PROBABILISTIC PROOF-CARRYING CODE

By

Michael Ian Sharkey

April 2012

A Thesis
submitted to the School of Graduate Studies and Research
in partial fulfillment of the requirements
for the degree of
Master of Computer Science[1]

© Michael Ian Sharkey, Ottawa, Canada, 2012

[1]The Master's Program is a joint program with Carleton University, administered by the Ottawa-Carleton Institute of Computer Science

Abstract

Proof-carrying code is an application of software verification techniques to the problem of ensuring the safety of mobile code. However, previous proof-carrying code systems have assumed that mobile code will faithfully execute the instructions of the program. Realistic implementations of computing systems are susceptible to probabilistic behaviours that can alter the execution of a program in ways that can result in corruption or security breaches. We investigate the use of a probabilistic bytecode language to model deterministic programs that are executed on probabilistic computing systems. To model probabilistic safety properties, a probabilistic logic is adapted to out bytecode instruction language, and soundness is proven. A sketch of a completeness proof of the logic is also shown.

Contents

Abstract **ii**

1 Introduction **1**

1.1 Overview of Thesis 2

1.2 New Results in this Thesis 3

I Background **5**

2 Mobile Code and Security **6**

2.1 Mobile Code 6

2.2 Aspects of Safe Code 7

2.2.1 Confidentiality 8

2.2.2 Integrity 8

2.2.3 Availability 9

2.3 Methods of ensuring safe mobile code 9

2.3.1 Digital Signing 9

2.3.2 Sandboxing 10

2.3.3 Proof Carrying Code 11

3 Semantics **15**

3.1 Denotational semantics 15

3.2 Operational semantics 16

3.3 Axiomatic semantics 16

3.3.1 Hoare logic 16

4 Probabilistic Programs **19**
4.1 Probabilistic semantics . . . 19
4.2 Probabilistic computing platforms . . . 20
4.2.1 Quantum computers . . . 20
4.2.2 Classical machines . . . 21
4.2.3 Distributed computers . . . 23
4.3 Probabilistic Logic . . . 24
4.3.1 Language . . . 26
4.3.2 Semantics . . . 28
4.3.3 Decidability . . . 29
4.4 Alternative Approaches . . . 30

5 Lua **31**
5.1 Introduction to Lua . . . 31
5.2 Lua VM . . . 32

II Probabilistic Proof-Carrying Code **35**

6 Simplified Probabilistic Lua **36**
6.1 Simplified Lua . . . 36
6.2 Simplified Probabilistic Lua Instruction Set . . . 36
6.3 Pseudo-Instructions . . . 38
6.4 Well-Formed Programs . . . 40
6.4.1 Forward-Branching . . . 40
6.4.2 Constrained Branching . . . 40
6.4.3 Return Instruction . . . 41
6.5 Motivating Examples . . . 41
6.5.1 Parity Check . . . 41
6.5.2 Error correction via Hamming Code . . . 42

7 Semantics of SPL **44**
7.1 “Many-Worlds” Operational Semantics . . . 44

7.1.1 Valuations 45
7.1.2 Operational Small-Step Semantics 46
7.2 Operational Semantics for the “Copenhagen” Virtual Machine 50
7.3 Modelling Errors 52
7.4 Termination 54
7.4.1 Probabilistic Termination 55
7.5 Proof of Equivalence of Many-Worlds and Copenhagen 56

8 Proof Rules for SPL Instructions **62**
8.1 Preliminaries 62
8.1.1 Body rule 63
8.1.2 Inference Rules 63
8.2 Instruction Proof Rules 64
8.2.1 Rules for Instructions that Update the Program Counter . . . 64
8.2.2 Rules for Instructions that Load from Memory Stores 65
8.2.3 Rules for Arithmetic instructions 65
8.2.4 Rules for Probabilistic Instruction 66
8.3 Conditional Branching 66
8.4 Special Shape of the Rules 68
8.4.1 Non-conditional branch instructions 68
8.4.2 Conditional Branch instructions 69
8.4.3 Distributive property 70
8.5 Computational Complexity 72

9 Soundness **73**
9.1 Soundness 73
9.1.1 Non-branching programs 74
9.1.2 Proof of conditional branching programs 79
9.2 Proof of Soundness of Weakest Preconditions 81
9.2.1 Discussion of Completeness 83

10 Implementation **85**
10.1 Compiler . . . 86
10.2 Virtual Machine . . . 88
10.3 Verification Condition Generator . . . 90
10.4 Proof Verifier . . . 92

11 Final Remarks **93**
11.1 Conclusions . . . 93
11.2 Summary of Contributions . . . 93
11.3 Future Work . . . 94

A Sample Programs **102**
A.1 Program 1: Parity Check . . . 102

B Coq code **106**

C Sample proof of probabilistic correctness **110**

1 Introduction

In today's world, we rely upon mobile code every day. For example, JavaScript is used to provide rich and interactive web applications by companies such as Google, Microsoft and Yahoo. This has led to an explosion in creative uses for code, and unparalleled access to data for web users.

Mobile code extends beyond JavaScript, however. Adobe's Flash, Sun/Oracle's Java Applets, Microsoft's ActiveX and Google's Native Client all use mobile code to provide developers with access to a user's machine for reasons of performance and efficiency.

For the purposes of this thesis, mobile code is defined by the existence of a trust boundary between the producer of a program, and the consumer of that program. This abstract view of mobile code is useful, as it might be that the physical computing system used to produce and consume the program are identical, but the logical view of the computing system divides into producer and consumer roles. For example, a program may be written and compiled on a mainframe computer using a low-privilege account. Running this compiled program with elevated privileges should be considered mobile code for the purposes of this thesis.

The advantages of mobile code are weighed down by one crucial disadvantage: the negative security implications of allowing arbitrary software to run on a user's computer. Software flaws in Java applets, ActiveX components and the Flash plugin can expose millions of machines to the possibility of malicious compromise [26]. This has created an industry devoted to arresting the spread of malicious code via a number of methods.

This thesis will look at one mechanism of ensuring the safety of programs loaded from untrusted sources, with a specific focus on a class of errors, hereon referred to as *probabilistic errors*. These errors could be the result of spurious environmental noise, or caused by malicious agents that have access to the physical medium that a

computation is being carried out upon. Our proofs ensure that classical programs that have been modified to exhibit probabilistic behaviours will meet a given probabilistic post-condition, such as limiting the probability of an erroneous result to a small value.

1.1 Overview of Thesis

In Chapter 2, we further 'discuss the definition of mobile code and how various real-world implementations meet the definition. We also discuss what it means for mobile code to be considered safe, and describe existing tools and technologies that have been developed to ensure mobile code's safety. Proof-carrying code is described in more detail, including its history and applications.

Chapter 3 discusses the notion of computer program meaning. Denotational, operational and axiomatic semantics are introduced, and examples of their usage are compared. Special focus is given to operational and axiomatic semantics, which are utilized in this thesis to give meanings to programs. Hoare logic, a type of axiomatic semantics, is described in further depth.

Chapter 4 is an introduction to probabilistic programs. After a review of the history of probabilistic semantics, an overview of probabilistic computing platforms is described. Chadha et al's [10] work on the Exogenous Probabilistic Propositional Logic is reviewed, and will be used as the assertion language in the proof-carrying code system. An overview of the Lua programming language is given in Chapter 5, which will form the basis of the probabilistic bytecode language used in this work.

In the main section of this thesis, we investigate the properties of a proof-carrying code system for a probabilistic bytecode language. The Simplified Lua and the Simplified Probabilistic Lua bytecode languages are described in Chapter 6. Simplified Lua is based upon the high-level Lua language, while the Simplified Probabilistic Lua bytecode is based upon the bytecode instruction set that the Lua language compiles into. Two different semantics for the bytecode language are given in Chapter 7; both semantics are given in the operational style, but have slightly different intentions. The "Many-World" semantics provides a deterministic view of probabilistic programs, which we use to prove our theoretical results in Chapter 9, while the "Copenhagen" semantics correspond more closely to computation on real-world computing systems

with probabilistic errors. We provide a proof that shows an equivalence between these two semantics. We also describe program transformation rules for modelling certain types of probabilistic computation errors.

An axiomatic semantics of the Simplified Probabilistic Lua bytecode language is presented in Chapter 8 as a set of proof rules. The soundness and completeness of these rules with respect to the operational semantics is given in Chapter 9.

Finally, Chapter 10 describes the technical details of the implementation of the Probabilistic Proof-Carrying Code system. Conclusions and future work are presented in Chapter 11.

1.2 New Results in this Thesis

Two semantics are given for a probabilistic bytecode language. The first semantics loosely corresponds to the Copenhagen interpretation of quantum mechanics, where a classical state is immediately and non-deterministically altered when a probabilistic event occurs. The second semantics loosely corresponds to the “Many-Worlds” interpretation of quantum mechanics, where a probabilistic state is deterministically transformed by the execution of a program.

These programs are proven to have a correspondence, such that the probabilistic state resulting from executing a program under “Many-Worlds” describes (in the limit) the distribution of results from executing under Copenhagen. This proof of correspondence ensures that proofs that assume one type of semantics will apply to the other.

This probabilistic bytecode language is based upon the bytecode language defined by the Lua programming language [35]. Although the results are specific to this probabilistic bytecode language, the results can be easily generalized to other languages.

This thesis defines a set of proof rules for a probabilistic byte-code language. The proof rules are synthesized from two results:

- a probabilistic while-language developed by Chadha [10] that includes a Hoare-style logic proven sound and complete

- a deterministic bytecode language developed by Bannwart and Mueller [4] that includes a Hoare-style logic proven sound and complete

The proof rules defined in this thesis are novel, and applicable to proving properties of real-world programs that experience probabilistic behaviour. These rules can be used to ensure that programs that exhibit probabilistic behaviours will satisfy assertions that describe probabilistic states. Such assertions could ensure the probability that an algorithm will return the correct result in the event of data corruption exceeding an acceptable threshold, or validate that probabilistic behaviour will not cause additional data corruption.

To model common probabilistic behaviours exhibited by (supposed) deterministic programs, a set of transformation rules is described. These transformations can be used to model real-world behaviours such as memory corruption or deliberate attacks by malicious actors. By performing source transformation, we can keep the probabilistic bytecode language simple, and specify the exact probabilistic behaviour we are interested in verifying. For instance, if a memory corruption event is very unlikely, but an instruction skip instruction is highly likely, the program can be transformed to investigate the latter occurrence, while ignoring the former.

Part I

Background

2 Mobile Code and Security

2.1 Mobile Code

As mentioned, mobile code is defined by the existence of a trust boundary between the producer of a program, and the consumer of that program.

However, the most common examples of mobile code involve both a trust boundary and a network of computers. The code producer and the code consumer will be physically separate computing devices, with a potentially untrusted communication medium joining them. We limit our discussion to mobile code that has a restricted notion of trust between the two parties; malicious mobile code (i.e., malware, worms, etc.) are not considered in this work.[1]

The World Wide Web has become reliant upon mobile code for high-functioning user interfaces. The ECMAScript programming language is supported by most modern web browsers. ECMAScript is a multi-paradigmatic programming language, with support for functional, object-oriented and imperative styles. Other technologies in use include Adobe's proprietary Flash platform, Oracle's (formerly Sun Microsystems's) Java applet, Microsoft's ActiveX and Silverlight platforms, and Google's NativeClient. Each technology provides a code consumer platform, with specialized functionality for high performance graphics, access to persistent storage or creating communication channels with the code producer.

A relatively new example of a code producer/consumer relationship is *cloud computing* services. Companies such as Amazon and Google have invested in a large number of data centres [34, 20], which house thousands of commodity PC's. These computers can be contracted to perform computationally intensive work, distribute workloads geographically for fault tolerance, or store quantities of data that exceed normal hard-drive storage limits. In the cloud computing scenario, the code producer

[1] It would be nice if malware authors included a proof of correctness with their malicious programs, but it's an unlikely outcome.

is an agent that has contracted the use of the cloud computing service, and Google/Amazon take on the role of the code consumer. Technologies such as Java and Python are commonly used as the programming language of choice [20], and frameworks like MapReduce [12] simplify the development of robust distributed algorithms.

As mentioned above, trust boundaries can create a producer/consumer separation. Perhaps the best example of this is found in network filtering. For efficiency, network filtering must be performed as soon as a network packet is available for inspection; in modern computer architectures, this occurs in a trusted environment known as kernel mode or ring 0 [49]. This trusted environment controls processes, communicates with attached hardware and mediates user interactions. Code running in this environment has access to all data accessible by all users of the computing system. Network filters are small predicates that describe which network packets are allowed to be accepted for further processing, and which packets should be discarded. Loading a network filter into the trusted kernel environment involves two actors: the kernel itself is the code consumer, and the filter author is the code producer.

Finally, computing environments that involve a heavy price to pay in the event of a failure can be considered to have a producer/consumer relationship. For example, a spacecraft's guidance or scientific instrumentation control systems must operate correctly, or else vast sums of money could go to waste. Issues with the Mars Polar Lander and the Cassini space probe can all be traced to software problems; although, in the latter case, the problem was not of human origin. Such computing systems are not limited to far-flung spaceships; computers linked to global financial markets or highly classified infrastructures must have a similar degree of confidence in the correctness of code before execution.

2.2 Aspects of Safe Code

Safety of running programs can broadly be considered the enforcement of three data security concepts, known as the CIA triad in the software security field: Confidentiality, Integrity and Availability. Different types of logics can describe the necessary properties of safe code under each of the concepts.

2.2.1 Confidentiality

Confidentiality is the protection of data in the event of a breech of security. Private data must be protected from malicious activity, or confidentiality is broken. Methods of ensuring confidentiality include encryption, remote storage or trust boundaries. Operating systems enforce confidentiality by separating the memory a process may access from all other processes running on the system, unless special overrides are allowed.

In the context of mobile code, confidentiality implies that the only data mobile code can access is that which is specifically granted ability to read. For instance, a chunk of code could be passed a parameter containing all the information that is allowed to be processed. Any accesses to data outside that small subset is disallowed.

Confidentiality guarantees can be met by using safety policies written in predicate logic. Proof rules describe how instructions interact with memory, and a deduction in predicate logic corresponds to a confidentiality guarantee that the mobile code does not access memory outside the bounds of the security policy.

2.2.2 Integrity

In the CIA triad, integrity refers to the restriction of data to legitimate users. This criteria can be met using a similar proof system as the one described above. Mobile programs must be limited to accessing memory that is strictly granted permission to that code.

However, integrity has another, more general, meaning. Code may be able to prove that accesses to memory are within the constraints given, but if the structure of the code is altered security guarantees could be lost. For instance, a program could alter a string of characters encoded in the standard C NULL-terminated encoding to remove the trailing terminator; further use of this structured data could led to security issues.

To mitigate this issue, proofs should enforce that any accesses to structured data should result in acceptable modifications to that data. This can be accomplished using predicate calculus as well.

2.2.3 Availability

Unless data is available, it is useless. Therefore, even if data is kept confidential and integrity is enforced, a malicious user can still achieve damage if the data cannot be accessed by legitimate users. Therefore, availability refers to the enforcement of access to legitimate users in the presence of malicious activity.

Mobile code should not be able to deny further access to data once run. Possible methods of compromising this ability include deleting data or exhausting all resources that legitimate users use to access their data, such as network sockets, process memory or CPU time.

Linear logic [19] is known as a logic of resources; rather than classical (or intuitionistic) predicate logic, where a premise may be used an arbitrary number of times in a proof, premises are "used up" in the course of a linear logic derivation. This can be used to model access to a limited resource: in this case, mobile code could only be allowed to open a minimum of network sockets, run for a small number of time slices or allocate a bounded amount of memory.

2.3 Methods of ensuring safe mobile code

2.3.1 Digital Signing

Digitally signed code relies upon public key cryptography and secure hash algorithms to ensure the integrity and authenticity of mobile code.

Upon receipt of a mobile code object, a code consumer will attempt to determine if a valid digital signature is attached. This digital signature consists of a cryptographic hash of the code object's encoded representation, which has been signed using the private key of the code's issuer.

This mechanism ensures that the code object has not been altered in transit; if an alteration had occurred, the cryptographic hash would no longer match, and the code object would be rejected. Additionally, the authenticity of the mobile code producer is established via the use of the public key to encrypt the hash. The only method of decrypting the hash to its plain-text representation is via the signing author's public key, which shows that the code could only have been produced by the signing agent.

Digitally signed code objects explicitly add trust into the process of executing

mobile code. Before mobile code is executed, a user must explicitly acknowledge that a trust relationship exists between the code producer and the code consumer (in this case, the user). If no trust relationship exists, the code is denied execution. Similarly, if the digital signature cannot be verified, due to an incorrect hash or a revoked key, the code is marked as untrusted and discarded.

Microsoft's Authenticode for ActiveX objects is an example of a digital signing protection mechanism; Java applets and browser extensions can also be signed for additional protection.

2.3.2 Sandboxing

Mobile code generally must interact with its environment to perform its intended function. For instance, a JavaScript code fragment may modify the contents of a static webpage after it has been downloaded and rendered, or communicate with a remote data server to retrieve up-to-date information [18].

In modern operating systems, all user-executed code operates in a restricted environment known as "user-mode". This is differentiated from "kernel-mode", where restrictions on executing code is lifted.

For instance, if a user chooses to execute a calculator program on a modern operating system, the program will run in user-mode, and only have access to a limited selection of the resources available in the computer system. Generally, this will include access to the file system, graphic memory, common libraries, and memory allocated to the calculator process. However, memory allocated to the processes that are executed by other users will be unreachable, and certain machine instructions (hardware interactions and memory mapping) will cause a hardware fault.

If the calculator program needs to interface with the entire user base of a computer system, or needs access to resources beyond those allowed for a user-mode program, a kernel-mode component can be loaded by a trusted administrator of the system. This component (sometimes referred to as a "driver") has much fewer explicit restrictions upon its behaviour, and is generally expected to abide by a set of conventions (as opposed to enforced rules). Driver components can access all physical memory, interact with attached hardware devices, or completely destroy the workings of a computer

system.

Although mobile code needs interaction to be useful, the set of behaviours available must be restricted. To do so, mobile code is run in a restricted environment, or "sandbox". Code running in a sandbox has access to a subset of the behaviours code run by a user would normally be able to access. This is done because the code is untrusted.

Sandboxes are implemented in a number of mobile code environments, including Java applets, Adobe Flash, and browser JavaScript interpreters. Each sandbox has a number of similar properties, including restricting access to the filesystem, arbitrary network connections or other confidential, private information.

Sandboxes also support configurable security policies, which can vary due to a broader security environment, or a deliberate requirement for access to resources. For example, a Java applet could be run in a sandbox that permits access to local files if the browser itself imposes further restrictions on the files available.

2.3.3 Proof Carrying Code

Proof-carrying code was developed by George Necula [40], and includes a logic that is an extension of the Hoare logic rules to an assembly program. Programs are compiled to a target assembly language, then a proof of the program's safety is determined and associated with the compiled output, which is then passed along with the mobile code to the code consumer.

An advantage of proof-carrying code is that the burden of responsibility for safety enforcement shifts from the code consumer unto the code producer. The code must be proven correct before a consumer will accept, load and execute the mobile code program. Proving properties of code can be somewhat difficult, but PCC relies on the ease of proof verification. The code consumer must only perform a simple validation step to ensure that the code and its proof are acceptable; upon acceptance, the code can be run without need for expensive sandboxing checks.

Every assembly instruction has a proof rule associated with it that is used to enforce a safety property. Safety properties could include a guarantee that typing assertions are respected, or that memory accesses are made to valid memory areas.

More complicated safety properties would require commensurate changes to the instruction proof rules.

In Necula's thesis [40], the proof rules are tailored to the instruction set targeted, which is a custom abstract assembly language known as the Safe Assembly Language, or SAL. In addition, a conversion from SAL to conventional computer instruction sets such as SPARC and Intel x86 is described.

Unlike code signing, the proof does not need to be accepted as valid based upon trust of the issuer. The proof is related to the mobile code program in a very specific way, and cannot be circumvented. If a proof (or its associated program) is corrupted in transit, one of two scenarios can occur. First, the proof is no longer a valid proof of safety, which results in the code consumer rejecting the program. Or, (more rarely) the proof is altered, yet still encodes a valid proof of safety, which results in the consumer accepting the code and executing the program. In this case, even though the program may be altered from its original state, the safety guarantees are still met, and the code will not cause security problems.

One disadvantage of proof-carrying code is the difficulty of generating the proof of safety. This burden is placed upon the code producer, a role not typically associated with mobile code development. *Certifying compilers* generate the bulk of a proof of safety from a program as the compiler executes, based upon hints and the type system of the source language; however, there may still be proof obligations that require manual intervention.

Another disadvantage is the difficulty in crafting the proof rules to ensure safety. The proof rules are specifically tailored to the security policy required; a new security policy, even one that is similar to the original, requires the generation of new proof rules, and attendant proofs of safety and soundness.

An attempt to solve the issue of developing new proof rules by using temporal logic was investigated by Bernard and Lee [8]. Temporal logic allows for predicates to refer to propositions that can change their truth value as time progresses. Many security policies are intimately linked with time; certain files can only be accessed after a successful login, while accessing the Internet after reading a sensitive file may be disallowed. This approach was applied to ActionScript Bytecode by DeVries [15],

using co-logic programming to simplify the verification of the proof rules for infinite streams.

Other work has been done on providing extensions to proof-carrying code to make the procedure easier or less reliant upon the explicit proof rules. Appel and Felty [3] investigated using higher-order logic to remove the requirement for individual proof rules for each instruction, by using the expressiveness of the meta-logic (Calculus of Inductive Constructions [9]) to define the basic types of the programming language and the operational semantics of the machine instructions.

Trusted components in Proof-Carrying Code

Proof-carrying code relies upon the safety of two fundamental components for correct and safe execution[2] .

1. Verification Condition Generator
2. Proof verification

The verification condition generator (or VCGen) is used by both the code consumer and the code producer to generate a set of proof obligations that must be proved before loading and execution may occur. The VCGen component uses the proof rules associated with the proof-carrying code system to compute the weakest-preconditions of all instructions in a program, and subsequently generates the set of proof skeletons required to be proven.

The proof verification system accepts a proof of safety and verifies that the proof provided by the code consumer is in fact a valid proof of safety.

Untrusted components in Proof-Carrying Code

Two additional components are involved in proof-carrying code. These components do not need to be trusted, which means they can be developed with less scrutiny and verification than the components described above.

The two components are:

[2]The loading system is a trusted component; however, in the context of proof-carrying code it must be assumed to be correct, as the code consumer must assume that the processing unit executing code will process instructions faithfully. This thesis partially explores the idea of unreliable instruction processing, but not to the extent where the CPU itself is untrustworthy.

1. Certifying compiler
2. Proving system

The certifying compiler is a component that helps generate proof hints along with the compiled output of a program. Most programming languages have some additional structure which is lost upon compilation. In the C programming language, type information of variables and functions is lost in the translation to machine code, but these typing annotations can be helpful in proving properties of the resultant compiled code. A certifying compiler can use these annotations to simplify the proof process, as additional information can be used to assist with the proof. However, the resulting proof must still be considered valid by the (trusted) proof verification component; if the certifying compiler's hints are invalid, then the proof will likely be too difficult to complete, and the code will not be accepted. Certifying compilers can also output invariant hints, such as partial correctness loop invariants.

The proving system used to generate the safety proof is an untrusted component. If a proof system generates an incorrect proof (accidentally or maliciously), the proof verifier will reject the invalid proof, and thus there is no guarantee that the code consumer can run the code safely.

3 Semantics

To reason about programs and behaviours, we first must have some notion of a program's meaning. *Programming language semantics* allow us to describe the meanings of programs using mathematical terminology.

There are several types of programming language semantics, broadly classed as operational, denotational, or axiomatic.

3.1 Denotational semantics

Denotational semantics was developed by Scott [50], and is used to transform a program into a mathematical function that takes a program state as input, and outputs (the meaning of) a transformed state. The intent of denotational semantics is to give a compositional description of the programming language semantics. Each command in the programming language is given an individual transformation, which may refer to the subcomponents inductively. For instance, the programming language tokens `1,2,3` and the operator `+` can be given the expected interpretation of the numbers $1, 2, 3$ and the mathematical operator $+$. Then, an expression such as `1 + 2` can be evaluated by evaluating the individual components, resulting in $1 + 2 = 3$. To fully formalize the semantics, the notion of a *domain* is required. A domain is a set with mathematical structure of elements that meet criteria for modelling a programming language. The full treatment of domain theory and its applications to denotational semantics is outside the scope of this work.

Denotational semantics can be difficult to work with, and can be difficult to properly formulate, and the mathematical structures can be very sophisticated. For example, semantics that include recursive definitions, such as `while`-loops, require least fixed points; semantics for programming languages with non-traditional features, such as concurrency or non-determinism require yet higher levels of abstraction [50].

