Technical Report

UCAM-CL-TR-566
ISSN 1476-2986

Number 566

UNIVERSITY OF CAMBRIDGE

Computer Laboratory

Formal verification of probabilistic algorithms

Joe Hurd

May 2003

15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500

http://www.cl.cam.ac.uk/

This technical report is based on a dissertation submitted December 2001 by the author for the degree of Doctor of Philosophy to the University of Cambridge, Trinity College.

Technical reports published by the University of Cambridge Computer Laboratory are freely available via the Internet:

http://www.cl.cam.ac.uk/TechReports/

Series editor: Markus Kuhn

ISSN 1476-2986

Abstract

This thesis shows how probabilistic algorithms can be formally verified using a mechanical theorem prover.

We begin with an extensive foundational development of probability, creating a higher-order logic formalization of mathematical measure theory. This allows the definition of the probability space we use to model a random bit generator, which informally is a stream of coin-flips, or technically an infinite sequence of IID Bernoulli($\frac{1}{2}$) random variables.

Probabilistic programs are modelled using the state-transformer monad familiar from functional programming, where the random bit generator is passed around in the computation. Functions remove random bits from the generator to perform their calculation, and then pass back the changed random bit generator with the result.

Our probability space modelling the random bit generator allows us to give precise probabilistic specifications of such programs, and then verify them in the theorem prover.

We also develop technical support designed to expedite verification: probabilistic quantifiers; a compositional property subsuming measurability and independence; a probabilistic while loop together with a formal concept of termination with probability 1. We also introduce a technique for reducing properties of a probabilistic while loop to properties of programs that are guaranteed to terminate: these can then be established using induction and standard methods of program correctness.

We demonstrate the formal framework with some example probabilistic programs: sampling algorithms for four probability distributions; some optimal procedures for generating dice rolls from coin flips; the symmetric simple random walk. In addition, we verify the Miller-Rabin primality test, a well-known and commercially used probabilistic algorithm. Our fundamental perspective allows us to define a version with strong properties, which we can execute in the logic to prove compositeness of numbers.

Contents

1 Introduction **11**
- 1.1 History of Formalization . . . 11
- 1.2 Introduction to Theorem Provers . . . 12
- 1.3 Formal Methods and Probability . . . 13
- 1.4 Formalizing Probabilistic Programs . . . 14
- 1.5 Example Probabilistic Programs . . . 17
- 1.6 The Miller-Rabin Primality Test . . . 18
- 1.7 Automatic Proof Tools . . . 18
- 1.8 How to Read this Thesis . . . 20

2 Formalized Probability Theory **23**
- 2.1 Introduction . . . 23
  - 2.1.1 The Need for Measure Theory . . . 23
  - 2.1.2 How to Create a Measure . . . 25
- 2.2 Measure Theory . . . 26
  - 2.2.1 Measure Spaces . . . 27
  - 2.2.2 Carathéodory's Extension Theorem . . . 30
  - 2.2.3 Functions between Measure Spaces . . . 31
  - 2.2.4 Probability Spaces . . . 33
- 2.3 Bernoulli($\frac{1}{2}$) Sequences: Algebra . . . 34
  - 2.3.1 Infinite Sequence Theory . . . 35
  - 2.3.2 The Algebra Generated by Prefix Sets . . . 36
  - 2.3.3 Canonical Forms . . . 37
  - 2.3.4 Properties of $(\mathcal{A}, \mu)$ . . . 39
- 2.4 Bernoulli($\frac{1}{2}$) Sequences: Probability Space . . . 40
  - 2.4.1 The Need for σ-algebras . . . 40
  - 2.4.2 Definition of the Probability Space . . . 41
  - 2.4.3 Construction of a Non-Measurable Set . . . 43
  - 2.4.4 Probabilistic Quantifiers . . . 44
- 2.5 Concluding Remarks . . . 46

3 Verifying Probabilistic Algorithms **49**
- 3.1 Introduction . . . 49
  - 3.1.1 Motivating Probabilistic Algorithms . . . 49
  - 3.1.2 Verifying Probabilistic Algorithms in Practice . . . 51
  - 3.1.3 A Notation for Probabilistic Programs . . . 53

3.1.4 Probabilistic Termination . . . 54
3.2 Measurability and Independence . . . 55
3.2.1 Measurability . . . 55
3.2.2 Function Independence . . . 57
3.2.3 Strong Function Independence . . . 58
3.3 Probabilistic Termination . . . 61
3.3.1 Probabilistic 'While' Loops . . . 61
3.3.2 Probabilistic Termination Conditions . . . 64
3.3.3 Proof Techniques for Probabilistic While Loops . . . 66
3.3.4 Probabilistic 'Until' Loops . . . 68
3.4 Execution in the Logic of Probabilistic Programs . . . 69
3.4.1 Introduction to Execution in the Logic . . . 69
3.4.2 Formalizing a Pseudo-random Bit Sequence . . . 70
3.4.3 Execution as an Automatic Proof Procedure . . . 71
3.5 Concluding Remarks . . . 72

4 Example Probabilistic Programs 75
4.1 Introduction . . . 75
4.2 The Binomial($n, \frac{1}{2}$) Distribution . . . 77
4.3 The Uniform(n) Distribution . . . 78
4.4 The Geometric($\frac{1}{2}$) Distribution . . . 81
4.5 The Bernoulli(p) Distribution . . . 82
4.6 Optimal Dice . . . 83
4.7 The Symmetric Simple Random Walk . . . 87
4.8 Concluding Remarks . . . 88

5 Verification of the Miller-Rabin Primality Test 91
5.1 Introduction . . . 91
5.1.1 The Miller-Rabin Probabilistic Primality Test . . . 91
5.1.2 The HOL Verification . . . 92
5.2 Computational Number Theory . . . 93
5.2.1 Definitions . . . 93
5.2.2 Underlying Mathematics . . . 94
5.2.3 Formalization . . . 95
5.3 Probability Theory . . . 98
5.3.1 Guaranteed Termination . . . 98
5.3.2 Definition of the Miller-Rabin Test . . . 99
5.3.3 A Compositeness Prover . . . 100
5.4 Extraction to Standard ML . . . 100
5.4.1 Random Bits . . . 101
5.4.2 Arbitrarily Large Natural Numbers . . . 102
5.4.3 Extracting from HOL to ML . . . 102
5.4.4 Testing . . . 103
5.5 Concluding Remarks . . . 105

6 Summary 107
6.1 Future Work . . . 108

A Higher-order Logic and the HOL Theorem Prover **111**
A.1 Terms and Types . . . 111
A.2 Theorems . . . 112

B Predicate Set Prover **115**
B.1 Introduction . . . 115
B.1.1 An Introduction to Predicate Subtyping . . . 115
B.1.2 Simulating Predicate Subtyping in HOL . . . 116
B.2 The Formalism . . . 117
B.2.1 Subtypes . . . 117
B.2.2 Subtype Constructors . . . 118
B.2.3 Subtype Rules . . . 119
B.2.4 Subtypes of Constants . . . 120
B.2.5 Subtype Judgements . . . 121
B.2.6 Subtype Derivation Algorithm . . . 122
B.3 Subtype-checking in HOL . . . 123
B.3.1 Debugging Specifications . . . 123
B.3.2 Logical Limits . . . 124
B.4 Predicate Set Prover and Applications . . . 125
B.4.1 Predicate Set Prover . . . 125
B.4.2 Proving Conditions During Rewriting . . . 126
B.5 Concluding Remarks . . . 127

C The HOL Probability Theories (Abridged) **129**
C.1 `measure` Theory . . . 129
C.2 `probability` Theory . . . 134
C.3 `prob_dice` Theory . . . 137

D Miller-Rabin Primality Test Extracted to Standard ML **141**
D.1 HOL `miller_rabin_ml` Theory . . . 141
D.2 ML Support Modules . . . 143
D.2.1 `HolStream` Signature . . . 143
D.2.2 `RandomBits` Signature . . . 143
D.2.3 `HolNum` Signature . . . 143
D.3 ML Miller-Rabin . . . 144
D.3.1 `HolMiller.sml` Structure . . . 144

Acknowledgements

I thank Mike Gordon for his critical advice and encouragement over the whole of my Ph.D. His door was always open, and the quality of this work has benefitted enormously from his supervision. Konrad Slind also gave generously of his time to help me with hol98 and many other aspects of theorem proving: he has practically been a second Ph.D. supervisor.

