,loc,cn,(mn,pd),pc))
((stk'',loc'',cn'',(mn'',pd''),pc''))  (cnF, (mnF,pdF),pcF);\
\  (CFS, (cnS,(mnS,pdS),pcS), (cnF,(mnF,pdF),pcF),
(xp'',hp'',(stk'',loc'',cn'',(mn'',pd''),pc'')) ,
(xp',hp',(stk',loc',cn',(mn',pd'),pc')), frs'', frs'):exec_block0 |]\
\      ==> (CFS, (cnS,(mnS,pdS),(pcS)), (cnF,(mnF,pdF) ,pcF),
(xp,hp,(stk,loc,cn,(mn,pd),pc)) ,
(xp',hp',(stk',loc',cn',(mn',pd'),pc')), frs, frs'):exec_block0"



end
```

B.4 Exec_block3.thy

```
(*****************************************************)
(* Block execution relation                          *)
(*****************************************************)

Exec_block3 = Exec  + Exec_block0 +

consts exec_block3 ::"(bytecode * boundary *boundary *
jvm_state * jvm_state) set"

syntax "@exec_block3" :: [ bytecode, boundary, boundary,
jvm_state , jvm_state] => bool  ("_ _ _ |- _ -block-> _" )



translations
  "CFS s f |- a -block-> b" == "(CFS, s,f,a,b):exec_block3"
inductive exec_block3

intrs

Stop "[| exec (CFS, a) = Some b ;\
\        inside (third_of a) s f; \
\  (pc_of f) < length (get_code CFS (cn_of s) (ml_of s));\
\  same_method_frs s (hd(snd(snd a))) ( hd(snd(snd b))) f ;\
\  outside  (third_of b) s f|] \
\        ==> CFS s f |- a -block-> b"


Continue "[|  exec (CFS, a) = Some c ;\
\             inside (third_of a) s f;\
\       (pc_of f) < length (get_code CFS (cn_of s) (ml_of s));\
\       same_method_frs s (hd(snd(snd a))) (hd(snd(snd c))) f ;\
\       CFS s f |- c -block->  b|]\
\              ==> CFS s f |- a -block-> b"


end
```

B.5 Block_pairs.thy

```
(*******************************************************)
(* Execution path relation                             *)
(*******************************************************)

Block_pairs = HOL + Exec + Assign + State_parts + Exec_block3
+ Exec_block_conds +
constdefs

 narrow_boundary1 :: "boundary => boundary"
 "narrow_boundary1 b == let (cn,ml,pc) = b in
                           (cn,ml,(pc-1))"

constdefs

block_pairs :: "[bytecode, boundary, boundary,(xcpt option *
 heap *frame list), (xcpt option * heap *frame list)] => bool"
("_ _ _ |- _ -execute-> _")

"CFS S F |- s -execute-> t ==
(s,t) : {(s,t). exec (CFS, s) = Some t
& same_method_frs S (hd(snd(snd s))) ( hd(snd(snd t))) F &
 inside (third_of s) S F }"

 block_pairs_trancl :: "[bytecode, boundary, boundary,(xcpt option *
 heap *frame list), (xcpt option * heap *frame list)] => bool"
("_ _ _ |- _ -execute^+-> _")

"CFS S F |- s -execute^+-> t ==
(s,t) : {(s,t). exec (CFS, s) = Some t &
same_method_frs S (hd(snd(snd s))) ( hd(snd(snd t))) F
& inside (third_of s) S F }^+"

 big_step :: "[  bytecode, boundary, boundary,(xcpt option *
heap *frame list), (xcpt option * heap *frame list)] => bool"
("_ _ _ |- _ -bigstep-> _")

"CFS S F |- a -bigstep-> b ==
(a,b): {(a,b).((pc_equals (hd(third_of a)) (pc_of F) &
 exec(CFS, a) = Some b) |  (EX c. ((pc_equals (hd(third_of c)) (pc_of F) )
 & exec (CFS,c) = Some b  &
(CFS S (narrow_boundary1 F) |- a -block-> c)))) }"


(*** series of big steps - but all within s to f **********)

big_step_trancl :: "[  bytecode, boundary, boundary,(xcpt option *
 heap *frame list), (xcpt option * heap *frame list)] => bool"
("_ _ _ |- _ -bigstep^+-> _")
```