3.2 Operational semantics

Structural operational semantics were introduced by Plotkin [46] as a way of describing the semantics of programming languages using a natural and simple approach. Operational semantics are useful for language implementers, as the rules can be used to develop a simple interpreter. Because of this, operational semantics are often used for describing the semantics of assembly language or bytecode languages.

Operational semantics are a formalization of "plain-English" descriptions of computer behaviour. Language semantics are described using a transition system between program states. A transition rule describes how an input state is transformed into an output state. The semantics of a program is given over a trace of the commands or instructions in the program's execution.

3.3 Axiomatic semantics

Denotational or operational semantics operate over program states, describing how an input state transforms to an output state. By contrast, *axiomatic semantics* describe how statements transform logical assertions as execution progresses. The meaning of a program is defined by the logical assertions that are true when executing the program. Hoare logic [24] is the best-known example of an axiomatic semantics.

3.3.1 Hoare logic

Developed by C. A. R. Hoare, Hoare logic is a method for proving the correctness of algorithms and programs. Axioms and inference rules are defined for an imperative programming language, with the intent to derive weakest preconditions for program fragments. These proof rules are used to ensure that a program, meeting some logical pre-condition, will terminate in a state that meets some logical post-condition.

Table 1 describes a simple programming language consisting of assignment, sequencing and while-loops, and the Hoare logic rules associated with this language. The notation $[E/x]\psi$ means to substitute the expression E for x in the assertion ψ. These rules can be used to determine the weakest pre-condition a program's initial state must meet for the terminating state to meet a given post-condition.

The `while`-loop inference rule in Table 1 will result in a pre-condition that ensures

Table 1: Example Hoare logic proof rules

1. $\{[E/x]\psi\}\texttt{x := E}\{\psi\}$

2. $$\frac{\{\psi_1\}S_1\{\psi_2\} \quad \{\psi_2\}S_2\{\psi_3\}}{\{\psi_1\}S_1;S_2\{\psi_3\}}$$

3. $$\frac{\{\psi \land E\}S\{\psi\}}{\{\psi\}\text{while } E \text{ do } S\{\psi \land \neg E\}}$$

partial correctness. The rule states, in effect:

> If the assertions ψ and E hold before the loop body S, and the assertion ψ holds after the completion of S, then the following is true: If the assertion φ is true before the execution of the while-loop with condition E and loop-body S, then when the loop terminates, ψ and $\neg E$ will hold

Notice that the rule does not state the outcome if the loop never terminates! Under partial correctness, the program

```
while true do
  i := i + 1;
end while
```

would meet any post-condition.

To account for this, a strengthened `while`-loop rule is introduced; the details can be found in Hoare's paper [24].

Hoare logics can be shown to have useful properties; two essential properties are *soundness* and *completeness*. Soundness of a Hoare logic implies that any valid derivation made using the proof rules will imply that, if the program is executed in a state that meets the program's pre-condition, it will terminate in a state that meets the program's post-condition. Completeness is the converse property; if a program begins in a state s satisfying a particular precondition, and terminates in a state s' that satisfies some post-condition, then there is a valid derivation in the proof system that expresses that relationship. Cook proved soundness and completeness of a Hoare logic for a simple imperative programming language [11]; much work on

proving soundness and completeness for various Hoare-style logics has been done [4, 10, 32].

4 Probabilistic Programs

4.1 Probabilistic semantics

A *probabilistic program* is a program that has access to a source of random information (with some probability distribution) that is used in the execution of the program itself. This source of randomness could be exposed via an explicit library function, and may be a pseudo-random source such as the ANSI C library function `rand()`, or a truly random source found via querying a source of randomness like a radioactive isotope. Other formulations include the notion of a probabilistic choice operator $\oplus_p$, which non-deterministically chooses between two commands with a biased probability p.

The semantics of a probabilistic programming language with a random library function has been investigated by Kozen [31] and Chadha et al [10]. Kozen's semantics are given in a denotational style for a programming language with `while` loops, while Chadha et al's semantics are for a language with a restricted notion of iteration (via repeated loop bodies) and a `toss` operator. Chadha's work is in support of a model for quantum algorithms.

Much work has been done in the programming language theory literature with languages containing a probabilistic choice operator $\oplus_p$ (alternatively, or_p). Jones [29] provides a denotational semantics for a language with while-loops using a probabilistic powerdomain, and exploiting Stone duality to derive a Hoare-style logic for the programming language. Den Hartog and de Vink [13] provide a semantics for a language with probabilistic choice, but an incomplete axiomatization limits the results [10].

The semantics described above are for a programming language with deterministic probability. Under this style, the program transforms input probability distributions to output probability distributions in a deterministic way. Given the same input distribution, the same output distribution will result. McIver and Morgan [37] extend the work of Kozen to investigate the relationship between deterministic probabilistic

computation and demonic non-determinism, and further extend it to angelic non-determinism. Demonic non-determinism invokes the imagery of a malicious demon, which can control the outcome of a demonic choice operator. This demon can observe the entire state of the program up to the current choice, and can choose the outcome that would maximize disruption; angelic non-determinism would be identical, but with the opposite intent.

In this thesis, we will focus on deterministic probabilistic programs. However, demonic non-determinism provides a good model for the malicious hardware attacks described in section 4.2.2.

Tsukada investigates using probability in a proof-carrying code system [55]. Unlike the work presented in this thesis, the probabilistic aspect derives from the interaction between the code consumer and the code producer. The consumer can be convinced a proof is correct with high probability, which improves the class of proofs efficiently provable within the system to PSPACE.

4.2 Probabilistic computing platforms

There are several computing platforms that exhibit probabilistic behaviours, either as an expected part of their design and implementation, or as an unexpected and unfortunate result of environmental interactions.

4.2.1 Quantum computers

Quantum computers are inherently probabilistic computing machines. Originally described by Richard Feynman [17] and further elaborated by David Deutsch [14] a quantum computer utilizes the non-intuitive properties of quantum mechanical objects, such as photons or individual electrons. Quantum computers could be used to simulate the properties of hard-to-observe quantum mechanical systems on easier-to-observe simulators. Further, quantum computers can utilize the concept of *entanglement* to compute results in less time than a classical computer. Shor's factoring algorithm [51] can compute the factors of a composite number in time $O((logN)^3)$, while Grover's quantum database search can locate an entry in an unsorted database in $O(N^{1/2})$ time [22].

Central to the operation of a quantum computer [30] is the idea of a measurement. Quantum computers operate over qubits, a quantum mechanical object (or collection of objects) that can exist in a super-position of states. Quantum gates transform qubits in deterministic ways to perform computations. The Toffoli gate is a classical universal gate that can be implemented as a quantum gate [54, 7] the Hadamard gate is used to put qubits into superpositional states [7], and the controlled-Not gate performs a reversible logical-NOT operation [7]. All quantum mechanical operations over qubits must be reversible (as a consequence of the unitary transformations involved). However, there is one non-reversible operation, known as *measurement*.

Measurement of a qubit in a quantum computer will collapse the super-position state, and return a result in reference to a basis. What is more, the measured value will be probabilistic; the probability of a measurement is related to the length of the projection vector onto the basis axis. This implies that quantum algorithms must have a probabilistic component.

4.2.2 Classical machines

Modern-day classical computers are based around a von Neumann architecture, where the program and the data are shared in a common memory. A central processing unit operates over instructions in the shared memory, fetching and executing the program via the use of a program counter register. These machines are physical artifacts, consisting of a delicate combination of transistors, voltage transformers, timing clocks and other processing elements. Computing systems also interact with the broader environment, using attached hardware devices such as additional RAM chips, hard drive storage, or network interface cards. All of these physical artifacts are susceptible to noise and errors. Network interface cards (wired or wireless) may experience line noise due to interference, RAM chips may lose data due to temperature shifts, or hard drives using magnetic platters spinning on mechanical plates can become damaged or faulty.

Errors in Computation

Computers are inherently error-prone devices. Errors can occur from a number of sources, some obvious, some subtle. For instance, the input values to an algorithm

will influenced by the input from a computer user, and thus unpredictable. A new user to a computer system may not have proper training, and errors in data entry could introduce values to an algorithm that are considered erroneous.

However, certain errors can be caused by issues that are ignored during program design and analysis. Computer processing units and data storage devices are sensitive to rare events that can cause errors in a computations. CPU's are sensitive to spikes in voltage applied to the processing circuits; such spikes can result in skipped instructions.

Faulty or mis-documented hardware may expose values that would be normally outside the range of values expected from that hardware component. Such an event could occur in an esoteric sensor on a spacecraft such as the Mars Polar Lander [42], or in commodity hardware devices such as hard disks and network interface devices. In a study by Google, Inc., the average failure rate of commodity hard drives was empirically studied, and found the AFR "varies from 1.7%, for drives that were in their first year of operation, to over 8.6%, observed in the 3-year old population" [45].

A similar study found that DRAM errors occurred with more frequency than expected [48]. These effects can be caused by a number of issues, such as excessive heat outside of tolerable levels, problems in the manufacturing process, or even stray cosmic rays. As a result, the stored memory of a computer's memory (either RAM or ROM) could be altered, including the modification of the data and/or instructions required by an algorithm to successfully compute a result. Errors due to cosmic rays are more likely in computer operations in harsh conditions like high radiation areas [52], but can also occur in everyday situations. Some memory units contain functionality to check memory for errors (ECC-memory), but this functionality is frequently disabled due to the performance impact upon programs [48].

In the above examples, the majority were due to harmful but benign sources, such as cosmic rays or voltages spikes. In some cases, the source of error-causing events can be caused by a malicious source, such as a hardware reverse-engineer or a saboteur. By applying excess heat to a memory unit, bit flipping errors can be made to occur at a higher frequency than normal [21]. Similarly, precisely controlling the voltage to a CPU can cause instructions in a delicate and trusted operation to be skipped [6].

Stack overflow exploits rely on applying a set of inputs to a program that are outside the expected range of values normal to the program. A program listening for a network connection will implicitly expect inputs to respect certain constraints, such as a 400-character limit on the length of user names or a valid range of numbers between 1 and 100. If the testing of a program doesn't supply values outside the normal range of values, errors in handling unusual inputs can be considered a probabilistic error.

4.2.3 Distributed computers

Distributed computing [33] is the process of creating and analyzing algorithms that are executed in a distributed setting. Distributed algorithms are defined in terms of computing nodes or entities, each of which is analogous to a traditional, independent computer. Nodes have an internal state, a set of instructions, and the ability to send and receive messages and respond to events. The nodes are arranged in a network; each node having a set of neighbouring nodes. Data is exchanged between computing nodes using messages, which can be sent as the result of an instruction and received as an event. A distributed algorithm consists of instructions that are run on each node that results in a specific outcome, such as the creation of a spanning tree or the calculation of a minimum value in the set of nodes.

Models of distributed computing often have a number of simplifying assumptions, such as bidirectional links, no failures of nodes, no message corruption, or synchronous message passing. Some of these simplifying assumptions are realistic when implementing solutions to real-world problems; some are not. In fact, a number of proofs have been given for the impossibility of deterministic solutions to leader election algorithms [2], given constraints that are often found in practical situations. For instance, unique identifiers are a common assumption made in developing leader election algorithms, but a practical implementation may not be able to guarantee uniqueness.

E. W. Dijkstra described the property of self-stabilization in distributed computing [16]. His example was of a token-ring network: a ring of processes that must meet a specification, namely that one and only one process can hold a token at a time. The difficulty is that the individual processes can only communicate with immediate neighbours, while the specification is dependent upon the state of all the processes

simultaneously. An individual process cannot determine if the specification holds.

Higham and Myers develop a self-stabilizing protocol for electing a leader in a synchronous ring of known size [23]. The algorithm begins with some set of nodes marked as producers, each of which flips a fair coin and records the result. The coin result is then sent to the neighbouring node, and a result is expected from the opposite neighbour. Nodes not marked as producers simply forward messages along the ring.

Itai and Rodeh developed a probabilistic algorithm that elects a leader in a unidirectional ring of known size with a termination probability of 1.0 [27]. The algorithm assumes nodes have access to a source of non-determinism, such as a random number generator. The algorithm is based upon the Chang-Roberts leader election algorithm for unidirectional rings [47]. Informally, the algorithm replaces the unique identifiers with a unique random stream of identifiers, with the downside of an unlikely (though possible) event where every node chooses the same random values at every iteration.

These algorithms rely upon a source of random values to perform their work. Probabilistic computation can perform tasks with acceptable likelihoods of success, even in the cases where deterministic results are impossible.

4.3 Probabilistic Logic

The logic of program assertions that we use in Probabilistic Proof-Carrying Code is the Exogenous Probabilistic Propositional Logic (or EPPL) of Chadha et al [10]. This logic is suited for describing and proving properties of probabilistic programs. A brief outline of the logic is given, for complete details see [10].

EPPL takes its name from the exogenous semantics approach to enriching a logic: models in the underlying logic are enriched with additional structure appropriate to the new logic. The EPPL semantic model is a discrete (sub)-probability space, giving the probability of possible valuations. Chadha's approach uses probability measures that are finitely additive, discrete and bounded, and these measures operate over values from a *real closed field*. The reasons for this choice are given in [10] and are required to ensure decidability; allowing full generality in the choice of measures would cause loss of decidability.

In EPPL, there are two memories, a finite collection of Boolean registers denoted by bm and a finite collection of registers containing a subset of the real values denoted xm, corresponding to the choice of IEEE-754 floating-point numbers [1] in language implementations, denoted $\mathcal{D}$. Chadha [10] also uses this constraint to prove completeness of the logic.

Informally, a valuation v is an assignment of valid values to the collection of registers. It is described in more detail in Section 4.3.2. The set of all valuations is denoted by $\mathcal{V}$. A measure over valuations μ is any map from $2^{\mathcal{V}}$ (the powerset of $\mathcal{V}$) to $\mathbb{R}^+$ (the set of non-negative real numbers), such that

- $\mu(\varnothing) = 0$
- $\mu(U_1 \cup U_2) = \mu(U_1) + \mu(U_2)$ if $U_1 \cap U_2 = \varnothing$

μ is a *probability measure* if in addition $\mu(\mathcal{V}) = 1$; that is, the measure of the complete space is 1. This is also known as a *probabilistic state*. In this work, we will only consider finite probability measures; that is, probability measures where the set of valuations is finite.

EPPL can be parameterized with a real closed field as the domain of measurement. A real closed field is an ordered field $\mathbb{K} = (K, +, \cdot, 1, 0, \leq)$, where

- every non-negative element of the set K has a square root in K, and
- every polynomial of odd degree with coefficients in K has at least one solution.

As in [10], the logic will be parameterized with $\mathcal{A}$, the real closed field consisting of real algebraic numbers. This set includes the rationals, square roots of positive elements and roots of polynomials of odd degree.

In this work, we assume that $\mathcal{D} \subseteq K$, and that the range of a measure μ will always be $\mathcal{A}$. In the case where the value $d \in \mathcal{D}$ is a natural number, we occasionally represent the value d in a hexadecimal format, denoted 0xh. For example, $d = 10$, $h = \text{A}$, and thus 10 is represented as 0x0A.

4.3.1 Language

EPPL is divided into two levels. The first level consists of *classical state formulas*, which reason about the values in the finite collection of boolean and real registers. To reason about the structures of the first level, *probabilistic state formulas* are defined. The grammar of EPPL statements is given by the grammar in Table 2, in BNF notation.

- *Real terms* operate over the set of real registers and real constants, with the mathematical operators + and $\cdot$.

- There are two register sets, the Boolean registers and the real registers. *Classical state formulas* operate over both sets of registers. In the case of the boolean registers, the logical connectives ($\wedge, \vee, \neg$, tt, $=$, $\Leftrightarrow$) are defined in terms of ff and $\Rightarrow$ (for example, $\neg p$ is defined as $p \Rightarrow$ ff). The $\leq$ relation over real numbers is used to lift the real terms into the Boolean domain. $p = p'$ is defined as $(p \leq p') \wedge (p' \leq p)$, $p > 0$ defined as $(0 \leq p) \wedge \neg(p = 0)$.

  These classical state formulas[1] are used to specify relations between program registers in a classical state; they correspond to the predicates in the standard Hoare logic [24].

- *Probabilistic terms* specify the relations between measures of classical formulas, and elements from $\mathcal{A}$, the set of real algebraic numbers. Probabilistic terms are the only method of describing program states in an assertion statement. Probabilistic terms can also reference logical variables, ranged over by the meta-variables y_i which contain values of the set $\mathcal{A}$. An assignment of logical variables to values is denoted ρ.

- *Probabilistic state formulas* operate over probabilistic terms, and are interpreted in terms of truth values. Similarly to classical state formulas, probabilistic state formulas include the boolean connective $\supset$ and the boolean constant fff. The remaining logical connectives over probabilistic state formulas ($\wedge, \vee, \neg$, ttt, $=$,

[1] We will use the names *classical state formulas* and *classical formulas* interchangeably in the rest of the work; similarly for *probabilistic state formulas* and *probabilistic formulas*.

$\iff$) are defined in terms of fff and $\supset$. Note that some of the logical connectives are overloaded between the classical state formulas and the probabilistic state formulas.

Table 2: Language for EPPL Formula

Real terms, $c \in \mathcal{D}$
$t \;:=\; xm \mid c \mid (t + t) \mid (t \cdot t)$
Classical state formulas
$\gamma \;:=\; bm \mid (t \leq t) \mid \mathrm{ff} \mid (\gamma \Rightarrow \gamma)$
Probability terms, $d \in \mathcal{A}$
$p \;:=\; y \mid d \mid (Pr(\gamma)) \mid (p + p) \mid (p \cdot p)$
Probabilistic state formulae
$\eta \;:=\; (p \leq p) \mid \mathrm{fff} \mid (\eta \supset \eta)$

Given a classical state formula γ, an expression e and a register r, we denote the substitution of e for r in the classical formula by γ^r_e. For example, $(r_1 = r_2)^{r_1}_3$ is the formula $(3 = r_2)$.

Further, the EPPL statement η^r_e is defined to be the result of replacing all instances of r in η by the expression e.

Definition 4.3.1. *Given a classical state formula γ and a probabilistic term p, Chadha [10] defines the γ-conditioned term p/γ to be the term obtained by replacing every occurrence of each measure term $Pr(\gamma')$ by $Pr(\gamma' \wedge \gamma)$. This can be similarly extended to probabilistic formulas; see Table 3 for the recursive definition of η/γ.*

Note that conditional formulas will appear in Chapter 7. For example, a formula that will be relevant to proof rules for conditional branch instructions is

$$\eta \supset (\eta'/\gamma')$$

The EPPL logic includes an abbreviated formula, denoted $\Box\gamma$, which stands for $\Pr(\gamma) = \Pr(\mathrm{tt})$. This operator can be considered a restricted form of the modal logic operator "necessity". Similarly, $\diamond\gamma$ stands for $\neg(\Box(\neg\gamma))$, and can be considered as a restricted "possibility" operator. However, these operators cannot be nested, and therefore are not equivalent to their modal logic counterparts.

Table 3: Conditional EPPL terms and formulas

Conditional terms		
d/γ	=	d, where $d \in \mathcal{A}$
y/γ	=	y, where y is a logical variable
$Pr(\gamma')/\gamma$	=	$Pr(\gamma \wedge \gamma')$
$(p + p')/\gamma$	=	$(p/\gamma + p'/\gamma)$
$(p \cdot p')/\gamma$	=	$(p/\gamma \cdot p'/\gamma)$
Conditional formulas		
fff$/\gamma$	=	fff
$(p \leq p')/\gamma$	=	$(p/\gamma \leq p'/\gamma)$
$(\eta \supset \eta')/\gamma$	=	$(\eta/\gamma \supset \eta'/\gamma)$

4.3.2 Semantics

A *classical valuation* is an assignment of values to the registers of a program state. Let Reg be the set containing all boolean registers bm and real number registers xm. Then, a valuation $v : Reg \to \mathcal{D}$ is a function from registers to elements of $\mathcal{D}$, where `pc` is a distinguished register whose value is restricted to the set $\mathbb{N}$.

Satisfaction of EPPL classical formulas is defined inductively in the straightforward manner, and denoted by $v \Vdash_{\mathbf{c}} \gamma$. Given $V \subseteq \mathcal{V}$, the *extent* of γ in V is defined to be $|\gamma|_V = \{v \in V : v \Vdash_{\mathbf{c}} \gamma\}$.

A sub-measure of μ defined by a classical formula γ is defined as a function μ_γ satisfying:

$$\mu_\gamma(V) = \mu(|\gamma|_V)$$

This sub-measure restricts the measure μ to those valuations that satisfy the classical formula γ. Valuations that satisfy the formula have the same measure as the original; valuations that don't satisfy the classical formula have measure 0.

The satisfaction of EPPL probabilistic formulas is now defined inductively in Table 4, where $[\![p]\!]_\rho\langle\mu\rangle$ denotes the evaluation of some EPPL term p in the probabilistic state μ with respect to a logical assignment ρ. Notice that the probabilistic formulas are only defined for the implication operator $\supset$ and fff. The remaining logical operations are defined in terms of these primitive operations.

With respect to an assignment ρ, the evaluation function $[\![_]\!]_\rho$ for probabilistic

terms has type $ProbTerm \to (2^{\mathcal{V}} \to \mathcal{A}) \to \mathcal{A}$. Evaluating a probabilistic term p with a probabilistic state μ and an assignment ρ, denoted $[\![p]\!]_\rho\langle\mu\rangle$ returns a value in the set $\mathcal{A}$.

The probabilistic formula satisfaction relation $[\![_]\!]_\rho$ overloads the notation defined above, but has type $ProbForm \to (2^{\mathcal{V}} \to \mathcal{A}) \to B$, where B is the type of Booleans. It will be clear from context which notation is used.

Table 4: Semantics of EPPL

Denotation of probability terms		
$[\![d]\!]_\rho\langle\mu\rangle$	=	d
$[\![y]\!]_\rho\langle\mu\rangle$	=	$\rho(y)$
$[\![Pr(\gamma)]\!]_\rho\langle\mu\rangle$	=	$\mu_\gamma(\mathcal{V})$
$[\![p_1 + p_2]\!]_\rho\langle\mu\rangle$	=	$[\![p_1]\!]_\rho\langle\mu\rangle + [\![p_2]\!]_\rho\langle\mu\rangle$
$[\![p_1 \cdot p_2]\!]_\rho\langle\mu\rangle$	=	$[\![p_1]\!]_\rho\langle\mu\rangle \cdot [\![p_2]\!]_\rho\langle\mu\rangle$

Satisfaction of probabilistic formulas		
$[\![(p_1 \leq p_2)]\!]_\rho\langle\mu\rangle = \text{true}$	iff	$[\![p_1]\!]_\rho\langle\mu\rangle \leq [\![p_2]\!]_\rho\langle\mu\rangle$
$[\![\text{fff}]\!]_\rho\langle\mu\rangle = \text{false}$		
$[\![\eta_1 \supset \eta_2]\!]_\rho\langle\mu\rangle = \text{true}$	iff	$[\![\eta_1]\!]_\rho\langle\mu\rangle$ implies $[\![\eta_2]\!]_\rho\langle\mu\rangle$

We say that an EPPL formula η is an *analytic formula* if it does not contain any measure terms (i.e., probabilistic terms of the form $\Pr(\gamma)$). A *valid analytic formula* is an analytic formula that holds for any logical assignment ρ.

4.3.3 Decidability

Chadha [10] provides a proof sketch of EPPL's decidability, as a straightforward consequence of EPPL being *weakly complete*[2]. He does this by showing that if an arbitrary EPPL formula η is consistent, then there is a model consisting of a probabilistic measure μ and an assignment ρ such that $[\![\mu]\!]_\rho\langle\eta\rangle$ is satisfied. He then restricts the values of registers to the real closed field of IEEE-754 floating-point numbers, which has a decidable first-order theory.

[2]Weakly complete means that $\vDash \eta$ implies $\vdash \eta$ for all η; i.e., the set of premises is finite.

4.4 Alternative Approaches

There are several approaches described in the literature that integrate probabilistic models with verification.

Tawfik and Neufeld [53] investigate the use of temporal Bayesian networks for reasoning about time-dependent probabilistic events. Ngo and Haddaway [41] present a probabilistic logic programming framework that represents conditional probabilities, and discuss the link between Bayesian networks and their logic. Similarly, Jaeger [28] describes recursive relational Bayesian networks, which can be used to represent probabilistic knowledge bases.

These works are not immediately applicable to our problem, as there focus is on using the programming logic as a semantics for building programs that include conditional probabilities, as opposed to verifying properties of probabilistic programs.

5 Lua

5.1 Introduction to Lua

Lua is a full featured dynamically typed programming language designed by a group at PUC-Rio, and has found a niche in embedded software systems and video games as a scripting language. The entire Lua system is written in portable ANSI C, and has been successfully ported to x86 on Windows and Linux, PowerPC and other hardware [35].