The automated reasoning group in Cambridge has provided an intellectually stimulating and especially friendly climate in which to discuss research. Special mentions go to Michael Norrish for theorem proving know-how, Mark Staples for the idea of formalizing probability theory, Ken Friis Larsen for ML advice, and Daryl Stewart for many discussions. John Harrison also provided much theorem proving information by email, including specific help with my research projects. On the mathematical front, David Preiss gave me some valuable help with measure theory, Thomas Forster taught me many pieces of logic and set theory over a cup of tea, and Des Sheiham has always been happy to discuss whatever mathematics was on my mind.

This research was generously funded by the EPSRC award 98318027 and a grant from the Isaac Newton Trust. I received additional travel money to attend conferences from the University of Cambridge Computer Laboratory, Trinity College and the TPHOLs 2000 conference. Mike Holcombe's Verification and Testing group at Sheffield University Department of Computer Science kindly provided bench space and a computer account for my visits in the 2000–2001 academic year, and Alan Bundy's group at the University of Edinburgh made me feel very welcome when I gave a talk there.

Mike Gordon, Konrad Slind, Judita Preiss, Michael Norrish and Des Sheiham carefully proof-read versions of this thesis, and between them eliminated a large number of lurking mistakes.

My family have all provided a great deal of encouragement over the years, and supported me in my academic studies. Finally, I met Judita at the very beginning of my Ph.D; she has been with me through it all and without her this thesis would not have been possible.

Chapter 1

Introduction

We begin by briefly surveying the history of formalization, introducing mechanical theorem provers and showing their application to software engineering. A gap is identified in the current practice of verifying programs in a theorem prover, namely the ability to model programs that make use of a random number generator in their operation. This introduces our formalization of probability theory in higher-order logic, which we use to prove the correctness of some probabilistic algorithms in the HOL theorem prover.

1.1 History of Formalization

It is a remarkable fact that reasoning can be modelled by performing purely syntactic manipulations inside a formal language. A formal language used for this purpose is called a logic, and over the centuries many logics have been created with the purpose of capturing one class of real-world truths or another. In *Elements*, Euclid (circa 300B.C.) defined a logic allowing the rigorous formalization of a large body of geometric truths, while the propositional logic of Boole (1847) carves out a class of truths that are highly relevant to electronic circuit design. The modal logic of Lewis (1918) has been used to reason about diverse computer science properties, and more recently Burrows, Abadi, and Needham (1990) developed the BAN logic to model the beliefs of participants engaged in security protocols.

An intellectual motivation for modelling reasoning in logic is the desire for consistency. If a system of reasoning can be shown to reduce in principle to manipulations in a consistent logic, then the system of reasoning must be consistent too. It is for this reason that Hilbert's programme aimed to create a consistent logic capturing mathematical truth, providing a solid foundation for mathematics just as Euclid had for geometry.[1] This project is tackled head-on in the three volumes of *Principia Mathematica*, a monumental work written over 10 years by Whitehead and Russell (1910). A logic is defined that does not suffer from known inconsistency traps, and (in pain-staking detail) the concepts and theorems of mathematics are reduced to syntactic manipulation. Managing the morass of logical detail that formalization generates is no small feat, and indeed in his autobiography Russell (1968) said of the enterprise: "my intellect never quite recovered from the

[1]Hilbert actually wanted a consistent logic capturing *the whole* of mathematical truth, but this lofty goal turns out to be unobtainable. The incompleteness theorems of Gödel (1931) show that no consistent formal system can capture all the truths of number theory, let alone the whole of mathematics.

strain."

A second intellectual motivation to model reasoning in logic is that syntactic manipulation can perform "reasoning almost mechanically by the eye, which otherwise would call into play the higher faculties of the brain", as put by Whitehead (1911). In this respect formalization can be regarded as a novel system of mathematical notation, abstracting sophisticated reasoning steps by the shuffling of symbols written in a suitable syntax. As Whitehead goes on to say: "Civilisation advances by extending the number of important operations which can be performed without thinking about them."

1.2 Introduction to Theorem Provers

As argued by de Bruijn (1970), Trybulec and Blair (1985), Boyer et al. (1994) and many others (a cogent summary is given by Harrison (1996a)), computers can play an important role in assisting the process of formalization. Proof assistants can help with book-keeping when the logical details are overwhelming, and automatic theorem provers can perform speedy mechanical proof search in the logic when the problem has been sufficiently broken down. An interactive theorem prover implements the axioms and rules of inference of a logic, and generally comes with a proof assistant and many automatic procedures for proving different types of goals.

A design philosophy of Milner (1978) developed for the Edinburgh LCF project (Gordon, Milner, and Wadsworth, 1979) allows the creation of highly flexible theorem provers that correctly implement any logic. The distinguishing feature of theorem provers in the 'LCF style' is a small logical kernel that contains all the code that must be trusted to ensure that wrong theorems are not created. Arbitrary programs may be written to automate common patterns of reasoning, but every theorem must eventually be created from pre-existing theorems by the scrupulous logical kernel, and in this way soundness is assured.

The logical kernel defines a type containing logical terms, and an *abstract* type containing logical theorems. An abstract type may only be accessed through its interface, denying direct access to the internal representation. Terms may be constructed and destructed at will, and similarly theorems may be destructed to give their underlying terms (i.e., hypotheses and conclusion). However, there is a strict control on the creation of theorems, and only the following methods can do this:

- For every axiom of the logic, a theorem value that represents this axiom is available.
- For every rule of inference of the logic having the form

$$\frac{\vdash \phi_1 \quad \vdash \phi_2 \quad \cdots \quad \vdash \phi_n}{\vdash \phi}$$

  a function is available that takes hypothesis theorems in the required form and produces a conclusion theorem.

Since the type of theorems is abstract, a strongly typed programming language will ensure that theorems may only be created using these methods: precisely the valid theorems of the logic.

The HOL theorem prover, developed in Cambridge by Gordon (1988), continues the LCF tradition. It implements higher-order logic, extending the simple type theory of Church (1940) with Hindley-Milner polymorphism (Milner, 1978). Many proof assistants and automatic proof procedures have been developed in HOL over the years, supporting the formalization of various mathematical theories including sets, natural numbers, groups and real numbers. An international initiative called the QED project (Anonymous, 1994) aims to formalize all of mathematics in theorem provers, and work in HOL has contributed to this endeavour.

As well as mathematics, it is also possible to formalize computer science designs in the theorem prover, and then prove that the formalized design satisfies a mathematical specification. As a simple example of this, a hardware circuit may be formalized in the logic, and a theorem may be proved of the form "for any 64-bit inputs a, b to the circuit, the 128-bit output c will be the numerical product of a and b." This is formal verification, and is now used in the chip design industry to spot errors before a costly fabrication process (Colwell and Brennan, 2000). Formal verification has successfully been applied to many areas of design, including hardware circuits (Curzon, 1994), programming languages (Norrish, 1998), security protocols (Paulson, 1999) and software (Harrison, 1997).

In addition to the intellectual benefits of formalizing mathematics in logic, formal verification provides a practical motivation: to build up the mathematical language necessary to express design specifications. These same mathematical theories can then be re-used to verify that the designs satisfy their specification, a task that may require considerable effort. For example, in Chapter 5 we shall meet a function witness that takes two natural numbers and returns true or false. However, expressing the specification of witness requires a theory of sets, and the subsequent verification involves formalizing a significant amount of group theory.

