MACQUARIE
University
SYDNEY · AUSTRALIA

Faculty of Science and Engineering

Macquarie University

Evaluating Kiama Abstract State Machines for a Java Implementation

by

Pongsak Suvanpong

A thesis submitted to Macquarie University for the degree of

Master Of Research

Department of Computing

October 9, 2015

Supervisor: A/Prof. Anthony Sloane

This page is intentionally left blank

The work in this thesis has not been submitted for a higher degree to any other university or institution.

[illegible]

Pongsak Suvanpong

Abstract

This thesis represents nine months of research in the area of embedded domain-specific programming languages. Kiama is a pure embedded lightweight language processing library in Scala. It provides classes for language processing paradigms such as attribute grammars, strategic term rewriting and abstract state machines (ASM) which can be used for analyzing, translating and executing languages.

In this thesis, we evaluate the ASM component of Kiama by implementing several complex machines for executing the dynamic semantics of the Java language and the Java Virtual Machine byte code into which it is translated. We use the book "*Java and the Java Virtual Machine: Definition, Verification and Validation*" by R. Stärk, J. Schmid and E. Börger, as our reference. The book describes the Java language version 1.2 using the ASM method.

We are able to implement the machines without any modifications to Kiama's ASM. The combination of Scala and Kiama allows us to closely replicate the book's ASM notations in executable code. However, we find a few problems with the Kiama ASM library for which we show workarounds. In addition, we discover a few bugs in the book's machine definitions and provide fixes in our implementation.

Table of Contents

Chapter 1 Introduction

Software is now used in many devices that can directly and indirectly affect our everyday lives. From a microwave oven in our kitchen to a car in our garage, a cellular phone in our hand, a pacemaker in the human body, to all modern airplanes are now being controlled by computer software. Hence, software is a crucial part of many products today. The failure of the software in these devices can be fatal. Hence, the software needs to be developed using techniques that ensure high reliability.

A software development process normally starts with capturing the requirements. The result of that process is an informal specification of the software. A formal model based on the informal specification is then created which can be used for mathematically validating the properties of the informal requirements. Abstract State Machines(ASM) which is the topic of this thesis is one way to formally model systems and software. To implement the software, the models are recoded using programming languages. Hence, programming languages are very important in the software engineering process.

To implement a programming language, typically, requires four main components: The syntax, semantic analyzer, the code generator and the execution engine. The syntax analyzer parses the textual form of a language into a structured form according to the grammar of the language. Using the structured form, the semantic analyzer makes sure that the program is correct according to the semantics defined by the language. The code generator takes the output from the semantic analyzer and linearized it into a sequence of instructions. The execution engine, then, executes those instructions.

Implementing a programming language normally requires tools such as parser generators, syntactic and semantics analyzers. These tools normally live in a separate piece of software. For example, JastAdd [1] may be used for attribute grammars for semantics analyzer, Tatoo [2] can be used for generating LALR parser generator and JFlex [3] is a flex like tool for generating lexical analyzer for Java. Another approach uses tools that are integrated into a library. Kiama [4] is such a tool. It is a pure embedded lightweight language processing library written in Scala. It moves aways from the traditional language engineering approach that uses many external tools. The tools in Kiama are set of classes and objects, packed in a library that can be invoked directly from the software that requires language processing. Some of these components in the Kiama library are, the attribute grammars, the strategic term rewriting and the abstract state machine. The attribute grammars which can be used for implementing the semantic analyzer. Strategic term rewriting can be use for translating a tree structure into a linear structure (code generator) and optimization. Abstract State Machines (ASM) [5] can be used for describing the dynamic semantics of programming languages and virtual machines.

In this study, we are interested in the ASM component in Kiama. An ASM is a formal method of specifying a system using a state based approach. A specification of a model is written using the ASM notations which have clear and precise meaning. Similar to Finite State Machines (FSM), ASM comprise states and rules. The rules update/transit the states. ASM uses discrete time-step execution model. But contrary to FSM, ASM use abstract states, not a single symbol state. ASM has a pseudo language to define rules to manipulate the abstract states. The rules can be viewed as the logic and states can be viewed as the inputs and outputs of ASM. These features allow ASM to solve a wider range of problems than FSM.

1.1 Motivation

There are number of ASM programming languages [6]–[11]. All of them are designed as an external tool and only for system modeling purposes, thus the models cannot be reused in the software implementation phrase. Hence, recoding is required if the models are built with these languages.

Kiama is a library. The components are just classes and functions. ASM models defined using Kiama can be invoked from application software written in any JVM languages which have support for interoperability between languages. Hence, the recoding process might be removed if the models are specified using Kiama ASM.

Most of the components in Kiama have been evaluated and tested in number of case studies [12], [13] except the ASM component. There is a sample that shows how Kiama ASM can be used. The sample is a simple machine for executing the RISC instructions set that was described in the book *Compiler Construction* [11]. The RISC example is a very small example, however so we want to see if Kiama ASM can be scaled to specify a complex system.

1.2 Contribution

In this thesis, we evaluate the ASM implementation in Kiama. *The research question that we are investigating in this study is whether the current implementation of Kiama ASM can be scaled up to a more complex problem than the simple RISC machine in the current Kiama library. The approach that we use is to implement complex nontrivial machines using the current ASM implementation in Kiama.*

The book *Java and the Java Virtual Machine: Definition, Verification and Validation* [15] has developed the formal ASM models of the semantics of the Java language and its virtual machine. In this thesis, we use the term *JBOOK* to refer to this book. The JBOOK uses the standard Java version 1.2 specification and divides the specification into 5 levels: *I, C, O, E,* and *T.* Each level is divided into two sub sections: the dynamic semantics and the virtual machine with its compiler. The I level describes the imperative core expressions and statements. The C level describes the procedural features (static methods). The O level describes the object-oriented features (dynamic method binding and inheritance).

The E level describes exception handling features and the T level describes the multi-threading features. On the JVM side, there is no the T level, but the N level that describes the native method interface features.

In addition to the formal mathematical versions of the ASM models, an appendix section of the JBOOK contains an executable implementation of the ASM models. The implementation uses a programming language called AsmGofer [8].

We consider the machines in the JBOOK to be complex and non-trivial, so to sufficiently evaluate the ASM component in Kiama, we implement the I, C and O levels of the dynamic semantics, the virtual machine and the compiler. Due to time limitation (nine months, including preparation of this thesis), we do not implement the E, the T and the N level. In addition, we feel that the I, C and O level cover the major part of the Java language and similar approaches and techniques to those used in the I, C and O can handle the E, the T and the N level.

Our evaluation criteria are:

1. Our code should closely replicate the mathematical definition of the machines used in the JBOOK with correct semantics.
2. Our transition rules map one-to-one to the transition rules that are used in the JBOOK.
3. The ASM models that we code using Kiama/Scala in this study, can be invoked from an application just as any other parts of Kiama.
4. We implement our machines for this study without modifying to any part of Kiama or using any special version of Scala.

To test our implementation we use the test cases available in the book. For each test case we expect the execution of the dynamic semantic machines and the virtual machine to produce the same result. The test cases are designed to exercise all the transition rules of the machines.

We also compare our implementation and the AsmGofer implementation from the book to show that Kiama/Scala is more suitable for specifying ASM than AsmGofer.

We have encountered some problems while we are implementing our code. We present the workarounds for the problems without modifying the Kiama ASM library.

The source code of this study is available at `https://bitbucket.org/psksvp/compx/overview` and the appendix sections contain the excerpt of the code.

1.3 Organization

This thesis is organized as follows. In chapter 2, we discuss the origin and the concept of ASM. We focus on the concept of abstract states, execution model, the notations and the constructs used in the ASM models presented in the JBOOK. We then look at the current implementation of ASM programming languages.

Chapter 3 describes Kiama ASM in more detail. We use an example that comes with the Kiama library to describe how the library can be used to write an ASM definition and the execution model. Some Scala techniques that are commonly used, are also described here.

Chapter 4 describes in detail the scope of this study and the assumptions that we have made.

Chapter 5 and 6 describes the techniques that we used for implementing the dynamic semantics ASMs and the JVM ASMs with its compiler respectively. These techniques are available in Scala which allow us to closely replicate the mathematical notation used in the JBOOK.

Chapter 7 focuses on testing and evaluation. The testing is done by using the test cases available in the JBOOK and some additional test cases that we wrote. To ensure correctness, we run each test case through the dynamic semantics machines and compile each test case and run it through the JVM machines. The outputs from both the dynamic semantics machines and the JVM machines must be the same. We also present problems that we found in the current implementation of ASM in Kiama and how we work around the problems. In addition, we compare our Kiama/Scala implementation with the Asm-Gofer implementation by the JBOOK's authors. Moreover, we find bugs in the definitions of the machines in the JBOOK and describe the fixes in our implementation.

Chapter 8 presents the conclusion and significance of this study. We also present an idea of an additional tool that might help developers easily debug ASM models written using Kiama.

Chapter 2 Related Works

Many systems of computation can be described using the concept of abstract mechanical machines which have discrete time-steps execution model. These machines have a number of states and a set of transition rules which govern the transition and updating of their states in each step. One such conceptual machine is the Turing machine [16] which can be used to model many computable problems. It is an idealized state machine which uses the concept of tape and the tape reader/writer machine. The tape has a series of symbols which are read by the reader head. The machine has a set of rules which govern how the machine responds to a certain symbol. For example if the reader found symbol ’E’ and the previous symbol that the reader read was ’Q’ then move the tape forward, and write out symbol ’1’.

Another well known method of state machine modeling is Finite State Machines (FSM). In a FSM, the behaviors of a system are described using a finite number of states and each state is represented by a single symbol. In contrast, with Turing machines which use only the reader to read a symbol, the FSM defines arbitrary events from the environment which may cause the current state to move to another state. Similar to Turing machines, a set of rules drives the state transition.

Turing machines are intended to accept strings as the input and produce strings as the output thus it can be difficult to express non-string operations such as floating point numerical computation. There are limitations in FSM as well. A state in a FSM is simply a single symbol and there is no hierarchy in a FSM model. A large number of states is required to model a large complex system. Thus, the transition rules become very complex and can be hard to manage. In addition, a single symbol per state limitation results in FSM variable-less which can be difficult to express certain problems. For example, it is not possible to use FSM to build a machine to do an unbounded counting, an external functionality (in this case a variable) must be added to implement counting which results in dependency to the underlining programming language which is used to implement the counting FSM.

2.1 Abstract State Machines

A generalization of the FSM is the Abstract State Machines (ASM) [5]. An ASM is a formal method for specifying a system using the state machine concept (state-based). The specification of a model is done using the ASM notations with a clear and precise programming language constructs. Similar to FSM, the ASM comprises *states* and *rules* that update/transit the states and the discrete time-step execution model. But contrary to the FSM, the ASM uses abstract states, not a single symbol. Moreover, ASM provides notations and language constructs to form rules to manipulate the abstract state. To develop a system using ASM, the system is viewed as having a number of states. A set of transition rules dictate if any state transition should occur.

ASM has been used to specify the semantics of many programming languages [15], [17]–[19], to specify real and virtual CPU architectures [20], [21] and to specify hard realtime systems [11], [22], [23]

2.1.1 States

Contrary to a single symbol per state in FSM, a *state* in ASM is an n-arity function $f(a_1,a_2,...,a_n)$ where a_1, a_2, ..., a_n are called *locations and f* is the state nam*e*. An ASM state could be thought as an unbounded n-dimension array or a multi-keyed hash table abstract data type. A state can be thought as a memory unit of ASM which allows the read/write operations. The location abstracts away the memory addressing.

2.1.2 Rule Constructs

Rules in ASM are the logic which decide whether any states should be updated. It is, sometime, called the *transition rule* which is in the form `if Condition then Updates` or using patterning matching `Pattern → Updates`.

The `Updates` to each state in the form `f(a₁, a₂, ..., aₙ) := t.`

The updates are done in parallel and is guarded against *inconsistency*. An inconsistency occurs when a state is updated more than one time to a different value in the a step. The symbol **:=** is used to specify an update which value on the right side of the symbol.

Since ASM is inherently a parallel machine, there are also constructs to simplify the operations on collection of states. The `forall x with p do R` construct is to simultaneous execute rule `R` for all state `x` which has property `p`. The `choose x with p do R` construct is the non-deterministic version of `forall`. It randomly picks a state **`x`** which has property `p` and execute rule `R`.

2.1.3 Execution Model

Contrary to the execution model of procedural programming languages where a program starts at an entry point and stops when it reaches the end, the execution model of ASM uses the concept of discrete time-step. In each step, it executes all the rules defined. If any rule is fired, they may update the machine’s states. The updates to the states at the current step will be seen at the next step, thus, the updates can be done in parallel. The execution of ASM terminates, if its states have not changed from the previous step. This state of an ASM is called the *fixed-point*.

To illustrate the ASM execution model, let's look at an algorithm to estimate the logarithm base 2 ($\log_2$) of an integer number n. An iterative method of this algorithm will keep dividing 2 to the number n until n is 1. Listing 1 shows the algorithm written in Scala.

```scala
def log2(n:Int):Int=
{
  var t = 0
  while(n > 1)
  {
    n = n / 2
    t = t + 1
  }
  t // result
}
```

Listing 1: Iterative method of estimating log base 2 written in Scala using a while loop.

To model the algorithm using ASM, we can take advantage of the ASM execution model to simplify the algorithm. In each step, we only need to divide the number N by 2, if N is greater than one, otherwise we skip the update. The number of steps for the machine to reach a fixed-point, will be the result of the algorithm. Listing 2 shows an ASM machine to estimate $\log_2(9)$ written using standard notations.

```
Function N: ⇒ Number
InitRule =
  N := 9

MainRule =
  if N > 1 then N := N / 2
  else skip
```

Step	Value of State N
0 (Init)	9
1	4
2	2
3	1

Listing 2: ASM model to estimate log2(9) (left). Keyword Function N is to declare a state which in this case a nullary function of type Number. The InitRule is executed once at when this machine starts. The MainRule is executed at each step until a fixed-point is reached. The result (right) of execution of the ASM to estimate $\log_2(9)$. The number of steps (without the init step) is the result of $\log_2(9)$.

2.2 ASM programming languages

Since the first publication on ASM [5] has been published which is also known as the *Lipari Guide*, several programming languages have been developed to execute ASM model in computers [24], [25]. These languages are not available, therefore in this report we only pick the languages that are still available to review.

There are several approaches to implement the ASM languages, we have categorized them into, pure ASM language, hybrid ASM specification language and ASM extension or library in another programming language.

2.2.1 Pure ASM language

A pure ASM language offers only the ASM constructs for system modeling, running and testing of the model only. It does not offer any direct integration to other programming language or operating system for application development. Hence, it can be used for system modeling only.

2.2.1.1 CoreASM

CoreASM [9] is a dynamically typed interpreted ASM language. It runs on the Java Virtual Machine (JVM). However, it does not offer any interface to the other JVM-based programming languages. The language implements most of the notations described in the Lipari guide.

The syntax of CoreASM aims to be close to the standard textual notation described in Lipari guide. A machine definition in CoreASM starts with the keyword `CoreASM < machine name >`. The state syntax uses the -> symbol to separate the left which is the name and arguments and the right hand side which is the return type of the function. Indentation is used to indicate a block. The CoreASM type system provides the NUMBER, STRING and ENUM types. The symbol := is used to update a state.

CoreASM provides a plugin architecture for the states and rules. The collection of states and rules can be packaged and reused in a later specification. In fact many of the ASM standard constructs are written in CoreASM language as a library.

The main disadvantage of CoreASM is the speed of execution, due to the fact that it is an interpreted environment. In addition, ASM model written in CoreASM cannot be reused in the software development phrase if CoreASM is not used as the main development language.

2.2.1.2 CASM

CASM [26] implementation is based on CoreASM syntax. However, CASM is statically typed while CoreASM is a dynamically typed. CASM requires types to be clearly declared for the arguments of the state and the rules. Type conversion cannot be done automatically.

The aim of CASM is to improve the execution performance which is lacking in the CoreASM implementation due to the use of an interpreted approach. CASM offers both an interpreter and a compiler. The compiler generates C++ code which requires a C++ compiler to generate the executable. The performance of the compiled ASM specification is much faster than CoreASM.

CASM can compile ASM model into C++, however it is not possible to execute the model from another C++ application.

2.2.1.3 Asmeta

Asmeta [10] is an attempt to steer away from just defining a programming language for specifying an ASM model, but to use a meta-modeling approach. Asmeta uses the Object Management Group (OMG) standard XML Metadata Interchange (XMI) for modeling the meta ASM model (AsmM). The definition of an ASM model in AsmM is served as medium between the ASM modeling and the ASM simulation engine.

AsmetaL is a dynamically typed ASM modeling language. It has a compiler which compiles AsmetaL to the AsmM XMI definition. The benefits of compiling the ASM modeling language to AsmM XMI definitions are, first the model validation (AsmetaV) and simulation engine (AsmetaS) can use the XMI directly which make them independent of modeling languages. Second, the models written in different ASM languages can be integrated, since they are compiled into a common AsmM XMI definition. AsmetaL has been used for runtime monitoring of Java programs [27].

2.2.2 Hybrid ASM language

Using this approach, the ASM language provides some interfaces to other programming languages or the running platform operating system. This allows easier application development. There are two approaches which this kind of ASM languages use, first is to allow calling sub-routines written in other programming languages, second is to compile the ASM specification into another programming language, e.g. C/C++ or Java, the compiled target languages source code can, then be further integrated with the application source code and compiled into the binary code of the target platform.

2.2.2.1 AsmL

AsmL [6] is a programming language with the ASM constructs and the execution model. It is designed to easily interface with the .NET platform. The AsmL compiler generates the .NET Common Language Runtime (CLR) code. Thus, AsmL can be used to write a complete application to run on the .NET platform. An interface is also provided by the AsmL for other .NET programming languages to load and run machines defined by AsmL. Moreover, AsmL also supports object oriented programming.

The design goal of AsmL is to be a complete design language from specification phase to coding and testing phase. The program source code can be written using plain text or embedded in an XML file which can have other documentation tags.

The execution model is a little different from the other ASM languages. The control of step is left to the programmer by using a keyword *step*. The keyword is used to advance the machine step by one. The statement `step until fixedpoint` will execute the rules within the block until all the states have reached a fixed-point.

AsmL sees the variables in CLR, including the abstract data type collection in the .NET library as states. State update can be done using the := operator. The type system in AsmL is the complete .NET platform type system, including the parameterized type.

2.2.2.2 XASM

The aim of XASM [7] is to be extensible, component based and able to interface with the C/C++ programming language. XASM is a compiled language. The XASM compiler generates C language code which the programmer can integrate with additional libraries using the XASM external language interface. This method offers both easy integration with an application written in C and an efficient executable. XASM supports most of the

ASM constructs described in the Lipari Guide.