The Lua language focuses on simplicity, mainly as an aid to portability and to reduce the size of the resulting interpreter. The interpreter runtime minus the Lua language parser and compiler is less than 100 kilobytes, fitting into embedded systems easily.

By limiting the virtual machine instructions to the subset of C supported by ANSI-standard compilers, a number of useful optimizations are not supported (such as GCC's computed goto, relying upon unused bits in pointer variables or supporting lightweight processes). Rather, a number of features are implemented in ANSI C that achieve equal results. Arrays and hash maps (known as a table in Lua) are amalgamated, but the runtime will switch the handling of the underlying memory depending upon how the table is used (i.e., using a table as an array will result in indexing into a block of linearly allocated memory). To compensate for lack of lightweight processes (or threads), the Lua VM natively supports a limited form of continuations known as coroutines. These coroutines consist of their own independent frames of execution, and naturally lead to producer-consumer relationships between processes.

The Lua runtime has found use in a number of settings, including scripting the behaviour of networking utilities, providing the logic for photo editing software or controlling the user interface of online video games.

5.2 Lua VM

The Lua virtual machine instruction set resembles the instruction set of RISC processors, with minor differences:

Lua's **IAdd** instruction accepts 3 parameters and performs the expected addition operation, but with some additional complexity. The **IAdd** instruction supports a third parameter type of RK, which means the parameter can be loaded from either a register or the constant store, depending upon the value of the parameter. The **IAdd** instruction also supports operator overloading, but for this project such behaviour is avoided.

Memory load and store instructions are replaced by a class of instructions that operate over different data sections, described below.

The Lua branch instructions are identical to standard RISC branching instructions, with one important exception. The **ITest** instruction compares its two arguments and increments the program counter by 1 if true and by 2 if false. Unlike RISC instruction, the branch offset is not a parameter to the **ITest** instruction; instead, if the test succeeds, the next instruction is assumed to be an unconditional jump (**IJmp**) to the branch destination. Rather than waste a virtual machine cycle fetching the next instruction, the **ITest** statement loads the **IJmp** destination and branches immediately.

Unlike the traditional von Neumann stored-program architecture, the Lua memory model is more complex. Instructions and data are stored in separate domains, and arbitrary access to RAM is not permitted. Rather, a number of load instruction categories exist for accessing data stores.

The basic unit of a Lua binary is a closure, which is a function and any associated data. At runtime, a closure can access associated data via a number of sources, all of which are mediated by the Lua VM. The data sources (and the instructions used to interact with these sources) are as follows:

1. the register store, accessed via instructions that reference registers

2. the constant store, accessed via **ILoadK** and instructions that support RK (the source is determined by the value of an RK parameter; values over 250 are loaded from the constant store)

3. a Lua table, accessed via **IGetTable** and **ISetTable**. Tables are the primary data structure in Lua, and are equivalent to Python's dictionary or Perl's associative arrays.

4. the global store, via **IGetGlobal** and **ISetGlobal**. The global store is a special case of a table. Unlike other languages, the global table is function-specific and can be redefined during runtime.

5. the upvalue store, accessed via **GetUpval** and **ISetUpval**. Lexically scoped variables are stored as upvalues, and are generally used in function closures.

The Lua memory model is not identical to a von Neumann architecture, but the differences do not impact the verification condition generation, nor the major structure of the proof obligations.

The Lua instruction set also includes an iterative **IForPrep** instruction, which is used to loop over the elements of a list by incrementing an internal loop counter variable.

The Lua bytecode language includes a specific instruction to provide an iterative loop. Two instructions, **IForPrep** and **IForLoop** are used to implement an iterative loop construct in the Lua virtual machine.

The Lua code snippet

$$\texttt{for } i = 1, 50 \texttt{ do } \ldots \texttt{ end}$$

can be compiled to the bytecode

```
ILoadK  r0  1
ILoadK  r1  50
ILoadK  r2  1
IForPrep  r0     n
...
IForLoop  r0    -n
```

This bytecode will load the consecutive registers r_0 through r_2 with the parameters controlling the loop (respectively, the beginning loop index value, the terminating loop index value, and the index step value), then will iterate through the loop body (assumed to be of length n) while updating the loop index value in each iteration. The iterative looping construct will not be used in this work, and so the instructions **IForPrep** and **IForLoop** will not be discussed further.

Part II

Probabilistic Proof-Carrying Code

6 Simplified Probabilistic Lua

6.1 Simplified Lua

To explore probabilistic proof-carrying code in this thesis, a simplified programming language is developed and described. We define Simplified Lua to be a subset of the Lua v5.0 programming language, which is suitable for writing small programs that compute numeric values that do not require procedure invocation nor unbounded iteration. This subset of Lua is an untyped, imperative programming language, where variables can take either Boolean or IEEE-754 floating-point number values. The language is compiled to a bytecode, described below in more detail. The Simplified Lua language is not probabilistic; rather, the compiler output is instrumented with probabilistic instructions, depending upon the model of probabilistic computation under investigation. Simplified Lua can be easily compiled to the probabilistic bytecode language that is described in Section 6.2.

Table 5: Statements in Simplified Lua

s ::= s ; s	x = e	if e then *s* else *s* end	return	skip

Table 6: Expressions in Simplified Lua

e ::= x	e + e	e - e	e * e	e == e	e < e	e >> e	e ˆ e	

A Simplified Lua program consists of a sequence of program statements, $s_1, s_2, \ldots, s_n$; the set of valid statements is given by the BNF grammar in Table 5. The statement `skip` does nothing. Conditional choice is provided by the `if` statement. The expression syntax is given in Table 6.

6.2 Simplified Probabilistic Lua Instruction Set

To investigate the properties of Probabilistic PCC, we introduce an operational semantics for a simplified instruction set based on the Lua 5.0 virtual machine

[35, 25, 36], hereafter referred to as the Simplified Probabilistic Lua instruction set (SPL). This instruction set shares many features in common with the Safe Assembly Language described in [39]. The instruction set is extended with a probabilistic choice pseudo-instruction, similar to the probabilistic choice operator in [13]. This operator is not part of Chadha's EPPL; rather, it is defined inductively from the fundamental SPL instruction set. It is included here as a fundamental operation to better reflect certain probabilistic errors, which will be described in the next section.

Table 7: Instruction set for Simplified Probabilistic Lua

ILoadK r k	Load a constant k into register r
ILoadBool r c	Load a boolean value c into register r
IMove r_1 r_2	Move from register r_1 to register r_2
IGetGlobal r k	Load from global memory the value indexed by k into register r
I$BinaryArith$ r_{dst} r_1 r_2	Binary arithmetic operations
I$UnaryArith$ r_{dst} r_{src}	Unary arithmetic operations
I$BinaryArith$ r_{dst} r_1 $K(k)$	Binary arithmetic operations (with constant)
I$UnaryArith$ r_{dst} $K(k)$	Unary arithmetic operations (with constant)
ITest r_1 r_2	Conditional branch (forward only)
ILt r_1 r_2	Conditional branch (forward only)
ILe r_1 r_2	Conditional branch (forward only)
IJmp n	Unconditional branch (forward only)
IToss r q	Probabilistic toss, biased by probabilistic parameter q and stored in r
INop	Does nothing
IReturn r	Terminates a computation

The instruction set is given in Table 7. Missing from the instruction set are procedure invocations, unbounded iteration in the form of negative branch offsets, and operations over other fundamental data types such as character strings.

The arithmetic instructions are defined over the floating point number set $\mathcal{D}$ as defined in Section 4.3. The conditional and unconditional branching instructions can only have positive branch offsets. The probabilistic toss instruction is parameterized by a real number q, $0 \leq q \leq 1$. The set of registers bm (registers containing Boolean values) and xm (registers containing IEEE-754 floating-point values) are not

differentiated by these instructions.

Register arguments can be one of two forms. The first and simplest is a number indexed into the machine registers, starting from 1. The second is a pseudo-register, indexed into the program's constant pool, and denoted $K(n)$, where n is the index of the constant to be loaded into the register. The constant pool consists of values of floating-point or boolean type.

The unconditional branching operatior **IJmp** is restricted to only accept positive branch offsets. This limitation enforces bounded iteration. The conditional branch instructions (**ITest**, **ILt**, and **ILe**) have a semantics that ensures that negative offsets cannot be included in an SPL program. The conditional branching instructions can only increment the program counter by one (in the true case) or two (in the false case). Generally, the instruction immediately succeeding a conditional branch will be a **IJmp**, which will branch to the code handling the true conditional case.

The only instruction that differs from a classical, non-deterministic language is the **IToss** instruction. This instruction tosses a coin, and assigns a Boolean result to a machine register depending upon the outcome of the toss. The probabilistic parameter q declares the likelihood of the coin toss resulting in a heads (equated with a Boolean **true** assignment).

The machine state includes the program instructions, represented as a sequence of SPL instructions, the constant pool, the global variable store and the registers. The constant pool stores values that are fixed at program compilation time, and include values such as mathematical constants π, e, or hard-coded strings (such as user messages or protocol headers). Booleans are not stored in the constant pool; rather, the **ILoadBool** instruction encodes the value of the Boolean constant into the format of the instruction. The program counter is a simple integer index into the program instruction list.

6.3 Pseudo-Instructions

For conciseness, a pseudo-instruction $\oplus_q$ is defined, similar to the probabilistic choice operator found in [13]. This operator relates two instructions, one of which will be non-deterministically chosen during a program's execution. Given instructions I_1

and I_2, the pseudo-instruction $I_1 \oplus_q I_2$ will toss a probabilistic coin, weighted with bias q towards landing *heads*. If the coin lands *heads*, the I_1 instruction is executed; otherwise, the I_2 instruction is executed instead. This pseudo-operator can be defined in terms of the above instructions:

$$
\begin{aligned}
InstrH \ \oplus_q \ InstrT \equiv\ & \text{IToss } r\ q \\
& \text{ITest } r\ true \\
& \text{IJmp } 3 \\
& InstrT \\
& \text{IJmp } 2 \\
& InstrH
\end{aligned}
$$

The probabilistic toss instruction **IToss** is first invoked, storing a probabilistic boolean value into a new register r. Then, the conditional branch instruction **ITest** is used to determine which of the boolean values has been assigned to the register. In the case of a *true* value, the leftmost instruction $InstrH$ is executed. Otherwise, the rightmost instruction $InstrT$ is executed, and an unconditional branch instruction is used to skip the $InstrH$ instruction.

To implement bounded iteration, two additional pseudo-instructions are defined. The instructions are named in correspondence with the **ForLoop** instructions in the Lua v5.0 virtual machine, and have a similar role. The **IForPrep** instruction initializes a loop body. The **ForLoop** instruction performs a three-step operation:

1. the loop counter register is incremented by the step-value

2. a less-than comparison is made between the loop counter and the looping end value

3. a negative-offset branch to the top of the loop body is performed if the loop counter is less than the end value; otherwise, the program counter is incremented and the loop is terminated.

To implement bounded iteration, a syntactic translation is performed, which is known as *loop unrolling*. The *loop-body* is defined to be the sequence of instructions between a **IForPrep** and a **IForLoop** instruction. The *loop-body* instructions are duplicated $(end - start)/incr$ times. The loop counter register is then destructively updated at the end of every duplicated loop body. The start, end and step parameters of the for loop must be constant values.

Suppose that we assume that loops have a bounded iteration count; for instance, 1000 iterations. Loops that have a run-time computed bound that is less than the maximum iteration count can be handled by jumping to the end of the unrolled loop if the counter exceeds the run-time computed limit. If the run-time computed bound exceeds the maximum iteration count, then execution of the loop will terminate as if the loop bound was exceeded.

6.4 Well-Formed Programs

An SPL program is a sequence of instructions chosen from the above set. SPL programs must meet certain criteria to be considered *well-formed*. Non-well-formed programs are invalid, and do not have a semantics.

6.4.1 Forward-Branching

All SPL programs with unconditional branch instructions must ensure that all offset are non-negative; an offset of 0 is equivalent to a no-operation instruction. Branch offsets are fixed by the compiler, and cannot be altered during the execution of a program. Bounded iteration is accomplished by repeated sequences of branching instructions, as described above.

6.4.2 Constrained Branching

All SPL programs with branch targets must ensure that the target(s) of the branch must result in the program counter residing within the range $(1, |\mathfrak{p}|)$, where $\mathfrak{p}$ is a sequence of instructions and $|\mathfrak{p}|$ is its length. This can be assumed by our proofs, as a verification algorithm can deterministically decide if a program's branch targets are within the accepted range.

6.4.3 Return Instruction

All SPL programs end with an **IReturn** instruction. This is enforced by a verification algorithm, and the Lua-to-SPL compiler automatically appends an IReturn instruction, whether the program source includes a return statement or not.

6.5 Motivating Examples

To motivate the work, sample probabilistic programs are introduced and the execution environment is described.

These examples assume that the probabilistic behaviour under study is a probabilistic corruption of data values. That is, a register value will contain a slightly different value than what would be expected in a deterministic execution of the program. Programs can be proven tolerant of these errors via algorithms designed to detect and correct data errors.

Programs that exhibit “instruction skipping” behaviour do not have a simple algorithmic solution. Such programs could be made robust by judicious instruction choices; specific details are beyond the scope of this work.

6.5.1 Parity Check

To correct errors in information transmission or retrieval, error detection and error correction algorithms are used to validate the correctness of a value. Redundancy is added to a message, and a simple procedure can be used to detect the existence of an error, or correct it completely.

A simple error detection method is the addition of a *parity bit*. A parity bit is one bit of information that is computed from the original bits of some value, generally via an `XOR` operation. The parity bit can be used to detect when a single bit-flip error has occurred, and notify the appropriate subsystem that data has been corrupted. Parity checks can only detect single-bit flip errors; double-bit flip errors will leave the parity bit in the same state as the non-error state.

The parity check algorithm is a simple example amenable to probabilistic verification. A simple version of the parity checking algorithm is shown in Example A.1 in Appendix A, expressed in the Simplified Lua language. This program computes

the parity of a given (integer) number by using the >> and & binary operators. The $a >> n$ operator performs the bitwise right shift operation; given a number a, >> will shift the bits of the binary representation n units rightward, resulting in a new number a'. For instance, assume $a = 10$, which has binary representation $b1010$. $a >> 1$ would shift the bits one unit rightward, resulting in $b101 = 5$. Similarly, the & and ˆ operators perform bitwise logical AND and XOR between two numbers, respectively. By applying $a\&1$ to a number a, we can determine the value of the least-significant bit of the binary representation. In the case where $a = 10$, $a\&1 = 0$.

The parity program iteratively computes the bitwise exclusive-OR operation between all the bits of the input number, and returns that value (either 0 or 1) as its output.

Example A.2 in Appendix A is the result of compiling Example A.1 to the Simplified Probabilistic Lua bytecode, introducing a probabilistic behaviour to the initial load of the input number. The input number will be retrieved from a specific register, with a high probability of success; however, a single-bit flip error will occur with a small but non-zero probability p_1, and a double bit-flip error will occur with a still smaller probability p_2. This is simulated using the **IXor**, **IAnd** and **IShr** instructions, which perform the bitwise XOR, bitwise AND and bit-shift right operations, respectively. The bits in the input value are individually inspected and combined via the **IXor** instruction to compute the parity bit value. We can use the probabilistic Hoare logic rules to verify either that no error has occurred, or an error in retrieval occurs with a probability of $1 - p_2$.

6.5.2 Error correction via Hamming Code

The automatic correction of errors is the natural step from error detection. Error correction adds additional redundancy to a stream of bits, so if an error occurs and the original value is corrupted, the redundant information can be used to recover the original, unaltered value.

An easy method of error correction involves sending multiple copies of the data, and if a bit in one of the copies differs from the others, an error occurred in transmission. Discovering the correct bit can be done by querying the remaining copies;

assume that "majority rules", and if the other two copies have a value (say, `1`), then the correct value should be 1. The number of copies should be an odd number, otherwise ties may result. Assuming that 3 copies of the data are used, this scheme would require 3 times more data in a transmission than the size of the uncorrectable message.

Hamming codes are an encoding scheme that has a reduced overhead compared to majority-rules, but has a decreased ability to correct errors. This encoding scheme can detect 2-bit errors, and correct 1-bit errors; in contrast, majority-rules can detect errors in every bit of a message. However, assuming that a communication channel is relatively noise-free, Hamming codes are a common algorithm for error-correction.

The probabilistic Hoare-style logic rules that we present could be used to prove that the probability of the original value being recovered is total; i.e., $Pr(\langle\text{return value}\rangle = \langle\text{original value}\rangle) = 1$.

7 Semantics of SPL

Two approaches are used to describe the meaning of probabilistic programs. The approaches parallel the two common interpretations of quantum mechanics, the Copenhagen and the Many-worlds [44] interpretations. However, these names should only be considered guides to the semantics, rather than as exact correspondences with the quantum mechanical philosophies.

Probabilistic programs are defined by the existence of a probabilistic toss instruction, which non-deterministically assigns a Boolean value to a specified register. The probabilistic toss instruction is independent of program register values, which differs from the demonic probabilistic choice in McIver [38].

The semantics differ in the handling of probabilistic states. Many-worlds semantics operate over a probabilistic state, as defined in Section 4.3. In contrast, Copenhagen-style semantics concretize probabilistic choices, and operate non-deterministically over a classical valuation. Small-step operational semantics are used to describe the semantics of the SPL language in both approaches.

7.1 "Many-Worlds" Operational Semantics

The "Many-Worlds" operational semantics is named in analogy to the interpretation of quantum mechanics; this is merely meant to give a flavour of the semantics, rather than imply a direct correspondence between this model and quantum properties.

Under the Many-World semantics, a program operates over a *generalized probabilistic state*. A program transforms an initial probabilistic state μ to a final state μ', such that for any chosen initial state, the final state is uniquely determined. A computation results in a probabilistic distribution of valuations, rather than a single valuation as in a classical computation.

7.1.1 Valuations

Like the Secure Assembly Language defined in [39] and physical computing systems, an executing program operates over a state consisting of a program counter $pc \in \mathbb{N}$ and a finite collection of memory cells. These cells take values from the booleans or the real number set $\mathcal{D}$. As execution progresses, the program counter is incremented or modified via a branch instruction. This counter is unavailable to the program; control flow can only be directed via the branch instructions.

To represent the state of execution of the Many-Worlds Simplified Probabilistic Lua (MW-SPL) machine, we recall the notion of a valuation from section 4.3. Here, 3 sets of indices are used for an SPL program.

1. registers: denoted $r_1, r_2, \ldots$
2. global variables: indices are character strings cs chosen from the ASCII alphabet, and denoted $Glb(cs)$
3. constants: denoted $k_1, k_2, \ldots$

Recall that in Section 4.3 a valuation had type $Reg \to \mathcal{D}$, where Reg corresponds to the first set of indices mentioned here. We extend Reg to now also include the second and third sets. A valuation will map all of these indices to their values. We make one extension to the set $\mathcal{D}$; constant store values can include members of the ASCII character string set, to act as indices into the global variable store; this will be clear from context and will not be remarked upon further.

The constant pool is used by instructions to load static data into registers for arithmetic operations or other computations, and cannot be changed by a program's execution. The global variable store is used to parameterize a program's execution. Recall that the values in the constant store are denoted $K(k_n)$, and globals denoted $Glb(c)$, for some $k_n \in$ constants and some $c \in cs$.

Execution of a program begins with a state s, consisting of a single valuation with the program counter set to 0, and all registers set to a pre-defined value such as 0. This initial valuation has probability 1.0. Using the small-step transition relation described below, a new state s' results. If the program counter points to an **IReturn**

r instruction, the execution is considered completed, and a value (or values) stored in r is treated as the computation's *return value*.

Definitions

An SPL program $\mathfrak{p}$ is a sequence of instructions, $I_1, I_2, I_3, \ldots I_{|\mathfrak{p}|}$. The nth instruction in a program $\mathfrak{p}$ is denoted by I_n. Alternatively, we can abstract from integer indices, and refer to instructions by their *label*. The instruction at a label l is denoted by I_l.

Function addition is denoted $f + f'$ and is defined as $(f + f')(v) = f(v) + f'(v)$. Multiplication by a number r is denoted $r \cdot f$, and defined $(r \cdot f)(v) = r \cdot f(v)$.

Given a register r and an expression e, we define the function $\delta^r_e : \mathcal{V} \to \mathcal{V}$ to be the function that maps a valuation to another one where the value of the expression e under the input valuation is assigned to the register r, and all other registers contain the same value; or, $\delta^r_e(v) = v[r := e]$. $(\delta^r_e)^{-1} : 2^{\mathcal{V}} \to 2^{\mathcal{V}}$ is the function taking each set $V \subseteq \mathcal{V}$ to the set of its pre-images; this is useful, as it allows us to determine the measure of a state after an assignment, by interrogating the original state measure function.

Two functions *njmp* and *step* are defined with respect to probabilistic states. Given a probabilistic state μ and natural number n, the *njmp* function is defined as $njmp(\mu, n) = \mu \circ (\delta^{pc}_{pc+n})^{-1}$, and is used to describe an increment of the program counter by a fixed offset. The *step* function is defined as $step(\mu) = njmp(\mu, 1)$, and is used to increment the program counter by 1.

7.1.2 Operational Small-Step Semantics

The Many-Worlds operational semantics relation defines a binary transition relation (actually, a functional relation) on probabilistic states μ, denoted $\to_{MW}$.

Recall that a probabilistic state μ is defined to be a probabilistic measure over the set of valuations $\mathcal{V}$. By the definition of a probabilistic measure,

$$\mu(U_1 \cup U_2) = \mu(U_1) + \mu(U_2) \text{ if } U_1 \cap U_2 = \varnothing$$

Recall the definition of μ_γ given a $\mathcal{V}$:

$$\mu_\gamma = \mu(|\gamma|_{\mathcal{V}})$$

Therefore, we can represent a probabilistic state μ as the sum of disjoint sub-states. In particular,

$$\mu = \mu_{\text{pc}=1} + \mu_{\text{pc}=2} + \cdots + \mu_{\text{pc}=|\mathfrak{p}|}$$

The single-step transitive relation of a probabilistic state μ given an SPL program $\mathfrak{p}$ can be given by

$$\mu'_1 + \mu'_2 + \cdots + \mu'_{|\mathfrak{p}|-1}$$

where μ'_l is given by $[[I_l]]\langle \mu_{\text{pc}=l} \rangle \rightarrow_{MW} \langle \mu'_l \rangle$.

A probabilistic sub-state is terminated when the program counter register pc = $|\mathfrak{p}|$; a probabilistic state μ is terminated when $\mu = \mu_{\text{pc}=|\mathfrak{p}|}$

We now define the single-step transition relation for each instruction. Unless otherwise stated, every instruction transition rule includes an implicit increment of the program counter register over all classical valuations in the probabilistic state. There is no transition relation for the **IReturn** instruction. Once a probabilistic sub-state is terminated, there is no further transitions possible.

Instructions that update machine registers

$[[\text{ILoadK } r\ k]]\langle \mu \rangle \rightarrow_{MW} \langle \mu \circ (\delta^r_{K(k)})^{-1} \rangle$

The **ILoadK** instruction populates a machine register with the value of the kth element of the constant pool.

$[[\text{IGetGlobal } r\ k]]\langle \mu \rangle \rightarrow_{MW} \langle \mu \circ (\delta^r_{Glb(K(k))})^{-1} \rangle$

The **IGetGlobal** instruction populates a machine register with the value from the global variable set. The ASCII name of the global variable is found by indexing into the constant pool.

$[[\text{ILoadBool } r\ k]]\langle \mu \rangle \rightarrow_{MW} \langle \mu \circ (\delta^r_b)^{-1} \rangle$ where b = true if k = 1, b = false otherwise

The **ILoadBool** instruction populates a machine register with the value of a constant Boolean. Unlike the **ILoadK** instruction, there is no need to reference the constant pool. Instead, the value of the constant (Boolean *true* or $false$) is encoded within the instruction itself.

$[[\text{IMove } r_d\ r_s]]\langle \mu \rangle \rightarrow_{MW} \langle \mu \circ (\delta^{r_d}_{r_s})^{-1} \rangle$

The **IMove** instruction populates a machine register with the value of another machine register.

$[\![\text{IAdd } r_d\, r_1\, r_2]\!]\langle\mu\rangle \rightarrow_{MW} \langle\mu \circ (\delta^{r_d}_{r_1+r_2})^{-1}\rangle$

The **IAdd** instruction computes the arithmetic sum of the values r_1 and r_2. These values may be registers, or they may be entries in the program's constant pool. The other arithmetic instructions (**ISub**, **IMul**, **IDiv**, **IXor**, **IShr**, **IShl** and **IMod**, **IAnd**) are handled similarly.

$[\![\text{I}Arith\ r_d\, r_1\, k]\!]\langle\mu\rangle \rightarrow_{MW} \langle\mu \circ (\delta^{r_d}_{r_1+K(k)})^{-1}\rangle$ A variant of the binary arithmetic instruction will operate over a register r_1 and the kth value from the constant pool. The resulting value is calculated in the same manner as the above case, and stored in the destination register r_d.