```
"CFS S F |- a -bigstep^+-> b ==
(a,b): {(a,b).((pc_equals (hd(third_of a)) (pc_of F) &
 exec(CFS, a) = Some b) | ( EX c. ((pc_equals (hd(third_of c)) (pc_of F) )
 & exec (CFS,c) = Some b  & (CFS S(narrow_boundary1 F) |- a -block-> c))))}^+"


end
```

B.6 Block_pairs_conds.thy

```
Block_pairs_conds = Block_pairs +

constdefs

excep_free :: "[instr list] => bool"

"excep_free (ys) == (ALL CFS xp1 hp1 frs1 xp' hp' frs' cn1 ml1 cn'
 ml' pcS pcF.(( ys = (fromto  pcS pcF (get_code CFS cn1 ml1))
         & (CFS (cn1,(ml1),pcS) (cn',(ml'),pcF)
           |- (xp1,hp1,frs1) -execute^+-> (xp',hp',frs'))) -->
   xp1 = None & xp' = None))"


excep_free_instr :: "[instr ] => bool"

"excep_free_instr (y) == (ALL xp stk loc pc  xp' stk' loc' pc' xs .
((xs!pc = y)  & (execCFSindep (xs,(xp,stk,loc,pc)) =
Some (xp',stk',loc',pc')) --> xp = None & xp' = None))"


well_formed_loop:: "[instr list]=> bool"
"well_formed_loop zs == ( ALL CFS cnS mlS cnF mlF pcS pcF xs ys.
( zs = (UBF (Goto_fwd (Suc (length xs)))#xs @( ys @
[CBB (Ificmplt_bwd (length xs + length ys))])) -->
( pcS < pcF &  get_code CFS cnS (mlS) ! pcS =
 UBF (Goto_fwd (length (xs) + 1))
 &  excep_free ys & list_all excep_free_instr ys &
 (xs = fromto (pcS + 1)  (pcS + (length xs)) (get_code CFS cnS (mlS))) &
(ys = fromto  (pcS + length xs + 1)  ( pcF-1) (get_code CFS cnS (mlS))) &
(simple_block xs) & simple_block ys & ys ~= [] &xs~=[] &
 get_code CFS cnS (mlS) ! pcF = CBB (Ificmplt_bwd (length (xs @ ys))) &
(pcF = pcS + length (xs @ ys) + 1)  &(simple_block ys) &
ref_trans_2 ys & linear ys &
correct_init_loop_state (cnS,(mlS), pcS) (cnF,(mlF),pcF) )))"


end
```

B.7 Exec_block_conds.thy

```
Exec_block_conds =  State_parts + ListSlice + Exec_block3 +

constdefs


is_branch :: "instr => bool"
"is_branch instr == case instr of
          LAS ins => False
| CO  ins => False
| MO  ins => False
| MA  ins => False
| CH  ins => False
| MI  ins => False
| MR  ins => False
| OS  ins => False
| CBF  ins => True
        | CBB  ins => True
| UBF  ins => True
        | UBB  ins => True"




(****** instructions which only alter stk, pc, and xp************)
consts
is_load ::"load_and_store => bool"
```

```
primrec

"is_load (X load idx) = True"

 "is_load (X store idx)  = False "

 "is_load (Bipush ival) = True"

 "is_load Aconst_null = True"

 "is_load (Iinc idx ival) = False"

 "is_load Iadd  = True"


constdefs
 simple_stk_op :: "instr => bool"
"simple_stk_op instr == case instr of
          LAS ins => (is_load ins)
| CO  ins => False
| MO  ins => False
| MA  ins => False
| CH  ins => True
| MI  ins => True
| MR  ins => False
| OS  ins => True
| CBF  ins =>False
        | CBB  ins => False
| UBF  ins => False
        | UBB  ins => False"


constdefs
 stk_op :: "instr => bool"
"stk_op instr == case instr of
          LAS ins => (is_load ins)
| CO  ins => False
| MO  ins => False
| MA  ins => False
| CH  ins => True
| MI  ins => True
| MR  ins => False
| OS  ins => True
| CBF  ins => True
        | CBB  ins => True
| UBF  ins => True
        | UBB  ins => True"

consts
all_stk_ops :: "instr list => bool"

recdef all_stk_ops  "measure (% xs. length xs)"