1.3 Formal Methods and Probability

Theorem provers can be used to formalize software in logic and reason about it, even proving that a program satisfies a mathematical specification. An early advocate of this was Turing (1949), and since then it has become a recognized approach to software engineering called formal methods. Much research effort is currently being spent on developing the formalisms that are needed to apply formal methods to new classes of programs, such as object-oriented software (Huisman, 2001), or software distributed across a network (Vos, 2000; Wansbrough et al., 2001).

This is not exactly our type of software, so it is not important for our work

One such class is probabilistic programs, i.e., programs that make use of a random number generator in their operation. Probabilistic programs can be very hard to test using conventional techniques. As a simple example, consider the case of a program dice supposed to output a sequence of standard dice throws. Suppose the first 20 observed outputs of dice are as follows:

$$4, 4, 4, 5, 3, 2, 1, 5, 6, 4, 2, 4, 5, 3, 3, 4, 5, 6, 1, 1, \ldots$$

Is dice operating correctly? On the face of it, we might answer yes, since the output sequence certainly looks random. We can make this scientific by defining a statistic—such

as the average of the first 20 throws—and then comparing how much the given sequence differs from the expected result. We would expect an average of $3\frac{1}{2}$ from a perfect dice, but in our example the average of these 20 values from dice is $3\frac{3}{5}$: a deviation of $\frac{1}{10}$. A numerical computation reveals that 85% of all sequences that might result from a perfect dice would have a deviation of at least $\frac{1}{10}$, and so this single test gives us a confidence of 15% that the program dice is incorrect.

This kind of statistical result is an inherent limitation of black-box testing. In theory, with enough output either the confidence will increase to one or else decrease to zero, but the burden of numerical computation may impose a practical limit before an acceptable level of certainty is reached. In contrast, formal methods promise the ability to prove rigorously that the program satisfies a probabilistic specification, by reasoning directly with the program.[2] To do this, we need to be able to give a precise logical meaning to probabilistic programs—a formal semantics—so that a probabilistic program $\hat{f}$ can be represented as a logical term f. Since specifications (probabilistic or not) are usually written as logical predicates, the problem of verifying that a program $\hat{f}$ satisfies a specification P is reduced to finding a proof of the formula $P(f)$.

Formalizing the semantics of probabilistic programs is a research area that has been tackled many times, beginning with a seminal paper by Kozen (1979). However, it is not a straightforward task to pick up one of these semantics and formalize it in a theorem prover (and indeed, has not been attempted). Sometimes sophisticated mathematical theories are used in semantics of probabilistic programs (e.g., Jones (1990) extends domain theory), but the formalization of such sophisticated mathematical theories in a theorem prover is a difficult research topic in itself. Other times a fresh logic is created specifically for the task of modelling probabilistic programs (e.g., Feldman and Harel (1984) introduce probabilistic dynamic logic), but this makes it difficult to use the theories and proof tools that are available in standard theorem provers.

Therefore, our philosophy is to develop the semantics of probabilistic programs within a conservative extension of higher-order, taking the opportunity wherever possible to build upon existing work. This approach greatly speeds up our development, allowing us to take the work further than would have been possible starting from scratch. In addition, our work then becomes available to others working within higher-order logic. Since the inception of the HOL system, this kind of reciprocation has been practiced by many users. As a result, there are now many theories, formalizations and automatic proof tools available for newcomers to use in their own projects.

1.4 Formalizing Probabilistic Programs

In Chapters 2 and 3 we present a simple modelling of probabilistic programs in higher-order logic, in which we can specify and verify any program equipped with a source of random bits. In the language of probability theory, these random bits are assumed to be independent, identically distributed (IID) Bernoulli($\frac{1}{2}$) random variables (as defined, for

[2] In an interesting approach that combines both formal methods and testing, Monniaux (2001) has recently shown how abstract interpretation can be used to calculate the confidence bounds of black-box testing.

example, in DeGroot (1989, page 145)).[3] Suppose we have a probabilistic 'function'

$$\hat{f} : \alpha \to \beta$$

that takes as input an element of type α, and uses a random number generator to calculate a result of type β. Given the specification

$$B : \alpha \times \beta \to \mathbb{B}$$

for $\hat{f}$ as a predicate[4] on pairs of elements from α and β, say a particular function application $\hat{f}(a)$ satisfies the specification if $B(a, \hat{f}(a))$ is true. Of course, since $\hat{f}$ is probabilistic, the application $\hat{f}(a)$ may meet the specification on one instance and fail it on another.[5]

Our approach is to model $\hat{f}$ with a higher-order logic function[6]

$$f : \alpha \to \mathbb{B}^\infty \to \beta \times \mathbb{B}^\infty$$

which explicitly takes as input a sequence $s : \mathbb{B}^\infty$ of random bits in addition to an argument $a : \alpha$, uses some of the random bits in a calculation exactly mirroring $\hat{f}(a)$, and then passes back a sequence of 'unused' bits with the result.

Mathematical measure theory is then used to define a probability measure function

$$\mathbb{P} : \mathcal{P}(\mathbb{B}^\infty) \to [0, 1]$$

from sets of bit sequences to real numbers between 0 and 1. The natural question *"for a given a, with what probability does $\hat{f}$ satisfy the specification?"* then becomes *"for a given a, what is the probability measure of the set*

$$\{s \mid B(a, \mathsf{fst}\ (f\ a\ s))\}$$

of bit sequences?"[7]

Example: Consider the dice probabilistic program which is supposed to output a sequence of fair dice throws. As a probabilistic 'function', it has type

$$\widehat{\mathsf{dice}} : \mathbb{I} \to \mathbb{N}$$

where $\mathbb{I}$ is the 'unit' type containing only the element $()$. A sequence of dice throws is obtained by repeatedly evaluating $\widehat{\mathsf{dice}}\ ()$. Using the method above, we can formalize this probabilistic program with a higher-order logic function of type

$$\mathbb{I} \to \mathbb{B}^\infty \to \mathbb{N} \times \mathbb{B}^\infty$$

[3] Commonly, IID is introduced as a property of a *finite* collection of random variables. We extend this to an infinite sequence by insisting that every finite prefix is IID.

[4] Predicates in higher-order logic are simply functions to the booleans $\mathbb{B}$.

[5] Note that the formula $\forall a.\ B(a, \hat{f}(a))$ does eliminate any non-determinism, since for each a the application $\hat{f}(a)$ is probabilistic.

[6] Functions in higher-order logic must be deterministic, so modelling $\hat{f}$ in this way has eliminated probabilistic non-determinism.

[7] The function fst picks the first component of a pair, in this case the β from $\beta \times \mathbb{B}^\infty$.

However, the $\mathbb{I}$ is redundant in this context, and so the type we actually use is

$$\mathsf{dice} : \mathbb{B}^\infty \to \mathbb{N} \times \mathbb{B}^\infty$$

Each of the input bit sequences is an oracle that determines the value of dice. The probability that dice outputs a '3' is the probability measure of the following set of bit sequences:

$$\{s \mid \mathsf{fst}\ (\mathsf{dice}\ s) = 3\}$$

Assuming dice is implemented correctly, the probability measure of this set will be equal to $\frac{1}{6}$. Informally, dice partitions the whole space of bit sequences into 6 equally-sized pieces according to the probability measure. The result of the application dice s is then determined by the piece that contains the input bit sequence s. □

Modelling probabilistic programs with higher-order logic functions in this way has a number of advantages:

- probabilistic programs can be represented as higher-order logic functions; we do not have to formalize another programming language in which to express the programs;
- since our modelling of probabilistic functions is the same as that used in pure functional programming languages, we can both borrow their monadic notation (Wadler, 1992) to elegantly express our programs, and easily transfer programs to and from an execution environment;
- when applying the theory to the verification of probabilistic programs, we need provide formal support for only one probability space: sequences of IID Bernoulli($\frac{1}{2}$) bits.