$[\![\text{IUnm } r_d\, r_s]\!]\langle\mu\rangle \rightarrow_{MW} \langle\mu \circ (\delta^{r_d}_{-r_s})^{-1}\rangle$

The **IUnm** instruction computes the negation of a floating-point number value stored in register r_s. The **INot** instruction performs a similar operation over the Boolean values.

Instructions that perform probabilistic tosses

$[\![\text{IToss } r_d\, q]\!]\langle\mu\rangle \rightarrow_{MW} \langle q \cdot \mu \circ (\delta^{r_d}_{true})^{-1} + (1-q) \cdot \mu \circ (\delta^{r_d}_{false})^{-1}\rangle$

The **IToss** instruction tosses a probabilistic coin, and divides the state into two portions; a state μ'_{true} where the contents of register r_d have been set to $true$, and a state $\mu'_{\text{false}}$where r_d is $false$. The parameter q controls the probability of a $true$ toss result; this value must be less than or equal to 1. In the case where $q = 1$, this instruction is identical to **ILoadBool** r_d **1**; similarly, $q = 0$, this reduces to **ILoadBool** r_d **0**.

There is some subtlety to the IToss instruction; consider the program in Example 7.1 starting in the initial probabilistic state $\mu_0 = (pc \mapsto 0) \mapsto 1.0$; μ_0 is the function mapping any valuation that assigns pc to value 0 to 1.0, and all other valuations to 0.0. This program will toss a probabilistic coin with a bias of 0.9, and store the results in register r_1. It will then toss a second coin with a bias of 0.7, and overwrite the contents of register r_1 with the result of that toss. We would expect this program's meaning to be identical to a program consisting of the second instruction.

Example 7.1: Overwriting Toss Registers

```
IToss r1 0.9
```

```
IToss r1 0.7
```

The first IToss instruction results in a probabilistic state μ_1

$$[(pc \mapsto 1, r1 \mapsto true) \mapsto 0.9] + [(pc \mapsto 1, r1 \mapsto false) \mapsto 0.1]$$

The second IToss instruction is computed in two parts:

$$[(pc \mapsto 2, r1 \mapsto true) \mapsto (0.7) \cdot (0.9)] + [(pc \mapsto 2, r1 \mapsto false) \mapsto (0.3) \cdot (0.9)] \quad (1)$$

$$[(pc \mapsto 2, r1 \mapsto true) \mapsto (0.7) \cdot (0.1)] + [(pc \mapsto 2, r1 \mapsto false) \mapsto (0.3) \cdot (0.1)] \quad (2)$$

These two sub-states are combined using function addition, resulting in the final state μ_2

$$[(pc \mapsto 2, r1 \mapsto true) \mapsto 0.7] + [(pc \mapsto 2, r1 \mapsto false) \mapsto 0.3]$$

which meets the intuitive expectation of the program.

Instructions that Modify the Control Flow

$[\![\text{IJmp } n]\!]\langle\mu\rangle \rightarrow_{MW} \langle njmp(\mu, n)\rangle$

The **IJmp** instruction unconditionally branches to an instruction that is n offsets from the current program counter. This offset must be a positive number, and must result in a state where the program counter is less than the total number of instructions in the SPL program.

$[\![\text{ILt } r_1\ r_2]\!]\langle\mu\rangle \rightarrow_{MW} \langle njmp(\mu_{r_1<r_2}, 1) + njmp(\mu_{\neg(r_1<r_2)}, 2)\rangle$

The **ILt** instruction is used to implement conditional branches. The values stored in registers r_1 and r_2 are compared, and if the comparison succeeds the program counter is incremented (which doesn't include the implicit increment). If the comparison fails, the program counter is incremented twice. Generally, the instruction immediately following **ILt** will be an **IJmp** instruction, which has the same restrictions as noted above: n must be positive, and must be bounded by the size of the program.

The instructions **ITest** and **ILe** are handled similarly, differing only in the comparison operator. **ITest** compares the registers for equality, while **ILe** tests for less-than-or-equality. All of these operations are defined only over the floating-point numbers.

Table 8: Operational Semantics for Copenhagen VM

$[[\text{ILoadK } r\ k]]\langle(v,\theta)\rangle \rightarrow_C \langle(v^r_{K(n)},\theta)\rangle$
$[[\text{ILoadBool } r\ k]]\langle(v,\theta)\rangle \rightarrow_C \langle(v^r_b\rangle,\theta)$ where b = true if c = 1, b = false otherwise
$[[\text{IGetGlobal } r_d\ k]]\langle(v,\theta)\rangle \rightarrow_C \langle(v^{r_d}_{Glb(K(k))},\theta)\rangle$
$[[\text{IMove } r_d\ r_s]]\langle(v,\theta)\rangle \rightarrow_C \langle(v^{r_d}_{v(r_s)},\theta)\rangle$
$[[\text{IAdd } r_d\ r_1\ r_2]]\langle(v,\theta)\rangle \rightarrow_C \langle(v^{r_d}_{v(r_1)+v(r_2)},\theta)\rangle$
$[[\text{ISub } r_d\ r_1\ r_2]]\langle(v,\theta)\rangle \rightarrow_C \langle(v^{r_d}_{v(r_1)-v(r_2)},\theta)\rangle$
$[[\text{IMul } r_d\ r_1\ r_2]]\langle(v,\theta)\rangle \rightarrow_C \langle(v^{r_d}_{v(r_1)\cdot v(r_2)},\theta)\rangle$
$[[\text{IXor } r_d\ r_1\ r_2]]\langle(v,\theta)\rangle \rightarrow_C \langle(v^{r_d}_{v(r_1)\oplus v(r_2)},\theta)\rangle$
$[[\text{IAnd } r_d\ r_1\ r_2]]\langle(v,\theta)\rangle \rightarrow_C \langle(v^{r_d}_{v(r_1)\wedge v(r_2)},\theta)\rangle$
$[[\text{IUnm } r_d\ r_s]]\langle(v,\theta)\rangle \rightarrow_C \langle(v^{r_d}_{-v(r_1)},\theta)\rangle$
$[[\text{IJmp } n]]\langle(v,\theta)\rangle \rightarrow_C \langle(\text{cjmp}(v,n),\theta)\rangle$
$[[\text{ILt } r_1\ r_2]]\langle(v,\theta)\rangle \rightarrow_C \langle(\text{cjmp}(v,1),\theta)\rangle$ if $v(r_1) < v(r_2)$
$[[\text{ILt } r_1\ r_2]]\langle(v,\theta)\rangle \rightarrow_C \langle(\text{cjmp}(v,2),\theta)\rangle$ if $\neg(v(r_1) < v(r_2))$
$[[\text{ILe } r_1\ r_2]]\langle(v,\theta)\rangle \rightarrow_C \langle(\text{cjmp}(v,1),\theta)\rangle$ if $v(r_1) \leq v(r_2)$
$[[\text{ILe } r_1\ r_2]]\langle(v,\theta)\rangle \rightarrow_C \langle(\text{cjmp}(v,2),\theta)\rangle$ if $\neg(v(r_1) \leq v(r_2))$
$[[\text{ITest } r_1\ r_2]]\langle(v,\theta)\rangle \rightarrow_C \langle(\text{cjmp}(v,1),\theta)\rangle$ if $v(r_1) = v(r_2)$
$[[\text{ITest } r_1\ r_2]]\langle(v,\theta)\rangle \rightarrow_C \langle(\text{cjmp}(v,2),\theta)\rangle$ if $\neg(v(r_1) = v(r_2))$
$[[\text{INop}]]\langle(v,\theta)\rangle \rightarrow_C \langle(v,\theta)\rangle$
$[[\text{IToss } r_d\ q]]\langle(v,\theta)\rangle \rightarrow_C \langle(v^{r_d}_{cointoss(q)}, q\cdot\theta)\rangle$ if cointoss(q) is heads
$[[\text{IToss } r_d\ q]]\langle(v,\theta)\rangle \rightarrow_C \langle(v^{r_d}_{cointoss(q)}, (1-q)\cdot\theta)\rangle$ if cointoss(q) is tails

INSTRUCTIONS THAT DO NOT MODIFY THE STATE

$[[\text{INop }]]\langle\mu\rangle \rightarrow_{MW} \langle\mu\rangle$

The **INop** instruction does not affect the valuation, besides the implied increment of the program counter.

7.2 Operational Semantics for the "Copenhagen" Virtual Machine

An operational semantics of the Copenhagen Simplified Probabilistic Lua (C-SPL) virtual machine is given in Table 8. The intention of the *Copenhagen virtual machine* is to provide a correspondence with a physical machine, where probabilistic errors result in non-deterministic output of a computation.

Unlike the Many-Worlds model, the state is not probabilistic; the state contains

a classical valuation. To simplify the presentation, we write v^r_e to denote $\delta^r_e(v)$ or $v[r := e]$. The program counter update functions $cjmp : \mathcal{V} \to (\mathbb{N} \to \mathcal{V})$ increments the distinguished program counter register by a fixed offset, and is defined $cjmp(v, n) = v^{\text{pc}}_{\text{pc}+n}$.

When a probabilistic choice between two instructions is required, the Copenhagen virtual machine queries a source of randomness that meets certain criteria; independence, uniformity and lack of periodicity. Based upon the result of this query, the classical valuation v is updated to v', based upon the transition relation of the chosen instruction.

The state is extended to include a real number value θ, which denotes the computed probability of the valuation v based upon the execution of the program. The program begins in a known and defined valuation, and therefore the initial probability value is 1.0. As the program executes and probabilistic choices are made, the probability value is updated accordingly.

We assume that probabilistic choices are independent, and therefore updates to the probability value can be easily computed: $\theta' = \theta \cdot \text{probability of the choice}$. Conditional probabilistic updates could be included, but is considered outside the scope of this work.

The advantages of this semantics is that it is similar to the behaviour of (classical) physical computers that exhibit probabilistic behaviour. When a cosmic ray strikes a memory unit, the execution of a program irrevocably changes, and the program continues executing with updated register contents.

The Copenhagen operational semantics relation $\to_C$ is given over classical valuations v. Every transition other than the branch instructions includes an implicit increment of the valuation's program counter register, as well as incrementing probabilistic value θ as required.

Like the Many-Worlds semantics, the **IToss** instruction tosses a probabilistic coin. However, rather than dividing the probabilistic state into two, the outcome of the coin toss is decided immediately and the contents of register r_d will be set to the tossed value. The parameter q controls the probability of a *true* toss result; this value must be less than or equal to 1. In the case where $q = 1$, this instruction is identical to

ILoadBool r_d **1**.

To illustrate, assume the virtual machine is in the initial state $(v_0, 1)$ where v_0 is the initial classical valuation where all registers are set to a pre-defined value, and is executing the following probabilistic program:

$$\begin{gathered}\text{ILoadK } r1\ \ 42 \oplus_{0.3} \text{ ILoadK } r1\ \ 24 \\ \text{IAdd } r2\ \ r1\ \ 1 \oplus_{0.65} \text{ IAdd } r2\ \ r1\ \ 2\end{gathered} \tag{3}$$

The virtual machine will query a random source (`/dev/urandom` on Linux, the Cryptographic Service API's on Microsoft Windows, or a Internet-based randomness service such as `www.random.org`) and choose an instruction (*left* or *right*) to execute. Assuming further than the leftmost instruction is chosen for the first probabilistic choice, and the rightmost instruction is chosen from the second, the virtual machine state will update the register `r1` with the constant value `42`, then update `r2` with the value of `r1` + 2 ; unchosen instructions are ignored. The updated probabilistic value θ'' is computed as follows:

$$\begin{aligned}\theta' &= 1.0 \cdot 0.3, \\ \theta'' &= \theta' \cdot (1.0 - 0.65) \\ &= 0.105\end{aligned}$$

Global memory and register sets are modelled as classical valuations; there is no uncertainty about the contents of registers nor global variables. Rather, the uncertainty is in the final output of the execution.

7.3 Modelling Errors

The **IToss** instruction along with the choice of an initial probabilistic state defines a method of modelling certain classes of probabilistic errors that manifest in realistic programs. By introducing IToss instructions into a SPL program, different erroneous behaviours can be expressed, depending upon the expected context of an execution. The choice of the initial probabilistic state will determine the process of execution.

To model classical programs, the initial state can be restricted to a register mapping with measure 1 and all other mappings measure 0, and no **IToss** instructions are allowed within the program body. This valuation models a machine where no probabilistic errors can occur; this is the ideal, but unlikely case.

To model programs that involve bit-flips, either due to transient events such as cosmic rays, or physical errors such as a noisy communications medium, the program begins in a classical state, but every access to a set of designated registers must be guarded by an IToss instruction (or the pseudo-instruction $\oplus_p$).

Example 7.2: Overwriting Toss Registers

```
IToss r1 0.0001
ITest r1 True
IJmp 3                    # Coin toss came up heads, no bit flip.

ILoadK r1 0x8042    # Coin toss came up tails, bit flip occurred!
IJmp 2

IMove r1 0x42
```

For example, the program in Example 7.2 could refer to an infrequent bit-flip event occurring during the assignment to the r_1 register. A probabilistic coin is tossed with a bias of 0.0001, and the result is stored in register `r1`. Based on the outcome of this result, one of two register assignments occur. In the case of a heads coin-toss, the register `r1` is overwritten with the arbitrary hexadecimal value 0x42, which has the following representation in binary notation: `0000000001000010`. In the other case, the register `r1` is assigned the hexadecimal value 0x8042, or binary `1000000001000010`. Notice that these two binary representations differ only in the leftmost digit.

Modelling programs that involve skipping instructions can be modelled similarly, with judicious insertion of **IToss**, **ITest** and **INop** instructions.

7.4 Termination

The SPL language can be shown to have termination properties that are desirable. First, we define the notion of the *transitive closure of the transition relation* for our SPL semantics.

We denote the transitive closure of a transition relation given an SPL program $\mathfrak{p}$ and probabilistic states μ and μ' as:

$$[\![\mathfrak{p}]\!]_\rho \langle \mu \rangle \rightarrow^* \langle \mu' \rangle$$

The specific transition relation will be specified by a subscript. The transitive closure of the single-step relation is given by applying the single-step transition relation iteratively until the execution terminates. In the case of the Copenhagen semantics, given a classical valuation v, this means $v(\text{pc}) = |\mathfrak{p}|$. In the case of the Many-World semantics, given a probabilistic state μ, this means $\mu = \mu_{\text{pc}=|\mathfrak{p}|}$.

For example, consider the simple program **ILoadK r1 1**; **IAdd r1 r1 r2**; **IReturn**. In the Many-Worlds semantics, assuming that the execution begins with the probabilistic state μ_0 (the probabilistic state which maps to probability 1.0 the valuation that assigns all registers the value 0 and the program counter initialized to 1, and all other valuations assigned the probability 0), the single-step transition relation is iteratively applied to μ_0:

$[\![\textbf{ILoadK r1 1}]\!]\langle \mu_0 \rangle \rightarrow_{MW} \langle \mu'_{\text{pc}=2} \rangle$
$[\![\textbf{IAdd r1 r1 r1}]\!]\langle \mu'_{\text{pc}=2} \rangle \rightarrow_{MW} \langle \mu''_{\text{pc}=3} \rangle$

Due to the lack of probabilistic toss and conditional branch instructions, there is no need to describe the disjoint sub-states associated with each possible program counter value; these substates will map all valuations to 0. The final state $\mu''_{\text{pc}=3}$ will map the valuation $[r_1 \mapsto 2, \text{pc} \mapsto 3]$ to the probability 1, and all other valuations to 0.

Similarly, in the Copenhagen semantics, assuming the execution begins with the classical state $(v_0, 1)$ (v_0 is the classical state where all registers are assigned the value 0 and the program counter is assigned the value 1), the single-step transition relation is iteratively applied to v_0:

$[\![\textbf{ILoadK r1 1}]\!]\langle(v_0, 1)\rangle \rightarrow_C \langle(v', 1)\rangle$
$[\![\textbf{IAdd r1 r1 r1}]\!]\langle(v', 1)\rangle \rightarrow_C \langle(v'', 1)\rangle$

The final state v'' will be the valuation $[r_1 \mapsto 2, \text{pc} \mapsto 3]$, and $\theta = 1$.

7.4.1 Probabilistic Termination

We now define the notions of *function termination*, *weak probabilistic termination* and *strong probabilistic termination*.

We define *classical function termination* to be the property such that the instruction at a program counter is the **IReturn** instruction. Upon execution of this instruction, the function is considered to be in a terminated state.

Weak probabilistic termination is the property that, for all initial states that meet the program pre-condition, the transitive closure of the small-step relation of the program results in a probabilistic state that has a non-zero probability of termination; i.e. assuming the Many-Worlds semantics, given a program $\mathfrak{p}$, an EPPL assertion P and probabilistic states μ_0 and μ,

$$\text{If } [\![P]\!]_\rho\langle\mu_0\rangle \text{ and } [\![\mathfrak{p}]\!]_\rho\langle\mu_0\rangle \rightarrow^*_{MW} \langle\mu\rangle, \text{ then } \mu_{\text{pc}=|\mathfrak{p}|}(\mathcal{V}) = 1$$

Strong probabilistic termination is the property that, for all initial states that meet the program pre-condition, the transitive closure of the small-step relation of the program results in all states entailing classical function termination; i.e., assuming the Many-Worlds semantics, given a program $\mathfrak{p}$, an EPPL assertion P and probabilistic states μ_0 and μ, then

$$\text{If } [\![P]\!]_\rho\langle\mu_0\rangle \text{ and } [\![\mathfrak{p}]\!]_\rho\langle\mu_0\rangle \rightarrow^*_{MW} \langle\mu\rangle, \text{ then } \mu_{\text{pc}=|\mathfrak{p}|}(\mathcal{V}) = 1$$

It is simple to show that strong probabilistic termination holds for the subset of SPL programs that do not include **IForPrep**/**IForLoop** instructions.

Proposition 7.4.1. *If $\mathfrak{p}$ is a well-formed SPL program, then for any initial probabilistic state μ_0, $\mathfrak{p}$ will terminate.*

Proof. (Sketch.) Because well-defined SPL programs enforce that all forward branches must have positive offsets, and all instructions other than **IReturn** increment the

program counter, and all programs must have an **IReturn** instruction appended, it can be shown by induction that a program will result in a terminated state. □

7.5 Proof of Equivalence of Many-Worlds and Copenhagen

The Many-Worlds and Copenhagen semantics are radically different in terms of handling probabilistic choice, but they are related, as we will prove in this section.

The Many-Worlds semantics is a functional relation from a probabilistic measure over valuations to another probabilistic measure; however, we need a notion of the probabilistic output of the Copenhagen virtual machine.

To describe the probabilistic measure associated to the output of Copenhagen virtual machine, we take the probabilistic measure to be the sum of all possible probabilistic choices made by a probabilistic program. In the cases where a probabilistic program is guaranteed to terminate, as in this thesis 7.4, the distribution will be finite.

Definition 7.5.1. *We define the function lift* $: \mathcal{V} \to (2^{\mathcal{V}} \to \mathcal{A})$*, which takes a valuation* v *and lifts it to a probability measure:*

$$lift_v(V) = \begin{cases} 1 & \text{if } v \in V \\ 0 & \text{otherwise} \end{cases}$$

Lemma 7.5.2. *Given an expression* e*, a register* r*, and a valuation* v

$$lift_v \circ (\delta_e^r)^{-1} = lift_{\delta_e^r(v)}$$

Proof. Recall that $(\delta_e^r)^{-1}(V) = \{v \in \mathcal{V} \mid (\delta_e^r)(v) \in V\}$. Thus we have:

$$\begin{aligned} lift_v((\delta_e^r)^{-1}(V)) &= \begin{cases} 1 & \text{if } v \in (\delta_e^r)^{-1}(V) \\ 0 & \text{otherwise} \end{cases} \\ &= \begin{cases} 1 & \text{if } \delta_e^r(v) \in V \\ 0 & \text{otherwise} \end{cases} \\ &= lift_{\delta_e^r(v)}(V) \end{aligned}$$

□

Lemma 7.5.3. *Given a valuation v, a real number θ in the range [0,1] and an SPL instruction $I \neq ITоss$*

$$[\![I]\!]\langle lift_v \rangle \rightarrow_{MW} \langle lift_{v'} \rangle \textit{ if and only if } [\![I]\!]\langle (v, \theta) \rangle \rightarrow_C \langle (v', \theta) \rangle$$

Proof. By cases:

- **IAdd** $r_d\ r_1\ r_2$ (remaining arithmetic instructions similar)

  By the definition of $\rightarrow_{\mathrm{MW}}$, we have $[\![\text{IAdd } r_d\, r_1\, r_2]\!]\langle lift_v \rangle \rightarrow_{MW} \langle lift_v \circ (\delta^{r_d}_{r_1+r_2})^{-1} \rangle$.

  By the definition of $\rightarrow_{\mathrm{C}}$, we have $[\![\text{IAdd } r_d\, r_1\, r_2]\!]\langle (v, \theta) \rangle \rightarrow_C \langle (v^{r_d}_{r_1+r_2}, \theta) \rangle$.

  We must show that $lift_v \circ (\delta^{r_d}_{r_1+r_2})^{-1} = lift_{v^{r_d}_{r_1+r_2}}$

  $$\begin{aligned} lift_v \circ (\delta^{r_d}_{r_1+r_2})^{-1} &= lift_{\delta^{r_d}_{r_1+r_2}(v)} && \text{by Lemma 7.5.2} \\ &= lift_{v^{r_d}_{r_1+r_2}} && \text{by notation defined in Section 7.2} \end{aligned}$$

- **ILoadK** $r_d\ k$ (remaining register load instructions similar)

  By the definition of $\rightarrow_{\mathrm{MW}}$, we have $[\![\text{ILoadK } r\ k)]\!]\langle lift_v \rangle \rightarrow_{MW} \langle lift_v \circ (\delta^{r}_{K(k)})^{-1} \rangle$.

  By the definition of $\rightarrow_{\mathrm{C}}$, we have $[\![\text{ILoadK } r\ k]\!]\langle (v, \theta) \rangle \rightarrow_C \langle (v^{r}_{K(k)}, \theta) \rangle$.

  We must show that $lift_v \circ (\delta^{r}_{K(k)})^{-1} = lift_{v^{r}_{K(k)}}$

  $$\begin{aligned} lift_v \circ (\delta^{r}_{K(k)})^{-1} &= lift_{\delta^{r}_{K(k)}(v)} && \text{by Lemma 7.5.2} \\ &= lift_{v^{r}_{K(k)}} && \text{by notation defined in Section 7.2} \end{aligned}$$

- **ILt** $r_1\ r_2$, such that given a valuation v, $v(r_1) < v(r_2)$ (remaining branch instructions and false case similar)

  By the definition of $\rightarrow_{\mathrm{MW}}$, we have

  $$[\![\text{ILt } r_1\ r_2]\!]\langle lift_v \rangle \rightarrow_{MW} \langle njmp((\mathrm{lift}_v)_{r_1<r_2}, 1) + njmp((lift_v)_{r_1 \not< r_2}, 2) \rangle$$

  .

By the definition of $\rightarrow_{\mathrm{C}}$, we have $[\![\text{ILt } r_1\ r_2]\!]\langle(v,\theta)\rangle \rightarrow_C \langle(\text{cjmp}(v,1),\theta)\rangle$.

We must show that $njmp((lift_v)_{r_1<r_2},1) + njmp((lift_v)_{r_1\nless r_2},2) = lift_{\text{cjmp}(v,1)}$

$$\begin{aligned}
njmp((\text{lift}_v)_{r_1<r_2},1)+njmp((lift_v)_{r_1\nless r_2},2) &= njmp((lift_v)_{r_1<r_2},1) \text{ by } v(r_1)<v(r_2)\\
&=(lift_v)_{r_1<r_2}\circ(\delta^{pc}_{pc+1})^{-1} \text{ by definition of } njmp\\
&=(lift_v)\circ(\delta^{pc}_{pc+1})^{-1} \text{ by definition of conditional sub-measure}\\
&=(lift_{\delta^{\text{pc}}_{\text{pc}+1}(v)}) \text{ by Lemma 7.5.2}\\
&=(lift_{v^{\text{pc}}_{\text{pc}+1}}) \text{ by notation of Section 7.2}\\
&=lift_{cjmp(v,1)} \text{ by definition of cjmp}
\end{aligned}$$

□

We will denote by U the Cartesian product of the set of classical valuations $\mathcal{V}$ with the real numbers in the range [0,1], where

$$U = \{(v,\theta) | [\![\mathfrak{p}]\!]_\rho\langle v_0,1\rangle \rightarrow^*_C \langle(v,\theta)\rangle\}$$

The set U is finite, due to the termination result of Proposition 7.4.1, and we will consider specific $U's$ below.