"all_stk_ops [] = True"

"all_stk_ops (x#xs) = ((stk_op x) & (all_stk_ops xs))"


consts
  get_cbf_branch :: "cond_branch_fwd => nat"

primrec

"get_cbf_branch  (Ifnull_fwd i) = i"

"get_cbf_branch ( Ifiacmpeq_fwd  X i) = i"

"get_cbf_branch ( Ificmplt_fwd i) = i"


consts
  get_cbb_branch :: "cond_branch_bwd => nat"

primrec

"get_cbb_branch  (Ifnull_bwd i) = i"

"get_cbb_branch ( Ifiacmpeq_bwd X i) = i"

"get_cbb_branch ( Ificmplt_bwd i) = i"


consts
  get_ubf_branch :: "uncond_branch_fwd => nat"

primrec
```

```
"get_ubf_branch  (Goto_fwd i) = i"


consts
  get_ubb_branch :: "uncond_branch_bwd => nat"

primrec

"get_ubb_branch  (Goto_bwd i) = i"


constdefs
get_branch :: "instr => nat option"
 "get_branch instr == case instr of
          LAS ins =>None
| CO  ins => None
| MO  ins =>None
| MA  ins => None
| CH  ins => None
| MI  ins => None
| MR  ins => None
| OS  ins => None
| CBF  ins => Some (get_cbf_branch ins)
        | CBB  ins => Some (get_cbb_branch ins)
| UBF  ins =>  Some (get_ubf_branch ins)
        | UBB  ins =>  Some (get_ubb_branch ins)  "

constdefs

is_target ::"[bytecode,cname,mname,param_desc, p_count] => bool"
"is_target CFS cn mn pd pc == ( ALL pc1.
                              (is_branch( (get_code CFS cn (mn,pd) ) ! pc1)
       --> pc1 + the(get_branch
                               ( (get_code CFS cn (mn,pd) ) ! pc1)) = pc))"


constdefs

 inrange :: "[p_count, p_count,p_count,p_count] => bool"

 "inrange a b s f == (s < a) & (b < f)"
```

```
linear :: "[instr list] => bool"

"linear (xs) ==( list_all not_branch xs)  & (list_all is_CFS_independent xs)"


constdefs

 simple_block :: "[instr list] => bool"
"simple_block (ys) == ALL CFS xp1 hp1 frs1 xp' hp' frs' cn1 ml1 cn'
                    ml' pcS pcF.( ys = (fromto  pcS pcF
                                        (get_code CFS cn1 ml1))
                       & inside  frs1 (cn1,ml1,pcS) (cn',ml',pcF)
                       & same_method_frs (cn1,ml1,pcS) (hd frs1)
                                          (hd frs')(cn',ml',pcF)
                       & pcF < length (get_code CFS cn1 ml1)
       & exec (CFS, (xp1,hp1,frs1)) = Some(xp',hp',frs')) -->
 inside  frs' (cn1,ml1,pcS) (cn',ml',pcF) |
        pc_equals (hd frs') (pcF + 1)"




constdefs

  insert_pc_frm :: "[frame, p_count] => frame"
  "insert_pc_frm s x  == let (stk, loc, cn,ml,pc) = s in
                                 (stk, loc, cn,ml,x)"


   add_pc_frm :: "[frame, p_count] => frame"
  "add_pc_frm s x  == let (stk, loc, cn,ml,pc) = s in
                                 (stk, loc, cn,ml,(pc + x))"
consts
  insert_pc_frames :: "(frame list* p_count) => frame list"

recdef insert_pc_frames "{}"

  "insert_pc_frames ([] ,x) = []"

  "insert_pc_frames ((y#ys), x) = [(insert_pc_frm y x)]@(ys)"

consts
```

```
  add_pc_frames :: "(frame list* p_count) => frame list"

recdef add_pc_frames "{}"

  "add_pc_frames ([] ,x) = []"