XASM provides two ways to interface with external C program, first a XASM machine specification can call external C functions. Second a machine specification which is compiled by XASM compiler into C code, can be embedded into a project. Both ways require the passing and returning of values into and from the machine from the C function must be typed based on the standard C language data types.

The modular design of the component based concept in XASM makes it easy to reuse the machine definitions. Each machine can be packaged into a component library. This feature allows easy structuring of large ASM projects. In addition, each submachine can be called like a function in which the submachine cannot update any of the states of the caller. Another way is to execute a machine as a submachine, in this case, the submachine can update the state of the caller.

Like CASM, XASM is a statically and strongly typed language. Every rule and state needs to have a type declared on their arguments. The XASM types are that of the C language but without the pointer type.

2.2.3 ASM extension in other programming languages

This kind of ASM implementation uses an existing programming languages and extends and/or modifies the languages to support the writing and running of an ASM specification. There are two approaches, first is to modify the semantics of a language to support ASM execution mode which as a result, makes the language incompatible with the original language.

The second approach is to implement the execution model of ASM as a library in a programming language. The advantage of this approach is, there is very little complexity, thus easy to maintain, because there is no language engineering involve. In addition, it is easy to integrate in an application development. However, if the language cannot express the notations and the constructs defined in the Lipari guide [5], this might be difficult to express some problems in a high level abstraction.

2.2.3.1 AsmGofer

AsmGofer [8] is a modification of Gofer [28] which is a Haskell-like functional programming language. AsmGofer does not add any new syntax to Gofer, however the internal evaluation engine of Gofer is modified to support the execution model of ASM which requires updating global states in each of the machine time steps. As a result, code written for AsmGofer will not run correctly on Gofer and vice versa. The type system in AsmGofer is that of Gofer. The JBOOK has used AsmGofer to implement their ASM models in executable form.

To define a state, AsmGofer provides a Gofer function (subroutine) `initVal`. For example to create a nullary state, AsmGofer provides a Gofer's function `initVal`.

```
initVal :: Eq a => String → a → a
```

Hence, to create a new unary state *f*, the following expression is used.

```
f = initVal “name” init
```

The parameter *name* is the name of the state which is used only when AsmGofer needs to print error messages related to the state. The state is kept in the variable *f*. The state update operator := is also defined by AsmGofer as a Gofer's function shown below.

```
(:=) :: AsmTerm a => a → a → a → Rule()
```

The function allows an update to a state in the form `f := 10` where `f` is a state.

Defining a rule in AsmGofer is the same as defining a Gofer function. Gofer's flow control statements can be used in the body of a rule. AsmGofer implements the `forall` and `choose` rules which are described in [6].

2.2.3.2 Kiama ASM

Kiama is a pure embedding language processing library. It is a library written in the Scala language. Since Kiama is our focus in this study, we have put the detail about the Kiama ASM in chapter 3.

2.3 Summary

Several ASM programming languages have been implemented to executed ASM model in computer. We have categorized them into: pure ASM language, hybrid ASM language and ASM that is hosted in another language. The last approach can be done by modifying an existing language or implement ASM as a Library.

ASM languages reviewed in this chapter except AsmGofer, are implemented using programming language development approach. The main advantage of this approach is the syntax of the language is very similar to the notations used in the Lipari guide.

The development of AsmGofer does not use the programming language development approach, however the runtime of the Gofer is modified to support the ASM execution model which as a result, code written in AsmGofer will not run correctly in Gofer and vice versa.

Kiama ASM is reviewed in Chapter 3.

Chapter 3 Kiama ASM

Kiama [4] is a lightweight language processing library for Scala [29]. The library provides components for language processing paradigms such as attribute grammars, strategy-based term rewriting and abstract state machines.

In general, a language processing pipeline comprises a parser, a semantic analyzer and a code generator or an execution engine (illustrated in Figure 1). In Scala, a parser can be built using the Scala parser combination library [30]. The Kiama attribute grammars can be used for semantic analyzing and the strategy-based term rewriting can be used for code optimization. The ASM component fits in the last part of the language processing pipeline as the execution engine which is the focus of this study.

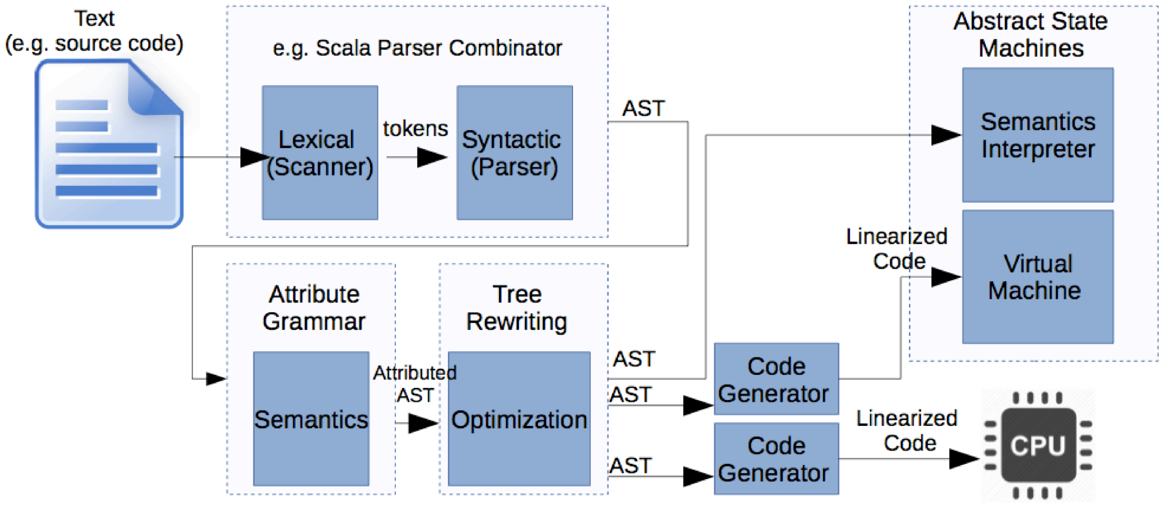

Figure 1: Typical Language Processing Pipeline in Kiama. NOTE: This pipeline is just one example of how the components in Kiama can be used for language processing.

3.1 ASM

Kiama ASM provides the ASM execution model in the abstract class `Machine`, the nullary (0-ary) state in the class `State[T]` and the unary (1-ary) state in the class `ParamState[T]`. To define a machine, a new class must be derived from the Kiama `Machine` class. The ASM rules are defined as the methods of the derived machine class. There are two predefined rules of the `Machine` class: `Machine.init` and `Machine.main`. `Machine.init` can optionally be overridden to initialize the machine while the rule *`Machine.main`* must be overridden to provide an entry point for the machine to start executing.

A nullary state in Kiama is an instance of the class `Machine.State[T]`. The class is a parameterized class which allows a state to be of any type `T` that exists in Scala. A unary state is an instance of the `Machine.ParamState[T, U]` class. The *`T`* is the type of the location and *`U`* is type of the value of the state at location *`T`*.

The := symbol is used as the operator to update a state. The symbol is actually the name of a method of the class `Machine.State[T]` and `Machine.ParamState[T, U]`. Each update to a state is recorded. At the end of step, all the updates are checked for consistency before the actual updates can occur. The `if_then_else` flow control in Scala is used as the `if_then_else` rule in ASM, since they have the same semantic.

Kiama ASM is a library, thus there is no modification to the Scala language to support the writing and running of ASM models. The current implementation of Kiama ASM does not provide any other predefined rules described in the JBOOK, for example the `forall` and the `choose` rules. However, some ASM constructs can easily be written in Scala. For example listing 3 (left), shows the `choose` rule written in Scala. The parameters to the method are a list of type T where T can be an instance of class `Machine.State` and a predicate function `fProp`. The function is used for determining if a state in the list of type T meets the selection properties. The anonymous function `rule` does operations on the states which has met with the properties. Listing 3 (right) shows an example call to the `choose` rule where `pc`, `sp` and `fp` are nullary states of type `Int`. From the example, the `choose` rule's parameters `sl` is `List(pc, sp, fp)`, `fProp` is a function `(s:State[Int]) => s.name == “pc”` which is the selection criterion and `rule` is `case s => s + 1` which increases the value of the state by one.

`def choose[T](sl: Seq[T],`
`           fProp:T=>Boolean)(rule:T => Unit)`
`{`
` val rnd = new scala.util.Random()`
` var s = sl(rnd.nextInt(sl.length))`
` if(fProp(s))`
`  rule(s)`
`}`
	`//example call to the choose rule`
	`val pc = new State[Int]("pc")`
	`val sp = new State[Int]("sp")`
	`val fp = new State[Int]("fp")`
	`choose(List(pc, sp, fp),`
	`       (s:State[Int]) => s.name == "pc")`
	`{`
	` case s => s + 1`
	`}`

Listing 3: ASM choose rule written in Scala. The function takes a sequence of states type T (Seq[T]) and a function that takes a state type T and returns a boolean (fProp:T=>Boolean). The function is used by the choose rule to determine if a state satisfies the selection properties. rule:T => Unit is an anonymous function that takes a state T and operate on it.

3.2 A Simple RISC ASM in Kiama

To illustrate how Kiama ASM can be used, let's look at the RISC example from the library. The example implements the RISC instruction set that was described in the book *Compiler Construction* [14]. The Instruction set comprises memory movement (load/store), integer arithmetic, bitwise operations, control flow, and input/output. The machine has 32 registers, each is 32 bits wide (one word). The full source code of this example is available in the Kiama source distribution[1].

3.2.1 States

In the RISC machine, the registers, the flags, the program counter, frame pointer and

1 `https://bitbucket.org/inkytonik/kiama.`

stack pointer are the states of the machine.

Registers

The RISC Registers are a 1-ary state in ASM, thus `ParamState` class is used.

```
type[2] RegNo = Int
val[3] R = new ParamState[RegNo,Int]("R")
```

The location of this state is the `RegNo` which is an alias for an Integer. Each location in the state stores a word (Integer), thus the class `ParamState` is instantiated with `ParamState[RegNo, Int]("R")`. The argument "R" is a string, defining the name of the state. It is used for debugging.

Program Counter, Frame Pointer and Stack Pointer

They are defined as special registers from number 28 to 31 of the state R. Hence they are defined as the reference to location 28 to 32 of the states R.

```
val PC = R(28)   // program counter
val FP = R(29)   // frame pointer
val SP = R(30)   // stack pointer
```

Flags

There are three flags: zero, condition and halt. In Kiama, they are nullary states which are defined as:

```
val Z = new State[Boolean]("Z")  // zero flag
val N = new State[Boolean]("N")  // condition flag
val halt = new State[Boolean]("halt") // halt flag
```

3.2.2 Instructions

Scala case classes[4] are used for representing the RISC instructions. In Scala, case classes are used as the immutable data holding classes. The data of a case class are specified in the parameters of the constructor. Scala automatically generates the accessor methods (getter) to the data. For example the MOV instruction type is defined as a case class below.

```
case class MOV(a:RegNo, b:RegNo, c:RegNo) extends Instr
```

The MOV class derives from an abstract class *Instr*, so that the class can be used with a polymorphic reference. The sequence of instructions is a Scala type `Seq[Instr]`[5]. The MOV class can be instantiated like below.

```
val m = MOV(1, 23, 2)
println(m.a) // accessor a, print 1
println(m.b) // accessor b, print 23
```

2 The *type* keyword defines a type alias.
3 The *val* keyword defined an immutable variable (single assignment).
4 http://www.scala-lang.org/old/node/107
5 The Scala trait *Seq* is a mixed-in type. Seq[T] provides common methods for accessing sequence structure like array and list.

The RISC machine keeps the instructions to be executed in a variable `code` of type `Seq[Instr]`. It indexes the sequence to get the instruction at the PC state index.

3.2.3 Machine and Rules

The `RISC` class derives from the abstract class `Machine`. The `init` method of the abstract class `Machine` is normally overridden to initialize the states. The `init` method is invoked once by the machine before the first step.

The `main` method of the Machine class must be overridden to provide an entry point of the RISC machine class. The `main` method is the first rule that the `Machine` class executes and it is executed repeatedly in every step until fixed-point is reached. Listing 4 shows the code of the `init` and `main` rule and Listing 5 shows the complete list of the rules that execute the RISC instruction.

```
override def init {
 PC   := 0
 R(0) := 0
 Z    := false
 N    := false
 halt := false
}
```

```
override def main {
 if(!halt) {
  try {
    arithmetic(code(PC))
    control(code(PC))
    memory(code(PC))
    inputoutput(code(PC))
  } catch {
    case e: Exception => halt := true
 }
}
```

Listing 4: The rule init and main of the RISC machine.

The RISC machine executes each instruction by using Scala *pattern matching*[6] of the instruction case classes. Pattern matching on case classes is done by putting the rules and actions inside a `match` expression.

```
 selector match { alternatives }
```

A sequence of alternatives starts with the `case` keyword followed by a pattern and the => symbol and followed by the action of the pattern. Scala will try to match each pattern in the sequence in the order that they are written. if the *`selector`* matches with any case classes in the *match* block, the data of the matched case class are automatically bound to the local variables that are passed to the pattern case statement. For example if the pattern

```
case MOV(x, y, z) => R (x) := R (z) << y
```

matches, the local variables x, y, z are automatically bound to the case class data a, b, c of the MOV class.

In each step, an instruction is selected by the main rule from `code` at the index which is the current value of the `PC` state. The machine continues running until the `halt` state is set to true, which prevents any further update to the states. Hence, the fixed-point is reached and the RISC machine terminates.

6 http://docs.scala-lang.org/tutorials/tour/pattern-matching.html

```
def arithmetic(instr : Instr) {
 instr match {
  case MOV(a,b,c)    => R(a) := R(c) << b
  case MOVI(a,b,im) => R(a) := im << b
  case MVN(a,b,c)    => R(a) := -(R(c) << b)
  case MVNI(a,b,im) => R(a) := -(im << b)
  case ADD(a,b,c)    => R(a) := R(b) + R(c)
  case ADDI(a,b,im) => R(a) := R(b) + im
  case SUB(a,b,c)    => R(a) := R(b) - R(c)
  case SUBI(a,b,im) => R(a) := R(b) - im
  case MUL(a,b,c)    => R(a) := R(b) * R(c)
  case MULI(a,b,im) => R(a) := R(b) * im
  case DIV(a,b,c)    => R(a) := R(b) / R(c)
  case DIVI(a,b,im) => R(a) := R(b) / im
  case MOD(a,b,c)    => R(a) := R(b) % R(c)
  case MODI(a,b,im) => R(a) := R(b) % im
  case CMP(b,c)      => Z := R(b) =:= R(c)
                        N := R(b) < R(c)
  case CMPI(b,im)    => Z := R(b) =:= im
                        N := R(b) < im
  case CHKI(a,im)=> if((R(a)<0)||(R(a)>=im))
                      R(a) := 0
  case _                =>
 }
}
```

```
def control(instr : Instr) {
 instr match {
  case b:BEQ if Z => PC:=PC+b.label.disp
  case b:BNE if !Z => PC:=PC + b.label.disp
  case b:BLT if N  => PC:=PC + b.label.disp
  case b:BGE if !N => PC:= PC + b.label.disp
  case b:BLE if Z||N =>PC:=PC + b.label.disp
  case b:BGT if !Z && !N =>
               PC:=PC+ b.label.disp
  case b:BR      => PC := PC + b.label.disp
  case b:BSR => LNK := PC + 1
                PC := PC + b.label.disp
  case RET(c) => PC := R(c)
                 if(R(c)=:= 0)
                   halt := true
  case _      => PC := PC + 1
 }
}
```

```
def memory(instr : Instr) {
 instr match {
  case LDW(a,b,im) => R(a):=Mem((R(b)+im)/4)
  case LDB(a,b,im) => halt:=true
  case POP(a,b,im) => R(a):=Mem((R(b)-im)/4)
                      R(b):= R(b) - im
  case STW(a,b,im) => Mem((R(b)+im)/4):=R(a)
  case STB(a,b,im) => halt := true
  case PSH(a,b,im) => Mem(R(b)/ 4) := R(a)
                      R(b) := R(b) + im
  case _           =>
 }
}
```

```
def inputoutput(instr : Instr) {
 instr match {
  case RD(a) => R(a):=console.readInt(“:”)
  case WRD(c) => emitter.emit(R(c))
  case WRH(c) =>
    emitter.emit((R(c):Int).toHexString)
  case WRL() => emitter.emitln
  case _     =>
 }
}
```

Listing 5: rules that execute RISC instruction. The default case _ (underscore character) causes the rules to do nothing.

3.3 Summary

In this chapter, we reviewed the Kiama ASM library. We looked at Kiama as a language processing library in general and where Kiama ASM fitted in the language processing pipeline.

To understand how to use Kiama ASM, we presented a sample RISC machine implementation from the Kiama ASM library. We showed how Scala case classes were used to represent the RISC instructions, how Scala pattern matching was used on the case classes, how ASM states could be defined and how the rules were defined as the methods of the RISC machine class.

Chapter 4 Scope of this study

This section describes the scope of this study in more detail. Since the JBOOK is our reference definitions of the Java dynamic semantics and the JVM machines, we briefly describe the JBOOK. This section, then, describes the scope of our implementation and the assumptions about what the machines accept.

4.1 The JBOOK

The JBOOK uses the Java language specification version 1.2 and specifies the dynamic semantics of the language and its JVM using ASM method. The dynamic semantics of the Java language and the JVM are described by dividing the language based on features into five levels: I, C, O, E and T. The JBOOK does not have the T level for the JVM but provides the N. For each level in the JVM, the JBOOK also presents the compiler which compiles the Java AST to the JVM instructions for that level. Table 1 summarizes the features and the machines defined at each level.

Level	Description	Machines defined
$Java_I$/JVM_I	The imperative core Java expressions and statements. The features of the core language comprise the basic primitive data types, the expressions (literal, variable declaration, unary, binary inline conditional), the statements which include the control flow like, the while loop, the if conditional and abruption (break and continue).	$execJavaExp_I$ $execJavaStm_I$ $execVM_I$
$Java_C$/JVM_C	The procedural features. This level introduces the static method call and static variables with class and interface scoping. The `return` abruption statement is added.	$execJavaExp_C$ $execJavaStm_C$ $execVM_C$ $switchVM_C$
$Java_O$/JVM_O	The object-oriented features. This level extends the previous level with class instances and instance method calls.	$execJavaExp_O$ $execJVM_O$
$Java_E$/JVM_E	The exception handling features. This level adds the exception handling statements: the try-catch block and throw statements.	$execJavaStm_E$ $execJavaExp_E$ $execVM_E$ $switchVM_E$
$Java_T$	The multithreading features.	$ExecJavaStm_T$
JVM_N	The native method interfacing features.	$execVM_N$

Table 1: Summarization of the levels of the Java language features that the JBOOK has divided.