Definition 7.5.4. *Given an SPL program $\mathfrak{p}$, an initial classical valuation v_0 and a set U, we define the probabilistic measure $\varphi^U : 2^{\mathcal{V}} \rightarrow \mathcal{D}$ generated by a program $\mathfrak{p}$ to be:*

$$\varphi^U(V) = \sum_{\substack{(v,\theta)\in U\\ v\in V}} \theta$$

Definition 7.5.5. *Given a set U, an expression e and a register r, we define $\Delta^r_e(U)$ to be*

$$\{(\delta^r_e(v),\theta) \mid (v,\theta)\in U\}$$

Similarly, given a set U and a real parameter q in the range [0,1], we define $\Theta_q(U)$ to be

$$\{(v, q\cdot\theta\ )|\ (v,\theta)\in U\}$$

Lemma 7.5.6. *Given a probabilistic measure φ^U,*

$$\varphi^U \circ (\delta_e^r)^{-1}(V) = \varphi^{\Delta_e^r(U)}(V)$$

Proof.

$$\begin{aligned}
\varphi^U \circ (\delta_e^r)^{-1}(V) &= \varphi^U((\delta_e^r)^{-1}(V)) \\
&= \sum_{\substack{(v,\theta)\in U \\ v\in(\delta_e^r)^{-1}(V)}} \theta && \text{by definition of } \varphi \\
&= \sum_{\substack{(v,\theta)\in U \\ \delta_e^r(v)\in V}} \theta && \text{by definition of pre-image} \\
&= \sum_{\substack{(\delta_e^r(v),\theta)\in\Delta_e^r(U) \\ v\in V}} \theta && \text{by definition of } \Delta \\
&= \varphi^{\Delta_e^r(U)}(V)
\end{aligned}$$

□

Lemma 7.5.7. *Given a probabilistic measure φ^U,*

$$\varphi^{\Theta_q(U)}(V) = q \cdot \varphi^U(V)$$

Proof.

$$\begin{aligned}
\varphi^{\Theta_q(U)}(V) &= \sum_{\substack{(v,\theta)\in\Theta_q(U) \\ v\in V}} \theta && \text{by definition of } \varphi \\
&= \sum_{\substack{(v,\theta)\in U \\ v\in V}} q \cdot \theta && \text{by definition of } \Theta \\
&= q \cdot \sum_{\substack{(v,\theta)\in U \\ v\in V}} \theta && \text{by arithmetic} \\
&= q \cdot \varphi^U(V)
\end{aligned}$$

□

Theorem 7.5.8. *The probabilistic distribution of results from running the Copenhagen virtual machine over a program is the same as the probabilistic distribution resulting from running the program using the Many-Worlds semantics, given that both start from a single classical valuation $v_0, 1$ (properly lifted into a probabilistic state in the case of Many-Worlds).*

$$\forall v_0, \forall \mu', \forall v'. [[\mathfrak{p}]]_\rho \langle lift_{v_0} \rangle \to^*_{MW} \langle \mu' \rangle \wedge [[\mathfrak{p}]]_\rho \langle (v_0, 1) \rangle \to^*_C \langle v', \theta \rangle \Rightarrow \mu' = \varphi^U$$

where U is Cartesian product of the set of classical valuations V and real numbers in the range [0,1].

Proof. (sketch) Induction on the number of probabilistic choices in a program.

- Base case, no **IToss** instructions

  Trivial; the transitive closure of the transition relation results in a equivalent relation between the Many-Worlds and Copenhagen semantics.

- Inductive case, n **IToss** instructions, need to prove for $n+1$ **IToss** instructions:

  We assume that $\mu = \varphi^U$ for n instructions. Without loss of generality, assume that the $n+1^{th}$ **IToss r q** instruction is the final instruction in the program[1]. We must show that $\mu' = \varphi^{U'}$.

  By the definition of the Many-World semantics, we know

  $$\mu' = q \cdot \mu \circ (\delta^r_{\text{true}})^{-1} + (1-q) \cdot \mu \circ (\delta^r_{\text{false}})^{-1}$$

  Similarly, by the definition of the Copenhagen semantics, we know that

  $$U' = \underbrace{\{(\delta^r_{\text{true}}(v), q \cdot \theta) \mid (v, \theta) \in U\} \cup \{(\delta^r_{\text{false}}(v), (1-q) \cdot \theta) \mid (v, \theta) \in U\}}_{\text{disjoint sets}}$$

  For the purposes of this proof sketch, we assume that there are no collisions between the set unions. This restriction can be lifted by extending the state further with a unique "trace" element that ensures uniqueness of the tuples.

[1] It can easily be shown that if an IToss instruction is not the last instruction, but there are no other IToss instructions within the remainder of the program, the result follows from Lemma 7.5.3

We can restate U' as

$$U' = \Theta_q \circ \Delta^r_{\text{true}}(U) \cup \Theta_{1-q} \circ \Delta^r_{\text{false}}(U)$$

.

$$\begin{aligned}
\varphi^{U'}(V) &= \sum_{\substack{(v,\theta)\in U' \\ v\in V}} \theta \\
&= \sum_{\substack{(v,\theta)\in \Theta_q\circ\Delta^r_{\text{true}}(U) \\ v\in V}} \theta + \sum_{\substack{(v,\theta)\in \Theta_{1-q}\circ\Delta^r_{\text{false}}(U) \\ v\in V}} \theta \\
&= \sum_{\substack{(v,\theta)\in \Delta^r_{\text{true}}(U) \\ v\in V}} q\cdot\theta + \sum_{\substack{(v,\theta)\in \Delta^r_{\text{false}}(U) \\ v\in V}} (1-q)\cdot\theta \\
&= q\cdot \sum_{\substack{(v,\theta)\in \Delta^r_{\text{true}}(U) \\ v\in V}} \theta + (1-q)\cdot \sum_{\substack{(v,\theta)\in \Delta^r_{\text{false}}(U) \\ v\in V}} \theta \\
&= q\cdot\varphi^{\Delta^r_{\text{true}}(U)}(V) + (1-q)\cdot\varphi^{\Delta^r_{\text{false}}(U)}(V) \\
&= q\cdot\varphi^{U}\circ(\delta^r_{\text{true}})^{-1}(V) + (1-q)\cdot\varphi^{U}\circ(\delta^r_{\text{false}})^{-1}(V) \\
&= q\cdot\mu\circ(\delta^r_{\text{true}})^{-1}(V) + (1-q)\cdot\mu\circ(\delta^r_{\text{false}})^{-1}(V) \\
&= \mu'(V)
\end{aligned}$$

Thus completes the proof.

□

8 Proof Rules for SPL Instructions

For each instruction in the Simplified Probabilistic Lua language, we define a proof rule that describes how pre-conditions and post-conditions are related. We show in Section 8.4 these rules can be used to derive a pre-condition from a given instruction and its post-condition. Recall that EPPL probabilistic formulas are denoted by η. Note that EPPL formulas with conditionals as defined by Definition 4.3.1 will be used in the proof rule for **ILt**.

8.1 Preliminaries

An SPL program $\mathfrak{p}$ is a sequence of SPL instructions. Recall that we denote I as an SPL instruction, along with the parameters associated with the instruction, and denote the instruction at a label l in a given SPL program $\mathfrak{p}$ as I_l. We assume that all instructions have labels and the SPL program has a list of EPPL assertions associated with each I_l in a program, denoted $\mathcal{E}$. The EPPL assertion associated with program label l is denoted η_l.

Similar to [4], we will present the notion of a *labelled instruction specification.* Given a labelled instruction I_l and an EPPL assertion precondition associated with that instruction η_l, we denote the labelled instruction specification $I_l \triangleright \{\eta_l\}$. The meaning of this specification is whenever a probabilistic state μ satisfies the labelled assertion η_l, upon termination of the instruction I_l, the labelled assertions of the successor[1] instruction will be satisfied by the resulting probabilistic state μ'. It should be noted that labelled instruction specifications refer only to their pre-condition.

As in [4], the meaning of a program assertion $I_l \triangleright \{\eta_l\}$ is defined in relation to the SPL program $\mathfrak{p}$ and the set $\mathcal{E}$. The rules can be used to prove (by induction on the number of instructions executed) that the pre-condition of the **IReturn** instruction holds if the program terminates. Then, if all instructions meet the validity constraints

[1] For a rigorous definition of successor, see Section 8.4

of Section 6.4, then the post-condition of the program $\mathfrak{p}$ is the pre-condition of the **IReturn** instruction, and the pre-condition of the program is the pre-condition of the first instruction.

When the label of the instruction is obvious, we will elide the subscript. For instance, INop $\triangleright \{\eta_l\}$ refers to a **INop** instruction at label l.

8.1.1 Body rule

We first define the notion of a valid derivation in the proof rule system for SPL.

Definition 8.1.1. *Let* $\{P\}\ \mathfrak{p}\ \{Q\}$ *denote the usual notion of Hoare triple where* P *is the precondition and* Q *is the postcondition of program* $\mathfrak{p}$*. Such a program has a* valid derivation*, denoted* $\vdash \{P\}\ \mathfrak{p}\ \{Q\}$*, if there is a proof using the rules for SPL presented here, ending with the following rule.*

$$\frac{\begin{array}{c}\eta_1 := P \\ \vdots \\ \eta_{|\mathfrak{p}|} := Q \\ \forall l \in \{1 \dots |\mathfrak{p}| - 1\} : I_l \triangleright \{\eta_l\}\end{array}}{\{P\}\ \mathfrak{p}\ \{Q\}}$$

Note that this means that a program $\mathfrak{p}$ can be verified if the program instructions can be individually verified. This rule breaks the proof up into one premise containing one labelled instruction for each assertion. Each of these assertions is proved in one step by applying a single inference rule, and there is one rule for each SPL instruction. We define the precondition of a program as the precondition of the first instruction, and the postcondition of a program is the precondition of the **IReturn** instruction.

8.1.2 Inference Rules

We adopt from [10] a number of proof rules for probabilistic programs. Their proofs of soundness can be directly adapted to our logic.

$$\frac{}{\eta \text{ if } \eta \text{ is an EPPL theorem}}\text{TAUT}$$

$$\frac{}{\{\eta\}\ \mathfrak{p}\ \{\eta\} \text{ if } \eta \text{ does not contain any terms of the form } \Pr(\gamma)}\text{Pr-FREE}$$

$$\frac{\{\eta \wedge (y = p)\}\ \mathfrak{p}\ \{\eta\}}{\{\eta^y_p\}\ \mathfrak{p}\ \{\eta\} \text{ if } y \text{ does not occur in } p \text{ or } \eta}\text{ELIMV}$$

$$\frac{\eta_0 \supset \eta_1 \qquad \{\eta_1\}\ \mathfrak{p}\ \{\eta_2\} \qquad \eta_2 \supset \eta_3}{\{\eta_0\}\ \mathfrak{p}\ \{\eta_3\}}\text{CONS}$$

$$\frac{\{\eta_0\}\ \mathfrak{p}\ \{\eta_2\} \qquad \{\eta_1\}\ \mathfrak{p}\ \{\eta_2\}}{\{\eta_1 \vee \eta_1\}\ \mathfrak{p}\ \{\eta_2\}}\text{OR}$$

$$\frac{\{\eta_0\}\ \mathfrak{p}\ \{\eta_1\} \qquad \{\eta_0\}\ \mathfrak{p}\ \{\eta_2\}}{\{\eta_0\}\ \mathfrak{p}\ \{\eta_1 \wedge \eta_2\}}\text{AND}$$

8.2 Instruction Proof Rules

8.2.1 Rules for Instructions that Update the Program Counter

INop

$$\frac{\eta_l \supset \eta_{l+1}}{\text{INop} \rhd \{\eta_l\}}$$

The **INop** instruction is included in this section, as it leaves a state unchanged, except for the implicit update to the program counter register. Therefore, the precondition of the successor instruction can be any formula implied by η_l, (since assertions cannot reference the program counter register). However, it should be noted that all instructions implicitly increment the instruction counter, so the categorization of **INop** is somewhat arbitrary.

IJmp

$$\frac{\eta_l \supset \eta_{l+n}}{\text{IJmp n} \rhd \{\eta_l\}}$$

The **IJmp** instruction is similar to the **INop** instruction; the only update that occurs is to the program counter register, except the update is an arbitrary offset (within the legal bounds of the program).

8.2.2 Rules for Instructions that Load from Memory Stores

This section includes instructions that load from any type of memory store; in SPL, the memory stores are the register list, the global memory store, or the constant pool.

ILoadK

$$\frac{\eta_l \supset {\eta_{l+1}}^{r_1}_{K(k)}}{\text{ILoadK r1 k} \rhd \{\eta_l\}}$$

IMove

$$\frac{\eta_l \supset {\eta_{l+1}}^{r_1}_{r_2}}{\text{IMove r1 r2} \rhd \{\eta_l\}}$$

IGetGlobal

$$\frac{\eta_l \supset {\eta_{l+1}}^{r_1}_{Glb(K(k))}}{\text{IGetGlobal r1 k} \rhd \{\eta_l\}}$$

Recall that the global memory store is a mapping from a string to a Lua value. The **IGetGlobal** instruction uses the constant pool memory store to retrieve the name of a global variable, then loads the value associated with that name from the global variable store.

8.2.3 Rules for Arithmetic instructions

The memory loading instructions can only load from a single memory store; **IGetGlobal** loads from the global variable store, **ILoadK** loads from the constant pool, and **IMove** loads from the register store. However, the arithmetic instructions have a special form that allows loading from either the register store, or the constant pool. Loading from the constant pool is indicated by a special flag in the register parameter, and is fixed at compile time. This is handled in the proof rules by replacing the register name with the constant pool value.

$$\frac{\eta_l \supset {\eta_{l+1}}^{r_1}_{r_2+r_3}}{\text{IAdd r1 r2 r3} \rhd \{\eta_l\}}$$

$$\frac{\eta_l \supset {\eta_{l+1}}^{r_1}_{r_2+d} \wedge K(k) = d}{\text{IAdd r1 r2 k } \rhd \{\eta_l\}} \text{ constant pool}$$

The remainder of the binary arithmetic (**IMul**, **IDiv**, **IXor**, etc.) instructions are handled similarly.

$$\frac{\eta_l \supset {\eta_{l+1}}^{r_d}_{-r_s}}{\text{IUnm } r_d \; r_s \;\; \rhd \{\eta_l\}}$$

The **INot** instruction is handled similarly.

8.2.4 Rules for Probabilistic Instruction

Table 9: Tossed terms and formulas

Tossed terms		
toss(r, q; d)	=	d, where $d \in \mathcal{A}$
toss(r, q; y)	=	y, where y is a logical variable
	=	
toss(r, q; $Pr(\gamma)$)	=	$q \cdot (Pr(\gamma^r_{\text{tt}})) + (1-q) \cdot (Pr(\gamma^r_{\text{ff}}))$
toss(r, q; $p + p'$)	=	toss(r, q; p) + toss(r, q; p')
toss(r, q; $p \cdot p'$)	=	toss(r, q; p) $\cdot$ toss(r, q; p')
Tossed formulas		
toss(r, q; fff)	=	fff
toss(r, q; $p \leq p'$)	=	toss(r, q; p) $\leq$ toss(r, q; p')
toss(r, q; $\eta \supset \eta'$)	=	toss(r, q; η) $\supset$ toss(r, q; η')

IToss

$$\frac{\eta_l \supset \text{toss}(\text{r}, q; \eta_{l+1})}{\text{IToss r } q \rhd \{\eta_l\}}$$

The **IToss** instruction tosses a probabilistic coin and assigns the result to the parameterized register r. The toss predicate function is used to transform an EPPL predicate, and is defined in Table 9.

8.3 Conditional Branching

Unlike the preceding instructions, the conditional branch instructions result in a probabilistic state where the program counter register can differ between substates. The proof rule for conditional branch instructions is more complicated than the non-branching instructions, and is similar to the alternative statement defined in [10], Table 7.

In the Copenhagen semantics, the execution of a conditional branch instruction will update the program counter to one of two locations, depending upon the state of the machine and the parameters to the instruction. For instance, **ILt r1 r2** will branch to the instruction immediately afterwards (labeled I_{true}) if $r_1 < r_2$, and branch to the instruction following I_{true} otherwise.

However, in the Many-Worlds semantics, when a branch instruction is executed, the probabilistic state is split into two sub-states; a sub-state where the classical criteria $r1 < r2$ is true, and a sub-state where $r1 < r2$ is false. We will refer to the two possible paths a conditional branch instruction may take as *traces*. Due to the structure of the SPL machine, we can consider a trace of a program $\mathfrak{p} = I_1, I_2, \ldots, I_{|\mathfrak{p}|}$ to be a suffix of the sequence of instructions $I_i, I_{i+1}, \ldots, I_{|\mathfrak{p}|}$, where $1 \leq i \leq |\mathfrak{p}|$.

By the termination result shown by Proposition 7.4.1, we know that any two traces of a program $\mathfrak{p}$ will have a common instruction, and thus a common post-condition assertion. This result will be assumed in the proof rule.

The proof rule does not assume that the conditional branch instruction is a result of compiling a Simplified Lua IF-statement. However, the two traces of execution that occur after executing the conditional branch instruction can be considered distinct sub-programs, even if the actual instructions executed in the trace overlap.

ILt Recall the definition of γ-conditioned EPPL statements from Definition 4.3.1.

Let l be the index of an **ILt** instruction. Let t' be the sequence of instructions $I_{l+1}, \ldots, I_{|\mathfrak{p}|}$, and t'' be the sequence of instructions $I_{l+2}, \ldots, I_{|\mathfrak{p}|}$.

The instruction specification $ILt \triangleright \{\eta_l\}$ holds if

- the last EPPL assertion in $\mathcal{E}$ is of the form $\{y' + y'' = \Pr(\gamma_0)\}$ for some logical variables y', y'' and some classical state formula γ_0.
- there exist $\mathcal{E}'$, a sequence of EPPL formulas associated with t' (of the same length as t'), and similarly $\mathcal{E}''$ associated with t'' where η' and η'' are the first formulas in $\mathcal{E}'$ and $\mathcal{E}''$, respectively, and the last formulas in $\mathcal{E}'$ and $\mathcal{E}''$ are $y' = \Pr(\gamma_0)$ and $y'' = \Pr(\gamma_0)$, respectively.
- all the premises (in the rule below) hold

$$\frac{\begin{array}{c}\eta_l \supset (\eta'/(r_1 < r_2) \wedge \eta''/(\neg(r_1 < r_2))) \\ \eta_{|\mathfrak{p}|} = \{y' + y'' = \Pr(\gamma_0)\} \\ \{\eta'\}\ t'\ \{y' = \Pr(\gamma_0)\} \\ \{\eta''\}\ t''\ \{y'' = \Pr(\gamma_0)\}\end{array}}{\text{ILt r1 r2} \rhd \{\eta_l\}}$$

If there exists a proof in the logic for the sub-programs t_1 and t_2 (each trace is given by considering the possible traces of the execution of the conditional branch instruction), we can then conclude that the instruction specification in the conclusion of the rule holds.

The **ILe** and **ITest** instructions are handled similarly. The **ILe** instruction will use the conditional $r_1 \leq r_2$, while the **ITest** instruction rule will use the conditional $r_1 = r_2$.

8.4 Special Shape of the Rules

Following [4, 5], we define a weakest precondition function wp, that calculates the weakest precondition of the instruction at label l from the instruction and the weakest preconditions of all successor instructions. Note that this function is defined so that if the weakest preconditions are used as the set of preconditions needed to apply the inference rules defined in the previous subsections, their premises become trivially true implications. For example, if I_l is **INop**, then the weakest precondition is defined to be η_{l+1} by the first line of Table 10, and the premise of the **INop** rule becomes $\eta_{l+1} \supset \eta_{l+1}$.

Given label l, we write $succ(l)$ to denote the *successor function*, which maps a label to a set of labels reachable from an instruction I_l.

$$succ(l) = \begin{cases} (l+1, l+2) & \text{if } I_l \text{ is } \textbf{ILt}, \textbf{ILe} \text{ or } \textbf{ITest} \\ (l+n) & \text{if } I_l \text{ is } \textbf{IJmp n} \\ (l+1) & \text{otherwise} \end{cases}$$

8.4.1 Non-conditional branch instructions

Following the approach of Bannwart-Mueller [4, 5], we examine the structure of the instruction rules. Every instruction has exactly one verification rule, in the form:

$$\eta_l \to \mathrm{wp}^1_{\mathfrak{p}}(I_l, (\eta_i)_{i \in succ(l)})$$

where $succ(l)$ is the function that returns the possible successor indices for a given instruction I_l.

Therefore, we can define the $\mathrm{wp}^1_{\mathfrak{p}}$ function based on our instruction specifications. Table 10 defines the values of the $\mathrm{wp}^1_{\mathfrak{p}}$ function for each instruction and set of labelled assertions for the successor instructions. Observe that $\mathrm{wp}^1_{\mathfrak{p}}(I_l, \mathrm{fff}) = \mathrm{fff}$ for all instructions I_l. This is because the only operations that are applied to the EPPL assertion are substitution, merge and toss, and these operations preserve fff, thus the result of applying $\mathrm{wp}^1_{\mathfrak{p}}$ to fff is fff, regardless of the instruction.

Table 10: Values of the $\mathrm{wp}^1_{\mathfrak{p}}$ function.

I_l	$\mathrm{wp}^1_{\mathfrak{p}}(I_l, (\eta_i)_{i \in succ(l)})$
INop	η_{l+1}
IJmp n	η_{l+n}
ILoadK r1 k	$\eta_{l+1}{}^{r_1}_{K(k)}$
IMove r1 r2	$\eta_{l+1}{}^{r_1}_{r_2}$
IGetGlobal r1 k	$\eta_{l+1}{}^{r_1}_{Glb(K(k))}$
I$BinaryOp$ r1 r2 r3	$\eta_{l+1}{}^{r_1}_{r_2\ op\ r_3}$
I$UnaryOp$ r1 r2	$\eta_{l+1}{}^{r_1}_{op\ r_2}$
IToss r p	$\mathrm{toss}(r, p; \eta_{l+1})$

8.4.2 Conditional Branch instructions

Conditional branch instructions do not follow the special form, and therefore require specific handling. The weakest precondition function for conditional branch instructions relies upon the merge partial function, defined in Table 11. This partial function operates over the structure of EPPL assertions; it can be shown that the weakest precondition generation function preserves the structure of the EPPL postcondition, which ensures that the merge function is defined when applied in the weakest precondition generation algorithm.

For example, $\mathrm{merge}(\{y = \Pr(\gamma) + 0.3\}, \{y = \Pr(\gamma')\}) = \{y = \Pr(\gamma) + 0.3 + \Pr(\gamma')\}$.

Table 11: Merging EPPL assertions

Merged terms		
merge(d, d)	=	d, where $d \in \mathcal{A}$
merge(y, y)	=	y, where y is a logical variable
merge(p, p')	=	$p + p'$
merge$(p_1 + p'_1, p_2 + p'_2)$	=	merge(p_1, p_2) + merge(p'_1, p'_2)
merge$(p_1 \cdot p'_1, p_2 \cdot p'_2)$	=	merge$(p_1, p_2) \cdot$ merge(p'_1, p'_2)
Tossed formulas		
merge(fff,fff)	=	fff
merge$(p_1 \leq p'_1, p_2 \leq p'_2)$	=	merge$(p_1, p_2) \leq$ merge(p'_1, p'_2)
merge$(\eta_1 \supset \eta'_1, \eta_2 \supset \eta'_2)$	=	merge$(\eta_1, \eta_2) \supset$ merge(η'_1, η'_2)

The weakest precondition for the conditional branch instructions is then given in Table 12.

Table 12: Values of the $\mathrm{wp}^1_{\mathfrak{p}}$ function for conditional branch instructions.

I_l	$\mathrm{wp}^1_{\mathfrak{p}}(I_l, (\eta_i)_{i \in succ(l)})$
ILt r1 r2	merge$(\eta_{l+1}/(r1 < r2), \eta_{l+2}/(\neg(r1 < r2)))$
ILe r1 r2	merge$(\eta_{l+1}/(r1 \leq r2), \eta_{l+2}/(\neg(r1 \leq r2)))$
ITest r1 r2	merge$(\eta_{l+1}/(r1 = r2), \eta_{l+2}/(\neg(r1 = r2)))$

8.4.3 Distributive property

Like in Bannwart-Mueller, we check that the $\mathrm{wp}^1_{\mathfrak{p}}$ instruction is distributive over the logical connectives. Since our connectives are defined in terms of fff and $\supset$, we only need to verify that $\mathrm{wp}^1_{\mathfrak{p}}$ distributes over $\supset$. This property is important to prove completeness of the Hoare logic. Since the proofs are straightforward for most instructions, we focus on the more interesting proof rules.