  "add_pc_frames ((y#ys), x) = [(add_pc_frm y x)]@(ys)"



constdefs
  insert_pc :: "[jvm_state, p_count] => jvm_state"
  "insert_pc  s x  == (let (xp, hp, frs) = s  in
                           (xp, hp, (insert_pc_frames (frs, x))))"


 add_pc :: "[jvm_state, p_count] => jvm_state"
  "add_pc  s x  == (let (xp, hp, frs) = s  in
                           (xp, hp, (add_pc_frames (frs, x))))"



minimise_frm :: "frame =>minframe"
"minimise_frm a == let (stk,loc,cn,ml,pc) = a in (stk, loc,pc)"

consts

  minimise_frs :: "frame list => minframe list"

  recdef minimise_frs "measure (% xs. length xs)"

"minimise_frs ([]) = []"
"minimise_frs (x#xs) = (minimise_frm x)#(minimise_frs xs)"



consts

  meta_frames :: "(frame list* frame list) => bool"

  recdef meta_frames "measure (% (xs, ys). length xs)"

  "meta_frames ([],[]) = True"
  "meta_frames ((x#xs),[]) = False"
  "meta_frames ([],(y#ys)) = False"
  "meta_frames ((x#xs), (y#ys)) = ((stk_of x = stk_of y) &
                                   (loc_of x = loc_of y) &
                                    meta_frames (xs,ys))"







 constdefs
(******* compare only xp, stk, loc of top of stk ***********)

  meta_eq ::" [jvm_state, jvm_state] => bool" ("_ =~= _")

  "a =~= b ==  (fst a) = (fst b) &
               stk_of (hd(snd(snd a )))= stk_of (hd(snd(snd b))) &
               loc_of (hd(snd(snd a))) = loc_of (hd(snd(snd b)))"



(******* should mention only xp, stk, loc of top of stk ***********)


meta_holds:: "[instr list, (jvm_state => bool),jvm_state] => bool"
"meta_holds ys q s == (ALL CFS cn1 ml1 pcS  pcF.
                                   ((ys) = (fromto  pcS pcF
                                             (get_code CFS cn1 (ml1)))
                                    -->q (insert_pc s pcS)))"



(******* should mention only xp, stk, loc of top of stk ***********)


  meta_inv :: " [(((xcpt option * heap *(opstack *locvars *cname
                *method_loc *p_count)list))=>bool) ] => bool"

  "meta_inv P == ALL s x. P (s) --> P ((insert_pc s x))"
 ref_trans_2 :: "[instr list] => bool"

"ref_trans_2 ys == ALL CFS xp1 hp1 frs1 xp' hp' frs' cn1 ml1 cn'
                  ml' pcS pcF.((( ys = (fromto  pcS pcF
                                   (get_code CFS cn1 ml1)))
                 & linear ys & pc_equals (hd frs1) pcS
                  & (CFS (cn1,(ml1),pcS) (cn',(ml'),pcF) |- (xp1,hp1,frs1)
                                                -block-> (xp',hp',frs')))
                  -->
                   (xp1 = xp' & hp1 = hp' &
                  ( EX a b. stk_of (hd(third_of (xp',hp',frs'))) =
                            (a#(b#(stk_of (hd(third_of (xp1,hp1,frs1))))))
                      &  loc_of (hd(third_of (xp1,hp1,frs1))) =
                         loc_of (hd(third_of (xp',hp',frs')))
                      & st_cn_of (hd(third_of (xp1,hp1,frs1))) =
                        st_cn_of(hd(third_of (xp',hp',frs')))
                      & st_ml_of (hd(third_of (xp1,hp1,frs1))) =
                        st_ml_of(hd(third_of (xp',hp',frs')))
                      & (tl(third_of (xp1,hp1,frs1))) =
                        (tl(third_of (xp',hp',frs'))))))"




end
```

B.8 Exec_instrs.thy

```
(*******************************************************************)
(* Sequence execution relation                                     *)
(*******************************************************************)
```

```
Exec_instrs = Exec  + State_parts +

consts exec_instrs ::"(instr list * minstate * minstate) set"

syntax "@exec_instrs" :: [instr list,minstate, minstate] =>
 bool  ("_ |- _ -instrs-> _" )


translations
  "xs |- a -instrs-> b" == "(xs ,a,b):exec_instrs"
inductive exec_instrs

intrs

Stop "[| execCFSindep (xs, a) = Some b ;\
\        xs ~= [];\
\        inlist xs ((snd a)); \
\  outlist  xs ((snd b))|] \
\        ==> xs |- a -instrs-> b"