The ASMs that the JBOOK defines use transition rules and pattern matching to describe how the Java language and the JVM instructions are evaluated. A transition rule consists of two parts: a pattern and an action. They are separated by an arrow ($\rightarrow$) where the pattern is on the left of the arrow. The action of a transition rule is performed if the pattern matching succeeds. To illustrate how the transition rules work, let look at an example how the machine executes the dynamic semantics of an inline condition expression which is part of execJavaExp$_I$. The transition rules to process the expression are shown in the last four lines of listing 6. The α, β *and* γ denote the position of each expression in a syntax tree and the ▸ symbol denotes the current node where `pos` is pointing. `pos` is a nullary state, its value is the position of a node that the machine is evaluating. At the beginning, `pos` is set to the top of the syntax tree.

```
$execJavaExp_I$ = case $context(pos)$ of
  $lit \rightarrow yield(JLS(lit))$
  $loc \rightarrow yield(locals(loc))$

  $uop\ {}^{\alpha}exp \rightarrow pos := \alpha$
  $uop$ ▸$val \rightarrow yieldUp(JLS(uop, val))$

$^{\alpha}exp_1$ bop $^{\beta}exp_2 \rightarrow pos := \alpha$
▸$val$ bop $^{\beta}exp_2 \rightarrow pos := \beta$
▸$val_1$ bop ▸$val_2 \rightarrow$ if $\neg(bop \in divMod \wedge isZero(val_2))$ then $yieldUp(JLS(bop, val_1, val_2))$

  $loc = {}^{\alpha}exp \rightarrow pos := \alpha$
  $loc =$ ▸$val \rightarrow locals := locals \oplus \{(loc, val)\}$
                              $yieldUp(val)$

$^{\alpha}exp_0\ ?\ {}^{\beta}exp_1 : {}^{\gamma}exp_2 \rightarrow pos := \alpha$
▸$val\ ?\ {}^{\beta}exp_1 : {}^{\gamma}exp_2 \rightarrow$ if $val$ then $pos := \beta$ else $pos := \gamma$
$^{\alpha}true\ ?$ ▸$val : {}^{\gamma}exp_2 \rightarrow yieldUp(val)$
$^{\alpha}false\ ?\ {}^{\beta}exp1 :$ ▸$val \rightarrow yeildUp(val)$
```

Listing 6: Java$_I$ transition rules for the core Java expression (execJavaExp$_I$). (source) fig. 3.2 in the JBOOK.

The left hand side of the rule $^{\alpha}$`exp`$_0$ `?` $^{\beta}$`exp`$_1$ `:` $^{\gamma}$`exp`$_2$ `→ pos := α` is the pattern of the inline condition syntax. The pattern notations that the JBOOK uses are very similar to the actual Java source code. If the pattern matching succeeds, the operation on the right hand side of the arrow (→) executes which sets `pos` to `α`. This means, at the next step, the machine will evaluate the $^{\alpha}exp_0$ node. Once the $^{\alpha}exp_0$ has been evaluated, the machine puts a node `val` which is the result of the evaluation in position α of `restbody`. `restbody` is a unary state which when given a position returns a node. Its purpose is to keep the results of the evaluation which could be seen as an update-to-date version of the syntax tree.

At the next step, the rule ▸`val ?` $^{\beta}$`exp`$_1$ `:` $^{\gamma}$`exp`$_2$ `→ pos := if val` **`then`** `pos := β` **`else`** `pos := γ` is fired. The action of the rule is to check if the value of `val` is true or false, if it is true $^{\beta}exp_1$ is evaluated, otherwise $^{\gamma}$`exp`$_2$ is evaluated at the next time step.

If `pos` is at `β` (▸) then the rule $^{\alpha}$`true ?` ▸`val :` $^{\gamma}$`exp`$_2$ `→ yieldUp(val)` is fired at the next step. As the result, at the `β` position in `restbody` would contain a `val` which is the result of the evaluation of $^{\beta}$`exp`$_1$. The action of this rule is to yield the result up. This is done using the `yieldUp` rule. `yieldUp` puts the `val` node in `restbody` at the position of inline condition node. In fact, it replaces the inline condition node with the `val` node in `restbody`. Table 2 shows the detail of the execution in each step of an expression `1==1 ? 1 + 2 : -3`.

step	transition rule	Source matched	pos	source at pos
1	${}^{\alpha}exp_0$? ${}^{\beta}exp_1$: ${}^{\gamma}exp_2$ → pos := α	`1 == 1 ? 1 + 2 : -3`	${}^{\alpha}exp_0$	1==1
2	${}^{\alpha}exp_1$ bop ${}^{\beta}exp_2$ → pos := α	1==1	${}^{\alpha}exp_1$	1
3	lit → yield(JLS(lit))	1	`Val(1)`	1
4	▸val bop ${}^{\beta}exp_2$ → pos := β	`Val(1) == 1 ? 1 + 2 : -3`	${}^{\beta}exp_2$	1
5	lit → yield(JLS(lit))	1	`Val(1)`	1
6	▸val_1 bop ▸val_2 →	`Val(1) == Val(1)`	`Val(true)`	1==1
	if *¬(bop ∈ divMod∧ isZero(val_2))*			
	then *yieldUp(JLS(bop, val_1, val_2))*			
7	▸val ? ${}^{\beta}exp_1$: ${}^{\gamma}exp_2$ →	`val(1) == val(1) ? 1 + 2 : -3`	${}^{\beta}exp_1$	1+2
	if *val* ***then*** *pos := β else pos := γ*			
8	${}^{\alpha}exp_1$ bop ${}^{\beta}exp_2$ → pos := α	`1 + 2`	${}^{\alpha}exp_1$	1
9	lit → yield(JLS(lit))	1	`Val(1)`	1
10	▸val bop ${}^{\beta}exp_2$ → pos := β	`Val(1) + 2`	${}^{\beta}exp_2$	2
11	lit → yield(JLS(lit))	2	`Val(2)`	2
12	▸val_1 bop ▸val_2 →	`Val(1) + Val(2)`	`Val(3)`	1 + 2
13	${}^{\alpha}true$? ▸val : ${}^{\gamma}exp_2$ → yieldUp(val)	`Val(true) ? Val(3) : -3`	`Val(3)`	

Table 2: Step by step evaluation of an inline condition 1 == 1 ? 1+2 : -3.

When any transition rule has reached a result, there are two kind of actions that it will do: `yield(result)` or `yieldUp(result)`. `yield(result)` replaces the current node in `restbody` with the result, while `yieldUp(result)` replaces the parent of the current node in `restbody` with the result. The nodes that depends on their children to reach a result, the transition rule will use `yieldUp`, while nodes that do not have children, the transition rule will use `yield`.

4.2 What we implement

We implement all the machines from the I to the O level in both the dynamic semantics part and the virtual machine part (including the compiler for each level). We feel that the I, C and O level cover the major part of the Java language and similar approaches and techniques to those used in the I, C and O levels can handle the E, the T and the N levels.

The JVM accepts a list of the abstract instruction as input, hence in each level of the JVM, the JBOOK describes a compiler for that level. We also implement the compiler, so that we can test the JVM machines. The compiler accepts the Java Abstract Syntax Tree (AST) as input and compiles the AST into a list of instructions for the JVM to execute.

Constructing an AST by hand can be difficult and time consuming which as a result will not allow us to do adequate testing. Hence, we implement a simple Java parser using the Scala parser combinator library [30]. Not all Java sources are accepted by the parser; we have made some assumptions about what our parser accepts which we describe in the next section.

Our testing ensure that given a Java program, the running of the machines for the dynamic semantics of the Java language and the JVM on the Java program, should produce the same result.

Figure 2 summarizes what we implement in this study.

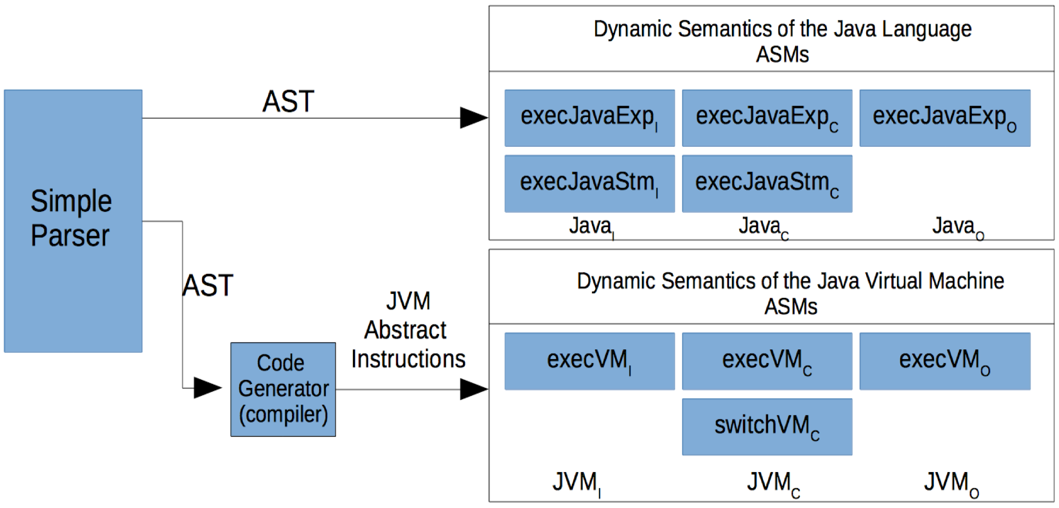

Figure 2: The components that have been implemented for this study.

4.3 Assumptions

In this study, we focus on the execution engine (figure 1) of the Java language. At the execution engine stage, the AST as been processed by other parts of a language processing pipeline. Hence, our parser assumes the following:

1. The Java source file is syntactically correct.
2. The Java source file has correct static semantics.
3. In each level, the JBOOK describes the constraints of the Java source which the machines accept. Our parser assumes these constraints are already enforced in the Java source code. For example, all static method calls must use the fully qualified name.

4.4 Summary

We implement the Java$_I$, Java$_C$, Java$_O$, JVM$_I$, JVM$_C$, JVM$_O$ machines and the compiler. We consider these machines to be non-trivial and suitable for evaluating the Kiama ASM library. Our research focuses on ASM, thus, we make some assumptions about what our implementation can accept. These assumptions are what the machine definitions in the JBOOK also expect.

Chapter 5 The Dynamic Semantics of Java in Kiama

This section describes our implementation of the dynamic semantics of the Java language in Scala/Kiama. First we show how the Java syntax tree can be represented in Scala, then we describe the implementation of Java$_{I}$, Java$_{C}$ and Java$_{O}$. In each section, we describe the techniques that are used to implement the states and the transition rules in Scala/Kiama. In the Java$_{I}$ section, we give a more detailed account of the techniques that we use than in the Java$_{C}$ and the Java$_{O}$ sections, because the same techniques that we use in Java$_{I}$ are reused throughout the implementation of Java$_{C}$ and Java$_{O}$.

5.1 Defining the Java syntax in Scala

The JBOOK defines the syntax as a set of production rules. The notations that the JBOOK uses, are close to the textual Java source. For example the expression (`Exp`) can be a literal (`Lit`), a local variable (`Loc`), a unary expression (`Uop Exp`), a binary expression (*`Exp Bop Exp`*), an inline condition (`Exp ? Exp : Exp`) or an assignment (*`Asgn`*). The `Asgn` is a rule which expands to `Loc = Exp`. Figure 3 shows the grammars of Java$_{I}$.

Exp := Lit | Loc | Uop Exp | Exp Bop Exp | Exp ? Exp : Exp | Asgn
Asgn := Loc = Exp
Stm := ; | Asgn; | Lab: Stm | **break** Lab; | **continue** Lab; | **if**(Exp) Stm **else** Stm | **while**(Exp) Stm | Block
Block := {Bstm$_1$….Bstm$_n$}
Bstm := Type Loc; | Stm
Phrase := Exp | Bstm | Val | Abr | Norm

Figure 3: The grammar of the core Java (Java$_{I}$) expressions and statements which the JBOOK defines.

Section 3.2.2 showed how case classes were used for representing the RISC instructions. The case classes do not require relationship between each others. However the grammar of the Java language is a tree structure which requires parent-child relationship between nodes. Hence, we make the class that represents the expression (*class* `Exp`) an abstract class, so that any classes extends from `Exp` can be used as a child of other case classes extending from class `Exp` (Listing 7).

`exp`	`abstract class Exp(children:Seq[Node]) extends Phrase(children)`
`Lit`	`case class Lit(representation:String) extends Exp(Nil)`
`Loc`	`case class Local(name:String) extends Exp(Nil)`
`Uop exp`	`case class UnaryOp(op:Operator, exp:Exp) extends Exp(List(exp))`
`exp`$_1$ `Bop exp`$_2$	`case class BinaryOp(op:Operator, exp1:Exp, exp2:Exp) extends Exp(List(exp1, exp2))`
`exp`$_0$ `? exp`$_1$`:exp`$_2$	`case class InlineCond(exp0:Exp, exp1:Exp, exp2:Exp) extends Exp(List(exp0,exp1,exp2))`
`loc = exp`	`case class Asgn(loc:Local, exp:Exp) extends Exp(List(loc, exp))`

Listing 7: On the left is the syntax notation used by the JBOOK. On the right is the corresponding Scala case classes of the Java expression (Exp) of Java$_{I}$. The hierarchy of these case classes is based on the grammar in Figure 3.

Figure 4 shows an example of the textual form of a Java expression and its corresponding case classes representation.

```
i = 2 + (8 * j) | Asgn(Local(i),                         |          Asgn
                |      BinaryOp(Lit(2),                  |          /  \
                |               Op(+),                   | Local(i)   BinaryOp
                |               BinaryOp(Lit(8),         |             /  |  \
                |                        Op(*),          |       Lit(2)  +   BinaryOp
                |                        Local(j))))     |                    /  |  \
                |                                        |              Lit(8)   *   Local(j)
```

*Figure 4: The textual form of a Java_I expression (i = 2 + (8 * j)) and its case classes representation. The case class can be visualized as a tree structure.*

5.2 Java_I

Java_I defines two machines to interpret the dynamic semantics of the imperative core Java expressions and statements. The expressions consist of the literal of primitive data types, unary and binary operations, assignment, and inline conditional. The statements consist of the control flow like the `if` condition and the `while` loop, abruption statements like the `break`, `continue` and `label`.

5.2.1 States in Java_I

Java_I defines three states: `pos`, `restbody` and `locals`.

`pos` is a nullary (0-ary) state. It acts as a pointer to the current location of a node on the AST that needs to be executed. The JBOOK defines it as `pos: Pos`. `Pos` is an abstract type for the position of a node in an AST. We use an instance of `class State[Int]` for `pos`. Each node in the AST is assigned a unique integer number, thus, we parameterize the state with the Scala `Int` class.

`restbody` is a unary (1-ary) state. It acts as the up-to-date state of an AST while the evaluation is in progress. The JBOOK defines it as `restbody: Pos → Phrase` which is a unary function that takes a `Pos` type and returns a `Phrase` (node type) in an AST. We implement `restbody` using the class `ParamState[Int, Node]`. `restbody` can be viewed as a dictionary where the key is a position (Int) and the value is the node at the position. A more detailed explanation of how `restbody` is used is given in section 5.2.2.

`locals` is also a unary state. It acts as the register of the local variables and their values. The JBOOK defines it as a map between AST node type `Loc` and another AST node type *`Val`*.

```
type Locals = Map(Loc, Val)
locals: Locals
```

`Val` is a node type which can be any of the Java primitive types.

```
type Val = boolean | byte | short | char | int | long | float | double
```

We implement `local` using the `class ParamState[String, Value[_]]`. The parameterized type `String` is used as the location of the state which is the name of the local variable. The type `Value[_]` is a parameterized class which corresponding with the JBOOK *`Val`* type (*`val`* is a keyword in Scala, hence we name the class `Value` to avoid any confusion.) In other word, `local` is a dictionary where the key is a string (name of a local variable) and the value is node type `Value[_]`. There are many meaning in Scala for the _ (underscore) character, but when it is used in the parameterized part of a class (e.g. `Value[_]`), it tells the Scala compiler to ignore the type. Listing 8 shows our implementation of the `Value[_]` classes based on the type `Val` in the JBOOK.

```
abstract class Value[T](val value:T, val typeT:Type) extends Phrase(Nil)
case class NullValue() extends Value[Byte](0, ByteType())
case class BoolValue(v:Boolean=false) extends Value[Boolean](v, BoolType())
case class ByteValue(v:Byte=0) extends Value[Byte](v, ByteType())
case class CharValue(v:Char='\u0000') extends Value[Char](v, CharType())
case class ShortValue(v:Short=0) extends Value[Short](v, ShortType())
case class IntValue(v:Int=0) extends Value[Int](v, IntType())
case class LongValue(v:Long=0L) extends Value[Long](v, LongType())
case class FloatValue(v:Float=0.0f) extends Value[Float](v, FloatType())
case class DoubleValue(v:Double=0.0d) extends Value[Double](v, DoubleType())
case class Ref(obj:Class) extends Value[String](obj.id, RefType(obj))
```

Listing 8: Our implementation of the Value classes.

5.2.2 Transition Rules of Java$_I$

The JBOOK combines the relevant transition rules in a sub-machine. A sub-machine is actually a rule. Listing 6 (page 18) shows the sub-machine execJavaExp$_I$ that contains the transition rules for the core Java expressions. We implement our machine by deriving a new class from Kiama's abstract `Machine` class. Each of the sub-machines is a method of the derived class and the states are the instance variables of the class. Listing 9 shows execJavaExp$_I$ in Kiama/Scala.

```
private def execJavaExpI: Unit=
{
 context(pos) match
 {
  case lit:Lit                                    => yieldResult(JLS(lit))
  case Local(name)                                => yieldResult(locals(name))
  case UnaryOp(op, Value(v))                      => yieldResultUp(JLS(op, v))
  case UnaryOp(_, exp)                            => pos := exp
  case BinaryOp(op, Value(left), Value(right))    => yieldResultUp(JLS(op, left, right))
  case BinaryOp(_, Value(_), exp2)                => pos := exp2
  case BinaryOp(_, exp1, _)                       => pos := exp1
  case Asgn(loc, Value(v))                        => locals(loc) := v
                                                     yieldResultUp(v)
  case Asgn(_, exp)                               => pos := exp
  case InlineCond(BooleanValue(_), Value(v), _)   => yieldResultUp(v)
  case InlineCond(BooleanValue(_), _, Value(v))   => yieldResultUp(v)
  case InlineCond(BooleanValue(v), exp1, exp2)    => if(v.value) pos := exp1 else pos := exp2
  case InlineCond(exp0, _, _)                     => pos := exp2
 }
}
```

Listing 9: execJavaExp$_I$ in Kiama/Scala. *Note: we name the rule yeild as yieldResult and yieldUp as yieldResultUp because yield is a keyword in Scala.*

The Scala pattern matching syntax is very similar to the notations used by the JBOOK. However, the transition rules in the JBOOK use the Java syntax notation as the patterns while our code uses case classes. If the code is compared with the notation used in the JBOOK, the sequence of the alternatives in our code are reverse of what are written in the JBOOK, because the pattern matching in Scala is done from the most specific case to the most general case. The reordering of the transition rules does not We also move the checking of divide by zero into the JLS function, instead of checking it at the action of the transition rule.