Conditional branching

$$\begin{aligned}
&\mathrm{wp}^1_{\mathfrak{p}}(\mathbf{ILt\ r1\ r2}, \eta_1 \supset \eta_2, \eta'_1 \supset \eta'_2) \\
\Longleftrightarrow&\mathrm{merge}((\eta_1 \supset \eta_2)/(r1 < r2),\ (\eta'_1 \supset \eta'_2)/(r1 \nless r2)) \\
\Longleftrightarrow&\mathrm{merge}((\eta_1/(r1 < r2) \supset \eta_2/(r1 < r2),\ \eta'_1/(r1 \nless r2) \supset \eta'_2/(r1 \nless r2)) \\
\Longleftrightarrow&\mathrm{merge}((\eta_1/(r1 < r2), \eta'_1/(r1 \nless r2))\ \supset\ \mathrm{merge}(\eta_2/(r1 < r2), \eta'_2/(r1 \nless r2)) \\
\Longleftrightarrow&\mathrm{wp}^1_{\mathfrak{p}}(\mathbf{ILt\ r1\ r2}, \eta_1, \eta'_1) \supset \mathrm{wp}^1_{\mathfrak{p}}(\mathbf{ILt\ r1\ r2}, \eta_2, \eta'_2)
\end{aligned}$$

As we can see, conditional branching distributes over the logical operator $\supset$. The conditional operator $/\gamma$ and the merge function both distribute over $\supset$ by definition. The remaining conditional branching instructions are similar.

Probabilistic Toss

$$\begin{aligned}
&\mathrm{wp}^1_{\mathfrak{p}}(\mathbf{IToss\ r\ q}, \eta_1 \supset \eta_2) \\
\Longleftrightarrow&toss(r, q; \eta_1 \supset \eta_2) \\
\Longleftrightarrow&toss(r, q; \eta_1) \supset toss(r, q; \eta_2) \\
\Longleftrightarrow&\mathrm{wp}^1_{\mathfrak{p}}(\mathbf{IToss\ r\ q}, \eta_1) \supset \mathrm{wp}^1_{\mathfrak{p}}(\mathbf{IToss\ r\ q}, \eta_2)
\end{aligned}$$

The $toss(r, q; \eta)$ operator distributes over $\supset$ by definition. The second to last step of the above proof follows from that definition, and we omit its proof.

Arithmetic operations

$$\begin{aligned}
&\mathrm{wp}^1_{\mathfrak{p}}(\mathbf{IAdd}\ r_d r_1 r_2, \eta_1 \supset \eta_2) \\
\Longleftrightarrow&(\eta_1 \supset \eta_2)^{r_d}_{r_1+r_2} \\
\Longleftrightarrow&\eta_1{}^{r_d}_{r_1+r_2} \supset \eta_2{}^{r_d}_{r_1+r_2} \\
\Longleftrightarrow&\mathrm{wp}^1_{\mathfrak{p}}(\mathbf{IAdd}\ r_d\ r_1\ r_2, \eta_1) \supset \mathrm{wp}^1_{\mathfrak{p}}(\mathbf{IAdd}\ r_d\ r_1\ r_2, \eta_2)
\end{aligned}$$

The substitution operator distributes over logical operations by definition.

The remaining instructions follow a similar pattern, and are considered trivial.

8.5 Computational Complexity

Computing the weakest precondition function has polynomial complexity for all instructions, except the **IToss** instruction. However, the complexity of computing $\mathrm{wp}_{\mathfrak{p}}^{1}$ becomes exponential with the number of **IToss** instructions in a program. This complexity limits the size of programs that can be verified to those that have a tractable number of probabilistic toss instructions. This limitation implies that the programs under study should exhibit only occasional probabilistic behaviours, rather than fully general probabilistic behaviour.

9 Soundness

We will now show that our proof logic is sound. Soundness and completeness are properties of our proof calculus that establish a relationship between the proof rules of the SPL language, and the semantics of the language. We will take the Many-Worlds semantics as the de facto semantics of the SPL language; by the result of Theorem 7.5.8, we can assume that a similar result will hold for the Copenhagen semantics. Soundness ensures that if a proof exists for an SPL program, there is a sequence of transitions in the Many-Worlds semantics that begin in a probabilistic state that satisfies the precondition assertion, and results in a probabilistic state that satisfies the postcondition assertion.

Completeness ensures that if a sequence of Many-Worlds transitions begin with a probabilistic state that satisfies an expressible EPPL precondition assertion, and terminates with a probabilistic state that satisfies an expressible EPPL postcondition assertion, then there exists a corresponding proof in our proof calculus. We choose the precondition to be the weakest precondition, and the proof must end in the body rule defined above, which means that each rule associated with an instruction is trivially true.

9.1 Soundness

The proof of soundness is split into two parts. The first part follows the treatment in Bannwart-Müller [4], with the simplification due to the lack of a procedure invocation mechanism. This proof part ignores the conditional branch instructions. Because of this, the proof is rather trivial, but included for the record.

The second part of the proof is based upon the soundness proof in Chadha's work [10], and includes the conditional branching instructions.

9.1.1 Non-branching programs

Definition 9.1.1. *Given an EPPL pre-condition P, post-condition Q and an SPL program $\mathfrak{p}$, we say that the triple $\{P\}\ \mathfrak{p}\ \{Q\}$ is satisfied, denoted $\vDash \{P\}\ \mathfrak{p}\ \{Q\}$, if for any logical variable assignment ρ and any probabilistic states μ and μ' such that*

$$[\![P]\!]_\rho\langle\mu\rangle \textit{ and } [\![\mathfrak{p}]\!]_\rho\langle\mu\rangle \to^*_{MW} \langle\mu'\rangle$$

we have that

$$[\![Q]\!]_\rho\langle\mu'\rangle.$$

Lemma 9.1.2. *Given a valuation v, a classical assertion γ, a register r and term e,*

$$v^r_e \Vdash_c \gamma \textit{ iff } v \Vdash_c \gamma^r_e$$

Proof. Trivial, by induction on the the structure of the assertion statement, γ. □

Recall the definition of *extent* of γ in V: $|\gamma|_V = \{v \in V : v \Vdash_{\mathbf{c}} \gamma\}$.

Lemma 9.1.3. *Given a term e, a register r, a classical assertion γ, then*

$$(\delta^r_e)^{-1}(|\gamma|_V) = |\gamma^r_e|_V$$

Proof.

$$\begin{aligned}
(\delta^r_e)^{-1}(|\gamma|_V) &= \{v \mid v \in (\delta^r_e)^{-1}(|\gamma|_V)\} \\
&= \{v \mid \delta^r_e(v) \in |\gamma|_V\} \\
&= \{v \mid v^r_e \in |\gamma|_V\} \\
&= \{v \mid v \in |\gamma^r_e|_V\} \text{ by Lemma 9.1.2} \\
&= |\gamma^r_e|_V
\end{aligned}$$

□

Lemma 9.1.4. *Given a probabilistic state* μ*, a register* r *and a term* e*, let* $\mu' = \mu \circ (\delta^r_e)^{-1}$*. Then*

$$[\![Pr(\gamma)]\!]_\rho\langle\mu'\rangle = [\![Pr(\gamma^r_e)]\!]_\rho\langle\mu\rangle.$$

We can extend this notion to all probabilistic terms:

$$[\![p]\!]_\rho\langle\mu'\rangle = [\![p^r_e]\!]_\rho\langle\mu\rangle$$

and probabilistic formulas:

$$[\![\eta]\!]_\rho\langle\mu'\rangle \text{ iff } [\![\eta^r_e]\!]_\rho\langle\mu\rangle$$

Proof. By Lemma 9.1.3,

$$(\delta^r_e)^{-1}(|\gamma|_V) = |\gamma^r_e|_V$$

hence

$$\mu((\delta^r_e)^{-1}(|\gamma|_V)) = \mu(|\gamma^r_e|_V).$$

Therefore,

$$\begin{aligned}
[\![Pr(\gamma)]\!]_\rho\langle\mu'\rangle &= \mu'(|\gamma|_V) \\
&= (\mu \circ (\delta^r_e)^{-1})(|\gamma|_V)) \\
&= \mu((\delta^r_e)^{-1}(|\gamma|_V)) \\
&= \mu(|\gamma^r_e|_V) \text{ by result above from Lemma 9.1.3} \\
&= [\![Pr(\gamma^r_e)]\!]_\rho\langle\mu\rangle
\end{aligned}$$

This can be applied to probabilistic terms and probabilistic formulas in a straightforward manner. □

Lemma 9.1.5. *Let* μ *be a probabilistic state,* q *be a constant real value in the range* $[0,1]$*,* r *be an SPL register and* $\mu' = q \cdot \mu \circ (\delta^r_{true})^{-1} + (1-q) \cdot \mu \circ (\delta^r_{false})^{-1}$*. Then,*

$$[\![Pr(\gamma)]\!]_\rho\langle\mu'\rangle = q \cdot [\![Pr(\gamma^r_{true})]\!]_\rho\langle\mu\rangle + (1-q) \cdot [\![Pr(\gamma^r_{false})]\!]_\rho\langle\mu\rangle$$

This can be extended to all probabilistic terms p

$$[\![p]\!]_\rho\langle\mu'\rangle = [\![toss(r,q;p)]\!]_\rho\langle\mu\rangle$$

and probabilistic formulas η

$$[\![\eta]\!]_\rho\langle\mu'\rangle \textit{ iff } [\![toss(r,q;\eta)]\!]_\rho\langle\mu\rangle$$

Proof. Trivial consequence of Lemma 9.1.4. □

Lemma 9.1.6. *Given classical formulas* γ *and* γ'*, probabilistic state* μ *and a logical assignment* ρ

$$[\![Pr(\gamma')/\gamma]\!]_\rho\langle\mu\rangle = [\![Pr(\gamma')]\!]_\rho\langle\mu_\gamma\rangle$$

This can be naturally extended to probabilistic terms and formulas.

Proof. By definition,

$$[\![Pr(\gamma')/\gamma]\!]_\rho\langle\mu\rangle = \mu(|\gamma' \wedge \gamma|_\mathcal{V}) = \mu(|\gamma'|_\mathcal{V} \cap |\gamma|_\mathcal{V}) = \mu_\gamma(|\gamma'|_\mathcal{V}) = [\![Pr(\gamma')]\!]_\rho\langle\mu_\gamma\rangle$$

□

Theorem 9.1.7. *If* $\vdash \{P\}\ \mathfrak{p}\ \{Q\}$*, then* $\vDash \{P\}\ \mathfrak{p}\ \{Q\}$*, if* $\mathfrak{p}$ *does not contain conditional branch instructions.*

Recall that $\vDash \{P\}\ \mathfrak{p}\ \{Q\}$ *means given EPPL assertions* P, Q*, SPL program* $\mathfrak{p}$*, logical assignment* ρ *and probabilistic states* μ, μ'*,*

$$\textit{If } [\![P]\!]_\rho\langle\mu\rangle \textit{ and } [\![\mathfrak{p}]\!]_\rho\langle\mu\rangle \rightarrow^*_{MW} \langle\mu'\rangle\textit{, then } [\![Q]\!]_\rho\langle\mu'\rangle$$

Proof. A proof of $\vdash \{P\}\ \mathfrak{p}\ \{Q\}$ must end with the application of the body rule. Thus we know that $\eta_1 = P$, $\eta_{|\mathfrak{p}|} = Q$ and for $l = 1, \ldots, |\mathfrak{p}| - 1$, we know that

$$I_l \rhd \{\eta_l\}$$

Let ρ be a logical variable assignment and let μ and μ' be probabilistic states such that $[\![P]\!]_\rho\langle\mu\rangle$ and $[\![\mathfrak{p}]\!]_\rho\langle\mu\rangle \rightarrow^*_{MW} \langle\mu'\rangle$. We must show that $[\![Q]\!]_\rho\langle\mu'\rangle$.

By the definition of $\rightarrow^*_{MW}$, we know that there exist $\mu_1, \ldots, \mu_{|\mathfrak{p}|}$ such that $\mu_1 = \mu$, $\mu_{|\mathfrak{p}|} = \mu'$ and for $l = 1, \ldots, |\mathfrak{p}| - 1$,

$$[\![I_l]\!]\langle\mu_l\rangle \rightarrow_{MW} \langle\mu_{l+1}\rangle$$

We show by cases on instruction I_l that if $I_l \rhd \{\eta_l\}$, $[\![\eta_l]\!]_\rho\langle\mu_l\rangle$ and $[\![I_l]\!]\langle\mu_l\rangle \rightarrow_{MW} \langle\mu_{l+1}\rangle$, then $[\![\eta_{l+1}]\!]_\rho\langle\mu_{l+1}\rangle$. When we instantiate l with $|\mathfrak{p}| - 1$, we get $[\![\eta_{|\mathfrak{p}|}]\!]_\rho\langle\mu_{|\mathfrak{p}|}\rangle$, which is $[\![Q]\!]_\rho\langle\mu'\rangle$, our desired result.

Register Loading Instructions

Case **IMove r1 r2**

By the definition of $\rightarrow_{MW}$, $\mu_{l+1} = \mu_l \circ (\delta^{r_1}_{r_2})^{-1}$.

We know $[\![\eta_l]\!]_\rho\langle\mu_l\rangle$ and we must show $[\![\eta_{l+1}]\!]_\rho\langle\mu_l \circ (\delta^{r_1}_{r_2})^{-1}\rangle$

By the **IMove** rule, we know: $\eta_l \supset \eta_{l+1}{}^{r_1}_{r_2}$

Thus $[\![\eta_l]\!]_\rho\langle\mu_l\rangle$ implies $[\![\eta_{l+1}{}^{r_1}_{r_2}]\!]_\rho\langle\mu_l\rangle$

By Lemma 9.1.4, we know that $[\![\eta_{l+1}{}^{r_1}_{r_2}]\!]_\rho\langle\mu_l\rangle = [\![\eta_{l+1}]\!]_\rho\langle\mu_l \circ (\delta^{r_1}_{r_2})^{-1}\rangle$.

Case: **ILoadK r1 k**

By the definition of $\rightarrow_{MW}$, $\mu_{l+1} = \mu_l \circ (\delta^{r_1}_{K(k)})^{-1}$.

We know $[\![\eta_l]\!]_\rho\langle\mu_l\rangle$ and we must show $[\![\eta_{l+1}]\!]_\rho\langle\mu_l \circ (\delta^{r_1}_{K(k)})^{-1}\rangle$

By the **ILoadK** rule, we know: $\eta_l \supset \eta_{l+1}{}^{r_1}_{K(k)}$

Thus $[\![\eta_l]\!]_\rho\langle\mu_l\rangle$ implies $[\![\eta_{l+1}{}^{r_1}_{K(k)}]\!]_\rho\langle\mu_l\rangle$

By Lemma 9.1.4, we know that $[\![\eta_{l+1}{}^{r_1}_{K(k)}]\!]_\rho\langle\mu_l\rangle = [\![\eta_{l+1}]\!]_\rho\langle\mu_l \circ (\delta^{r_1}_{K(k)})^{-1}\rangle$.

Case: **IGetGlobal r1 k**

By the definition of $\rightarrow_{MW}$, $\mu_{l+1} = \mu_l \circ (\delta^{r_1}_{Glb(K(k))})^{-1}$.

We know $[\![\eta_l]\!]_\rho\langle\mu_l\rangle$ and we must show $[\![\eta_{l+1}]\!]_\rho\langle\mu_l \circ (\delta^{r_1}_{Glb(K(k))})^{-1}\rangle$

By the **IGetGlobal** rule, we know: $\eta_l \supset \eta_{l+1}{}^{r_1}_{Glb(K(k))}$

Thus $[\![\eta_l]\!]_\rho\langle\mu_l\rangle$ implies $[\![\eta_{l+1}{}^{r_1}_{Glb(K(k))}]\!]_\rho\langle\mu_l\rangle$

By Lemma 9.1.4, we know that $[\![\eta_{l+1}{}^{r_1}_{Glb(K(k))}]\!]_\rho\langle\mu_l\rangle = [\![\eta_{l+1}]\!]_\rho\langle\mu_l \circ (\delta^{r_1}_{Glb(K(k))})^{-1}\rangle$.

As we can see, the register loading instructions are similar in structure; the only difference between the instructions is the valuation function used to determine the loaded value.

Arithmetic Instructions

Case: **IAdd r1 r2 r3**

By the definition of $\rightarrow_{MW}$, $\mu_{l+1} = \mu_l \circ (\delta^{r_1}_{r_2+r_3})^{-1}$.

We know $[\![\eta_l]\!]_\rho\langle\mu_l\rangle$ and we must show $[\![\eta_{l+1}]\!]_\rho\langle\mu_l \circ (\delta^{r_1}_{r_2+r_3})^{-1}\rangle$

By the **IAdd** rule, we know: $\eta_l \supset \eta_{l+1}{}^{r_1}_{r_2+r_3}$

Thus $[\![\eta_l]\!]_\rho\langle\mu_l\rangle$ implies $[\![\eta_{l+1}{}^{r_1}_{r_2+r_3}]\!]_\rho\langle\mu_l\rangle$

By Lemma 9.1.4, we know that $[\![\eta_{l+1}{}^{r_1}_{r_2+r_3}]\!]_\rho\langle\mu_l\rangle = [\![\eta_{l+1}]\!]_\rho\langle\mu_l \circ (\delta^{r_1}_{r_2+r_3})^{-1}\rangle$.

The rest of the arithmetic instructions follow similarly.

Unconditional Branching

Case: **IJmp n**

By the definition of $\to_{MW}$, $\mu_{l+1} = njmp(\mu_l, n)$.

We know $[\![\eta_l]\!]_\rho\langle\mu_l\rangle$ and we must show $[\![\eta_{l+1}]\!]_\rho\langle njmp(\mu_l, n)\rangle$

By the **IJmp** rule, we know: $\eta_l \supset \eta_{l+n}$

Thus $[\![\eta_l]\!]_\rho\langle\mu_l\rangle$ implies $[\![\eta_{l+n}]\!]_\rho\langle\mu_l\rangle$

Since assertions cannot reference the program counter, $[\![\eta_{l+n}]\!]_\rho\langle njmp(\mu_l, n)\rangle$

Case: **INop**

By the definition of $\to_{MW}$, $\mu_{l+1} = \mu_l$.

We know $[\![\eta_l]\!]_\rho\langle\mu_l\rangle$ and we must show $[\![\eta_{l+1}]\!]_\rho\langle\mu_l\rangle$

By the **IJmp** rule, we know: $\eta_l \supset \eta_{l+1}$

Thus $[\![\eta_l]\!]_\rho\langle\mu_l\rangle$ implies $[\![\eta_{l+1}]\!]_\rho\langle\mu_l\rangle$

Probabilistic instructions

Case: **IToss r q**

By the definition of $\to_{MW}$, $\mu_{l+1} = q \cdot \mu \circ (\delta^r_{true})^{-1} + (1-q) \cdot \mu \circ (\delta^r_{false})^{-1}$.

We know $[\![\eta_l]\!]_\rho\langle\mu_l\rangle$ and we must show $[\![\eta_{l+1}]\!]_\rho\langle q \cdot \mu \circ (\delta^r_{true})^{-1} + (1-q) \cdot \mu \circ (\delta^r_{false})^{-1}\rangle$

By the **IToss** rule, we know: $\eta_l \supset \mathrm{toss}(\mathrm{r}, q; \eta_{l+1})$

Thus $[\![\eta_l]\!]_\rho\langle\mu_l\rangle$ implies $[\![\mathrm{toss}(\mathrm{r}, q; \eta_{l+1})]\!]_\rho\langle\mu_l\rangle$.

By Lemma 9.1.5, $[\![\mathrm{toss}(\mathrm{r}, q; \eta_{l+1})]\!]_\rho\langle\mu_l\rangle =$

$$[\![\eta_{l+1}]\!]_\rho\langle q \cdot \mu \circ (\delta^r_{true})^{-1} + (1-q) \cdot \mu \circ (\delta^r_{false})^{-1}\rangle$$

□

9.1.2 Proof of conditional branching programs

By Theorem 9.1.7, we know that programs that do not contain conditional branch instructions meet the soundness property. We will now investigate programs that include conditional branch instructions.

The conditional branch instructions are the most interesting from a soundness perspective. These instructions are the only mechanism by which a probabilistic program state can result in a new probabilistic state where the program counter registers differ; however, recall from Section 7.4 that the probabilistic states will eventually result (via the transitive closure of relations) in a state where the program counter registers agree.

Therefore, we prove that the EPPL program assertions are met in the two successor states when the probabilistic states are restricted by the updated program counter values.

Lemma 9.1.8. *If* $\vdash \{P\}\ \mathfrak{p}\ \{y' + y'' = Pr(\gamma_0)\}$ *and* $I_1 =$ ***ILt r1 r2*** *then*

$$\vDash \{P\}\ \mathfrak{p}\ \{y' + y'' = Pr(\gamma_0)\}$$

Proof. Given an SPL program $\mathfrak{p}$ with first instruction $I_1 =$ **ILt r1 r2**, traces t', t'' that correspond to the two successor paths after the execution of **ILt** that do not contain conditional branch instructions, classical assertion $\gamma = (r_1 < r_2)$,

1. We know that $\vdash \{P\}\ \mathfrak{p}\ \{y' + y'' = \Pr(\gamma_0)\}$ is provable by the body rule.

2. Therefore, by the body rule, **ILt r1 r2** $\triangleright \{P\}$

3. By the **ILt** rule, we know there exist $\mathcal{E}'$ and $\mathcal{E}''$ such that η_1 and η_2 are the first formulas in $\mathcal{E}'$ and $\mathcal{E}''$, respectively, and $\vdash\ \{\eta'\}\ t'\ \{y' = \Pr(\gamma_0)\}$, $\vdash \{\eta''\}\ t''\ \{y'' = \Pr(\gamma_0)\}$, and $P \supset \eta'/\gamma \wedge \eta''/\neg\gamma$

4. By Theorem 9.1.7, $\vDash \{\eta'\}\ t'\ \{y' = \Pr(\gamma_0)\}$ and $\vDash \{\eta''\}\ t''\ \{y'' = \Pr(\gamma_0)\}$

5. Assume we begin in a state μ such that $[[P]]_\rho\langle\mu\rangle$. We know from 3 that $[[\eta'/\gamma \wedge \eta''/\neg\gamma]]_\rho\langle\mu\rangle$.

6. $[[\eta'/\gamma \wedge \eta''/\neg\gamma]]_\rho\langle\mu\rangle$ means that $[[\eta'/\gamma]]_\rho\langle\mu\rangle$ and $[[\eta''/\neg\gamma]]_\rho\langle\mu\rangle$.

7. By Lemma 9.1.6, $[[\eta'/\gamma]]_\rho\langle\mu\rangle = [[\eta']]_\rho\langle\mu_\gamma\rangle$; similarly $[[\eta''/\neg\gamma]]_\rho\langle\mu\rangle = [[\eta'']]_\rho\langle\mu_{\neg\gamma}\rangle$.

8. By the definition of $[[\mathbf{ILt\ r1\ r2}]]\langle\mu\rangle \rightarrow_{MW} \langle\mu_1\rangle$, we know that $\mu_1 = \text{njmp}(\mu_\gamma, 1) + \text{njmp}(\mu_{\neg\gamma}, 2)$. By the definition of traces, $\mathfrak{p}$ executed from the state $\text{njmp}(\mu_\gamma, 1)$ is equivalent to the program t', executed in the initial state μ_γ; similarly, $\mathfrak{p}$ executed from the state $\text{njmp}(\mu_{\neg\gamma}, 2)$ is t'' executed in the initial state $\mu_{\neg\gamma}$.

9. By the definition of $\vDash$ and 4 above, we know that if $[[\eta']]_\rho\langle\mu_\gamma\rangle$ and $[[t']]_\rho\langle\mu_\gamma\rangle \rightarrow^*_{MW} \langle\mu'\rangle$, then $[[y' = \text{Pr}(\gamma_0)]]_\rho\langle\mu'\rangle$. Similarly, we know that if $[[\eta'']]_\rho\langle\mu_{\neg\gamma}\rangle$ and $[[t'']]_\rho\langle\mu_{\neg\gamma}\rangle \rightarrow^*_{MW} \langle\mu''\rangle$, then $[[y'' = \text{Pr}(\gamma_0)]]_\rho\langle\mu''\rangle$.

10. From 9, the definition of the semantics of EPPL formulas and the definition of the extent of a formula in a set of valuations (defined just before Table 4), we know that $[[y' = \text{Pr}(\gamma_0)]]_\rho\langle\mu'\rangle$ has the meaning $\rho(y') = \mu'(|\gamma_0|_\mathcal{V})$; similarly, $[[y'' = \text{Pr}(\gamma_0)]]_\rho\langle\mu''\rangle$ means $\rho(y'') = \mu''(|\gamma_0|_\mathcal{V})$.