Continue "[|  execCFSindep (xs, a) = Some c ;\
\             xs ~= [];\
\             inlist xs ((snd a));\
\       xs |- c -instrs->  b|]\
\              ==> xs |- a -instrs-> b"


end
```

B.9 Triple.thy

```
(*******************************************************************)
(* Pre- and Post-condition  relation                                *)
(*******************************************************************)

Triple = List + Exec_block0 + Exec_block3  + Exec_instrs +
         Block_pairs_conds + Ref_trans +


consts
```

```
triple:: " ((((xcpt option * heap *(opstack *locvars *cname *method_loc
               *p_count)list))=>bool)* instr list*
                (((xcpt option * heap *(opstack *locvars *cname *method_loc
                   *p_count)list))=>bool) => bool)"

recdef triple"{}"


"triple(p, [], q) = (ALL xp hp frs
     xp' hp'  frs' .(((p((xp, hp, frs)))
     --> q((xp', hp', frs')))))"


"triple(p, (y#ys), q) = (ALL CFS  xp hp frs cn1 ml1 cn' ml'  xp' hp'
  frs'  pcS  pcF.
(((CFS (cn1,(ml1),pcS) (cn', (ml'),pcF)  |- (xp, hp, frs) -block->
                                                       (xp',hp',frs'))
   & ((y#ys) = (fromto  pcS pcF
                                        (get_code CFS cn1 (ml1))))
   & st_pc_of (hd frs) = pcS
   & p((xp, hp,frs))-->
    q((xp', hp', frs')))))"




constdefs

  assert :: "[((opstack *locvars)=>bool) , (xcpt option * heap *
       (opstack *locvars *cname *method_loc *p_count)list)] => bool"

  "assert P == (%(xp,hp,frs). (let (stk, loc,cn,ml,pc) = hd frs in
           P (stk,loc)))"

end
```

B.10 Wpc.thy

```
(*********************************************************************)
(* Weakest Precondition for a list of commands                       *)
(*********************************************************************)

constdefs

wp :: "[ instr list, ((((xcpt option * heap *((opstack *locvars *cname *
 method_loc *p_count)list ) ))) => bool)] =>
           ((((xcpt option * heap *((opstack *locvars *cname *method_loc *
               p_count)list )))) => bool)"


"wp ys q == (%(xp,hp,frs).
(ALL CFS  xp' hp'  frs' cn1 ml1 cn' ml'  pcS  pcF .
(((CFS  (cn1,(ml1),pcS)  (cn', (ml'),pcF)|-
                         ((xp, hp, frs)) -block-> (xp', hp',frs'))
& ((ys) = (fromto  pcS pcF (get_code CFS cn1 (ml1))))
& st_pc_of(hd frs) = pcS))-->
          q(xp', hp', frs')))"



not_term_state :: "jvm_state => bool"
"not_term_state s == ((fst s) = None & (third_of s) ~= [])"



(*****************************************)
(* Sequence weakest precondition         *)
(*****************************************)

 meta_wp :: "[ instr list, ((opstack *locvars) => bool)] =>
             ((((xcpt option * heap *((opstack *locvars *cname *method_loc
                 *p_count)list )))) => bool)"


" meta_wp ys q == (%(xp,hp,frs). (let (stk,loc,cn,ml,pc) = (hd  frs) in

(ALL  xp' stk' loc' pc'.
(not_term_state (xp,hp,frs) &
                 ys |- (xp,stk,loc,0) -instrs->(xp',stk',loc',pc'))-->
```

```
        assert q (xp', hp, (stk',loc',cn,ml,(pc + pc'))#(tl frs)))))"