5.2.2.1 The Scala Extractor Pattern and the `restbody` state

In general, an execution of a Java program in an AST form can be viewed as a series of actions: traversing, evaluating and replacing. The traversing action moves through the tree and does pattern matching. The evaluating action evaluates the matched tree node using the action on right-hand side of the pattern matching rule. The current node is then replaced with the result of the evaluating action. For example an expression node `1 + 2` when the traversing action gets to this node, the left side of this node is a literal node (`Lit(1)`), so it gets evaluated into a value node (`Val(1)`). The `Lit(1)` node is replaced with a `Val(1)` node. The same thing happens with the right side node `Lit(2)`, which gets replaced with `Val(2)` node. And last, the whole expression is then replaced with a `Val(3)` node which is the result of this evaluation. Figure 5 illustrates this example.

```
      +              +               +
      /\       =>    /\       =>     /\    =>  Val(3)
 Lit(1) Lit(2)   Val(1) Lit(2)   Val(1) Val(2)
```

Figure 5: An execution of 1 + 2 expression by way of replacing each node with the result of the evaluation of the node.

The transition rules in the JBOOK appears as the node replacing action is being done, however, `restbody` is where the result of each evaluation is kept. For example with the transition rules below, the node `‣val` in the second rule is the result of evaluating the node $^{\alpha}exp_1$ of the first rule. The JBOOK does not mention that the `‣val` node is actually pulled from `restbody`.

$^{\alpha}exp_1\ bop\ ^{\beta}exp_2 \rightarrow pos := \alpha$
$^{\blacktriangleright}val\ bop\ ^{\beta}exp_2 \rightarrow pos := \beta$
$^{\blacktriangleright}val_1\ bop\ ^{\blacktriangleright}val_2 \rightarrow$ ***if*** $\neg(bop \in divMod \wedge isZero(val_2))$ ***then*** $yieldUp(JLS(bop, val_1, val_2))$

One of our aims in this study is to be able to write our Scala code as closely as possible to the notation used in the JBOOK. Our implementation uses *Scala extractor patterns*[7] to accomplish this [31]. In short, a Scala extractor object is an object that has a method name `unapply`. When an extractor object is used in a pattern, the `unapply` method is invoked by default. The return value from the `unapply` method appears as the argument in the Scala match-case block. The return value also indicates whether the pattern matched or not, rather than just providing the value which is used for sub-matching.

The `unapply` method, in our case, is used to pull the up-to-date node from `restbody`. For example, we use an extractor object `Value` in the transition rules for the binary operator to pull the up-to-date node of type `Value[_]` from `restbody` (listing 10). In listing 10, what appear to be the argument of the extractor object `Value` is actually what the `unapply`

7 http://docs.scala-lang.org/tutorials/tour/extractor-objects.html

method returns and is matched against the identifier patterns left and right. The underscore character (_) in this case is a wildcard pattern that matches anything. Listing 11 shows the code of the extractor objects that we define to pull different types of an AST node including the `Value[_]` node. They derive from an abstract parameterized class `SingleExtractor`. The derived classes specify the node type in the parameterized type of the class `SingleExtractor`. The extractor objects are reused throughout our implementation in Java$_C$ and Java$_O$.

```
  case BinaryOp(op, Value(left), Value(right)) => yieldResultUp(JLS(op, left, right))
  case BinaryOp(_, Value(_), exprRight)        => pos := exprRight
  case BinaryOp(_, exprLeft, _)                => pos := exprLeft
```

Listing 10: The extractor object Value is used for pulling the up-to-date node from restbody.

```
// T is an AST node type
abstract class SingleExtractor[T:ClassTag]
{
 def extract(e:Node):Option[T]=
 {
  val n:Node = restbody(e)
  n match
  {
   // if the type of n is the same with T
   case v:T => Some(v)
   case _   => None  // unless return none
  }
 }
}
```

```
object Value extends SingleExtractor[Value[_]]
{
 def unapply(e:Node) = extract(e)
}
object BooleanValue extends SingleExtractor[BoolValue]
{
 def unapply(e:Node) = extract(e)
}
object RefValue extends SingleExtractor[Ref]
{
 def unapply(e:Node) = extract
}
object Normal extends SingleExtractor[Norm]
{
 def unapply(e:Node) = extract(e)
}
```

Listing 11: The restbody extractor classes (Value, BooleanValue, RefValue and Normal). Each is used for matching a different type of Node in restbody. The unapply(e:Node) method of an extractor class is invoked by default when it is used in a pattern. The unapply(e:Node) method invokes the base class (SingleExtractor) method extract(e:Node) which looks up a single node from restbody of a specified type T.

The extractor object `Value` only pulls a single node from `restbody`. However, there are some other cases where the list of children of a node needs to be pulled from `restbody`. An example of this case is the `Block` node. The children of a Block node are statements (`Stm`). The processing of these statements is done one by one which normally yields a `Norm` node as a result, except for the abruption statements (`break`, `continue` and `return`). Listing 12 (left) shows the transition rules defined by the JBOOK to process `Block` node. The first rule checks for an empty block. The second rule matches a non-empty block, and sets `pos` to the first statement in the block, so that it gets processed in the next step. The third rule matches with a non-empty block that all statements have been processed (`Norm` node is a result of processing statement). The forth rule matches with a partially processed block where some statements in the block have been processed (`Norm`) and some have not been processed (`Stm`).

$\{\}$	$\rightarrow$ *yield(Norm)*
$\{^{\alpha 1}stm1 \ldots {}^{\alpha n}stm_n\}$	$\rightarrow pos := \alpha_1$
$\{^{\alpha 1}Norm \ldots \blacktriangleright Norm\}$	$\rightarrow$ *yieldUp(Norm)*
$\{^{\alpha 1}Norm \ldots \blacktriangleright Norm^{\alpha i+1}\ stm_{i+1} \ldots {}^{\alpha n}stm_n\}$	$\rightarrow pos := \alpha_{i+1}$

```
case Block(Nil,Nil) => yieldResult(Norm())
case Block(_,Nil)   => yieldResultUp(Norm())
case Block(_,stm::_)=> pos := stm
```

Listing 12: The transition rules to process block statements. JBOOK is on the left and Kiama/Scala is on the right.

We implement another extractor object which pulls the children of a node from `restbody`. It works by splitting the list of the children into two lists where the first list contains just the `Norm` nodes (already processed). The second list contains the `Stm` nodes which have not been processed. Hence, we can write our transition rules as in listing 12 (right). The first argument of the `Block` extractor is a list of `Stm` nodes that have been processed and the second is the list of unprocessed `Stm` nodes. We only need three rules, because the second and the fourth rules are written together in our third rule (`Block(_, stm :: _)`). If all the *`Stm`* nodes in that block have been processed, the second list is empty (`Block(_, Nil)`). Otherwise, the head of the second list is the next `stm` node to be processed (`Block(_, stm :: _)`). In Scala, the :: symbol is a method of the List[8] class. When using it in a pattern, the left hand side of the :: symbol is bound to the head node of the list and the right hand side is bound to the rest of the list.

We encapsulate these functionalities in an abstract class `SequenceExtractor`, so that we can share it among different type of nodes. Listing 13 shows the source code of the extractor object `Block`. The `SequenceExtractor` class is reused in the processing of arguments list in the implementation of the Java method call in Java$_C$ and Java$_O$ (class `Exprs`).

```
abstract class SequenceExtractor[T:ClassTag](listProcessedNodeType:List[Node])
{
 def extract(e:Node):Option[(List[Node], List[Node])]=
 {
  e match
  {
   case v: T =>
    val ls = childrenInRestbodyOf(e)
    if(Nil == ls)
      Some((Nil, Nil)) // empty sequence
    else
     findNode.index(notOfTheseTypes = listProcessedNodeType, inList = ls) match
     {
      case Some(idx) => Some(ls.splitAt(idx))// still some nodes to be processed
      case None      => Some(ls, Nil)        // all has been processed
     }
  }
 }
}
object Block extends SequenceExtractor[AST.Block](List(Norm()))
{
 def unapply(e:Node) = extract(e)
}
object Exprs extends SequenceExtractor[AST.Exprs](Group.valueNodeList)
{
 def unapply(e:Node) = extract(e)
}
```

Listing 13: The extractor object Block and Exprs. The objects extend from an abstract class SequenceExtractor. The unapply method returns two lists, the first list contains list of nodes that have been processed and the second contains the nodes which still need to be processed.

5.2.2.2 Scala Implicit conversion and the `pos` State

The JBOOK does not specify the type of `pos` state. It uses an abstract type `Pos`. We use an integer to represent the position of a node in an AST. In the JBOOK, `pos` is assigned to an

8 http://www.scala-lang.org/api/current/index.html#scala.collection.immutable.List

abstract position using greek alphabets (α, β and γ). Each AST node in our implementation has an `Int` value attached with it. Every node type in our implementation is derived from an abstract class `Node`. The method `position` of the root class `Node` returns its position in an AST. Normally if we want to code an expression written using the notation in the JBOOK like

$uop\ {}^{\alpha}exp\ \rightarrow pos := \alpha,$

our code would look like below

```
case UnaryOp(_, expr) => pos := expr.position
```

In Scala, an implicit function is a function that has the `implicit` keyword modifier. When a variable is assigned with a value from different type, the Scala compiler will search for an implicit function that can provide a conversion from the assigned type to the variable type. With an implicit function, we can rewrite the above code like below.

```
case UnaryOp(_, expr) => pos := expr
```

This simplifies our code and makes it easy to read. We do not just use the implicit conversion technique here, we have used it to greatly simplify the code in the implementation of the Java compiler which we describe in section 6.3.

5.3 Java$_C$

Java$_C$ introduces the dynamic semantics and the syntax of the procedural Java language. It adds support for the procedural call of static methods. Before calling a static method, all the states of Java$_I$ are pushed on to a stack frame and initialized to a new environment of the method call. Before returning from a method call, the top of the stack frame is popped. The states in Java$_I$ are set to the values that are popped from the stack frame.

5.3.1 States in Java$_C$

`globals` is a unary state where the location of the state is the `Field.ID` and the value of the state is a node type `Value[_]`. A `Field.ID` is a string that contains the fully qualified name of a static field. The value of each static fields are kept in this state through the lifetime of the program.

`meth` is a nullary state of type `Method`. A `Method` is a node type. The state `meth` is pointing to the current method that is being executed.

`classState` is a unary state where the location is the `Class` node and the value is a type `ClassState`. `ClassState` can be the following values; `Linked`, `InProgress`, `Initialized` and `Unusable`.

`superClass` is also a unary state. It is a dictionary where the key is a `Class` node and the value is also a `Class` node which is the super class of the key.

`frames` is a nullary state of type `List`. It serves as the stack frame between method calls.

Each item in the stack contains the caller method pointer (`meth`), `restbody`, `pos` of the node where the invocation occurs and `local` of the caller.

5.3.2 Transition Rules of Java$_C$

Similar to the transition rules of Java$_I$, the transition rules of Java$_C$ are also divided into the expression (execJavaExp$_C$) and the statement (execJavaStm$_C$).

The execJavaExp$_C$ sub-machine contains transition rules to process the getting/setting of static the fields value which are stored in the `global` state. The transition rules for invocation of a method in execJavaExp$_C$ require the arguments of the method to be evaluated before the invocation can be processed. Each of the arguments of a method invocation is an expression which the transition rules of Java$_I$ handles. An invocation is ready to take place when all the arguments have been processed into values (`Value[_]` node).

execJavaStm$_C$ contains transition rules to process the `return` statement. The statement is an abruption in addition to `break` and `continue` which are defined by Java$_I$.

In Java$_C$, `restbody` contains the AST of the method that is currently running. Thus, Java$_C$ requires an entry point method to be specified before running.

The JBOOK defines two rules to support the invocation of methods: `invokeMethod` and `exitMethod`. `invokeMethod` puts all the states defined in Java$_I$ on the stack frame and `exitMethod` does the opposite which is to restore the environment of the caller method. Listing 14 shows the implementation of the rules.

```
private def invokeMethod(nextPos:Int, method:Method, vals:Vals): Unit=
{
 firstPos = positionInBodyOfNode(method.body)
 frames := push(frames, (meth.value, restbody.value, nextPos, locals.value))
 meth := method
 pos := firstPos
 restbody := copyFromBody(fromNode=method.body)
 locals := makeLocals(method.signature.parameterList, vals)
}
```

```
private def exitMethod(result:Node): Unit=
{
 val (oldMeth, oldPgm, oldPos, oldLocal) = top(frames)
 meth := oldMeth
 pos := oldPos
 locals := oldLocal
 frames := pop(frames)
 firstPos = oldMeth.body.position
 if(methNm(meth) == "<clinit>" && result.isInstanceOf[Norm])
 {
  restbody := oldPgm
  classState(classOf(meth)) := ClassState.Initialized
 }
 else
   restbody := updateTree(result, oldPos, oldPgm)
}
```

Listing 14: The implementation of the invokeMethod and exitMethod rules in Scala/Kiama.

If the callee method returns a value, a corresponding `Value[_]` node is passed as a parameter to the `exitMethod` rule. `restbody` of the caller at the position of the method invocation is replaced with that `Value[_]` node and otherwise, for methods with no return value, the position is replaced with a `Norm` node.

We have found a problem with the definition of `exitMethod` in the JBOOK. The explanation of the problem is in section 7.4.1.

5.4 Java_O

Java_O introduces the object-oriented aspect of the Java language. The `Ref` and `Null` type are added to the type system. Java_O uses heaps to model the dynamic of objects' state. A heap is storage for keeping track of the values of the instance fields of a class. The JBOOK defines it as is:

data Heap = Object(Class, Map(Class/Field, Val))

In Scala, we define a heap as:

```
abstract class Heap
case class Object(clazz:Class, fields:MutableMap[Field, Value[_]]) extends Heap
```

Java_C keeps the values of all static fields in the `global` state, but they are kept uniquely by using the fully qualified name of the fields. However, Java_O keeps the values of all instance fields in the `heap`.

5.4.1 State of Java_O

Java_O adds one more state to keep track of the heap objects. It is a map from the `Ref` type to the `Heap` class.

```
val heap = new ParamState[Ref, Heap]("heap")
```

5.4.2 Transition Rules of Java_O

The JBOOK defines a sub-machine execJavaExp_O which consists of transition rules to process the following: the `this` reference is a `Ref` type. In every instance method call, `this` is stored in the `local` state (Java_I defines the state). The `new` keyword is used for creating a new instance of a class. The transition rule ensures that the class is initialized. Then, it creates a heap object for the class and place the heap in the `heap` state.

The transition rules for setting/getting of the values of instance fields of a class are similar to the setting/getting of static fields except instance variable (type `Ref`) needs to be resolved by looking up the value from the associated heap object.

The `instanceof` keyword is used for checking if a class is derived from another class. The transition rule searches the class hierarchy and yields up a boolean value `Node`.

The transition rule for casting of one class to another searches for a sub-class of the target. The machine stops, if the sub-class is not found.

The transition rules to process instance method calls have to determine the call kind. If the method belongs to the class, the call kind is *special*. However, if the method belongs to the parent class, the call kind is *super*. The last kind of method call searches the method from within the class and up to the parent classes, this call kind is *virtual*. To determine the kind of method calls, the JBOOK defines an abstract rule `callKind` for which our implementation is shown in listing 15.

```
private def callKind(pos:Int):CallKind=
{
 restbody(pos) match
 {
  case InstanceInvk(RefValue(ref), m, Values(params)) =>
   val c:Class = classOf(ref)
   val msig = makeSignature(m, params)
   c.lookupMethod(msig) match
   {
    case Some(method) => CallKind.Special
    case None         => env.superClassOf(c).lookupMethod(msig) match
                          {
                           case Some(method) => CallKind.Super
                           case None         => CallKind.Virtual
                          }
   }
 }
}
```

Listing 15: The implementation of callKind rule in Scala.

The visibility of the methods and fields are enforced at the earlier stages in the language processing pipeline.

5.5 JLS

The JBOOK defines an abstract function JLS to handle Java literal recognition and the computation for unary and binary operators. The notations that the JBOOK uses appear as there are three overloads of the function: one for converting literal to a `Value[_]` node (*JLS(lit))* and one for the computation of unary operator (*JLS(uop, val))* and another one for the computation of the binary operator (*JLS(bop, $val_{1,}$ val_2))*. To keep the notations that the JBOOK uses, we implement these functions in Scala as an object (a singleton) with three different overload of the `apply` methods. They allow the JLS object be used like a function. Listing shows our implementation of the JLS object.

```
object JLS
{
 def apply(lit:AST.Lit):Node = litParser.parse(lit.representation)
 def apply(op:Operator, v:Value[_]):Node=
 {
  v match
  {
   case BoolValue(t)   => doUnary(op, t)
   case ByteValue(t)   => doUnary(op, toInt(t))
   case CharValue(t)   => doUnary(op, toInt(t))
   case ShortValue(t)  => doUnary(op, toInt(t))
   case IntValue(t)    => doUnary(op, t)
   case FloatValue(t)  => doUnary(op, toDouble(t))
   case DoubleValue(t) => doUnary(op, toDouble(t))
  }
 }
 def apply(op:Operator, left:Value[_], right:Value[_]):Node=
 {
  val result = doBinary(op, left, right)
  systemValueToJLSValue(result)
 }
}
```

Listing 16: The implementation of the JLS function in Scala. The apply methods allow us to call the object just like a function: JLS(lit) invokes `apply(lit:AST.Lit)`, JLS(uop, val) invokes `apply(op:Operator, v:Value[_])` and JLS(bop, $val_{1,}$ val_2) invokes `apply(op:Operator, left:Value[_], right:Value[_])`

We use the Scala combinator parser library for literal recognition. In particular, we use the *JavaTokenParser*[9] class of the library to do most of the work. The class can recognize whole numbers, decimal numbers, string and floating numbers.