11. Let $\mu^{\text{final}} = \mu' + \mu''$. We now show that $[[y' + y'' = \text{Pr}(\gamma_0)]]_\rho\langle\mu^{\text{final}}\rangle$.

$$\begin{aligned}
[[\text{Pr}(\gamma_0)]]_\rho\langle\mu^{\text{final}}\rangle &= \mu^{\text{final}}(|\gamma_0|_\mathcal{V}) && \text{by Def.n of } [[.]] \text{ and extent} \\
&= \mu'(|\gamma_0|_\mathcal{V}) + \mu''(|\gamma_0|_\mathcal{V}) && \text{by Def.n of } \mu^{\text{final}} \\
&= \rho(y') + \rho(y'') && \text{by 10} \\
&= \rho(y' + y'') && \text{by Def.n of } \rho \text{ and } + \\
&= [[y' + y'']]_\rho\langle\mu^{\text{final}}\rangle && \text{by Def.n of } [[.]]
\end{aligned}$$

We have shown that $[[y' + y'' = \text{Pr}(\gamma_0)]]_\rho\langle\mu^{\text{final}}\rangle$ when $[[P]]_\rho\langle\mu\rangle$ and $[[\mathfrak{p}]]_\rho\langle\mu\rangle \rightarrow^*_{MW} \langle\mu^{\text{final}}\rangle$, as needed.

□

Theorem 9.1.9. *If* $\vdash \{P\}\ \mathfrak{p}\ \{Q\}$*, then* $\vDash \{P\}\ \mathfrak{p}\ \{Q\}$*, for programs* $\mathfrak{p}$ *containing instructions of any valid type.*

Proof. (Sketch)

We proceed by strong induction on the number of conditional branch instructions in a program. The base case of no conditional branch instructions follows from Theorem 9.1.7.

Our inductive hypothesis is, if $\vdash \{P'\}\ t\ \{Q'\}$ for EPPL assertions P' and Q' and the program t has n or fewer conditional branch instructions, then $\vDash \{P'\}\ t\ \{Q'\}$. We will assume without loss of generality that the $n+1^{th}$ conditional branch instruction is the first instruction in the program. The traces of the program that result from the conditional branch will have $\leq n$ conditional branch instructions. By the **ILt** rule, we can assume that $\vdash \{\eta'\}\ t'\ \{Q\}$ and $\vdash \{\eta''\}\ t''\ \{Q\}$; by the induction hypothesis, we can then conclude that $\vDash \{\eta'\}\ t'\ \{Q\}$ and $\vDash \{\eta''\}\ t''\ \{Q\}$.

The result can be shown using proof similar to Lemma 9.1.8, except that the inductive hypothesis is used instead of Theorem 9.1.7. □

Notice that by appealing to Lemma 9.1.8, we restrict the postcondition assertion Q to be of the form $y' + y'' = \Pr(\gamma_0)$. This may seem overly restrictive; we show in the next section that we can recover full generality in a straightforward manner.

9.2 Proof of Soundness of Weakest Preconditions

We now consider the soundness of the assertions generated by the weakest precondition function $\text{wp}^1_{\mathfrak{p}}$. Much of the proof will be identical to the proof described in [10], and so parts of the proof will be simplified. The soundness of $\text{wp}^1_{\mathfrak{p}}$ for non-conditional branching instructions follows directly from the soundness proof above, and so will be elided. Additionally, we will show the result for the **ILt** instruction; the **ILe** and **ITest** instructions are proved similarly.

We first consider the special case where the program post-condition assertion η is of the form $y = p$, for some logical variable y and a probabilistic term p.

Lemma 9.2.1. *Given an SPL program $\mathfrak{p}$, logical variable y, probabilistic term p,*

$$wp^*(\textbf{\textit{ILt r1 r2}};\mathfrak{p}, y = p) = \{y = p'\}$$

and

$$\vdash \{wp^*(\textbf{\textit{ILt r1 r2}};\mathfrak{p}, y = p)\}\ \textbf{\textit{ILt r1 r2}};\mathfrak{p}\ \{y = p\}$$

for some probabilistic term p', where $wp^*(\mathfrak{p},\eta)$ is the repeated application of the $wp^1_{\mathfrak{p}}$ function over the program $\mathfrak{p}$ and η is $\eta_{|\mathfrak{p}|}$.

Proof. We assume that for non-branching programs, $\vdash \{\mathrm{wp}^*(\mathfrak{p}, y = p')\}\ \mathfrak{p}\ \{y = p\}$ and $\vdash \mathrm{wp}^*(\mathbf{ILt\ r1\ r2}; \mathfrak{p}, y = p'') = \{y = p\}$, as it is a trivial consequence of Lemma's 9.1.2, 9.1.4, 9.1.3 and 9.1.5.

To prove the corresponding lemma for conditional branching programs, we proceed by induction on the structure of p. We will prove the results for the **ILt** instruction; the other conditional branching instructions are proved similarly.

- For constants and logical variables, the result follows trivially.
- For p of the form $\Pr(\gamma_0)$, we can assume by the inductive hypothesis that $\vdash \{y_1 = p_1\}\ t_1\ \{y_1 = \Pr(\gamma_0)\}$ and $\vdash \{y_2 = p_2\}\ t_2\ \{y_2 = \Pr(\gamma_0)\}$, where t_1 and t_2 are the traces after the execution of **ILt** for the true and false cases, respectively.

  By the proof rule for **ILt**, we know

  $$\{y_1 = p_1/(r_1 < r_2) \wedge y_2 = p_2/\neg(r_1 < r_2)\}\ \mathbf{ILt\ r1\ r2}; \mathfrak{p}\ \{y_1 + y_2 = \Pr(\gamma_0)\}$$

  Let $\eta^\dagger = (y = y_1 + y_2) \wedge ((y_1 = p_1)/(r_1 < r_2)) \wedge ((y_2 = p_2)/\neg(r_1 < r_2))$. Since $\eta^\dagger \supset (y_1 = p_1/(r_1 < r_2) \wedge y_2 = p_2/\neg(r_1 < r_2))$, by the CONS rule, we know that $\vdash \{\eta^\dagger\}\ \mathbf{ILt\ r1\ r2}; \mathfrak{p}\ \{y_1 + y_2 = \Pr(\gamma_0)\}$.

  Also, $\eta^\dagger \supset (y = y_1 + y_2)$, and $\vdash \{y = y_1 + y_2\}\ \mathbf{ILt\ r1\ r2}; \mathfrak{p}\ \{y = y_1 + y_2\}$, since analytic formulas are unchanged by program execution. By the And rule, we know

  $$\vdash \{\eta^\dagger\}\ \mathbf{ILt\ r1\ r2}; \mathfrak{p}\ \{y_1 + y_2 = \Pr(\gamma_0) \wedge (y = y_1 + y_2)\}$$

  We can now replace the logical variables y_1 and y_2 in $\eta^\dagger$ and the postcondition assertion, giving (*)

  $$\vdash \{y = p_1/(r_1 < r_2) + p_2/\neg(r_1 < r_2)\}\ \mathbf{ILt\ r1\ r2}; \mathfrak{p}\ \{y = \Pr(\gamma_0)\}$$

  By the definition of $\mathrm{wp}^1_{\mathfrak{p}}$,

  $$\begin{aligned} &\mathrm{wp}^1_{\mathfrak{p}}(\mathbf{ILt\ r1\ r2}, (\{y = p_1\}, \{y = p_2\})) \\ &= \mathrm{merge}(\{(y = p_1)/(r_1 < r_2)\}, \{(y = p_2)/\neg(r_1 < r_2)\}) \\ &= \{y = p_1/(r_1 < r_2) + p_2/\neg(r_1 < r_2)\} \end{aligned}$$

From (*) and the previous equation, we have our desired result, with $p' = p_1/(r_1 < r_2) + p_2/\neg(r_1 < r_2)$.

The proof for p of the form $p_1 + p_2$ and $p_1 \cdot p_2$ is similar.

□

Theorem 9.2.2. *For any SPL program* $\mathfrak{p}$ *and any conditional-free EPPL formula* η*,*

$$\vdash \{wp^*(\mathfrak{p}, \eta)\}\ \mathfrak{p}\ \{\eta\}$$

Proof. The structure of the proof of this theorem is identical to Theorem 6.6 in [10], but appeals to the equivalent lemmas that have been defined above.

Outline: The postcondition assertion $\eta_{|\mathfrak{p}|}$ is re-written as the conjunction of a set of assertions $y_i = p_i$, logically AND'ed with the conjunction of the equivalent comparison operations over the logical variables y_i. Repeated applications of the CONS, Pr-FREE and ELIMV rules provide our desired result for assertions of arbitrary form. □

9.2.1 Discussion of Completeness

Completeness states that if $\vDash \{P\}\ \mathfrak{p}\ \{Q\}$, then $\vdash \{P\}\ \mathfrak{p}\ \{Q\}$; a program that begins in a probabilistic state μ that satisfies P and ends in a probabilistic state μ' that satisfies Q will have a corresponding proof of correctness in our logic. We can use Theorem 9.2.2 to show our Hoare logic is complete. Similar to the approach in [10], we can show that our logic is complete by appealing to the completeness of EPPL (proven in that same paper). The proof is identical to Theorem 6.7 in [10], appealing to the lemmas and theorems found in this thesis where appropriate.

It can be easily shown that $\vDash \{\eta'\}\ \mathfrak{p}\ \{\eta\}$ iff $\vDash_{\text{EPPL}} \eta' \supset \text{wp}^*(\mathfrak{p}, \eta)$ (corresponding to Chadha's Corollary 6.4) for an SPL program $\mathfrak{p}$, where $\vDash_{\text{EPPL}}$ refers to the semantics of EPPL formulas in [10].

Theorem 9.2.3. *For any SPL program* $\mathfrak{p}$ *and any conditional free formula* η*,* $\vdash \{wp^*(\mathfrak{p}, \eta)\}\ \mathfrak{p}\ \{\eta\}$.

Proof. Let $p_1, p_2, \ldots, p_n$ be all the comparison terms occurring in η. Pick n distinct logical variables $y_1, y_2, ..., y_n$ that do not occur in η. Let $p'_1, p'_2, \ldots, p'_n$ be the terms $\text{wp}^*(\mathfrak{p}, p_1), \text{wp}^*(\mathfrak{p}, p_2), \ldots, \text{wp}^*(\mathfrak{p}, p_n)$ respectively and let $\eta^\dagger$ be the formula obtained

from η by replacing each occurrence of a comparison formula $(p_i \leq p_j)$ by $(y_i \leq y_j)$. Finally, take

$$\eta_a \equiv \eta^\dagger \wedge (\bigwedge_i (y_i = p_i)) \text{ and } \eta_b \equiv \eta^\dagger \wedge (\bigwedge_i (y_i = p_i')).$$

Clearly, the following hold:

- $\eta^\dagger$ is an analytical EPPL formula;
- ${\eta^\dagger}^{y_1 y_2 \ldots y_n}_{p_1 p_2 \ldots p_n}$ is η;
- $(\eta_a \supset \eta^\dagger)$ and $(\eta_b \supset \eta^\dagger)$ are EPPL theorems;
- $(\eta_b \supset (y = p_i'))$ are EPPL theorems for all $1 \leq i \leq n$;
- $\mathrm{wp}^*(\mathfrak{p}, \eta)$ is ${\eta^\dagger}^{y_1 y_2 \ldots y_n}_{p_1' p_2' \ldots p_n'}$

By axiom Pr-FREE, $\vdash \{\eta^\dagger\}\ \mathfrak{p}\ \{\eta^\dagger\}$; by Lemma 9.2.1 (Lemma 6.5 in [10]), $\vdash \{y_i = p_i'\}\ \mathfrak{p}\ \{y = p_i\}$ for all $1 \leq i \leq n$. Since $(\eta_b \supset \eta^\dagger)$ and $(\eta_b \supset (y = p_i'))$ are EPPL theorems for all $1 \leq i \leq n$, by application of CONS it follows that

$$\vdash \{\eta_b\}\ \mathfrak{p}\ \{\eta^\dagger\} \text{ and } \vdash \{\eta_b\}\ \mathfrak{p}\ \{y = p_i\}$$

for all $1 \leq i \leq n$. Several applications of the inference rule AND then give $\vdash \{\eta_b\}\ \mathfrak{p}\ \{\eta_a\}$; since $(\eta_a \supset \eta)$ is an EPPL theorem, another application of CONS yields $\vdash \{\eta_b\}\ \mathfrak{p}\ \{\eta\}$. Finally, several applications of ELIMV show that

$$\vdash \{{\eta^\dagger}^{y_1 y_2 \ldots y_n}_{p_1' p_2' \ldots p_n'}\}\ \mathfrak{p}\ \{\eta\}$$

□

Chadha's proof of completeness can now be stated for our logic as:

Theorem 9.2.4. *If* $\vDash \{\eta'\}\ \mathfrak{p}\ \{\eta\}$*, then there is a proof in our logic such that* $\vdash \{\eta'\}\ \mathfrak{p}\ \{\eta\}$*.*

Proof. Suppose that $\vDash \{\eta'\}\ \mathfrak{p}\ \{\eta\}$. By Corollary 6.4 in [10], $\vDash_{\mathrm{EPPL}} (\eta' \supset \mathrm{wp}^*(\mathfrak{p}, \eta))$. By completeness of EPPL, $\vdash (\eta' \supset \mathrm{wp}^*(\mathfrak{p}, \eta))$. Theorem 9.2.3 (Theorem 6.6 in [10]) implies that $\vdash \{\mathrm{wp}^*(\mathfrak{p}, \eta)\}\ \mathfrak{p}\ \{\eta\}$, whence $\vdash \{\eta'\}\ \mathfrak{p}\ \{\eta\}$ by CONS. □

10 Implementation

The implementation of the Probabilistic Proof-Carrying Code system depends upon 4 inter-related components. These components are all designed in support of the broad goal of a verifiable mobile code fragment; however, only 2 of the components are *trusted components*. A trusted component is a subsystem that must be accepted as a base of trust between the agents cooperating in the mobile code system. Such components should remain small and easily verifiable, as the security of the larger system depends upon their correctness.

I implemented these four subsystems to support PPCC:

1. A compiler from Simplified Lua to the Simplified Probabilistic Lua VM

   this includes a component to insert probabilistic instructions to model errors

2. A virtual machine that interprets a SPL program, under the Copenhagen-style semantics

3. A Verification Condition generator (or, VCGen)

   Coq is used as the prover subsystem

4. An automatic proof verifier

The Coq system is used by the code producer to prove the verification conditions and the code consumer to verify the proof.

The final two items in the list (the VCGen and proof verifier) are the trusted components of the Probabilistic Proof-Carrying Code system. Proof-carrying code is meant to provide a guarantee that the transfer and subsequent loading of the mobile code is safe and correct.

10.1 Compiler

The first subsystem of PPCC is a compiler, that accepts programs written in Simplified Lua and outputs a sequence of Simplified Probabilistic Lua instructions. This compiler is based upon the standard Lua compiler tool, but rewritten in the Haskell programming language.

Haskell was chosen as the implementation language due to the ease of handling structured data via pattern matching and type inferencing. The code is structured in an imperative style (using Haskell's State monad).

The compiler does not include a parser to take a Simplified Lua program written in plain text format to an abstract syntax tree. This decision was made to reduce the time of implementation. Instead, programs must be directly input as pre-parsed AST's. For example, the simple program

Example 10.1: Sample Program 1

```
local a = 5
local b = -55
return 44 + b
```

will be encoded as

Example 10.2: Sample Program 1, encoded

```
sample1 = [
  Assign 'a' (Value (LuaNum 5)),
  Assign 'b' (Value (LuaNum (-55))),
  Return [PrimitiveApp LuaPlus (Value (LuaNum 44)) (Var 'b')]
  ]
```

The lack of a parser is not a major limitation, as it simplifies the compiler and reduces the need for defining precedence and dealing with ambiguous grammars.

Unlike Chadha's probabilistic programming language [10], the Simplified Lua programming language does not include probabilistic statements. Programs are written with the assumption that a program will execute deterministically.

To model probabilistic programs, a utility program was written which we will refer to as the Error-Inserter-Program (EIP) that operates on the output of the compiler

to insert probabilistic instructions as appropriate. For instance, to model singular bit-flips errors when loading data from the global memory store, the EIP translates every instance of an **IGetGlobal** instruction with the $\oplus_p$ pseudo-instruction. The choice of the probabilistic parameter p must be chosen by the user of the EIP utility; higher values of p correspond with higher error rates.

Given the program (to calculate interest on a banking savings account):

Example 10.3: Sample Program 1, encoded

```
IGetGlobal r1 'savingsRate'
IGetGlobal r2 'accountBalance'
IMul r3 r1 r2
```

Given an error rate of 0.001, the program will be rewritten to:

Example 10.4: Sample Program 1, encoded

```
IGetGlobal r1 'savingsRate'
IToss r9 0.001
ITest r9
IXor r1 r1 0x00100

IGetGlobal r2 'accountBalance'
IToss r9 0.001
ITest r9
IXor r2 r2 0x01000

IMul r3 r1 r2
```

The choice of the bit-flip mask (in the above example, 0x01000 and 0x00100 are hexadecimal representations of bit-flip masks) is made randomly during the rewrite process. A simple modification could include all possible bit-flip masks (0x1, 0x2, 0x3, 0x4, . . .); a bit position can be toggled by combining a bit-flip mask with a value via the **IXor** operation. However, for the purposes of this work, the existence of a single bit-flip is sufficient.

10.2 Virtual Machine

The virtual machine component is also written in Haskell, and implements the Copenhagen-style operational semantics of the Simplified Probabilistic Lua bytecode language. This virtual machine interprets a sequence of instructions, and will use a pseudo-random number generator to simulate the probabilistic choice operator. The standard Lua v5.0 virtual machine is taken as a reference implementation.

The input to the virtual machine is a list of SPL instructions, which is considered a program. A simple validation procedure is executed to validate that the program meets the validation criteria (only forward jumps, terminating with an **IReturn** instruction, etc). The virtual machine implements the Copenhagen semantics described in Section 7.2.

The execution of the machine operates over a global state vector, which consists of:

- the register state
- the global memory state
- the program's list of instructions
- the constant pool
- the program counter

The virtual machine implements the register state as a list containing elements of type LuaValue, which is a disjoint union of the valid Lua types (Numbers, Strings, Booleans, etc). The global memory state is a mapping from a String to a LuaValue; the string refers to the global variable identifier, and the LuaValue is the associated data item stored in the variable. Global variables are used to represent the parameters a program accepts; these global variables can have a probability distribution.

To execute a program, the virtual machine constructs an initial state. The registers are set to a special invalid value, which will cause an error to occur if an operation acts upon it. The global memory are set to the user-chosen input values. If required,

these values can be set using a probability distribution, to simulate the expected distribution of values of normal input.

The initial state also includes a pseudo-random number generator seed, which is used to provide the values of the probabilistic choice instruction IToss. This seed can be explicitly set, which assists in replaying executions. The random seed can also be chosen based upon a non-deterministic source, such as the current time or the contents of `/dev/random`.

Finally, the initial program counter value is assigned a value of 0, and the interpretation process begins. The interpreter is written in a *Fetch-Decode-Process-Loop* style. The instruction in the list at the program counter position is retrieved, and the instruction's parameters are determined. The program counter is then incremented as an implicit step of the instruction processing step; the branching instructions can update the program counter value further, but must assume that the program counter now points to the next instruction.

The processing step implements the required operational step described by the Copenhagen semantics of Table 8. The **IToss** instruction is handled by querying the pseudo-random number generator for a result, which is used to determine the outcome of the biased coin toss. To simplify the implementation, a probabilistic choice meta-instruction is also included, which performs the coin toss and executes an instruction (denoted the *left* or the *right* instructions), without requiring a series of **IToss** and conditional jumps to perform the same task. This simplifies the error-modelling utility component, but adds to the complexity of the virtual machine.

Haskell's lazy-by-default semantics caused issues with efficiency. The naive implementation of the virtual machine creates thunks (stub functions that compute a value upon request) during the process of interpretation, which are invoked once the output of the computation is required. For short computations, this poses no difficulty. However, for longer computations, the thunks begin to populate memory and can affect running times. To address this, the virtual machine enforces a strict evaluation semantics for the interpretation process.

To investigate the probabilistic properties of the virtual machine, we performed a statistical sampling. The virtual machine was run multiple times, and the end result

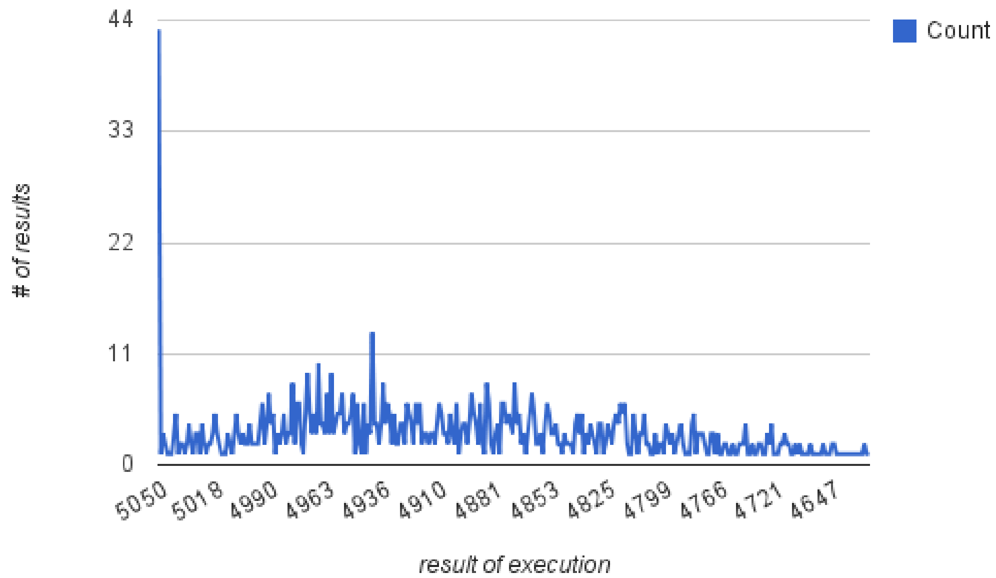

Figure 1: Sample distribution of results

of the computation is sampled. Each invocation uses a unique pseudo-random seed value, to better model randomness. The results of the cumulative calculations can be plotted to visualize the probabilistic distribution of outputs.

Figure 1 shows a sample distribution of results when sampling the faulty adder example program. The faulty adder computes the sum of the integers from 1..100, and therefore we expect the sum to be the value $5050 = \sum_{i=1}^{100} i$. However, an integer may be skipped with a small probability, which results in a sum less than the correct value. In this chart we see that the correct sum of 5050 is the end result of 43 of 1000 executions. This chart is not meant to represent full experimental results; rather, it shows how the Copenhagen machine approximates the Many-Worlds semantics.

10.3 Verification Condition Generator

The verification conditioner generator is the core of the proof-carrying code system, and is the first trusted component described in this work. The verification condition

generator (or VCGen) is a unique component, as it is required by both the code producer and the code consumer.

VCGen accepts two input values: an SPL program consisting of a sequence of SPL bytecode instructions and a constant pool, and a program post-condition consisting of an EPPL predicate. The post-condition predicate describes the required outcome upon the termination of a SPL program execution. The output will be a set of proof obligations, which must be discharged by the code consumer before the mobile code can be sent to the consumer. The proof obligations will be encoded in a Coq-compatible format, which is used by the code producer to prove the correctness.

Communication must occur between the code consumer and the producer for the success of verification condition generation. The choice of the program pre- and post-conditions is decided by the code consumer, as the onus is upon them to ensure safety of their computing platform. The code producer must be informed of the conditions required for execution before discharging the proof obligations; if the conditions differ between the producer and consumer, then the proof of correctness will likely be invalid.

This communication of conditions is taken for granted in this work. An encoding of pre- and post-conditions into a readable format could be published on a Web site, or described as part of a financial transaction between the consumer and the producer.

The verification condition generator implements the $\mathrm{wp}^1_{\mathfrak{p}}$ function described in section 8, with some simplifying properties. The bytecode instructions are divided into classes with similar proof rule schemas.

- Binary arithmetic : **IAdd**, **ISub**, etc.
- Conditional branching : **ILt**, **ITest**, etc.
- Memory loads : **ILoadK**, **ILoadBool**, **IGetGlobal**

Working from the last instruction, the proof rules are read "backwards", working from one (or more) instruction post-conditions to an instruction pre-condition. The pre-condition of a successor instruction becomes the post-condition of the predecessor instruction; the program's pre-condition is the pre-condition of the first instruction

in the program. In the case of conditional branch instructions, the post-conditions of the instruction are the pre-conditions of the target instructions.

10.4 Proof Verifier

The second trusted component is used solely by the code consumer to ensure the safety of the mobile code before loading. The verification condition generator is used by the code consumer to generate the proof obligations of the mobile code provided by the code producer. However, rather than performing the time-intensive task of proving the correctness, the consumer must simply verify the provided proof is a legitimate proof of safety.

In this work, the proof verifier is the Coq [43] automated theorem prover. The input to the proof verification component is the proof obligation for verifying correctness, plus the producer's proof of same, written as a Coq-readable proof. The output is a boolean value; True if the proof is accepted, False otherwise. The determination of the reason for the proof's lack of acceptance is outside the scope of this work.