A central concept in our model is probabilistic independence, where two events A and B are independent if

$$\mathbb{P}(A \cap B) = \mathbb{P}(A)\mathbb{P}(B)$$

Independence can be naturally lifted to probabilistic functions, and this allows us to calculate the probability of complicated results. Unfortunately it is difficult to show that a given probabilistic function is independent, because the natural property of function independence is not compositional. However, we create a compositional property of independence by strengthening the natural property, and in this way we can show that every probabilistic function constructible using our monadic constructors is automatically independent.

The formalization of probability theory and the compositional independence property provide the technology to address a tricky problem: how to represent and reason about probabilistic programs that do not necessarily terminate, but terminate with probability 1. A probabilistic version of the 'while' loop is introduced in the form of a new monadic operator that preserves the compositional independence property: at this point the monadic operators are expressive enough to construct any probabilistic program that terminates with probability 1.[8] A formalization of important probabilistic termination results from the literature completes the framework we need to specify and verify real probabilistic programs.

[8] The set {coin_flip, unit, bind, prob_while} of monadic operators has this property.

1.5 Example Probabilistic Programs

In Chapter 4 we demonstrate the application of our formal framework by verifying a selection of example probabilistic programs. First to be verified are sampling algorithms for four probability distributions, and we then move on to the symmetric simple random walk and the 'optimal dice' programs of Knuth and Yao (1976).

For our purposes, a sampling algorithm is a converter program that takes as input a sequence of Bernoulli($\frac{1}{2}$) samples and outputs a sequence of samples from another probability distribution. It is a curious fact that sampling algorithms exist for all probability distributions in the literature,[9] and so it is sufficient for us to treat as primitive only the probability space of Bernoulli($\frac{1}{2}$) sequences: all the rest can be obtained by creating an appropriate sampling algorithm.

Sampling algorithms for the well-known probability distributions tend to be small programs that each involve some interesting reasoning about probability to prove their correctness. They are thus ideal vehicles for developing the notation and proof techniques needed for practical verification of probabilistic programs. In addition, once a suite of sampling algorithms has been verified for some common distributions, they can be used as subroutines in the definition of higher-level textbook probabilistic algorithms.

Using our higher-order logic model, we formally verify sampling algorithms for the Binomial($\frac{1}{2}, n$), Uniform(n), Geometric($\frac{1}{2}$) and Bernoulli(p) probability distributions. The Geometric($\frac{1}{2}$) distribution poses special problems for formal treatments of probabilistic algorithms, since the distribution is over an infinite set. The Bernoulli(p) is also tricky, since the parameter p can be any real number in the range $[0, 1]$. However, we implement sampling algorithms for each distribution and prove them correct.

In addition to sampling algorithms we also use our framework to formalize the DDG tree[10] notation for probabilistic programs, introduced by Knuth and Yao (1976). It is demonstrated that finite DDG trees can be naturally represented using our formal framework, and two examples of DDG trees from Knuth's paper are verified. These two DDG trees are probabilistic programs that 'optimally' generate dice throws and sums of two dice throws—optimal in the sense that the expected number of Bernoulli($\frac{1}{2}$) samples that they require is minimal among all dice generating programs.

The last verification example in this chapter is the symmetric simple random walk: a probabilistic process with a compelling intuitive interpretation. A drunk starts at point n (the pub) and is trying to get to point 0 (home). Unfortunately, every step he makes from point i is equally likely to take him to point $i + 1$ as it is to take him to point $i - 1$. As every probabilist knows, the drunk will eventually get home with probability 1, and so the probabilistic program that simulates the drunk has the probabilistic termination property. The probabilistic program that simulates the random walk does not fit neatly into a standard program scheme that guarantees probabilistic termination. It is an advantage of our formal framework that definition of probabilistic programs is not tied to any particular program scheme, and so we can define the random walk, prove that it terminates, and show that it satisfies some characteristic properties.

[9] Williams (1991, page 35) puts it like this: "every experiment you will meet in this (or any other) course can be modelled via the triple $([0, 1], \mathcal{B}[0, 1], \text{Leb})$."

[10] DDG stands for Discrete Distribution Generating.

1.6 The Miller-Rabin Primality Test

In Chapter 5 we verify a classic random algorithm: the Miller-Rabin primality test. Based on a number theoretic result of Miller (1975), Rabin (1976) introduced a primality test that produces correct results with provably high probability. It is particularly relevant to modern cryptography, where large primes are frequently required for public key encryption algorithms (e.g. RSA).

Using our HOL probability theory, the primality test itself is simple to define and has a concise probabilistic specification. However, the correctness proof is fairly advanced and relies on some computational number theory that has not previously been formalized in any theorem prover. With the help of automatic proof tools—including our own predicate set prover described in Appendix B—we are able to formalize the required mathematics and verify that our Miller-Rabin implementation satisfies its probabilistic specification.

The benefits of formalization are made particularly apparent here, since the specification that we prove the implementation satisfies is stronger than the specification given in most algorithm textbooks. In fact, this extra strength allows the creation of an automatic proof procedure that uses the Miller-Rabin primality test to prove that numbers are composite without requiring knowledge of their factors. Using this, we can prove HOL theorems such as

$$\vdash \neg\mathsf{prime}(2^{2^8}+1)$$

showing that the 8th Fermat number is composite.

To highlight the software engineering benefit of this formal methods research, the verified HOL version of the Miller-Rabin test is manually extracted to the ML programming language. With some careful work designed to preserve as much as possible the context in which it was verified, it is possible to execute the program in ML. We extensively analyse the performance of this 'partially-verified version' of the Miller-Rabin primality test, and demonstrate its efficacy on numbers up to 2000 bits long.[11]

1.7 Automatic Proof Tools

The formalization of a mathematical proof into logic requires it to be reduced to axioms using only primitive rules of inference. The difficulty of this task stems not from deciding how each goal should be reduced to simpler subgoals (usually this is obvious from the mathematical proof), but rather from managing the sheer volume of logical detail that arises when reducing mathematical proofs to logic. Computer theorem provers provide a large benefit in keeping track of the logical dependencies that would otherwise have to be manually recorded, and thereby giving us confidence that our proof is valid.

However, the number of subgoals that can arise in the formalization of even a simple mathematical theorem can be overwhelming.[12] To really aid the effort, the computer must do more than correctly account for each subgoal: it must automatically prove some without requiring any interaction with the user.

[11] At time of writing, an RSA modulus of 512 bits has been factored, 1024 bits is generally used, and 2048 bits is considered prudent. Two primes each 2000 bits long give a 4000 bit RSA modulus, so our Miller-Rabin implementation serves well the current security requirements.

[12] A tactic proof in one of our theories that 2 is a prime took 10712 primitive rules of inference, though this is mainly due to the use of profligate automatic proof tools.

Though not the focus of this thesis, developing new proof procedures to speed up formalization is an integral part of our research. Improving the tools makes it possible to formalize more challenging mathematical theories, and then actually performing the formalization highlights the limitations of the existing proof procedures and stimulates the next cycle of tool-building. Both activities contribute to the theorem proving community, since theories can be built upon and tools can be applied by all other users of the theorem prover, whether they are interested in further formalization or using the theories to verify a particular design.

Two possible misconceptions about automatic proof tools may be clarified here. Firstly, adding a new proof tool can never result in false theorems. In fact, the LCF design of the HOL theorem prover actively encourages users to write new proof tools in the ML programming language, secure in the knowledge that the logical kernel will foil any attempt to violate soundness. Secondly, by using automatic proof search we are *not* saying that we are no longer interested in the proofs of theorems. There is an important distinction between what is mathematically obvious and what is logically obvious,[13] and all a bare theorem prover can do is make logically obvious steps. It is therefore necessary to create sophisticated proof tools to automatically reduce mathematically obvious steps to a string of logically obvious steps.