Handling the unary and binary operators requires type computation. For example, the expression `1 + 1.2` has result of type `double`, since `1.2` is a `double` literal. Table 3.3 and 3.2 in JBOOK define how the type computation can be done for the Java unary and binary operators. We have encoded the tables in Scala for which the code is shown in listing 17. To encode the table, we use pattern matching to match the operators with the operands. The function `max` in the source code returns the type that has the widest size.

```scala
def typeOf(op:Operator, opdType:Type):Type=
{
 (Operator.toString(op), opdType) match
 {
  case ("!", _:BoolType)        => BoolType()
  case ("~", a:IntegralType)    => max(a, IntType())
  case ("-"|"+", a:NumericType) => max(a, IntType())
 }
}
def typeOf(op:Operator, opdLType:Type, opdRType:Type):Type=
{
 (Operator.toString(op), opdLType, opdRType) match
 {
  case ("*"|"/"|"%"|"+"|"-", a:NumericType, b:NumericType)  => max(a, b, IntType())
  case ("<<"|">>"|">>>", a:IntegralType, b:IntegralType)    => max(a, IntType())
  case ("<"|"<="|">"|">=", a:NumericType, b:NumericType)    => BoolType()
  case ("=="|"!=", a, b) if(subType(a, b) || subType(b, a)) => BoolType()
  case ("&"|"^"|"|", a:IntegralType, b:IntegralType)        => max(a, b, IntType())
  case ("&"|"^"|"|", _:BoolType, _:BoolType)                => BoolType()
  case ("||" | "&&", _:BoolType, _:BoolType)                => BoolType()
 }
}
```

Listing 17: The type calculation for the Java unary and binary operators.

5.6 Summary

In this chapter, we described the techniques that we used to implement the Java dynamic semantic machines. These techniques include pattern matching, case classes, implicit functions, and extractor patterns.

Scala pattern matching is used as our main coding pattern. The machines pattern matches the case classes which are used for representing the abstract syntax tree. In addition, the type computation and the compiler (section 6.3) also use pattern matching extensively.

We use Scala extractor patterns to pull the up-to-date nodes from `restbody` which stores the results of evaluation. In effect, this process makes it appear as if the nodes of an AST are replaced by their evaluation results.

The implicit functions make the code more readable and frees us from the details of converting between types.

9 http://www.scala-lang.org/api/2.10.2/index.html#scala.util.parsing.combinator.JavaTokenParsers

Chapter 6 The Dynamic Semantics of the JVM and the Java Compiler in Kiama

This chapter describes our implementation of the dynamic semantics of the JVM and the Java compiler. We first describe how the JVM instructions are implemented, then we describe each level of the JVM machines that we implement. Last, we describe the implementation of the Java compiler which the JBOOK has left out the details of many functions.

6.1 The Abstract Instructions

To simplify the transition rules of the JVM, the JBOOK groups the actual JVM instructions into abstract instructions based on the similarity of their functions. For example the arithmetic and conditional operations are grouped into a `Prim(p)` abstract instruction where p is the type of the operator. The JBOOK summarizes all the abstract instructions in its reference section C.8 page 355.

Similar to the RISC example (section 3.2), we implement these abstract instructions using the Scala case classes. For example listing 18 shows the definition of the instructions supported by the JVM_I.

```
abstract class Instruction
case class Prim(opcode:PrimOp) extends Instruction
case class Load(t:MoveType, x:RegNo) extends Instruction
case class Store(t:MoveType, x:RegNo) extends Instruction
case class Dupx(s1:Size, s2:Size) extends Instruction
case class Pop(s:Size) extends Instruction
case class Goto(lab:LabelDef) extends Instruction
case class Cond(opcode:PrimOp, lab:LabelDef) extends Instruction
case class Halt() extends Instruction
```

```
data Instr = Prim(PrimOp)
           | Load(MoveType,RegNo)
           | Store(MoveType,RegNo)
           | Dupx(Size,Size)
           | Pop(Size)
           | Goto(Offset)
           | Cond(PrimOp,Offset)
           | Halt
```

Listing 18: The instructions for JVM_I, defined in Scala (left) and the notations used by the JBOOK (right).

6.2 The JVM ASM

The JVM is a word (4 bytes) stack-based machine. All the primitive data types in the Java language are mapped into one word or two words: boolean, char, short, int and float are mapped to a word and long and double are mapped into two words.

Our Scala/Kiama JVM ASM implementations are similar to the RISC example (section 3.2). Pattern matching and case classes are used extensively in the implementation. Implicit functions are used to simplify our code, for example list concatenation in the implementation of the compiler. The implementation is more straightforward than the implementation of the dynamic semantics of the Java language machines because the input to the JVM is just a list of instructions which is a linear structure. Executing a list of instructions is like walking on a straight line which is simpler than walking on a tree. Moreover, there are no node replacing actions (`restbody`) like in the machines for the dynamic semantics of the Java language, because a list of instructions that the JVM executes doesn't evolve as the evaluation proceeds since the complete state of the machine is represented by other states. The pattern matching in the JVM implementation is also simpler than the pattern matching of the dynamic semantics of the Java language.

Our Scala implementation is a one-to-one mapping with the mathematical definitions in the JBOOK. Listing 19 shows a side by side comparison of JVM_I between the mathematical definition in the JBOOK and our Kiama/Scala implementation.

```scala
private def execVMi(inst:Instruction): Unit =
{
 inst match
 {
  case Prim(p) =>
   val (opdP, ws) = split(opd, argSize(p))
   opd := opdP ::: JVMS(p, ws)
   pc := pc + 1
  case Dupx(s1, s2) =>
   val (opdP, ws1::ws2::_) = splits(opd, List(s1, s2))
   opd := opdP ::: ws2 ::: ws1 ::: ws2
   pc := pc + 1
  case Pop(s) =>
   val (opdP, ws) = split(opd, s)
   opd := opdP
   pc := pc + 1
  case Load(t, x) =>
   if(1 == size(t))
    opd := opd :+ reg.value(x)
   else
    opd := opd :+ reg.value(x) :+ reg.value(x + 1)
   pc := pc + 1
  case Store(t, x) =>
   val (opdP, ws) = split(opd, size(t))
   if(1 == size(t))
    reg(x) := ws(0)
   else
   {
    reg(x) := ws(0)
    reg(x + 1) := ws(1)
   }
   opd := opdP
   pc := pc + 1
  case Goto(offset) => pc := offset
  case Cond(p, offset) =>
   val (opdP, ws) = split(opd, argSize(p))
   opd := opdP
   if(1 == JVMS(p, ws).head)
    pc := offset
   else
    pc := pc + 1
  case Halt() => halt := "Halt"
  case _ =>
 }
}
```

```
execVM_I(instr) = case instr of
  Prim(p) → let(opd', ws) = split(opd, argSize(p))
             if p ∈ divMod ⇒ sndArgIsNotZero(ws) then
               opd := opd' · JVMS(p, ws)
               pc  := pc + 1

  Dupx(s1,s2) → let(opd',[ws_1,ws_2])=splits(opd,[s_1,s_2])
                  opd :=  opd' · ws_2 · ws_1 · ws_2
                  pc  := pc + 1

  Pop(s) → let(opd', ws) = split(opd, s)
             opd :=  opd'
             pc := pc + 1

  Load(t,x) → if size(t)=1 then opd := opd · [reg(x)]
                else opd := opd · [reg(x), reg(x + 1)]
                pc := pc + 1

  Store(t,x) → let(opd',ws) = split(opd,size(t))
                 if size(t) = 1 then
                   reg := reg ⊕ {(x, ws(0))}
                 else
                   reg := reg ⊕ {(x,ws(0)),(x+1,ws(1))}
                 opd := opd'
                 pc := pc + 1

  Goto(o) → pc := o

  Cond(p,o) → let(opd', ws) = split(opd,argSize(p))
               opd := opd'
               if JVMS(p,ws) then
                 pc := o
               else
                 pc := pc + 1

  Halt → halt := “Halt”
```

Listing 19: JVM_I ASM, Kiama/Scala Implementation on left and the mathematical definition of the machine in the JBOOK.

6.2.1 JVM_I

There are number of states defined by JVM_I: `halt`, `pc`, `opd`, and `reg`. `halt` is a nullary state of type `string`, if it is set to “Halt”, the JVM is terminated. `pc` (program counter) is a nullary state of type `Int`. It points to the current instruction. `opd` is a nullary state of type `List[Int]`. It acts as the operand stack. `reg` is a unary state where the location is a `RegNo` type which is an alias for a string and the value is the `Int` type. `reg` is similar to the `local` state in Java_I. It keeps track of the values of the local variables.

JVM_I defines instructions, transition rules for the core Java expressions and statements. These instructions include the primitive (`Prim`) operations for unary and binary operators, `Load`, `Store` and `Pop` for stack operations, direct (`Goto`) and conditional jump (`Cond`). The transition rules map these instructions to their semantics. For example, the semantic of the

`Prim(p)` is to take the operand from the top of the stack and do the calculation (`JVMS`) based on the primitive `p` (listing 20). The result of the calculation is pushed back on the stack. The JBOOK defines JVMS as an abstract function which we explain in section 6.4. We have added one extra instruction: `LabelDef` which has no execution semantics and its purpose is explained in section 6.3.2.

```
case Prim(p) => val (opdP, ws) = split(opd, argSize(p))
                opd := opdP ::: JVMS(p, ws)
                pc := pc + 1
```

Listing 20: Transition rule for the abstract instruction Primp(p). The ::: symbol is a method of the List class. It concatenates two lists together. JVMS is an abstract function defined by the JBOOK. Section 6.4 describes our implementation of JVMS.

6.2.2 JVM$_C$

JVM$_C$ defines instructions and transition rules for static method calls and static fields. Getting and setting values of static fields are handled by the `GetStatic` and `PutStatic` instructions. The `InvokeStatic` instruction is used for static method calls and the `Return` instruction is used for returning a value from a method call.

The states in JVM$_C$ are defined to keep track of class state, method calls and the values of static fields. `classState` is a unary state where the location is a `Class` node and the value is the state of the `Class` which can be `Linked` or `Initialized`. `stack` is a nullary state of type `List[Frame]`. It is used for saving and restoring the states of JVM$_I$ for method calls. The type `Frame` is a tuple of all state types defined by JVM$_I$. `globals` is a unary state whose location is the fully qualified name for a static field and the value is the `Value[_]` node of the static field. `meth` is a nullary state of type `Method` which is a node type. It points to the current method that is being executed by the JVM. `code` is nullary state of type `List[Instruction]`. It contains the list of instructions of a `Method` which `meth` state is pointing. `switch` is a nullary state of type `Switch` which can be: `NoSwitch`, `Call`, `Result` and `InitClass` which are the modes that the JVM can be in. When the JVM is in the `Call` mode, it pushes the current `Frame` on top of the `stack` state while the `Result` mode pops the previous frame from top of the `stack` state. The `InitClass` mode is used when a class is first used and has not been initialized.

JVM$_C$ defines rules `pushFrame`, `popFrame` and `switchVM`$_C$. `pushFrame` and `popFrame` are used in the `Call` and `Result` modes. We have found some problems with the rules that the JBOOK defines, and have devised fixes which we explain in more detail in section 7.4.2. The logic of context switching is defined in the `switchVM`$_C$ rule where listing 21 and 22 show the JBOOK definition and our implementation respectively.

```
switchVM_C = case switch of
    Call(meth, args) → if ¬isAbstract(meth) then
                            pushFrame(meth, args)
                            switch := Noswitch
    Result(res)      → if implicitCall(meth) then popFrame(0, []) else popFrame(1, res)
    InitClass(c)     → if classState(c) = Linked then classState(c) := Initialized
                          forall f ∈ staticFields(c)
                            global(c/f) := default(type(c/f))
                          if c = Object ∨ initialized(super(c)) then switch := Noswitch
                          else switch := InitClass(c)
```

Listing 21: The definition of switchVM$_C$ which the JBOOK defines.

```
private def switchVMc(): Unit=
{
 switch.value match
 {
  case Call(method, args) => if(method.access != AbstractModifier())
                              {
                                pushFrame(method, args)
                                switch := Noswitch()
                              }
  case Result(res)        => if(implicitCall(meth)) popFrame(0, List[Word]()) else popFrame(1, res)
                             switch := Noswitch()
  case InitClass(c)       => val cs:ClassState = classState(c)
                              if(cs == ClassState.Linked)
                              {
                               classState(c) := ClassState.Initialized
                               for(field <- staticFields(c))
                                 globals(field) := JVMS.valueToWords(JLS.valueFromType(field.fType))
                               if(c == JLS.Object ||initialized(superClassOf(c))) switch := Noswitch()
                               else switch := InitClass(superClassOf(c))
                              }
 }
}
```

Listing 22: Rule switchVM$_C$ in Scala. The JVM uses it to do context switching.

6.2.3 JVM$_O$

JVM$_O$ defines instructions for instance method calls. `InvokeSpecial` and `InvokeVirtual` are for early and late bound method invocations. The selection between the two instructions is done at compile time using the `callKind` (listing 15) rule defined by JVM$_O$. The `GetField` and `PutField` instructions are for getting and setting values of instance variables on heap. `New` is for instantiation of a class. `InstanceOf` for checking class hierarchy and `Checkcast` for type casting.

There is one state in JVM$_O$ which is the heap state. The definition and purpose are the same as the heap state in Java$_O$. However, the `Ref` type, which is a node type, needs to be converted to an integer number before it can be used with the `opd` state. The JBOOK defines the number as an address type (32 bits word). We need a lookup table which when given an address, returns a `Ref` node. The JBOOK does not define this but it makes as the `Ref` node can magically be converted into an address and vice versa.

The transition rules of JVM$_O$ uses the same context switching mode as in JVM$_C$.

6.3 The Java Compiler

The JBOOK presents a set of functions ($\mathcal{E}, \mathcal{B}, \mathcal{S}$) in each level. Given a Java AST, the functions compile the AST to a series of abstract instructions which the JVM can execute. The

$\mathcal{E}$ function compiles the expressions, $\mathcal{B}$ compiles the control flows and $\mathcal{S}$ compiles the statements. Listing 23 shows the definition of function $\mathcal{E}$ to compile the Java$_I$ core expressions.

```
ℰ(lit)                  = Prim(lit)
ℰ(loc)                  = Load(T(loc), l̅o̅c̅)
ℰ(loc = exp)            = ℰ(exp)·Dupx(0, size(T(exp)))·Store(T(exp), l̅o̅c̅)
ℰ(!exp)                 = ℬ₁(exp, una₁)·Prim(1)·Goto(una₂)·una₁·Prim(0)·una₂
ℰ(uop exp)              = ℰ(exp)·Prim(uop)
ℰ(exp₁ bop exp₂)        = ℰ(exp₁)·ℰ(exp₂)·Prim(bop)
ℰ(exp₀ ? exp₁ : exp₀)  = ℬ₁(exp0, if₁)·ℰ(exp₀)·Goto(if₂)·if₁·ℰ(exp1)·if₂
```

Listing 23: The compiler function $\mathcal{E}$. It compiles the core Java$_I$ expressions to a list of virtual instructions. una$_1$, una$_2$, if$_1$ and if$_2$ are labels not abstract instructions.

The definition of the function $\mathcal{E}$ maps each type of expression to a list of abstract instructions. The definition of the function shows labels are concatenated to the list of instructions which the JBOOK has left out the detail how the labels are removed before execution.

The notation $\overline{\texttt{loc}}$ is used as an abstraction for the location of a local variable `loc` in the JVM registers. In addition, the JBOOK defines a function T which when given an expression, computes the type of the expression. However, the JBOOK does not define how this computation can be accomplished. The sub-sections below explain how we implement these abstract functions in Scala.

6.3.1 The Java AST Compiler in Scala

Instead of using many overloaded functions $\mathcal{E}$ like in listing 23, we use the pattern matching of the syntax tree case classes, to implement the compiler. Listing 24 shows our implementation of the function $\mathcal{E}$ in Scala. The ::: symbol in our code is the method of the Scala List class. The method allows the concatenation of two or more lists. The JBOOK uses the · (middle dot) notion for this purpose.

```
private def E(node:Node): List[Instruction] =
{
 node match
 {
  case Lit(lit)                    =>  Prim(lit)
  case loc:Local                   => Load(T(loc), Bar(loc))
  case Asgn(loc, exp)              => E(expr) ::: Dupx(0, size(T(exp))) ::: Store(T(exp), Bar(loc))
  case UnaryOp(Op.NOT, exp)        => val una1 = LabelDef("una1") val una2 = LabelDef("una2")
                                      B1(exp, una1) ::: Prim(1) ::: Goto(una2) ::: una1 :::
                                     Prim(0) ::: una2
  case uop@UnaryOp(_, exp)         => E(exp) ::: Prim(uop)
  case bop@BinaryOp(_, exp1, exp2)  => E(exp1) ::: E(exp2) ::: Prim(bop)
  case InlineCond(exp0, exp1, exp2) =>  val if1 = LabelDef("if1") val if2 = LabelDef("if2")
                                       B1(exp0, if1) ::: E(exp2) ::: Goto(if2) ::: if1 :::
                                       E(exp1) ::: if2
 }
}
```

Listing 24: Scala implementation of the compiler E function. Each instance of LabelDef (una1, una2, if1, and if2) is stripped from the list of instructions before the JVM executes it.

6.3.2 Label Generator in Scala

We define an abstract instruction `LabelDef`. The JBOOK does not define this abstract instruction, but we need it in the compilation stage to temporary occupy a location in a list of instructions which the compiler is generating code. The location of a `LabelDef` instruction in a list of instructions represents the location of the next instruction which is a destination of a `Goto` or a `Cond` instruction. Before the JVM executes a list of instructions, it strips the `LabelDef` instructions from the list and computes the absolute location for each of the `Goto` and the `Cond` instruction in the list.

6.3.3 The T(exp) Function in Scala

We use the Java language specification version 1.2 as the guide to write the T function. The specification defines for each type of expression what is the type of the result of the evaluation of that expression. Listing 17 shows the unary and binary expression type calculation in Scala. Listing 25 shows our implementation of the function T. Our function T uses pattern matching to match each expression and returns the type. If any expression has multiple parts, then the function recurses on each of the part to have the type resolved. For example, the binary operator needs to have the type of the left and the right operand resolved before the function can compute the type of the whole expression.

```
private def T(e:Phrase): MoveType
{
   e match
   {
    case t: Type                      => t
    case lit: Lit                     => JLS.typeOf(lit)
    case v: Value[_]                  => v.typeT
    case loc: Local                   => T(loc)
    case Asgn(loc, exp)               => JLS.assignmentResultType(T(loc), T(exp))
    case FieldAsgn(cf, exp)           => JLS.assignmentResultType(T(cf), T(exp))
    case InlineCond(_, texp, fexp)    => JLS.inlineCondResultType(T(texp), T(fexp))
    case StaticFieldRef(c, f)         => T(env.fieldForName(c, f))
    case Field(_, _, fType, _)        => fType
    case Method(_, _, signature, _)   => signature.returnType
    case StaticInvk(c, m, _)          => T(env.methodForName(c, m))
    case UnaryOp(op, expr)            => JLS.typeOf(op, T(expr))
    case BinaryOp(op, lexp, rexp)     => JLS.typeOf(op, T(lexp), T(rexp))
   }
}
```

Listing 25: Type calculation for the compiler.