The axioms of the EPPL logic are encoded into Coq, and are made available to the code consumer for use in proofs; see Appendix B for the complete code listing. These axioms (and axiom schemas) are embedded into the Coq logic, the calculus of inductive constructions. Unlike the semantics of the Copenhagen virtual machine, there is no need to model randomness for probabilistic choices. The logic is deterministic, operating over distributions of valuations.

11 Final Remarks

11.1 Conclusions

A proof-carrying code system for a probabilistic bytecode language has been developed, using the Exogenous Probabilistic Proposition Logic presented by Chadha et al [10] as the logic of assertions. The proof rules for the instructions of the Simplified Probabilistic Lua bytecode language have been proven sound and complete with respect to the "Many-World" semantics given in Subsection 7.1.2.

A correspondence has been proven between the two semantics given in this work, which shows the applicability of the proof rules to practical computing systems that exhibit probabilistic behaviour. This proof is sketched in Section 7.5.

Without unbounded iteration, Simplified Probabilistic Lua programs have the property of strong probabilistic termination. This simplifies our proofs, as well as simplifies the instruction set.

Finally, Section 7.3 describes how probabilistic errors that occur in classical computing hardware can be modelled using the probabilistic toss instruction with additional code insertion, allowing behaviour such as random memory corruption or control flow errors to be investigated. Probabilistic proof-carrying code can ensure that the probability of erroneous outcomes is a low chance occurrence.

11.2 Summary of Contributions

1. I have derived a set of sound and complete proof rules for a bytecode language that includes probabilistic instructions explicitly, limited branching and arithmetic operations. These proof rules form the basis of a weakest precondition function, suitable for determining the weakest precondition necessary for a given post-condition to be met upon termination of a program.

2. I developed a suite of tools to illustrate the probabilistic proof-carrying code

system, including a verification condition generator, a probabilistic virtual machine, and a compiler from Simplified Lua to the Simplified Probabilistic Lua bytecode language.

3. Finally, I created a number of program transformation rules to model common probabilistic errors found in classical computing systems. These transformations allow code producers to determine the probability of certain erroneous outputs given the existence of probabilistic behaviour.

11.3 Future Work

The simple programming language described in this work has a number of limitations that reduce its utility; a lack of procedures, no unbounded iteration, no data structuring operations, etc.

Handling procedures via replacement of invocations with the body of the procedure should lead to a similar proof result as Bannwart-Mueller [4]. However, the additional properties of Lua closures may require some specialized axioms to handle the various cases.

There has been some work done on proof correctness for probabilistic programming languages that include unbounded termination [13, 38]. These approaches should be translatable to the SPL bytecode Hoare-style logic rules to allow for proving correctness of programs with possible unbound termination. Another aspect of loops worth further investigation is the impact of weakly terminating probabilistic programs.

The Simplified Probabilistic Lua language does not include data structuring instructions, such as **IGetTable** and **IPutTable**. These instructions could also be handled with proof rules, which would provide a full Hoare logic style logic for a probabilistic Lua virtual machine.

The proof rules, suitably extended with support for procedures, loops and data structuring operations, could be applied to other bytecode languages, including the Java Virtual Machine and Microsoft's Common Language Runtime. Such a task would increase the mainstream appeal of the proof-carrying code system.

The proof rules allow for simplification of the classical formulas. The naive implementation of the verification condition generator can result in very large proof obligations, which can be trivially reduced. For example, the composition of an **IToss** instruction and an **ITest** conditional branch instruction may result in probability sub-measures that are trivially false and thus ignorable. Similarly, simple properties should be automatically provable by the proof system, with or without hints from the compiler. To aid the proving of more complicated properties, the introduction of a typing system that can express probabilistic relationships would be an interesting endeavour.

Trusting the entire Coq automatic prover greatly increases the size of the trusted components, though the checker is only a small part of the full Coq system. Even so, the development of a proof verification subsystem with a smaller footprint would enhance the trustworthiness of the Probabilistic Proof-Carrying Code system.

Bibliography

[1] IEEE Standard for Floating-Point Arithmetic. Technical report, Microprocessor Standards Committee of the IEEE Computer Society, 3 Park Avenue, New York, NY 10016-5997, USA, August 2008.

[2] Dana Angluin. Local and global properties in networks of processors (extended abstract). In *Proceedings of the twelfth annual ACM symposium on Theory of computing*, STOC '80, pages 82–93, New York, NY, USA, 1980. ACM.

[3] Andrew W. Appel and Amy P. Felty. A semantic model of types and machine instructions for proof-carrying code. In *Proceedings of the 27th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '00, pages 243–253, New York, NY, USA, 2000. ACM.

[4] F. Y. Bannwart and P. Müller. A logic for bytecode. Technical Report 469, ETH Zurich, 2004.

[5] Fabian Bannwart and Peter Müller. A program logic for bytecode. *Electr. Notes Theor. Comput. Sci.*, 141(1):255–273, 2005.

[6] Hagai Bar-El, Hamid Choukri, David Naccache, Michael Tunstall, and Claire Whelan. The sorcerers apprentice guide to fault attacks. 2004.

[7] Adriano Barenco, Charles H. Bennett, Richard Cleve, David P. DiVincenzo, Norman Margolus, Peter Shor, Tycho Sleator, John A. Smolin, and Harald Weinfurter. Elementary gates for quantum computation. *Phys. Rev. A*, 52(5):3457–3467, Nov 1995.

[8] Andrew Bernard and Peter Lee. Temporal logic for proof-carrying code. In *Conference on Automated Deduction*, pages 31–46, 2002.

[9] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*. Springer, 2004.

[10] Rohit Chadha, Luís Cruz-Filipe, Paulo Mateus, and Amílcar Sernadas. Reasoning about probabilistic sequential programs. *Theor. Comput. Sci.*, 379(1-2):142–165, 2007.

[11] Stephen A. Cook. Soundness and completeness of an axiom system for program verification. *SIAM J. Comput.*, 7(1):70–90, 1978.

[12] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51:107–113, January 2008.

[13] Jerry den Hartog and Erik P. de Vink. Verifying probabilistic programs using a Hoare like logic. *Int. J. Found. Comput. Sci.*, 13(3):315–340, 2002.

[14] D. Deutsch. Quantum Theory, the Church-Turing Principle and the Universal Quantum Computer. *Proc. Royal Society, Series A*, 400:97–117, 1985.

[15] Brian W. DeVries, Gopal Gupta, Kevin W. Hamlen, Scott Moore, and Meera Sridhar. Actionscript bytecode verification with co-logic programming. In *Proceedings of the ACM SIGPLAN Fourth Workshop on Programming Languages and Analysis for Security*, PLAS '09, pages 9–15, New York, NY, USA, 2009. ACM.

[16] Edsger W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Commun. ACM*, 17:643–644, November 1974.

[17] Richard ER Feynman. Simulating physics with computers. *International Journal of Theoretical Physics*, 21(6):467–488, June 1982.

[18] J. Garrett. Ajax: A new approach to web applications. `http://www.adaptivepath.com/ideas/ajax-new-approach-web-applications`, 2005. [Online; accessed 02-September-2011].

[19] J.-Y. Girard. Linear logic. *Theoretical Computer Science* **50** *(1)*, pages 1–102, 1987.

[20] Inc. Google. Google App Engine. `http://appengine.google.com/`, 2011. [Online; accessed 01-September-2011].

[21] Sudhakar Govindavajhala and Andrew W. Appel. Using memory errors to attack a virtual machine. In *Proceedings of the 2003 IEEE Symposium on Security and Privacy*, SP '03, pages 154–165, Washington, DC, USA, 2003. IEEE Computer Society.

[22] Lov K. Grover. A fast quantum mechanical algorithm for database search. In *Annual ACM Symposium ON THEORY OF COMPUTING*, pages 212–219. ACM, 1996.

[23] Lisa Higham and Steven Myers. Self-stabilizing token circulation on anonymous message passing rings (extended abstract). 1998.

[24] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 26:53–56, January 1983.

[25] Roberto Ierusalimschy, Luiz Henrique de Figueiredo, and Waldemar Celes Filho. The evolution of Lua. In *HOPL'07*, pages 1–26, 2007.

[26] Adobe Systems Incorporated. Flash player statistics. `http://www.adobe.com/products/player_census/flashplayer/`, 2011. [Online; accessed 01-September-2011].

[27] Alon Itai and Michael Rodeh. Symmetry breaking in distributed networks. *Inf. Comput.*, 88(1):60–87, 1990.

[28] Manfred Jaeger. Complex probabilistic modeling with recursive relational bayesian networks. *Annals of Mathematics and Artificial Intelligence*, 32(1-4):179–220, August 2001.

[29] Claire Jones. *Probabilistic non-determinism.* PhD thesis, Edinburgh, Scotland, UK, 1989. UMI Order No. GAXDX-94930.

[30] Phillip Kaye, Raymond Laflamme, and Michele Mosca. *An Introduction to Quantum Computing.* Oxford University Press, Inc., New York, NY, USA, 2007.

[31] Dexter Kozen. Semantics of probabilistic programs. *J. Comput. Syst. Sci.*, 22(3):328–350, 1981.

[32] Dexter Kozen and Jerzy Tiuryn. On the completeness of propositional Hoare logic. *Inf. Sci.*, 139(3-4):187–195, 2001.

[33] Gérard Le Lann. Distributed systems - towards a formal approach. In *IFIP Congress*, pages 155–160, 1977.

[34] Amazon Web Services LLC. Amazon Web Services. `http://aws.amazon.com/`, 2011. [Online; accessed 01-September-2011].

[35] PUC-Rio Lua Team. Lua Programming Language. `http://www.lua.org/`, 2011. [Online; accessed 01-September-2011].

[36] Kein-Hong Man. A no-frills introduction to Lua 5.1 VM instructions. 2006.

[37] Annabelle McIver and Carroll Morgan. Demonic, angelic and unbounded probabilistic choices in sequential programs. *Acta Inf.*, 37:329–354, January 2001.

[38] Annabelle McIver and Carroll Morgan. *A probabilistic approach to information hiding*, pages 441–460. Springer-Verlag New York, Inc., New York, NY, USA, 2003.

[39] George C. Necula. Proof-carrying code. In *Principles of Programming Languages*, pages 106–119, 1997.

[40] George C. Necula. *Compiling with Proofs.* PhD thesis, 1998.

[41] Liem Ngo and Peter Haddawy. Probabilistic logic programming and bayesian networks. In *In Asian Computing Science Conference*, volume 1023, pages 286–300, 1995.

[42] James Oberg. Why the Mars probe went off course. *IEEE Spectr.*, 36:34–39, December 1999.

[43] Christine Paulin-Mohring. Inductive definitions in the system Coq; rules and properties. In M. Bezem and J. F. Groote, editors, *Proceedings of the International Conference on Typed Lambda Calculi and Applications*, volume 664, pages 328–345. Springer Verlag Lecture Notes in Computer Science, 1993.

[44] Roger Penrose. *The Road to Reality: A Complete Guide to the Laws of the Universe*. Vintage, January 2007.

[45] Eduardo Pinheiro, Wolf-Dietrich Weber, and Luiz A. Barroso. Failure trends in a large disk drive population. In *FAST'07: Proceedings of the 5th conference on USENIX Conference on File and Storage Technologies*, page 2, Berkeley, CA, USA, 2007. USENIX Association.

[46] Gordon D. Plotkin. A structural approach to operational semantics. *J. Log. Algebr. Program.*, 60-61:17–139, 2004.

[47] Nicola Santoro. *Design and Analysis of Distributed Algorithms (Wiley Series on Parallel and Distributed Computing)*. Wiley-Interscience, 2006.

[48] Bianca Schroeder, Eduardo Pinheiro, and Wolf-Dietrich Weber. DRAM errors in the wild: a large-scale field study. In *Proceedings of the eleventh international joint conference on Measurement and modeling of computer systems*, SIGMETRICS '09, pages 193–204, New York, NY, USA, 2009. ACM.

[49] Michael D. Schroeder and Jerome H. Saltzer. A hardware architecture for implementing protection rings. *Commun. ACM*, 15:157–170, March 1972.

[50] Dana S. Scott. Data types as lattices. *SIAM J. Comput.*, pages 522–587, 1976.

[51] Peter W. Shor. Algorithms for quantum computation: Discrete logarithms and factoring. In *Proc. 35th Symp. Found. Comp. Sci.*, page 124, 1994.

[52] G Swift. A brief history of memories in space from a see (single event effects) perspective. 2000.

[53] Ahmed Y. Tawfik and Eric Neufeld. Temporal bayesian networks. In *Proceedings of the TIME-94 – International Workshop on Temporal Representation and Reasoning*, pages 85–92, 1994.

[54] Tommaso Toffoli. Reversible computing. In *International Colloquium on Automata, Languages and Programming*, pages 632–644, 1980.

[55] Yasuyuki Tsukada. Interactive and probabilistic proof of mobile code safety. *Automated Software Engg.*, 12:237–257, April 2005.

A Sample Programs

A.1 Program 1: Parity Check

This program computes a simple parity check for a 4-bit plus parity bit input value. The post-condition is simple as well: the parity check value must be 0 with greater than 99% probability.

This program computes the parity of a given (integer) number by using the >> and & binary operators. The $a >> n$ operator performs the bitwise right shift operation; given a number a, >> will shift the bits of the binary representation n units rightward, resulting in a new number a'. For instance, assume $a = 10$, which has binary representation $b1010$. $a >> 1$ would shift the bits one unit rightward, resulting in $b101 = 5$. Similarly, the & and ôperators perform bitwise logical AND and XOR between two numbers, respectively. By applying $a\&1$ to a number a, we can determine the value of the least-significant bit of the binary representation. In the case where $a = 10$, $a\&1 = 0$.

The parity program iteratively computes the bitwise exclusive-OR operation between all the bits of the input number, and returns that value (either 0 or 1) as its output.

Example A.1: Parity Check

```
local a = inputValue  -- read from a global variable, 'inputValue'
local parity = 0

parity = parity ^ (a & 1)
a = a >> 1
parity = parity ^ (a & 1)
a = a >> 1
parity = parity ^ (a & 1)
a = a >> 1
parity = parity ^ (a & 1)
```

```
a = a >> 1
parity = parity ^ (a & 1)
a = a >> 1

return parity
```

The program is compiled to the Simplified Probabilistic Lua bytecode language, and probabilistic toss instructions are inserted to model bit flip errors. A probabilistic coin is tossed with bias 0.01, and if the coin lands heads, a bit is set in the `r0` register using the XOR operation. The choice of the value 8 is arbitrary. A second probabilistic coin with bias 0.001 is tossed, and if it lands heads, a second bit flip error is emulated, by flipping a second arbitrary (but unique) bit via XOR. Then, parity is calculated by a combination of Boolean bit-wise AND, XOR and SHR, which will result in the parity stored in register `r1`.

Example A.2: Parity Check

```
1  -- local a = inputValue
2  IGetGlobal r0, 'inputValue'
3
4  -- Toss a coin, and with probability 0.01,
5  --  flip a bit in the inputValue
6  IToss r2 0.01
7  ITest r2 true
8  IXor r0 r0 0x8
9  IXor r0 r0 0x8
10
11 IToss r2 0.001
12 ITest r2 true
13 IXor r0 r0 0x2
14 IXor r0 r0 0x2
15
16
17 -- local parity = 0
18 ILoadK r1 K(0)
19
20 -- parity = parity ^ (a & 1)
21 IAnd r2 r0 K(1)
22 IXor r1 r1 r2
23 -- a = a >> 1
24 IShr r0 r0 K(1)
25
26 -- parity = parity ^ (a & 1)
```

```
27 IAnd r2 r0 K(1)
28 IXor r1 r1 r2
29 -- a = a >> 1
30 IShr r0 r0 K(1)
31
32 -- parity = parity ^ (a & 1)
33 IAnd r2 r0 K(1)
34 IXor r1 r1 r2
35 -- a = a >> 1
36 IShr r0 r0 K(1)
37
38 -- parity = parity ^ (a & 1)
39 IAnd r2 r0 K(1)
40 IXor r1 r1 r2
41 -- a = a >> 1
42 IShr r0 r0 K(1)
43
44 -- parity = parity ^ (a & 1)
45 IAnd r2 r0 K(1)
46 IXor r1 r1 r2
47 -- a = a >> 1
48 IShr r0 r0 K(1)
49
50 -- return parity
51 IReturn r1
```

B Coq code

Example B.1: Parity Check

```
1  Require Import Reals.
2  Require Import Reals.ROrderedType.
3  Require Import Bool.
4  Require Import Bool.IfProp.
5  Require Import Lists.List.
6
7
8  Inductive numberTerm : Set := r : nat -> numberTerm
9             | rigid : R -> numberTerm
10            | c : nat -> numberTerm
11            | ntplus : numberTerm -> numberTerm -> numberTerm
12            | ntmul : numberTerm -> numberTerm -> numberTerm.
13
14 Inductive classicalTerm : Set := rigidB : bool -> classicalTerm
15            | b : nat -> classicalTerm
16            | ctlt : numberTerm -> numberTerm -> classicalTerm
17            | ff : classicalTerm
18            | ctimplies : classicalTerm -> classicalTerm -> classicalTerm
19            | cteq : numberTerm -> numberTerm -> classicalTerm.
20
21 Inductive probTerm : Set := prob : classicalTerm -> probTerm
22            | pbplus : probTerm -> probTerm -> probTerm
23            | pbmul  : probTerm -> probTerm -> probTerm
24            | pbconst : R -> probTerm.
25
26 Inductive probForm : Set := problt : probTerm -> probTerm -> probForm
27            | probimplies : probForm -> probForm -> probForm
28            | probfff : probForm
29            | probttt : probForm
30            | probeq : probTerm -> probTerm -> probForm.
```

```
31
32 Definition measure : Type := (list (nat ->R)) -> R.
33
34 Parameter bmem : nat -> bool.
35 Parameter probstate : measure.
36 Parameter registers : measure.
37
38 Variable A: Type.
39 Variable valuation_set : list (nat -> R).
40
41
42 Open Scope R_scope.
43
44 Fixpoint numberDenote (nt : numberTerm) (registers : nat -> R) : R :=
45   match nt with
46     | r reg => registers reg
47     | rigid rval => rval
48     | ntplus n1 n2 => (numberDenote n1 registers) + numberDenote n2 registers
49     | ntmul n1 n2 => numberDenote n1 registers * numberDenote n2 registers
50     | c val => 0%R
51   end.
52
53 Fixpoint classicalDenote ( ct : classicalTerm )
54       (registers : nat -> R) : bool :=
55   match ct with
56     | ff => false
57     | cteq n1 n2 =>  Reqb (numberDenote n1 registers)
58                           (numberDenote n2 registers)
59     | rigidB b1 => b1
60     | ctimplies c1 c2 =>  orb (negb (classicalDenote c1 registers))
61                                   (classicalDenote c2 registers)
62     | b mem => bmem mem
63     | ctlt n1 n2 => match Rcompare (numberDenote n1 registers)
64                              (numberDenote n2 registers) with
65         |Lt => true
66         |_ => false
67       end
```

```
68   end.
69
70 Definition extent (probstate : list (nat -> R) )
71                   (ct : classicalTerm) : list (nat -> R) :=
72                      filter (classicalDenote ct) probstate.
73
74 Fixpoint probTDenote (p : probTerm ) (registers_measure : measure) : R :=
75   match p with
76     | prob ct => registers_measure (extent valuation_set ct)
77     | pbplus p1 p2 => (probTDenote p1  registers_measure) +
78                        (probTDenote p2  registers_measure)
79     | pbmul p1 p2  => (probTDenote p1  registers_measure) *
80                        (probTDenote p2  registers_measure)
81     | pbconst rval => rval
82   end.
83
84
85 Fixpoint probFDenote ( p : probForm )  (registers : measure) : Prop :=
86   match p with
87     | probfff => False
88     | probttt => True
89     | probimplies p1 p2 => (probFDenote p1 registers) -> (probFDenote p2
   registers)
90     | problt p1 p2 => probTDenote p1  registers < probTDenote p2  registers
91     | probeq p1 p2 => probTDenote p1  registers = probTDenote p2  registers
92   end.
93
94 Axiom Meas0 : (probFDenote (probeq (prob ff) (pbconst 0) ) registers) = True .
95
96 Axiom PMP: forall (n1: probForm), forall (n2: probForm),
97    (probFDenote (probimplies n1 n2)  registers) = True /\ (probFDenote n1
   registers) = True
98       ->    (probFDenote n2  registers) = True.
99
100 Axiom total_measure:   registers valuation_set = 1.
101
102 Axiom filter_fine: forall f: (nat->R)->bool, forall v: list (nat->R),
```

```
103      (f = ( fun _:(nat->R) => true)) -> (v = filter f v ).
```

C Sample proof of probabilistic correctness

In this chapter, we will demonstrate a simple proof of correctness, utilizing the soundness theorem to ensure that our program will meet a postcondition assertion over all executions.

We begin with the following simple SPL program:

Example C.1: Parity Check

```
1  ILoadK r1 1
2  ILoadK r2 2
3
4  IAdd r4 r1 r2
5
6  IToss r3 0.8
7
8  ITest r3 True
9  ISub r4 r1 r4
10 IAdd r4 r1 r4
11
12 IReturn r4
```

This simple program includes arithmetic, probabilistic behaviour and a conditional branch over a probabilistic state.

The EPPL postcondition assertion that must be met (denoted Q) will be $\Pr(r_4 > 3) > 0.5$.

Working backwards, the final instruction is **IReturn**, so the weakest precondition of this instruction is the postcondition itself.

The next instruction is **IAdd r4 r1 r4**. By the definition of $\mathrm{wp}^1_{\mathfrak{p}}(\textbf{IAdd r4 r1 r4}, Q)$, the resulting assertion is

$$\Pr((r_1 + r_4) > 3) > 0.5$$

Notice that this instruction is the target of a conditional branch (the **ITest** instruction), but no special handling of that fact is required. However, we will need to refer to this assertion shortly, so we will denote it η_{heads}.

Continuing, the assertion for the **ISub r4 r1 r4** instruction results in

$$\Pr((r_1 + (r_1 - r_4)) > 3) > 0.5$$

We denote this assertion η_{tails}.

The weakest precondition assertion for the conditional branch instruction ITest r3 True refers to η_{heads} and η_{tails}. Via $\text{wp}^1_{\mathfrak{p}}(\textbf{ITest r3 True}, (\eta_{\text{heads}}, \eta_{\text{tails}}))$, we generate:

$$\text{merge}(\eta_{\text{heads}}/(r_3 = True), \eta_{\text{tails}}/(r_3 \neq True))$$

This merged assertion is

$$\Pr((r_1 + r_4) > 3 \wedge r_3 = True) + \Pr((r_1 + (r_1 - r_4)) > 3 \wedge r_3 \neq True) > 0.5$$

We will denote this assertion as η_{ITest}

The probabilistic toss instruction **IToss r3 0.8** generates the precondition $\text{toss}(r_3, 0.8; \eta_{\text{ITest}})$. By the definition of $\text{toss}(r, q; \eta)$, this is the assertion

$$\begin{aligned} & 0.8 \cdot \Pr((r_1 + r_4) > 3 \wedge True = True) \\ + & 0.2 \cdot \Pr((r_1 + r_4) > 3 \wedge False = True) \\ + & 0.8 \cdot \Pr((r_1 + (r_1 - r_4)) > 3 \wedge True \neq True) \\ + & 0.2 \cdot \Pr((r_1 + (r_1 - r_4)) > 3 \wedge False \neq True) > 0.5 \end{aligned}$$

In the interests of brevity, we will simplify this assertion; such simplification cannot be done in general, as the merge function requires that simplification does not occur while generating weakest preconditions.

Note that the probability of True $\neq$ T is 0 (and similarly for False). Therefore, the assertion simplifies to:

$$0.8 \cdot \Pr((r_1 + r_4) > 3) \;+\; 0.2 \cdot \Pr((r_1 + (r_1 - r_4)) > 3) > 0.5$$

We continue the process of generating the weakest precondition for **IAdd r4 r1 r2**, generating

$$0.8 \cdot \Pr((r_1 + r_1 + r_2) > 3) \; + \; 0.2 \cdot \Pr((r_1 + (r_1 - (r_1 + r_2))) > 3) > 0.5$$

Finally, we combine the generation of the two remaining **ILoadK** instructions:

$$\begin{aligned} & 0.8 \cdot \Pr((1 + 1 + 2) > 3) \; + \; 0.2 \cdot \Pr((1 + (1 - (1 + 2)) > 3) > 0.5 \\ = \; & 0.8 \cdot \Pr(4 > 3) \; + \; 0.2 \cdot \Pr(-1 > 3) > 0.5 \\ = \; & 0.8 \cdot 1 + 0.2 \cdot 0 > 0.5 \end{aligned}$$

This precondition reduces to *True*, so the the program will always meet the post-condition, regardless of the register assignments of the initial state.