There is usually a trade-off between the range of goals that can be proved and the depth which can be searched before a combinatorial explosion occurs. With this in mind, we briefly survey how the main proof procedures used in modern theorem provers can help with formalization:

First-order deductive provers: These use algorithms such as resolution or model elimination to perform general first-order proof search. In practice they are extremely useful for finishing off easy goals, but are quickly swamped when confronted with deeper problems.

Conditional rewriters: These take theorems of the form

$$\forall \vec{v}.\ C(\vec{v}) \Rightarrow (A(\vec{v}) = B(\vec{v}))$$

and rewrite instances of A in the goal to the corresponding B. For each instance, the condition C must be proved before the rewrite can take place (usually by a decision procedure or by a recursive call to the conditional rewriter). These tools are used to simplify goals, and their main failing is that the condition prover is too weak in some situations (leading to helpful conditional rewrites never being used), and too aggressive in others (leading to unacceptable performance).

Decision Procedures: There are many classes of formula for which there exists an automatic procedure that will always succeed in either proving or refuting the goal. In our formalization work we have extensively used decision procedures for Presburger arithmetic: this is the class of number[14] formulas that include addition, subtraction and comparison operators (i.e., no multiplications). As well as directly solving

[13] As pointed out by John McCarthy.

[14] In HOL there are decision procedures for Presburger arithmetic over natural numbers, integers and reals.

many subgoals, they can also be used in the middle of a complicated proof to prove a lemma such as

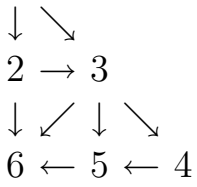
$$\vdash \quad \forall a, b, c, d.\ (a = b) \wedge (c = d) \Rightarrow (a + c = b + d)$$

that can be immediately used to reduce the current goal. The main limitation of decision procedures is that most goals do not naturally fall into a decidable class, and must first be reduced using other tools.

Appendix B describes an automatic proof tool we developed to support the verification of the Miller-Rabin primality test. It works by simulating predicate subtyping—a feature present in the logic of the PVS theorem prover (Owre et al., 1999)—by using HOL sets. The proof tool is able to tackle a class of 'compositional properties' that frequently occur as conditions of conditional rewrites, and was found to be particularly useful on problems of group theory.

1.8 How to Read this Thesis

In the interest of easy browsing, some effort has been made to keep different chapters as independent as possible, but inevitably some concepts are built upon as the work progresses. Figure 1.1 suggests some reading paths through the thesis.



Chapter	Topic
1	Introduction
2	Formalized Probability Theory
3	Verifying Probabilistic Algorithms
4	Example Probabilistic Programs
5	The Miller-Rabin Primality Test
6	Summary

Figure 1.1: Suggested Reading Paths Through This Thesis.

A brief introduction to higher-order logic and the HOL theorem prover is given in Appendix A. This primer explains the types, terms, theorems and rules of inference of higher-order logic, and their implementation in the HOL theorem prover. For more details the interested reader is referred to Gordon and Melham (1993). In the following list we limit ourselves to explaining a few symbols that might not be familiar to everyone, and set out the notation conventions we use throughout the thesis:

- **Theorems:** Formulas proved in a theorem prover are referred to as theorems, and always displayed with a $\vdash_{system}$ prefix, where the *system* is the theorem prover used. A bare $\vdash$ means the same as $\vdash_{HOL}$.

- **Types:** Examples of HOL types are: $\mathbb{B} = \{\top, \bot\}$ (booleans); $\mathbb{N} = \{0, 1, 2, \ldots\}$ (natural numbers); $\mathbb{R}$ (real numbers); α, β (type variables); $\alpha \to \beta$ (function spaces); $\alpha \times \beta$ (pairs); α^* (lists); and $\mathcal{P}(\alpha)$ (sets).

- **Constants:** HOL constants are displayed in standard mathematical notation when possible (e.g., $+$, $*$, mod, $\forall$ and $\in$), or sans serif when not (e.g., prime, image, group and cyclic). Note that higher-order logic is expressive enough to define as constants both functions (such as fst and snd which allow us to pick the first and second component of a pair) and also mathematical operators (such as $\circ$ and funpow which denote function composition and function power respectively).

- **λ-Calculus:** The simply typed λ-calculus of Church (1940) is the term language of higher-order logic. Therefore, many HOL definitions have much in common with functional programs, and various programming-language constructs such as if ... then ... else have been defined in higher-order logic. In addition, the HOL function definition package (called TFL) aids the definition of total recursive functions.

- **Lists:** There are many list operations defined as higher-order logic constants. The list constructors are [] and cons, but note that cons 1 [] may also be written as $1 :: []$ or [1]. List operations include the head and tail destructors hd and tl, length function length, membership predicate mem, and the higher-order list function map.

- **Sets:** Sets containing elements of type α are modelled in higher-order logic by functions $\alpha \to \mathbb{B}$, and the notation $\{x \mid P(x)\}$ stands for $\lambda\, x.\ P(x)$. It is possible to define polymorphic higher-order constants representing all the usual set operations such as $\in$, $\cup$, image and $\bigcup$. In addition, for each α there exists a universe set $\mathcal{U}_\alpha = \{x : \alpha \mid \top\}$ that contains every element of type α.[15] We sometimes write a bare $\mathcal{U}$ for $\mathcal{U}_\alpha$ if the type α can be deduced from context. Finally, $A \dot{\to} B$ denotes the set of functions between the sets A and B, and $\bigcup_{x \in s} f(x)$ is a useful shorthand for $\bigcup$(image f s).

- **Mathematics:** When doing informal mathematics, we follow the convenient custom of confusing the group G with its carrier set; in HOL we explicitly write set G for the carrier set (and $*_G$ for the operation). Also, we rely on context to disambiguate the following cases: $|S|$ meaning the cardinality of the set S; $|g|$ meaning the order of the group element g; and $a \mid b \mid c$ meaning that both a divides b and b divides c.

[15] Since the HOL logic specifies that all types are disjoint, so must be these universe sets.

Chapter 2

Formalized Probability Theory

To build our semantics of probabilistic programs upon a solid foundation, we formalize in HOL a rigorous theory of probability based on mathematical measure theory. Soundness is ensured by constructing the probability space of Bernoulli($\frac{1}{2}$) *sequences in a purely definitional extension of higher-order logic. We emphasize the development of notations to simplify our later use of this theory, and as part of this introduce probabilistic versions of the quantifiers into higher-order logic.*

2.1 Introduction

The work described in this chapter is a combination of two technical fields: mathematical measure theory and interactive theorem proving. The author appreciates that few readers will be fluent in both disciplines.

To aid the reader who is familiar with interactive theorem proving but is perhaps rusty on the mathematics, we have tried to make the chapter self-contained by defining all the concepts we use. Nevertheless, this is a poor substitute for a mathematical textbook that carefully explains the difficult ideas: two relevant ones are *Probability with Martingales* (Williams, 1991) and *Probability and Measure* (Billingsley, 1986). Also, if the main interest of the reader is the verification of probabilistic algorithms, then it may make sense to skip directly to Chapter 3 and refer back to this chapter as required.

For the reader who is conversant with measure theory but not with theorem proving, we have attempted to render our theorems in something close to mathematical notation, and provide in Appendix A a quick guide to the fundamental HOL concepts. The purpose of this chapter is to create a version of measure theory that can be input to a machine, and in Appendix C we give an abridged version of this end-product. Finally, for general background on formalization we recommend a paper of Harrison (1996a), and for a detailed introduction to the HOL theorem prover refer to Gordon and Melham (1993) or the current hol98 tutorial (Slind and Norrish, 2001).