6.3.4 The $\overline{\text{loc}}$ Notation in Scala

The JBOOK defines this notation to convert a local variable to a register number. The register (the `reg` state of JVM$_I$) in this case is just a state in JVM similar to the `local` state of Java$_I$ which is a look up table from a local variable name to its value. However the JVM register is a word (4 bytes) register where each entry stores only a word. Thus, a double precision floating point (double) variable occupies two entries in the register. We implement the $\overline{\text{loc}}$ notation by using the local variable name as the location (key) of the `reg` state. If any variables require two entries, the next entry will have the "+1" string appended at the end of the variables name. For example, if the type of a variable `foo` is `double` (two words), the `reg` state will have two entries, the first one is "`foo`" which is associated with the high word and the second entry is "`foo+1`" which is associated with the low word of the value of the variable.

6.4 JVMS

Similar to the JLS function (section 5.5), the JBOOK defines an abstract function JVMS to handle the operations for the `Prim` and `Cond` abstract instructions. There are four cases that the JVMS function needs to handle:

- Converting literals to words: our implementation uses the JLS function to convert literals to a `Value[_]` node. The node is then converted to a list of words using the Scala bitwise operators.
- Unary operation: We first convert a list of words to a `Value[_]` node. The JLS function is used for calculating the result, which is then converted back to a list of words.
- Binary operation: This implementation is the same as the unary operation except there are two operands.
- `Cond` operation: The abstract instruction `Cond` handles two native JVM instructions: `Ifne` and `Ifeq`. The operand of the instruction is a word which can be 0 or 1. `Ifne` puts 1 back on the operand stack if it is given a 1, otherwise 0. `Ifeq` puts 1 back on the operand stack if it is given 0, otherwise 1.

6.5 Summary

This chapter described the implementation of the JVM and the Java to JVM compiler in Kiama/Scala. The implementation is more straightforward than the implementation of the dynamic semantic of the Java language machines. The techniques that we use to implement them are very similar to the RISC machine example from the Kiama library. Pattern matching and implicit functions are used extensively.

The implementation of the compiler is a little more complicated, because the JBOOK defines many abstract functions and does not clearly define how they can be implemented. For example, the `T(exp)` function for computation of the type of an expression, the $\overline{\text{loc}}$ operator which converts a local variable to a register number and the labels generation for jump destinations. We implement these abstractions in our code.

Chapter 7 Testing and Evaluation

This chapter describes how we test and evaluate our implementation. We first describe the testing. Then, we show the evaluation based on the criteria that we set. We also describe some minor problems of Kiama, and the workarounds that we use. Then, we compare our implementation with the AsmGofer implementation that the JBOOK provides. And last we show a few bugs that we found in the JBOOK and describe our fixes.

7.1 Testing

To test our implementation, we implemented a simple Java parser using the Scala Combinator Parser library [30]. The parser combinator concept is well known. Several simple parsers can be combined to produce a complex parser. It allows us to write our parser in just few hundred lines of code. More importantly, having a parser as an integral part of Scala has simplified our work, since because our parser is written in Scala, it is easily integrated with the rest of our code.

We used the example Java code available in the JBOOK to test both the dynamic semantics of the Java language and the JVM. These programs exercise all of the transition rules. We also wrote additional test cases, where the test code in the JBOOK lacks, for example recursion, nested function calls and floating point arithmetic. The source code is available at `https://bitbucket.org/psksvp/compx/overview`.

For each test case, we generated the AST from the test Java source using our parser. We ran the dynamic semantics machines with the AST. For the JVM, the compiler compiles the AST to a list of the abstract instructions for the JVM to run. The dynamic semantic machines and the JVM were required to produce the same result from the AST.

7.2 Evaluation

We found the combination of Kiama and Scala allows us to closely replicate the mathematical definition of the machines in JBOOK. Kiama ASM has all the features needed to allow us to implement the states and the execution model of ASM presented in the JBOOK. The Scala pattern matching syntax, implicit functions and extractor patterns allow us to closely replication the JBOOK mathematical notation. The following sections elaborate our evaluation, based on the criteria we set in section 1.2.

7.2.1 Replication of the mathematical notation and definition

We were able to accomplish this criteria by using many techniques which we have described in chapter 5 and 6. The notations are not exactly the same, but they are very close.

The JBOOK book uses pattern matching notation to define the transition rules which we use Scala pattern matching. The sample code below shows the transition rules for unary operators; on the left is the JBOOK mathematical notation and our Kiama/Scala code is on the right. The matching order that the JBOOK uses is from the most general ($\text{uop}\ {}^{\alpha}\text{exp} \rightarrow \text{pos} := \alpha$) to the most specific ($\text{uop}\ {}^{\blacktriangleright}\text{val} \rightarrow \text{yieldUp(JLS(uop, val))}$) while the Kiama/Scala code is the opposite, because Scala pattern matching will try to match each pattern in the sequence in the order that they are written. The reordering of the patterns do not change the clarity of the machines.

$uop\ {}^{\alpha}exp \rightarrow pos := \alpha$	`case UnaryOp(uop, Value(v)) => yieldResultUp(JLS(uop, v))`
$uop\ {}^{\blacktriangleright}val \rightarrow yieldUp(JLS(uop, val))$	`case UnaryOp(_, exp)        => pos := exp`

The JBOOK uses greek letter notation to refer to a node (`pos := α`) while we use the node itself (`pos := exp`). We feel that our code is clearer to read than using the greek letter notation.

We cannot replicate the pattern (the left hand side of the transition rule before the → symbol). The notation that the JBOOK uses, is close to the textual notation of Java code, but we use the cases classes for the pattern. Listing 7 (page 21) summarizes the Java$_I$ syntax comparison between the JBOOK and Scala. After we have finalized our implementation, we discover that we might be able to replicate the JBOOK notations of the syntax tree by using the Concrete Object Syntax [32].

7.2.2 One-to-one mapping of the transition rules

Listing 26 shows the mathematical definition of Java$_I$ expression machines and our implementation in Kiama/Scala. The listing shows a one-to-one mapping between the transition rules. The same result is seen for the Java$_C$ and Java$_O$. Our JVM implementation also sees the same result, where our implementation is a one-to-one mapping with the mathematical definition in the JBOOK (listing 19 page 28, listing 23 page 36 and listing 24 page 36).

7.2.3 Reusability of our implementation

Kiama ASM is an embedded solution so ASM definitions can easily be used by other Scala programs and any JVM languages in general. Listing 31 shows a Java source program is defined as a string. The Parser class parses the source and produces an AST. The method `execute` of class `JavaSemanticMachine` and class `JavaVirtualMachine` takes the AST and execute it. `JavaSemanticMachine` executes the AST directly while `JavaVirtualMachine` invokes the compiler to compile the AST before executing. The classes can be invoked from any Scala applications to execute Java source code.

7.2.4 No modification to Kiama to implement our code

In this study, we want to implement the case study ASM models without modifying any part of Kiama. The main reason was to see the limitations and problems of Kiama when we scale up the complexity of the ASM models. We are able to implement all the ASM models that we planned without changing Kiama at all.

We encountered some problems while we were implementing the ASMs for this study.

However, we consider that these problems are minor problems because they did not stop us from completing our work. The sub-sections below explains what we have done to work around the problems. These workarounds might suggest areas of investigation for future updates of the Kiama ASM library.

```
$execJavaExp_I$ = case context(pos) of
  lit → yield(JLS(lit))
  loc → yield(locals(loc))

  uop $^\alpha$exp → pos := $\alpha$
  uop $^\blacktriangleright$val → yieldUp(JLS(uop, val))

 $^\alpha exp_1$ bop $^\beta exp_2$ → pos := $\alpha$
 $^\blacktriangleright$val bop $^\beta exp_2$ → pos := $\beta$
 $^\blacktriangleright val_1$ bop $^\blacktriangleright val_2$ → if $\neg$(bop $\in$ divMod$\wedge$ isZero($val_2$)) then yieldUp(JLS(bop, $val_1$, $val_2$))

  loc = $^\alpha$exp → pos := $\alpha$
  loc = $^\blacktriangleright$val → locals := locals $\oplus$ {(loc, val)}
                          yieldUp(val)

 $^\alpha exp_0$ ? $^\beta exp_1$ : $^\gamma exp_2$ → pos := $\alpha$
 $^\blacktriangleright$val ? $^\beta exp_1$ : $^\gamma exp_2$ → if val then pos := $\beta$ else pos := $\gamma$
 $^\alpha$true ? $^\blacktriangleright$val : $^\gamma exp_2$ → yieldUp(val)
 $^\alpha$false ? $^\beta$exp1 : $^\blacktriangleright$val → yeildUp(val)
```

```scala
private def execJavaExpI: Unit=
{
 val node = context(pos)
 node match
 {
  case lit:Lit                                  => yieldResult(JLS(lit))
  case Local(name)                              => yieldResult(locals(name))
  case UnaryOp(op, Value(v))                    => yieldResultUp(JLS(op, v))
  case UnaryOp(_, exp)                          => pos := exp
  case BinaryOp(op, Value(left), Value(right))  => yieldResultUp(JLS(op, left, right))
  case BinaryOp(_, Value(_), exp2)              => pos := exp2
  case BinaryOp(_, exp1, _)                     => pos := exp1
  case Asgn(loc, Value(v))                      => locals(loc) := v
                                                   yieldResultUp(v)
  case Asgn(_, exp)                             => pos := exp
  case InlineCond(BooleanValue(_), Value(v), _) => yieldResultUp(v)
  case InlineCond(BooleanValue(_), _, Value(v)) => yieldResultUp(v)
  case InlineCond(BooleanValue(v), exp1, exp2)  => if(v.value) pos := exp1 else pos := exp2
  case InlineCond(exp0, _, _)                   => pos := exp2
 }
}
```

Listing 26: The JBOOK definition of $execJavaExp_I$ (top) and Kiama/Scala implementation (bottom) shows a one-to-one mapping between the transition rules.

7.2.4.1 AST, body and position of a node in body

The processing of loop statements like the `while(){}` loop and the `label: … continue label` abruption, require the original copy of the sub-structure of the statements to replace the sub-structure in `restbody` of the statements in each iteration. For example, in an iteration of a while loop, each statement in the body of the loop is transformed into a `Norm` node. Thus, at the next iteration, the machine needs the original copy of the body of the while loop to continue the execution. Hence, the JBOOK defines `body` to be a state to store the syntax tree of a program that is being executed. It is initialized at the beginning and never changed throughout the execution of the program. `body` is a unary state. When given an integer number, it returns a node at that position. In each iteration of any loop statement, the sub-structure of the loop is copied from `body` into `restbody`.

We define `body` as a `ParamState[Int, Node]` in our implementation. And to keep track of the position of nodes in `body`, we add an additional unary state `positionInBodyOfNode`. When given a node, `positionInBodyOfNode` returns the original position of the node in `body`. In other words, the state is a reverse lookup of the `body` state. We need the `positionInBodyOfNode` state, because to copy the sub-structure of a loop node from `body`, we need to know the position of the node in `body`.

`positionInBodyOfNode` is an instance of class `ParamState[Node, Int]`. Internally `ParamState[Node, Int]` uses a Scala mutable map to store the reference of a `Node` class as the key and `Int` class as the value. A problem occurs when we initialize the `positionInBodyOfNode` state from an AST. To clearly understand the problem, let's look at an example of initializing the `possitionInBodyOfNode` state with a simple binary expression `1 + 1`. The parser would generate an AST like this `BinaryOp(Operator.plus, Lit(1), Lit(1))` which can be visualized as follows:

```
      BinaryOp
     /   |     \
    +  Lit(1) Lit(1)
```

If we try to initialize `possitionInBodyOfNode` with the AST, we would get an `InconsistentUpdateException`, because the Kiama updater thinks that the left operand `Lit(1)` and the right operand `Lit(1)` are the same node because Kiama's ASMs use value equality to compare locations not reference equality. The ASM does not allow a state to be updated at the same location with different values in a step. Listing 27 shows an example code of this problem.

```
val m = makeMachine()
val s = new m.ParamState[Node, Int]("s")
s(BinaryOp(Operator.plus, Lit("1"), Lit("1"))) := 1
s(Lit("1")) := 2
s(Lit("1")) := 3
m.performUpdates()
```

Listing 27: An example problem scenario where two different case classes Lit that have the same data "1". Even though the two Lit("1") nodes are two different objects, the Kiama updater thinks they are the same since they have the same value.

In Kiama, while the `Machine` class is running, a series of updates to a state updates within a step is recorded in a sequence (`Seq[Update]`). At the end of a step the `performUpdates` method is called. To check for inconsistency before the actual update, the method builds a list of unique updates to compare with the intended updates list. If they are not the same, an `InconsistentUpdateException` is raised. In Scala, an expression `Lit(1) == Lit(1)` evaluates to `true`, because the content of the two objects are compared, though they are two different references. The `performUpdates` method builds a list of the unique updates by calling on the method `groupBy` of the `Seq` class. Internally, the `groupBy` method may be using the == operator to compare the key of each update, thus `Lit(1)` is the same as another `Lit(1)`. Hence, the location `Lit(1)` appears twice on the intended update list while `Lit(1)` appears only once in the unique updates list. Hence, a mistaken of inconsistency update is detected.

To work around the problem, instead of using the reference of a Node as the location of the `possitionInBodyOfNode` state, we use a unique identification string as the location. The identification string is set to a uuid value when a Node class is instantiated. Hence, the `possitionInBodyOfNode` state is `ParamState[Node.ID, Int]`.

7.2.4.2 Accessing value of a unary state

To access the current value of a location in a unary state, the notation used by the JBOOK is `f(l)` where `f` is the state name and `l` is the location. In Kiama, the `ParamState` class provides us with same notation. In Scala, any classes or objects that define a method `apply`, can use the notation. The `apply` method allows the class and the object with the `apply` method be used like a function call. For example `"Scala is cool"(4)` yields `'a'` as the result, because the Scala String class implements the `apply` method to return a character at the index given by the parameter of the `apply` method.

The `apply` method of the class `ParamState` takes a location as its parameter. The apparent meaning of this is that the method should return the value of the state at the location specified in the parameter. However, the method returns an instance of the class `ParamUpdater`. To make this appears as if the `apply` method is returning the current value, the Kiama `Machine` class provides an implicit function which implicitly converts an instance of `ParamUpdater` to the last updated value of that state at the current step. This strategy works most of the time, but there are cases where it does not work.

To illustrate this problem, let's look at our implementation of the `context` rule which is used for determining the next node that is needed to be evaluated. Listing 28 shows the code of the rule. The code on the left does not work correctly because the inferred type of the variable *n* is the `ParamUpdater` class, hence the statements `n.isInstanceOf[Bstm]` and `n.isInstanceOf[Expr]` would always yield `false`. In this case the Scala compile does not use the implicit function mentioned in the previous paragraph, because the type of variable `n` is inferred from the return type of the `restbody` function (the `apply` method of the `ParamUpdater` class*)*. However the code on the right is working, because when the variable node is declared, the type of the node is specified (`val n:Node`). Hence, the Scala type checker uses the implicit function to convert the instance of the class `ParamUpdater` to type `Node`. Listing 29 shows the implicit function that is used for the conversion.

```
private def context(aPos:POS): Node =
{
 val n = restbody(aPos)
 if(aPos =:= firstPos    ||
    n.isInstanceOf[Bstm] || n.isInstanceOf[Expr])
  restbody(aPos)
 else
  restbody(up(aPos))
}
```

```
private def context(aPos:POS): Node =
{
 val n:Node = restbody(aPos)
 if(aPos =:= firstPos    ||
    n.isInstanceOf[Bstm] || n.isInstanceOf[Expr])
  restbody(aPos)
 else
  restbody(up(aPos))
}
```

Listing 28:context(pos). The code on the left does not work because, the type of `val n` is not a Node type while the code on the right works because `val n` is explicitly declared as type Node. It works, because the Scala compiler automatically uses the implicit function in Listing 29 to convert `ParamUpdater` to Node.

```
implicit def paramUpdaterToU[T,U,V >: U] (up : ParamUpdater[T,U]) : V = up.state.value (up.t)
```

Listing 29: The implicit function that can convert an instance of ParamUpdater to the type of the value of a state

This problem also occurs when an instance of the class `ParamUpdater` is used in the conditional part of an `if` statement or the selector part of a match statement. To work around this problem, we explicitly call the method `value` of the class `ParamUpdater`. The method returns the current value of the `restbody` state at location `pos`. Listing 30 shows this work-around.

```
case Load(t, x) => if(1 == size(t)) opd := opd :+ reg.value(x) //opd :+ reg(x)
                   else             opd := opd :+ reg.value(x) :+ reg.value(x + 1)
                   pc := pc + 1
```

Listing 30: The transition rule for the Load instruction of JVMI shows looking up the value at location x of the reg state requires calling the method value. The :+ symbol is a method of the List class which appends an element to the end.

7.3 Comparison with the AsmGofer Implementation

This section compares our implementation with the implementation by the JBOOK in AsmGofer. The AsmGofer implementation is considered to be a reference implementation. Hence, the comparison supports our aims for this study.

7.3.1 Transition Rules

In our implementation, we are able to code the machine definitions that map one-to-one with the definitions in the JBOOK. The AsmGofer implementation does not map one-to-one and there are many noises which makes the code hard to read.

Our implementation uses Scala pattern matching on the case classes while the AsmGofer implementation uses pattern matching on lists called. The AST in the AsmGofer implementation is represented using the zippers [33] method where a node is represented by a list (path) from the root node to the node. The AsmGofer implementation uses `Term` as the type of the node.

Table 3 shows a side by side comparison of the transition rules for the literal (`Lit`) processing and local variable (`Local`) value lookup. The Scala/Kiama code is closer to the mathematical notation used by the JBOOK than the AsmGofer implementation. The AsmGofer code has extra scaffolding in the pattern matching terms which make the code harder to read and it can be difficult to debug.

The AsmGofer code has extra transition rules to do some error handling. For example the rule `(Term(Lit(NoValue),_)[],_)->fail(nullPointerException)` is for handling the unrecognized literal error. Normally, the checking of literals should be done earlier with the syntactic analyzer, however, the JBOOK uses an abstract function `JLS` for this purpose, hence, we implement the function to do the same which as a result, allows our code to closely replicate the JBOOK notation. The detail of the JLS function is given in section 5.5. The AsmGofer supports the String class while our implementation does not, because the

JBOOK does not specify the type string.