  end
```

Bibliography

[1] Isar Homepage. http://isabelle.in.tum.de/Isar.

[2] Sun Microsystems Inc. http://www.sun.com.

[3] The Java Homepage. http://java.sun.com.

[4] The PVS Homepage. http://pvs.csl.sri.com/.

[5] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques and Tools.* Addison Wesley, 1986.

[6] Andrew Appel. *Modern Compiler Implementation in Java.* Cambridge University Press, 1998.

[7] David Aspinall. Proof General homepage. http://zermelo.dcs.ed.ac.uk.

[8] A. Avron, F. Honsell, and I. Mason. An overview of the edinburgh logical framework, 1989.

[9] John Barnes. *High Integrity Ada, The SPARK Approach.* Addison-Wesley, 1997.

[10] J. Camilleri and T. Melham. Reasoning with Inductively Defined Relations in the HOL Theorem Prover. Technical Report 265, University of Cambridge Computer Laboratory, 1992.

[11] George C. Necula Christopher Colby, Peter Lee. A Proof-Carrying Code Architecture for Java. In *Proceedings of the 12th International Conference on Computer Aided Verification (CAV00), Chicago*, 2000.

[12] Richard M. Cohen. Guide to the dJVM Model Version 0.5 Alpha. Technical report, Computational Logic Inc., 1996.

[13] David L. Detlefs, K. Rustan M. Leino, Greg Nelson, and James B. Saxe. Extended Static Checking. Technical Report 159, Compaq Systems Research Center, 1998.

[14] Sophia Drossopoulou and Susan Eisenbach. Java is Type Safe - Probably. In *Proceedings of the 11th European Conference on Object Oriented Programming*, 1997.

[15] R.W. Floyd. Assigning Meanings to Programs. *Proceedings of the Symposium of Applied Mathematics*, 19, 1967.

[16] Michael J. C. Gordon. Mechanizing Programming Logics in Higher Order Logic. In *G. Birtwistle and P. A. Subrahmanyam (editors), Current Trends in Hardware Verification and Automated Theorem*. Springer Verlag, 1988.

[17] Michael J.C. Gordon. From LCF to HOL: a short history. In *Proof, Languages and Interaction*. MIT Press, 2001.

[18] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Addison-Wesley, 1996.

[19] Tim Harris. A just-in-time Java bytecode compiler. CST Part II project dissertation, University of Cambridge, 1997.

[20] C.A.R. Hoare. An Axiomatic Basis for Computer Programming. *Communications of the ACM*, 12(10), October 1969.

[21] M. Huisman and B. Jacobs. Java Program Verification via a Hoare Logic with Abrupt Termination. In *Approaches to Software Engineering (FASE'00)*. Springer, 2000.

[22] Joseph Hummel, Ana Azevedo, David Kolson, and Alexandru Nicolau. Annotating the Java Bytecodes in Support of Optimization. *Concurrency: Practice and Experience*, 9(11):1003–1016, November 1997.

[23] Sun Microsystems Inc. Hotspot virtual machine white papers . http://java.sun.com/products/hotspot.

[24] Sun Microsystems Inc. The Java Tutorial . http://java.sun.com/docs/books/tutorial/index.html.

[25] B. Jacobs. The LOOP Project. http://www.cs.kun.nl/˜bart/LOOP/.

[26] Bart Jacobs, Joachim van den Berg, Marieke Huisman, Martijn van Berkum, Ulrich Hensel, and Hendrik Tews. Reasoning about Java Classes Preliminary Report. In *Proceedings of Object-Oriented Programming Systems, languages and Applications*, 1998.

[27] Matt Kaufmann, Panagiotis Manolios, and J Strother Moore. *Computer-Aided Reasoning: An Approach*. Kluwer Academic Publishers, 2000.

[28] Douglas Kramer. The Java Platform, a White Paper. Technical report, Sun Microsystems Inc., 1996.

[29] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification.* Addison-Wesley, 1997.

[30] R Macgregor, D Durbin, J Owlett, and A Yeomans. *Java Network Security.* Prentice Hall PTR, 1998.

[31] Erik Meijer and John Gough. Technical Overview of the Common Language Runtime. Technical report, Microsoft Research, 2001.

[32] R. Milner. Logic for Computable Function: description of a machine implementation. Technical Report STAN-CS-72-288, Stanford University, 1972.

[33] M.J.C. Gordon and T.F. Melham, editors. *Introduction to HOL: a theorem proving environment for Higher Order Logic.* Cambridge University Press, 1993.