2.1.1 The Need for Measure Theory

Recall from Section 1.4 that given a probabilistic 'function' $\hat{f} : \alpha \to \beta$ and a specification $B : \alpha \times \beta \to \mathbb{B}$, we model $\hat{f}$ with a higher-order logic function

$$f : \alpha \to \mathbb{B}^\infty \to \beta \times \mathbb{B}^\infty$$

that explicitly passes around an infinite sequence of random bits. The probability that $\hat{f}(a)$ satisfies the specification B is then the 'probability measure' of the set of sequences

$$\{s \mid B(a, \mathsf{fst}\ (f\ a\ s))\}$$

Thus the first goal of this chapter is make this precise, by formally defining a probability measure

$$\mathbb{P} : \mathcal{P}(\mathbb{B}^\infty) \to [0, 1]$$

from sets of infinite bit sequences to real numbers between 0 and 1, matching our intuition that all sequences are 'equally likely to occur'. To begin with, we list some desirable properties that we might expect any measure function to satisfy.

Definition 1 *Properties that a Measure Function $\mu : \mathcal{P}(X) \to \mathbb{R}$ may satisfy.*

Positivity:

$$\forall\, E.\ 0 \leq \mu(E),\ \textit{with}\ \mu(\emptyset) = 0 \tag{2.1}$$

Monotonicity:

$$\forall\, E, E'.\ E \subset E' \ \Rightarrow\ \mu(E) \leq \mu(E') \tag{2.2}$$

Additivity:

$$\forall\, E, E'.\ E \cap E' = \emptyset \ \Rightarrow\ \mu(E \cup E') = \mu(E) + \mu(E') \tag{2.3}$$

Countable Additivity:

$$\forall\, (E_n).\ (\forall\, i \neq j.\ E_i \cap E_j = \emptyset) \ \Rightarrow\ \mu\Big(\bigcup_n E_n\Big) = \sum_n \mu(E_n) \tag{2.4}$$

In addition, μ is a probability measure *if the measure $\mu(X)$ of the whole space is equal to 1.*

Unfortunately, a celebrated result of Banach and Tarski (see Wagon, 1993) showed that if the Axiom of Choice is assumed,[1] then there can exist sets that are non-measurable. A non-measurable set has the property that whichever real number is defined to be its measure, a contradiction may be derived. To illustrate this result with a classic example, it is possible to dissect the surface of a 2-dimensional sphere into five disjoint pieces, so that by using only rigid transformations they can be re-assembled into two spheres of exactly the same size as the original! Clearly these five pieces can have no well-defined 'area'.

The Banach-Tarski paradox also applies to bounded sets of real numbers. Suppose there existed a measure function

$$\mu : \mathcal{P}[0, 1] \to \mathbb{R}$$

that satisfies all of the above properties, and maps open intervals[2] (a, b) to their length $b - a$. Then it is possible to construct a set $N \subset [0, 1]$, such that whatever value $\mu(N)$ may

[1] As it is in the higher-order logic we use, in common with most of mathematics.

[2] The open interval (a, b) is defined to be the set $\{x \mid a < x < b\}$, and the closed interval $[a, b]$ is the set $\{x \in \mathbb{R} \mid a \leq x \leq b\}$.

take, a contradiction can be derived. In terms of our definitional extension of higher-order logic, the Banach-Tarski paradox effectively prevents any definition of μ from satisfying all of the required properties.[3] And since infinite sequences of bits have similar properties to the real numbers between 0 and 1,[4] it seems reasonable that the Banach-Tarski paradox might also be encountered when trying to define a probability function on sets of sequences. Indeed, this caution is justified in Section 2.4.3, where we define in HOL a non-measurable set Ω of sequences, proving in the theorem prover that Ω can have no possible measure.

2.1.2 How to Create a Measure

To develop a solid foundation upon which to stand a theory of probability, Kolmogorov (1950) and others developed mathematical measure theory: this avoids the contradictions of non-measurable sets by explicitly restricting the domain of measure functions. Before presenting in detail our construction of a probability measure for sets of infinite bit sequences, we will first illustrate the steps of the procedure by showing how Lebesgue integration is defined on sets of real numbers between 0 and 1.

The first step is to choose the generating sets $\mathcal{G}$ of the measure, for which we select all the open intervals in $[0, 1]$:

$$\mathcal{G} = \{(a, b) \mid 0 \leq a < b \leq 1\} \subset \mathcal{P}[0, 1]$$

We also assign a real-valued measure to each of the generating sets; in our case it is natural to assign the length $b - a$ to the open interval (a, b). It is usually a simple matter to check that at this point our measure, *when restricted to the generating sets*, satisfies the properties of Definition 1.

We next perform two closure steps, designed to increase the number of sets that it is possible to measure. The first step jumps to the smallest algebra containing the generating sets.

Definition 2 *An algebra $\mathcal{F} \subset \mathcal{P}(\mathcal{U})$ of sets:*

1. *contains the empty set, i.e., $\emptyset \in \mathcal{F}$;*
2. *is closed under complements, i.e., $\forall A \in \mathcal{F}.\ A^c \in \mathcal{F}$;*
3. *is closed under finite unions, i.e., $\forall A, B \in \mathcal{F}.\ A \cup B \in \mathcal{F}$.*

The measure μ must also be extended so that it is defined on every set in the algebra. It is usually straightforward to define an extended measure and verify that it satisfies the properties of Definition 1 (when restricted to sets in the algebra). Often the sets of the algebra have a finite representation, making it possible to reason about them directly. In our example, the smallest algebra $\mathcal{X}$ containing the generating sets $\mathcal{G}$ is the set of disjoint

[3] The extra work involved in a definitional extension shows its worth at times like this: it would be all too easy to introduce a subtle inconsistency by adding a new constant μ and asserting as axioms the desired properties.

[4] A great deal of intuition can be gained by identifying the bit sequence $(x_0, x_1, x_2, \ldots)$ with the real number $0.x_0x_1x_2\cdots$ (written in binary), ignoring the fact that the distinct sequences $(0, 1, 1, 1, \ldots)$ and $(1, 0, 0, 0, \ldots)$ encode the same real number.

finite unions of intervals, where each interval may either be open (a, b), closed $[a, b]$ or half-open $(a, b], [a, b)$.

The final step in the construction of the measure is the jump to the smallest enclosing σ-algebra.

Definition 3 *A σ-algebra $\mathcal{F} \subset \mathcal{P}(\mathcal{U})$ of sets:*

1. *is an algebra;*

2. *is closed under* countable *unions, i.e.,*

$$\forall\, A_0, A_1, A_2, \ldots \in \mathcal{F}.\ \left(\bigcup_{n \in \mathbb{N}} A_i\right) \in \mathcal{F}$$

We write $\sigma(\mathcal{G})$ for the smallest σ-algebra containing $\mathcal{G}$.

This last step is the important one: in our running example the smallest σ-algebra containing $\mathcal{G}$ defines the Borel sets $\mathcal{B}[0, 1]$, and the extension of our measure μ defines Lebesgue integration. Unfortunately, performing this step is complicated, and generally requires Carathéodory's extension theorem to show that a measure exists with the desired properties. In addition, it is usually the case that sets of the σ-algebra are not finitely representable, and new proof techniques are needed to effectively reason about them. But despite the complications it raises, the step is necessary to ensure that the theory we formalize is comprehensive: "every subset of $\mathbb{R}$ which you meet in everyday use is an element of $\mathcal{B}$" (Williams, 1991, page 17). Indeed, as we shall see in Section 2.4.1, the set of sequences that we need to effectively tackle probabilistic termination appear only at the σ-algebra step.