JBOOK	Scala/Kiama	AsmGofer

lit → yield(JLS(lit))
loc → yield(locals(loc))
	`case lit:Lit      => yieldResult(JLS(lit))`
	`case Local(name) => yieldResult(locals(name))`
		`(Term(Lit(NoValue),_)[],_) ->`
		`fail(nullPointerException)`
		`(Term(Lit(lit),_)[],_) | isString(lit) ->`
		`createStringObject(stringVal(lit))`
		
		`(Term(Lit(lit),_)[],_) -> yield(Val(lapply(lit)))`
		
		`(Term(LocAcc(loc),t)[],_) ->`
		`yield(Val(locals#(loc)))`

Table 3: Transition rules for literal processing and local variable look up. On the left is the notation used in the JBOOK, in the middle is our Scala/Kiama code and on the right is the AsmGofer code.

7.3.2 Node position representation and pos

Every node in an AST has a position associated with it. The JBOOK uses abstract positions where the greek letters (α, β *and* γ) represent the positions. Our implementation uses an integer to represent the position of a node in an AST. AsmGofer implementation also uses integers.

The JBOOK uses the notation `pos := α` to assign `pos` to a node which the position is indicated by `α`. Our Scala/Kiama implementation uses `pos := node` where `node` is a node in an AST that is going to be processed in the next step. An implicit function converts the node to its position automatically. The AsmGofer implementation uses a state to look up the position of a node. The state is defined as followed `down :: (Pos,Nat) -> Pos` which is a 2-ary state which when given a `Pos` and a `Nat` (natural number) returns the position of the nth child of a node at `Pos`. For example, `pos := down(pos, 0)` where `down(pos, 0)` is the first child of a node at `pos`. Table 4 shows the transition rules to process a unary operator written using the JBOOK, Scala/Kiama and AsmGofer notation.

JBOOK	*uop* $^{\alpha}$*exp* → pos := α *uop* $^{\blacktriangleright}$*val* → *yieldUp(JLS(uop, val))*

Scala/Kiama	`case UnaryOp(op, Value(v)) => yieldResultUp(JLS(op, v))`
	`case UnaryOp(_, exp)        => pos := exp`

AsmGofer	`(Term(Una(op),_)[exp],_)-> pos := down(pos,0)`
	`(_,Term(Una(op),_)[Term(Val(val),t)[]])|pos==down(up(pos),0)->`
	`yieldUp(Val(uapply(op,t,val)))`

Table 4: The comparison of the transition rules to process a unary operator. The transition rules in Scala/Kiama are written in reverse, because pattern matching in Scala requires more specific patterns to appear before the more general ones.

As we see from table 4, our code in Scala/Kiama is more intuitive to write and easier to understand than the AsmGofer code. The code `pos := exp` gives an intuition right away that the `exp` is assigned to `pos` which to be processed in the next step while the AsmGofer expression `pos := down(pos,0)` does not give any intuition. In addition, the use of the side condition `pos==down(up(pos),0)` in the pattern of the second rule makes pattern more complex than the Scala/Kiama version. We do not need the side condition because the pattern is already ensured that `pos` is pointing to the `▸val` node.

7.3.3 Reusability of the Java ASMs.

The ASMs that we implement for this study can be invoked from any JVM language that supports interoperability with Scala. Listing 31 shows an example of invoking the dynamic semantic machine to interpret a Java code. AsmGofer, in contrast, is implemented by modifying the Gofer runtime. It supports no interoperability with anything except itself.

Though Gofer and AsmGofer have the same syntax, the execution model of AsmGofer is different from that of Gofer. As a result, Gofer code and AsmGofer code may have different semantics, thus they cannot be mixed.

In contrast to Kiama ASM, the execution model and the semantics of ASM is encapsulated in the `Machine` class, thus, Kiama ASM can be mixed with normal imperative or functional style Scala code. This offers great reusability.

```
val source = “””
class Main
{
  public static double sine(double x)
  {
    double term = 1.0;
    double sum = 0.0;
    for(int i = 1; term != 0.0; i = i + 1)
    {
      term = term * (x / i);
      if(1 == i % 4)
        sum = sum + term;
      else if(3 == i % 4)
        sum = sum - term;
    }
    return sum;
  }

  public static void entryPoint()
  {
    double s = Main.sine(30.0);
    sys.print(s);
  }
}
“””
```

```
val p   = new Parser
val ast = p.parseProgram(source)

JavaSemanticMachine.execute(ast,
                            "Main",
                            "entryPoint")

JavaVirtualMachine.execute(ast,
                           "Main",
                           "entryPoint")
```

Listing 31: An example shows how the machine classes we have defined using Kiama can be used to execute a Java source. `JavaVirtualMachine.execute` *invokes the compiler to compile the AST before executing while* `JavaSemanticMachine.execute` *executes the AST directly.*

7.4 The JBOOK bugs

While we encoded the machine definitions using Kiama/Scala, we encountered several problems with the JBOOK definitions. After some hand tracing of the definitions, we have found where the problems are and provide fixes in our implementation. This section provides detail descriptions of the problems and the fixes that we made.

7.4.1 *firstPos* is not restored

In Java$_C$, the JBOOK presents the rule `exitMethod` whose purpose is to set the execution environment back to the caller environment. The environment includes all the states defined by Java$_I$ (`pos, locals and restbody`) and the state `meth` (current method) defined by Java$_C$. The environment is popped from a stack (the state `frames` of Java$_C$). Listing 32 shows the definition of the rules.

```
exitMethod(result) =
  let(oldMeth, oldPgm, oldPos, oldLocals) = top(frames)
  meth   := oldMeth
  pos    := oldPos
  frames := pop(frames)
  if methNm(meth) = “<clinit>” ∧ result = Norm then
    restbody                 := oldPgm
    classState(classNm(meth)) := Initialized
  else
    restbody := oldPgm[result/oldPos]
```

Listing 32: The exitMethod rule. (JBOOK page 66)

We found a problem with the definition of the rule when we run the dynamic semantic machine with a Java source file that makes nested methods call, the machine would not reach a fixed-point. The cause of the problem is the variable `firstPos` which is defined by Java$_I$ to indicate the position of a root node in `restbody` which is the start point of the execution. In Java$_I$ the value of `firstPos` remains constant. However, Java$_C$ extends the meaning of `firstPos` to be the top node of the body of the current executing method. Hence, the value of `firstPos` changes according which method the machine is currently executing (the state `meth` defined by Java$_C$).

The rule `exitMethod` defined by the JBOOK does not set the value of `firstPos` back to the top node of the caller method. As a result, the transition rules which are responsible for returning from a method call are never activated. Listing 33 shows the transition rules defined by execJavaStm$_C$. The rules depend on the value of `firstPos` being point to the current top node of a method that is being executed. The conditional statements in listing 33 always evaluates to false if `firstPos` is not set to the position of the top node of the caller method. This problem does not occur if there is only one layer method call or there is no method call from an entry point method. Listing 14 (page 28) shows our `exitMethod` implementation which sets `firstPos` back to the caller method body.

```
Return      →  if pos = firstPos ∧ ¬null(frames) then exitMethod(Norm)
Return(val) →  if pos = firstPos ∧ ¬null(frames) then exitMethod(val)
```

Listing 33: The transition rules of execJavaStm$_C$ that process returning from a method call. (JBOOK Fig. 4.5)

7.4.2 JVM `pushFrame` and `popFrame`

JVM$_C$ defines two rules: `pushFrame` and `popFrame`, to support invoking and existing from method calls. `pushFrame` is used to save the environment of the current method before invoking another method. `popFrame` is used to restore the environment back to that of the caller method. Listing 34 shows the JBOOK's definition of `pushFrame` and `popFrame`.

The environment consists of the program counter (`pc`), the operand stack (`opd`), the local register (`reg`) and the byte codes of the method to be invoked (code). The `code` state is a unary state which when passed a program counter, returns an instruction that is going to be executed in the next step.

`pushFrame(newMeth, args) =`
`  stack := stack · [(pc, reg, opd, meth)]`
`  meth  := newMeth`
`  pc    := 0`
`  opd   := []`
`  reg   := makeRegs(args)`
	`popFrame(offset, result) =`
	` let(stack',[(pc', reg', opd', meth')] = split(stack, 1)`
	`  pc    := pc' + offset`
	`  reg   := reg'`
	`  opd   := opd' · result`
	`  meth  := meth'`
	`  stack := stack'`

Listing 34: The JBOOK definition of the rules pushFrame and popFrame.

The compilation of a Java program is done by compiling all methods of all classes into byte-codes. A class file organizes byte-codes of all methods of a class. Hence, a Java class has a corresponding class file. Therefore, given a class/method, the JVM loads the byte-codes from the class/method's class file to execute.

The definitions of `pushFrame` and `popFrame` in the JBOOK do not mention the `code` state at all. The `code` state is a list of instructions (byte-codes) of the current executing method. The `code` state cannot remain the same, it must be set to the byte-code of the method that is going to be invoked, otherwise the JVM would execute the byte-codes of the caller method again. In addition, at `popFrame`, the `code` state must be set back the byte-code of the caller method.

Our implementation fixes this problem by adding a unary state `codeOfMethod`. The state when given a method, returns the byte-code of the method. In `pushFrame`, we set the code state to the byte-code of the method that is going to be invoked and set it back to the caller method at `popFrame`. Listing 35 shows our implementation.

```
private def pushFrame(newMeth:AST.Method, args:Args): Unit =
{
 stack := push(stack, (pc.value, reg.value, opd.value, meth.value))
 meth := newMeth
 pc := 0
 opd := List[Word]()
 reg := makeRegs(newMeth, args)
 code := codeOfMethod(newMeth)
}
```

```
private def popFrame(offset: Offset, result:List[Word]): Unit =
{
 val (pcP, regP, opdP, methP) = top(stack)
 pc := pcP + offset
 reg := regP
 opd := opdP ::: result
 meth := methP
 stack := pop(stack)
 code := codeOfMethod(methP)
}
```

Listing 35: Our implementation of pushFrame and popFrame with fixes (codeOfMethod state is added).

An interesting observation about the definition of `pushFrame` and `popFrame` in the JBOOK is the stack is manipulated differently from `invokeMethod` and `exitMethod`. `pushFrame` and `popFrame` do not use the `push` and `pop` definition that the JBOOK describes in section 2.3 (page 27 in the JBOOK). However, they use list operations where `pushFrame` concatenates the environment of the calling method at the bottom of the `stack` state. While `popFrame` uses `split` to split the stack frame to simulate `pop`. We feel that this is inconsistent. Therefore, we use `push` and `pop` in our implementation.

7.4.3 Declaration of variable with assignment

The JBOOK provides some example Java expressions and statements that the transition rules should be able to process, however the JBOOK example 3.2.2 and 3.2.3 (shown below) cannot be processed by the transition rules.

```
int i = 2;
int j = (i = i * i) + i;
```

The Java$_I$ machines do not define any transition rule that can process the declaration of variables with assignment statement, for example `int i = 2;`. The statement contains both a declaration of a variable statement (`int i`) and an assignment expression (`i = 2`). The statement `int i` is processed by the rule `Type x; → yield(Norm)` and the expression `i = 2` is processed by the following rules.

```
loc = ᵅexp → pos := α
loc = ▸val → locals := locals ⊕ {(loc, val)}
             yieldUp(val)
```

To fix this problem, we add two more transition rules to process the statement. Listing 36 shows the rules which is part of execJavaStmI.

```
①  case VarDeclAsgn(vtype, local, Value(v)) =>  locals(local) := v
                                                yieldResultUp(Norm())
②  case VarDeclAsgn(_, _, expr)             =>  pos := expr
```

Listing 36: The extra rules we add to process the variable declaration with assignment statement.

The `int i = 2;` statement would, first, get processed by the rule ②. Once the literal `2` has been processed which yields a `Value(2)` node, rule ① would associate the variable `i` with the `Value(2)` to the `locals` state of Java$_I$.

7.5 Summary

Our machines are tested with the sample Java code in the JBOOK. In addition, we wrote some extra testing code to fill in gaps where the JBOOK test codes are lacking. The Scala parser combinator library allowed quick implementation of our simple parser.

We have met the criteria that we set at the beginning. The criteria aim to make coding executable ASM using Scala/Kiama as close as possible to the ASM mathematical definition. However, we have encountered some minor problems with Kiama for which we showed the workarounds.

The comparison with the AsmGofer implementation by the JBOOK showed that our Scala/Kiama implementation is much closer to the JBOOK mathematical definitions. We have also pointed out bugs in the mathematical definitions and the fixes provided in our implementation.

Chapter 8 Conclusion

In this study, we have used Kiama ASM to implement the dynamic semantics of the Java language, the virtual machine and the compiler for the Java language. We use the JBOOK as our reference. The JBOOK describes the dynamic semantics and the virtual machine using the ASM method. Our aim was to see if Kiama ASM can be scaled up to implement complex machines. The evaluation criteria was to be able to closely replicate the mathematical definitions that are used in the JBOOK.

We have shown that the combination of Scala and Kiama allows our code to closely replicate the machine definitions in the JBOOK. The transition rules in our code map one-to-one to the rules in the JBOOK. The techniques that we use to accomplish are Scala pattern matching, implicit functions and extractor patterns. The Kiama ASM library provides us with the ASM execution model and state.

We found some minor problems with Kiama ASM. We were able to provide the workarounds without modifying any part of Kiama ASM. While we are coding and testing our implementation, we also found bugs in the JBOOK and provide fixes in our code.

The JBOOK has also implemented their machines using AsmGofer. The comparison between our Scala/Kiama code and the AsmGofer code shows that our code is more readable and closer to the mathematical definitions. In addition, Kiama is a library, the components are callable, thus what we have implemented can easily be reused by any JVM languages that support interoperability with JVM language. The AsmGofer implementation cannot be reused because AsmGofer modifies the runtime of Gofer, thus, the implementation cannot be interoperable with any other programming languages.

8.1 Significance of this study

This study shows that ASM implementation as a library can be used to specify complex ASM models. A Library approach is less complex than implementing an ASM programming language, because the number of lines of code required to implement a library is much less than implementing a programming language. The programming language approach can provide syntax which is closer to what the Lipari guide has used. However we have shown that using the expressiveness of Scala and the Kiama ASM library, we are able to closely replicate the mathematical notations of the reference models presented in the JBOOK.

The significance of this result is one can take ASM mathematical definitions and code them up in Scala to execute them and be able to use the code in an application development. We believe that the techniques that we have used to accomplish the goals in this study can be applied to other applications of ASM.

8.2 Future work

ASM can be thought of as a programming paradigm. The ASM programming model consists of states and rules. The execution model uses discrete time-steps where in every step, all rules are executed which may update states. The updates to states at step t are not visible until the next step t + 1. Hence, debugging ASM rules can be difficult. Internally, Kiama ASM provides some diagnostics when states are updated inconsistently. However, that is not enough when we have to deal with complex transition rules such as those of the JBOOK. In this study, we use the debugger provided with our integrated development environment and paper and pencil to manually trace to debug our code. We feel that there should be a graphical debugger designed specifically for debugging ASM rules. The debugger should allow us to see n-arity states visually in the current and the next step while we are single stepping through the rules. The would be a good future addition to Kiama ASM.

Currently Kiama ASM allows only nullary and unary states. There are applications that need more than one arity state; e.g. cellular automata simulation. We think that this feature can be done using the Scala macro, which should allow n-arity state be generated at compile time.

References

[1] T. Ekman and G. Hedin, “The JastAdd system—modular extensible compiler construction,” *Sci. Comput. Program.*, vol. 69, no. 1, pp. 14–26, 2007.

[2] J. Cervelle, R. Forax, and G. Roussel, “Tatoo: an innovative parser generator,” in *Proceedings of the 4th international symposium on Principles and practice of programming in Java*, 2006, pp. 13–20.

[3] G. Klein, “Jflex user’s manual,” *Available -Line Www Jflex Accessed August*, 2010.

[4] A. M. Sloane, “Lightweight language processing in Kiama,” in *Generative and Transformational Techniques in Software Engineering III*, Springer, 2011, pp. 408–425.

[5] Y. Gurevich, “Evolving algebras 1993: Lipari guide,” *Specif. Valid. Methods*, pp. 9–36, 1995.

[6] M. F. Group and others, “Introducing AsmL: A tutorial for the abstract state machine language,” Technical report, Microsoft FSE Group, 2001.

[7] M. Anlauff, “XASM-an extensible, component-based abstract state machines language,” in *Abstract State Machines-Theory and Applications*, 2000, pp. 69–90.

[8] J. Schmid, *Introduction to AsmGofer*. Citeseer, 2001.

[9] R. Farahbod, “CoreASM: An Extensible Modeling Framework & Tool Environment for High-level Design and Analysis of Distributed Systems,” Citeseer, 2009.

[10] A. Gargantini, E. Riccobene, and P. Scandurra, “A Metamodel-based Language and a Simulation Engine for Abstract State Machines.,” *J UCS*, vol. 14, no. 12, pp. 1949–1983, 2008.

[11] M. Ouimet and K. Lundqvist, “The TASM toolset: specification, simulation, and formal verification of real-time systems,” in *Computer Aided Verification*, 2007, pp. 126–130.

[12] S. Premaratne, A. M. Sloane, L. G. Hamey, and others, “An Evaluation of a pure embedded domain-specific language for strategic term rewriting,” in *Formal and Practical Aspects of Domain-Specific Languages: Recent Developments*, IGI Global, 2013, pp. 81–108.

[13] A. Ekroth and F. Mulder, “Compiler in Scala and Kiama in comparison with JastAdd,” 2015.

[14] N. Wirth, N. Wirth, N. Wirth, and N. Wirth, *Compiler construction*, vol. 1. Addison-Wesley Reading, 1996.

[15] R. Stärk, J. Schmid, and E. Börger, *Java and the Java Virtual Machine*, vol. 24. Springer-Verlag, 2001.

[16] A. M. Turing, “On computable numbers, with an application to the Entscheidungsproblem. A correction,” *Proc. Lond. Math. Soc.*, vol. 2, no. 1, pp. 544–546, 1938.

[17] E. Börger and D. Rosenzweig, “A mathematical definition of full Prolog,” *Sci. Comput. Program.*, vol. 24, no. 3, pp. 249–286, 1995.

[18] E. Boerger, U. Glässer, and W. Muller, “A formal definition of an abstract VHDL’93 simulator by EA-Machines,” in *Formal Semantics for VHDL*, Springer, 1995, pp. 107–139.

[19] R. e Karges, “Abstract state machine semantics of SDL,” *J. Univers. Comput. Sci.*, vol. 3, no. 12, pp. 1382–1414, 1997.