[34] J Strother Moore. Proving Theorems about Java-like Byte Code. In E.-R. Olderog and B. Steffen, editors, *Correct System Design—Recent Insights and Advances*, volume 1710 of *LNCS*. Springer-Verlag, 1999.

[35] George C. Necula. Proof-Carrying Code. Design and Implementation. In H. Schwichtenberg and R. Steinbruggen (eds.), Proof and System Reliability (lecture notes for a summer course in Marktoberdorf, 2001).

[36] George C. Necula. Proof-Carrying Code. In *Proceedings of the 24th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, January 1997.

[37] George C. Necula and Peter Lee. The Design and Implementation of a Certifying Compiler. Technical report, School of Computer Science, Carnegie Mellon University, 1998.

[38] Tobias Nipkow and David von Oheimb. Java$_{light}$ is Type-Safe—Definitely. In *25th ACM Symposium on Principles of Programming Languages, January 19-21, 1998, San Diego*, 1998.

[39] Tobias Nipkow, David von Oheimb, and Cornelia Pusch. μJava: Embedding a Programming Language in a Theorem Prover. In Friedrich L. Bauer and Ralf Steinbrüggen, editors, *Foundations of Secure Computation*, number 175 in NATO Science Series F: Computer and Systems Sciences. IOS Press, 2000. http://isabelle.in.tum.de/Bali/papers/MOD99.html.

[40] Lawrence C. Paulson. Natural Deduction as Higher-Order Resolution. *Journal of Logic Programming*, 3, 1986.

[41] Lawrence C. Paulson. The Foundation of a Generic Theorem Prover. *Journal of Automated Reasoning*, 5, 1989.

[42] Lawrence C. Paulson. *Isabelle, a Generic Theorem Prover*, volume 828 of *LNCS*. Springer, 1994.

[43] Lawrence C. Paulson. A Generic Tableau Prover and its Integration with Isabelle. Technical report, University of Cambridge Computer Laboratory, 1998.

[44] The GNU Project. GCJ: The Gnu Compiler for Java. http://gcc.gnu.org/java/.

[45] Cornelia Pusch. Formalizing the Java Virtual Machine in Isabelle/HOL. Technical report, Technische Universität München, June 1998.

[46] Cornelia Pusch. Proving the Soundness of a Java Bytecode Verifier in Isabelle/HOL, September 1998.

[47] Claire L. Quigley. Proof for Optimization: Programming Logic Support for Java JIT Compilers. In *Supplementary Proceedings of 2001 International Conference on Theorem Proving in Higher Order Logics*, number EDI-INF-RR-0046 in Informatics Research Report. Division of Informatics, University of Edinburgh, September 2001.

[48] Claire L. Quigley. A Programming Logic for Java Bytecode Programs. In *Proceedings of the 16th International Conference on Theorem Proving in Higher Order Logics*, volume 2758 of *Lecture Notes in Computer Science*. Springer-Verlag, September 2003.

[49] J. Rothe, B. Jacobs, and H. Tews. The Coalgebraic Class Specification Language CCSL. *Journal of Universal Computer Science*, 7(2), 2001.

[50] Robert R. Schneck and George C. Necula. A Gradual Approach to a More Trustworthy, yet Scalable, Proof-Carrying Code. In *Proceedings of the Conference on Automated Deduction (CADE'02), Copenhagen, July 2002.*

[51] John V. Guttag Stephen J. Garland and James J. Horning. An Overview of Larch. In Peter E. Lauer, editor, *Functional Programming, Concurrency, Simulation and Automated Reasoning*, volume 693 of *LNCS*. Springer-Verlag, 1993.

[52] Don Syme. Proving Java Type Soundness. Technical report, Computer Laboratory, University of Cambridge, June 1997.

[53] P. Thomas and Ray Weedon. *Object-Oriented Programming in Eiffel* . Addison Wesley, 1997.

[54] J. van den Berg and B. Jacobs. The LOOP compiler for Java and JML. In *Tools and Algorithms for the Construction and Analysis of Software (TACAS)*, LNCS. Springer, 2001.

[55] Bill Venners. *Inside the Java Virtual Machine.* McGraw-Hill, 1998.

[56] Glyn Winskel. *The Formal Semantics of Programming Languages.* The MIT Press, 1993.

[57] Hongwei Xi and Robert Harper. A Dependently Typed Assembly Language. Technical Report CSE-99-008, Oregon Graduate Institute, 1999.

[58] Hongwei Xi and Frank Pfenning. Dependent Types in Practical Programming. In *Proceedings of ACM SIGPLAN Symposium on Principles of Programming Languages(POPL '99)*, pages 228–242, 1999.