In Sections 2.2, 2.3 and 2.4 we formalize some general measure theory, construct the algebra of sequence sets, and apply Carathéodory's extension theorem to define the probability space of Bernoulli($\frac{1}{2}$) sequences. The end result of this is a σ-algebra $\mathcal{E}$ of measurable sets, and a measure $\mathbb{P}$ defined on $\mathcal{E}$ which is positive, increasing, additive and countably additive. In the language of probability theory, $\mathcal{E}$ is a set of events and $\mathbb{P}$ is a probability measure.

2.2 Measure Theory

In this section we take a general theory of measure based on sets (Williams, 1991), and formalize in HOL the definitions and theorems that are needed to define a probability measure on sets of bit sequences. Some of this formalization work is not new: there have been two versions of measure theory developed in the Mizar[5] theorem prover. The first version was created by Nędzusiak (1989) to support a rigorous theory of probability, and the second was a more comprehensive effort by Białas (1990) to define Lebesgue integration. Therefore, in this section we will concentrate on the aspects that are either novel to our higher-order logic development, or particularly support our later verification of probabilistic algorithms.

[5] `http://www.mizar.org/`

2.2.1 Measure Spaces

The HOL definitions of algebras and σ-algebras correspond closely to the mathematical versions given in Definitions 2 and 3.

Definition 4 *Algebras and σ-algebras in HOL*

$$\begin{aligned}\vdash\ & \forall \mathcal{F}. \\ & \quad \mathsf{algebra}\ \mathcal{F} = \\ & \quad \emptyset \in \mathcal{F}\ \wedge\ (\forall A.\ A \in \mathcal{F}\ \Rightarrow\ A^c \in \mathcal{F})\ \wedge \\ & \quad \forall A, B.\ A \in \mathcal{F}\ \wedge\ B \in \mathcal{F}\ \Rightarrow\ A \cup B \in \mathcal{F}\end{aligned} \tag{2.5}$$

$$\begin{aligned}\vdash\ & \forall \mathcal{F}. \\ & \quad \mathsf{sigma\_algebra}\ \mathcal{F} = \\ & \quad \mathsf{algebra}\ \mathcal{F}\ \wedge\ \forall \mathcal{C}.\ \mathsf{countable}\ \mathcal{C} \wedge \mathcal{C} \subset \mathcal{F} \Rightarrow \left(\bigcup \mathcal{C}\right) \in \mathcal{F}\end{aligned} \tag{2.6}$$

$$\vdash\ \forall \mathcal{F}.\ \mathsf{sigma}\ \mathcal{G} = \bigcap \{\mathcal{F} \mid \mathsf{sigma\_algebra}\ \mathcal{F}\ \wedge\ \mathcal{G} \subset \mathcal{F}\} \tag{2.7}$$

Using the HOL theory of sets we can now prove some basic results, such as the following propositions:

$$\vdash\ \forall \mathcal{F}, A, B.\ \mathsf{algebra}\ \mathcal{F}\ \wedge\ A \in \mathcal{F}\ \wedge\ B \in \mathcal{F}\ \Rightarrow\ A \cap B \in \mathcal{F} \tag{2.8}$$

$$\vdash\ \forall \mathcal{F}, \mathcal{C}.\ \mathsf{algebra}\ \mathcal{F}\ \wedge\ \mathsf{finite}\ \mathcal{C}\ \wedge\ \mathcal{C} \subset \mathcal{F}\ \Rightarrow\ \left(\bigcup \mathcal{C}\right) \in \mathcal{F} \tag{2.9}$$

$$\vdash\ \forall \mathcal{G}.\ \mathsf{sigma\_algebra}\ (\mathsf{sigma}\ \mathcal{G}) \tag{2.10}$$

This last proposition shows that $\sigma(\mathcal{G})$ is well-defined: by construction there is always a unique smallest σ-algebra containing $\mathcal{G}$.

Conventionally, measure theory defines measures to be functions from the set $\mathcal{P}(X)$ of subsets of a set X to the extended real numbers $\overline{\mathbb{R}} = \mathbb{R} \cup \{-\infty, +\infty\}$. This is also the formalism used by Białas (1990) in Mizar. The extended real numbers are used to create a more uniform theory, so that it is meaningful to write

$$\int_{x \in \mathbb{R}} 1\, dx = +\infty$$

instead of having to say that the left hand side does not have a well-defined value, but diverges to $+\infty$.

Notwithstanding, in our HOL formalization we model measures with functions from $\mathcal{P}(\alpha)$ to the standard real numbers $\mathbb{R}$ (constructed in HOL by Harrison (1998)).[6] We briefly experimented with creating a new type $\overline{\mathbb{R}}$ of extended real numbers, together with lifted versions of the usual field operations. However, we felt that the complications that this generated (transfer functions between the types, dealing with $(+\infty) + (-\infty)$, etc.) more than cancelled out the gain in uniformity described above, and that a simpler theory resulted from sticking to the standard real numbers. Thus our theory only directly applies

[6] It may appear to be a restriction that the domain of our measures is $\mathcal{P}(\alpha)$ (where α is an entire higher-order logic type) instead of the conventional $\mathcal{P}(X)$ (where X is any set). However, we can model a measure μ on X with the measure $\bar{\mu}(A) = \mu(A \cap X)$.

when the measure that we are modelling is finite (i.e., $\mu(\mathcal{U}) < \infty$). If in the future infinite measures are required, the cleanest solution may be to bypass the extended reals entirely and jump to the hyper-reals, which have been formalized in higher-order logic by Fleuriot (2001). In any case, probability measures satisfy $\mu(\mathcal{U}) = 1$, so there are no problems using the standard reals. Indeed, this is the approach taken in the development of probability theory in Mizar by Nędzusiak (1989).

The fundamental object of our interest is the measure space $\mathcal{M} = (\mathcal{F}, \mu)$, which is a set $\mathcal{F}$ of measurable sets paired with a measure function μ. Measure spaces have HOL type

$$\mathcal{P}(\mathcal{P}(\alpha)) \times (\mathcal{P}(\alpha) \to \mathbb{R})$$

and we use the following functions to recover their components:

$$\vdash \ \forall \mathcal{F}, \mu.\ \mathsf{measurable\_sets}\ (\mathcal{F}, \mu) = \mathcal{F} \tag{2.11}$$

$$\vdash \ \forall \mathcal{F}, \mu.\ \mathsf{measure}\ (\mathcal{F}, \mu) = \mu \tag{2.12}$$

Having decided upon our representation, it is a simple matter to define the HOL versions of the measure properties that we introduced in Definition 1.