[20] M. Srivas and M. Bickford, “Formal verification of a pipelined microprocessor,” *Softw. IEEE*, vol. 7, no. 5, pp. 52–64, 1990.
[21] A. Dold, “A formal representation of Abstract State Machines using PVS,” *Verifix-Rep. Ulm*, vol. 6, 1998.
[22] M. Ouimet, G. Berteau, and K. Lundqvist, “Modeling an electronic throttle controller using the timed abstract state machine language and toolset,” in *Models in Software Engineering*, Springer, 2007, pp. 32–41.
[23] M. Ouimet and K. Lundqvist, “The Timed Abstract State Machine Language: Abstract State Machines for Real-Time System Engineering.,” *J UCS*, vol. 14, no. 12, pp. 2007–2033, 2008.
[24] B. Beckert and J. Posegga, “leanEA: A lean evolving algebra compiler,” in *Computer Science Logic*, 1996, pp. 64–85.
[25] G. Del Castillo, I. \DJur\d janović, and U. Glässer, “An evolving algebra abstract machine,” in *Computer Science Logic*, 1996, pp. 191–214.
[26] R. Lezuo and A. Krall, “Using the CASM language for simulator synthesis and model verification,” in *Proceedings of the 2013 Workshop on Rapid Simulation and Performance Evaluation: Methods and Tools*, 2013, p. 6.
[27] P. Arcaini, A. Gargantini, and E. Riccobene, “Runtime monitoring of Java programs by Abstract State Machines,” 2010.
[28] M. P. Jones, “The implementation of the Gofer functional programming system,” Citeseer, 1994.
[29] M. Odersky, L. Spoon, and B. Venners, “Programming in Scala,” *Éc. Polytech. Fédérale Lausanne*, 2005.
[30] A. Moors, F. Piessens, and M. Odersky, “Parser combinators in Scala,” 2008.
[31] B. Emir, M. Odersky, and J. Williams, “Matching objects with patterns,” in *ECOOP 2007–Object-Oriented Programming*, Springer, 2007, pp. 273–298.
[32] E. Visser, “Meta-programming with concrete object syntax,” in *Generative programming and component engineering*, 2002, pp. 299–315.
[33] G. Huet, “The zipper,” *J. Funct. Program.*, vol. 7, no. 05, pp. 549–554, 1997.

Appendix 1 execJavaStm$_I$

```scala
private def execJavaStmI: Unit=
{
 context(pos) match
 {
  case LabelStm(_, Normal(_))        => yieldResultUp(Norm())
  case LabelStm(lab, Break(brk))     => if(lab.name == brk.label) yieldResultUp(Norm())
                                        else yieldResultUp(AST.Break(brk.label))
  case LabelStm(lab, Continue(cont)) =>
   if(lab.name == cont.label)
   {
    val theOriginalNode:Node = body(pos)
    replaceInRestbody(fromBodyNode = theOriginalNode, deep=true)
    yieldResult(theOriginalNode)
   }
   else
    yieldResultUp(AST.Continue(cont.label))
  case LabelStm(_, stm)       => pos := stm
  case BreakStm(lab, semi)    => yieldResult(AST.Break(lab.name))
  case ContinueStm(lab, semi) => yieldResult(AST.Continue(lab.name))

  case _:Phrase if restbody.value(pos).isInstanceOf[Abr] =>
   if(pos.value != firstPos && propagatesAbr(restbody(up(pos))))
   {
    val abr:Node = restbody(pos)
    yieldResultUp(abr)
   }

  case Block(Nil, Nil)    => yieldResult(Norm())  // empty block {}
  case Block(_, Nil)      => yieldResultUp(Norm())// all stm in block has been processed
  case Block(_, stm :: _) => pos := stm


  case If(BooleanValue(_), Normal(_), _) => yieldResultUp(Norm())
  case If(BooleanValue(_), _, Normal(_)) => yieldResultUp(Norm())
  case If(BooleanValue(v), exp1, exp2)   => if(true == v.value) pos := exp1
                                            else pos := exp2
  case If(exp0, _, _)  => pos := exp0

  case While(BooleanValue(_), Normal(_)) => val theOriginalNode = body(up(pos))
                                            replaceInRestbody(theOriginalNode, deep=true)
                                            yieldResultUp(theOriginalNode)
  case While(BooleanValue(b), stm)       => if(true == b.value) pos := stm
                                            else yieldResultUp(Norm())
  case While(exprTest, stm)              => pos := exprTest

  case VarDecl(vtype, local)               => locals(local) := JLS.valueFromType(vtype)
                                              yieldResult(Norm())
  case VarDeclAsgn(vtype, local, Value(v))=> locals(local) := v
                                               yieldResultUp(Norm())
  case VarDeclAsgn(_, _, expr)             => pos := expr

  case Semi()                  => yieldResult(Norm())
  case ExprStm(Value(_), semi) => yieldResultUp(Norm())
  case ExprStm(expr, semi)     => pos := expr
  case Print(Value(v))         => println(v.value)
                                 yieldResultUp(Norm())
  case Print(expr)             => pos := expr
 }
}
```

Appendix 2 execJavaExp$_C$ and execJavaStm$_C$

```
private def execJavaExpC: Unit =
{
 context(pos) match
 {
  case StaticFieldRef(c, f) => if(initialized(c)) yieldResult(globals((c,f)))
                               else initialize(c)
  case FieldAsgn(StaticFieldRef(c, f), Value(v)) => if(initialized(c))
                                                    {
                                                     globals((c,f)) := v
                                                     yieldResultUp(v)
                                                    }
                                                    else
                                                      initialize(c)
  case FieldAsgn(StaticFieldRef(_, _), expr) => pos := expr
  case StaticInvk(c, m, Values(params))      => if(initialized(c))
                                                  invokeMethod(up(pos), (c,m), params)
                                                else
                                                  initialize(c)
  case StaticInvk(_, _, params) => pos := params
  case Exprs(Nil, Nil)          => yieldResult(Vals(Nil))// empty param list
  case Exprs(valList, Nil)      => yieldResultUp(makeValuesNode(valList)) //all done
  case Exprs(_, expr :: _)      => pos := expr // next to eval is expr
 }
}
```

```
private def execJavaStmC: Unit =
{
 context(pos) match
 {
  case Static(Return(returnAbr)) => yieldResultUp(returnAbr)
  case Static(block) =>  val c:Class = classOf(meth)
                         if(c == JLS.Object || initialized(superClass(c))) pos := block
                         else initialize(superClass(c))
  case ReturnStm(Void(v), semi)      => yieldResult(AST.Return(v))
  case ReturnStm(Value(v), semi)     => yieldResultUp(AST.Return(v))
  case ReturnStm(expr, semi)         =>  pos := expr
  case LabelStm(_, Return(Void(v)))  => yieldResultUp(AST.Return(v))
  case LabelStm(_, Return(Value(v))) =>  yieldResultUp(AST.Return(v))
  case AST.Return(Void(_))  => if(pos =:= firstPos && false == frames.isEmpty)
                                exitMethod(Norm())
  case AST.Return(Value(v)) =>  if(pos =:= firstPos && false == frames.isEmpty)
                                 exitMethod(v)
  case ExprStm(Normal(_), semi) =>  yieldResultUp(Norm())
 }
}
```

Appendix 3 execJava$_O$

```
private def execJavaExpO:Unit=
{
 context(pos) match
 {
  case This() => yieldResult(locals("this"))
  case NewClassInvk(c, param) =>
   if(initialized(c))
   {
    val ref = Ref(c)
    heap(ref) := Object(c, env.makeInstanceFieldsMap(c))
    yieldResult(ref)
   }
   else
    initialize(c)
  case InstanceFieldRef(RefValue(ref), field) => yieldResultUp(getField(ref, field))
  case InstanceFieldRef(exp, field) => pos := exp
  case FieldAsgn(InstanceFieldRef(RefValue(ref), field), Value(v)) =>
   setField(ref, field, v)
   yieldResultUp(v)
  case FieldAsgn(InstanceFieldRef(RefValue(ref), field), exp2) => pos := exp2
  case FieldAsgn(InstanceFieldRef(exp1, field), exp2)          => pos := exp1
  case InstanceOf(RefValue(ref), c) => val bval = env.subClass(classOf(ref), c)
                                       yieldResultUp(BoolValue(bval))
  case InstanceOf(exp, c) => pos := exp
  case CastRef(c, RefValue(ref)) => if(env.subClass(classOf(ref), c)) yieldResultUp(ref)
                                    else sys.error("ref cannot be casted to class”)
  case CastRef(c, exp) => pos := exp
  case InstanceInvk(RefValue(ref), m, Values(params)) =>
   val msig:MethodSignature = makeSignature(m, params)
   val c:Class = callKind(up(pos)) match
   {
    case CallKind.Virtual => env.lookup(classOf(ref), msig)
    case CallKind.Super   => env.lookup(superClass(classOf(ref)), msig)
    case CallKind.Special => classOf(ref)
   }
   invokeMethod(up(pos), (c,m), Vals(ref :: params.values))
  case InstanceInvk(RefValue(_), _, params) => pos := params
  case InstanceInvk(exp, _, _)              => pos := exp
 }
}
```

Appendix 4 execVM$_C$ and switchVM$_C$

```
private def execVMc(inst:Instruction): Unit =
{
 execVMi(inst)
 inst match
 {
  case GetStatic(_, field) => if(initialized(field.parent[AST.Class]))
                              {
                               opd := opd ::: globals(field)
                               pc := pc + 1
                              }
                              else
                               switch := InitClass(field.parent[AST.Class])
  case PutStatic(_, field) => if(initialized(field.parent[AST.Class]))
                              {
                               val (opdP, ws) = split(opd, size(field))
                               globals(field) := ws
                               opd := opdP
                               pc := pc + 1
                              }
                              else
                               switch := InitClass(field.parent[AST.Class])
  case InvokeStatic(_, method) => if(initialized(method.parent[AST.Class]))
                                  {
                                   val (opdP, ws) = split(opd, argSize(method))
                                   opd := opdP
                                   switch := Call(method, ws)
                                  }
                                  else
                                   switch := InitClass(method.parent[AST.Class])
  case Return(t)               => val (opdP, ws) = split(opd, size(t))
                                  switch := Result(ws)
 }
}

private def switchVMc(): Unit=
{
 switch.value match
 {
  case Call(method, args) =>
             pushFrame(method, args)
             switch := Noswitch()
  case Result(res) =>
          if(implicitCall(meth))
           popFrame(0, List[Word]())
          else
           popFrame(1, res)
          switch := Noswitch()
  case InitClass(c) =>
           val cs:ClassState = classState(c)
          if(cs == ClassState.Linked)
          {
           classState(c) := ClassState.Initialized
           for(field <- environment.staticFields(c))
           {
            globals(field) := JVMS.valueToWords(JLS.valueFromType(field.fType))
           }
           if(c.name.identifier == JLS.Object.name.identifier ||
             initialized(environment.superClassOf(c)))
            switch := Noswitch()
           else
            switch := InitClass(environment.superClassOf(c))
          }
 }
}
```

Appendix 5 execVM$_O$

```
private def execVMo(inst:Instruction): Unit =
{
 execVMc(inst)
 inst match
 {
  case New(c) =>
   if(initialized(c))
   {
    val ref = AST.Ref(c)
    heap(ref) := Object(c, environment.makeInstanceFieldsMap(c))
    opd := opd :+ addressOf(ref)
    pc := pc + 1
   }
   else
    switch := InitClass(c)
  case GetField(_, cf) =>
   val (opdP, r :: _) = split(opd, 1)
   opd := opdP ::: getField(r, cf)
   pc := pc + 1
  case PutField(_, cf) =>
   val (opdP, r :: ws) = split(opd, 1 + size(cf))
   setField(r, cf, ws)
   pc := pc + 1
   opd := opdP
  case InvokeSpecial(_, cm) =>
   val (opdP, r :: ws) = split(opd, 1 + size(cm))
   opd := opdP
   switch := Call(cm, r :: ws)
  case InvokeVirtual(_, cm) =>
   val (opdP, r :: ws) = split(opd, 1 + size(cm))
   opd := opdP
   switch := Call(environment.lookup(classOf(r), cm), r :: ws)
  case InstanceOf(c) =>
   val (opdP, r :: _) = split(opd, 1)
   val sc = if(environment.subClass(classOf(r), c)) 1 else 0
   opd := opdP :+ sc
   pc := pc + 1
  case Checkcast(c) =>
   val r = top(opd)
   if(true == environment.subClass(classOf(r), c))
    pc := pc + 1
 }
}
```

```
private def E(node:Node): List[Instruction] =
{
 node match
 {
  case Lit(lit)                      =>  Prim(lit)
  case loc:Local                     => Load(T(loc), Bar(loc))
  case Asgn(loc, exp)                => E(expr) ::: Dupx(0, size(T(exp))) ::: Store(T(exp), Bar(loc))
  case UnaryOp(Op.NOT, exp)          => val una1 = LabelDef("una1") val una2 = LabelDef("una2")
                                       B1(exp, una1) ::: Prim(1) ::: Goto(una2) ::: una1 :::
                                       Prim(0) ::: una2
  case uop@UnaryOp(_, exp)           => E(exp) ::: Prim(uop)
  case bop@BinaryOp(_, exp1, exp2)   => E(exp1) ::: E(exp2) ::: Prim(bop)
  case InlineCond(exp0, exp1, exp2)  =>  val if1 = LabelDef("if1") val if2 = LabelDef("if2")
                                        B1(exp0, if1) ::: E(exp2) ::: Goto(if2) ::: if1 :::
                                        E(exp1) ::: if2
  case StaticFieldRef(c, f)                    => val cf:AST.Field = (c, f)
                                                   GetStatic(T(cf), cf)
  case FieldAsgn(StaticFieldRef(c, f), expr) => val cf:AST.Field = (c, f)
                                                 E(expr) ::: Dupx(0, size(T(expr))) :::
                                                 PutStatic(T(cf), cf)
  case StaticInvk(c, m, exprs)                 => val cm:AST.Method = (c, m)
                                                 E(exprs) ::: InvokeStatic(T(cm), cm)
  case Exprs(exprs)                            => exprs.map((exp:Node) => E(exp)).flatten.toList
  case This()                                  => Load(MoveType(AddrType()), "0")
  case NewClassInvk(c, params)                 => New(c) ::: Dupx(0, 1)
  case InstanceFieldRef(exp, f)                =>val cf:AST.Field = (classNameOf(exp), f)
                                                E(exp) ::: GetField(T(cf), cf)
  case FieldAsgn(InstanceFieldRef(exp1, f), exp2) =>val cf:AST.Field = (classNameOf(exp1), f)
                                                 E(exp1) ::: E(exp2) :::
                                                 Dupx(1, size(T(cf))) ::: PutField(T(cf), cf)
  case InstanceInvk(exp, m, params) =>
     val cm:AST.Method = (classNameOf(exp), m)
     E(exp) ::: E(params) ::: (callKind(cm) match
      {
             case CallKind.Virtual => InvokeVirtual(T(cm), cm)
             case CallKind.Super   => InvokeSpecial(T(cm), cm)
             case CallKind.Special => InvokeSpecial(T(cm), cm)
       })
  case AST.InstanceOf(exp, c) =>  E(exp) ::: JavaVirtualMachine.InstanceOf(c)
  case CastRef(c, exp) => E(exp) ::: Checkcast(c)
 }
}
private def B1(node:Node, lab:LabelDef): List[Instruction] =
{
 node match
 {
  case BoolValue(true) => Goto(lab)
  case BoolValue(false) => List()
  case UnaryOp(Operator.NOT, expr) => B0(expr, lab)
  case InlineCond(expr0, expr1, expr2) =>
   val if1 = LabelDef("if1")
   val if2 = LabelDef("if2")
   B1(expr0, if1) ::: B1(expr2, lab) ::: Goto(if2) ::: if1 ::: B1(expr1, lab) ::: if2
  case expr: Phrase => E(expr) ::: Cond(Ifne(), lab)
 }
}
private def B0(node:Node, lab:LabelDef): List[Instruction] =
{
 node match
 {
  case BoolValue(true) => Nil
  case BoolValue(false) => Goto(lab)
  case UnaryOp(Operator.NOT, expr) => B1(expr, lab)
  case InlineCond(expr0, expr1, expr2) =>
   val if1 = LabelDef("if1")
   val if2 = LabelDef("if2")
   B1(expr0, if1) ::: B0(expr2, lab) ::: Goto(if2) ::: if1 ::: B0(expr1, lab) ::: if2
  case expr: Phrase => E(expr) ::: Cond(Ifeq(), lab)
 }
}
```

```
private def S(node:Node):List[Instruction]=
{
 node match
 {
  case Semi() =>
   Nil
  case VarDeclAsgn(typeId, loc, expr) =>
   typeOfNode.set(loc, typeId)
   E(expr) ::: Dupx(0, size(T(expr))) ::: Store(T(expr), Bar(loc))
  case VarDecl(typeId, loc) =>
   typeOfNode.set(loc, typeId)
   E(JLS.defaultLiteralFromType(typeId)) ::: Dupx(0, size(T(typeId))) ::: Store(T(typeId), Bar(loc))
  case ExprStm(expr, semi) =>
   E(expr) ::: Pop(size(T(expr)))
  case prt@Print(expr) =>
   E(expr) ::: Prim(prt)
  case Block(blockStmSeq) =>
   blockStmSeq.map((stm:Node) => S(stm)).flatten.toList
  case If(exp, stm1, stm2) =>
   val if1 = LabelDef("if1")
   val if2 = LabelDef("if2")
   B1(exp, if1) ::: S(stm2) ::: Goto(if2) ::: if1 ::: S(stm1) ::: if2
  case While(exp, stm) =>
   val while1 = LabelDef("while1")
   val while2 = LabelDef("while2")
   Goto(while1) ::: while2 ::: S(stm) ::: while1 ::: B1(exp, while2)
  case LabelStm(lab, stm) =>
   val labC = LabelDef(lab.name + "C")
   ContinueJavaLabel2LabelDefMap(lab) = labC
   val labB = LabelDef(lab.name + "B")
   BreakJavaLabel2LabelDefMap(lab) = labB
   labC ::: S(stm) ::: labB
  case ContinueStm(lab, semi) =>
   val labC = ContinueJavaLabel2LabelDefMap(lab)
   Goto(labC)
  case BreakStm(lab, semi) =>
   val labB = BreakJavaLabel2LabelDefMap(lab)
   Goto(labB)
  /// Java C
  case Static(stm) =>
   S(stm)
  case ReturnStm(_:VoidValue, semi) =>
   JavaVirtualMachine.Return(MoveType(VoidType()))
  case ReturnStm(exp, semi) =>
   E(exp) ::: JavaVirtualMachine.Return(T(exp))
  case _ =>
   Nil
 }
}
```