Definition 5 *Properties of Measure Spaces*

$$\begin{aligned}
&\vdash \ \forall \mathcal{M}. && (2.13)\\
&\quad \mathsf{positive}\ \mathcal{M} = \\
&\quad \mathsf{measure}\ \mathcal{M}\ \emptyset = 0\ \wedge \\
&\quad \forall A.\ A \in \mathsf{measurable_sets}\ \mathcal{M}\ \Rightarrow\ 0 \leq \mathsf{measure}\ \mathcal{M}\ A \\
&\vdash \ \forall \mathcal{M}. && (2.14)\\
&\quad \mathsf{increasing}\ \mathcal{M} = \\
&\quad \forall A, B. \\
&\qquad A \in \mathsf{measurable_sets}\ \mathcal{M}\ \wedge\ B \in \mathsf{measurable_sets}\ \mathcal{M}\ \wedge\ A \subset B\ \Rightarrow \\
&\qquad \mathsf{measure}\ \mathcal{M}\ A \leq \mathsf{measure}\ \mathcal{M}\ B \\
&\vdash \ \forall \mathcal{M}. && (2.15)\\
&\quad \mathsf{additive}\ \mathcal{M} = \\
&\quad \forall A, B. \\
&\qquad A \in \mathsf{measurable_sets}\ \mathcal{M}\ \wedge\ B \in \mathsf{measurable_sets}\ \mathcal{M}\ \wedge \\
&\qquad A \cap B = \emptyset\ \Rightarrow \\
&\qquad \mathsf{measure}\ \mathcal{M}\ (A \cup B) = \mathsf{measure}\ \mathcal{M}\ A + \mathsf{measure}\ \mathcal{M}\ B \\
&\vdash \ \forall \mathcal{M}. && (2.16)\\
&\quad \mathsf{countably_additive}\ \mathcal{M} = \\
&\quad \forall f. \\
&\qquad f \in (\mathcal{U}_{\mathbb{N}} \dot{\to} \mathsf{measurable_sets}\ \mathcal{M})\ \wedge \\
&\qquad (\forall m, n.\ m \neq n \Rightarrow f(m) \cap f(n) = \emptyset)\ \wedge \\
&\qquad \Big(\bigcup_{n \in \mathcal{U}_{\mathbb{N}}} f(n)\Big) \in \mathsf{measurable_sets}\ \mathcal{M}\ \Rightarrow \\
&\qquad \sum_{n \in \mathcal{U}_{\mathbb{N}}} \mathsf{measure}\ \mathcal{M}\ (f(n)) = \mathsf{measure}\ \mathcal{M}\ \Big(\bigcup_{n \in \mathcal{U}_{\mathbb{N}}} f(n)\Big)
\end{aligned}$$

Following Williams (1991), in the definition of countable additivity we assume that the countable union set is measurable.[7]

Some combinations of these properties imply others, and the most useful theorems of this kind are as follows:

$$\vdash \ \forall \mathcal{M}. \quad \mathsf{algebra}\ (\mathsf{measurable\_sets}\ \mathcal{M}) \ \wedge \ \mathsf{positive}\ \mathcal{M} \ \wedge \ \mathsf{additive}\ \mathcal{M} \ \Rightarrow \ \mathsf{increasing}\ \mathcal{M} \tag{2.17}$$

$$\vdash \ \forall \mathcal{M}. \quad \mathsf{algebra}\ (\mathsf{measurable\_sets}\ \mathcal{M}) \ \wedge \ \mathsf{positive}\ \mathcal{M} \ \wedge \ \mathsf{countably\_additive}\ \mathcal{M} \ \Rightarrow \ \mathsf{additive}\ \mathcal{M} \tag{2.18}$$

$$\vdash \ \forall \mathcal{M}, f. \quad \mathsf{algebra}\ (\mathsf{measurable\_sets}\ \mathcal{M}) \ \wedge \ \mathsf{positive}\ \mathcal{M} \ \wedge \ \mathsf{increasing}\ \mathcal{M} \ \wedge \ \mathsf{additive}\ \mathcal{M} \ \wedge \ f \in (\mathcal{U}_{\mathbb{N}} \dot{\rightarrow} \mathsf{measurable\_sets}\ \mathcal{M}) \ \wedge \ (\forall m, n.\ m \neq n \Rightarrow f(m) \cap f(n) = \emptyset) \ \Rightarrow \ \mathsf{summable}\ (\mathsf{measure}\ \mathcal{M} \circ f) \tag{2.19}$$

So assuming an algebra of measurable sets and a positive measure, we have that additivity implies increasing, countable additivity implies additivity, and increasing and additive imply that the sequence of disjoint set measures can be summed. This last result is halfway toward showing that countable additivity holds, the only fact remaining to be shown is that the sum (which must now exist) is equal to the measure of the union of these disjoint sets.

We may now formalize the concept of a measure space.[8]

Definition 6 *Measure Spaces*

$$\vdash \ \forall \mathcal{M}. \quad \mathsf{measure\_space}\ \mathcal{M} = \mathsf{sigma\_algebra}\ (\mathsf{measurable\_sets}\ \mathcal{M}) \ \wedge \ \mathsf{positive}\ \mathcal{M} \ \wedge \ \mathsf{countably\_additive}\ \mathcal{M} \tag{2.20}$$

To speed up the development, some properties of measure spaces are also formalized, including a HOL version of the important Monotone Convergence Theorem.

[7] It is not necessary to assume the union set is measurable in the definition of additivity, since the measurable sets always form an algebra whenever these properties are applied in our development.

[8] Using this property it would now be possible to define a new HOL type of α measure spaces (an inhabitant has measurable sets $\emptyset$ and $\mathcal{U}$ having measure 0 and 1 respectively—this is also a probability space). This would have the positive effect of removing some side-conditions from theorems about measure spaces, but would require extra reasoning with the bijection functions for the new type. It was not clear that a net gain in simplicity would result, and so we chose not to follow this route in the present formalization.

Theorem 7 (Monotone Convergence Theorem)

$$\begin{aligned}\vdash\ & \forall \mathcal{M}, A, f. \\ & \mathsf{measure\_space}\ \mathcal{M}\ \wedge\ f \in (\mathcal{U}_{\mathbb{N}} \dot{\to} \mathsf{measurable\_sets}\ \mathcal{M})\ \wedge \\ & (\forall n.\ f(n) \subset f(\mathsf{suc}\ n))\ \wedge\ (A = \bigcup_{n \in \mathcal{U}_{\mathbb{N}}} f(n))\ \Rightarrow \\ & \lim_{n\to\infty} (\mathsf{measure}\ \mathcal{M}\ (f(n))) = \mathsf{measure}\ \mathcal{M}\ A\end{aligned} \tag{2.21}$$

The conditions on f in this theorem are stated as "$(f(n))_n$ monotonically converges to A", and often written as $(f(n))_n \uparrow A$.

2.2.2 Carathéodory's Extension Theorem

Definition 6 tells us precisely what a measure space is, but offers little in the way of hints on how to construct one. Since an algebra $\mathcal{F}$ is generally finitely representable, it is perhaps reasonable to believe that it will be possible to directly define a particular measure μ_0 on sets in $\mathcal{F}$, and prove that μ_0 has the required properties of positivity and countable additivity. However, if we cannot even write down the general form of a set in $\sigma(\mathcal{F})$, then it is not at all clear how we can extend μ_0 to a measure μ that is positive and countably additive on $\sigma(\mathcal{F})$. This is precisely what the celebrated extension theorem of Carathéodory allows us to do.

Theorem 8 (Carathéodory's Extension Theorem)

$$\begin{aligned}\vdash\ & \forall \mathcal{M}_0. \\ & \mathsf{algebra}\ (\mathsf{measurable\_sets}\ \mathcal{M}_0)\ \wedge\ \mathsf{positive}\ \mathcal{M}_0\ \wedge \\ & \mathsf{countably\_additive}\ \mathcal{M}_0\ \Rightarrow \\ & \exists \mathcal{M}. \\ & \quad (\forall A. \\ & \qquad A \in \mathsf{measurable\_sets}\ \mathcal{M}_0\ \Rightarrow\ \mathsf{measure}\ \mathcal{M}\ A = \mathsf{measure}\ \mathcal{M}_0\ A)\ \wedge \\ & \quad \mathsf{measurable\_sets}\ \mathcal{M} = \mathsf{sigma}\ (\mathsf{measurable\_sets}\ \mathcal{M}_0)\ \wedge \\ & \quad \mathsf{measure\_space}\ \mathcal{M}\end{aligned} \tag{2.22}$$

Given an algebra $\mathcal{A}$ and a positive, countably additive measure μ_0 on $\mathcal{A}$, there exists a measure space $(\sigma(\mathcal{A}), \mu)$ such that $\mu = \mu_0$ on $\mathcal{A}$.

The proof of this theorem requires two new definitions. Firstly, given a set system $\mathcal{F}$ and a measure function λ we may define the λ-system of $\mathcal{F}$, which contains every set $L \in \mathcal{F}$ satisfying

$$\forall A \in \mathcal{F}.\ \lambda(L \cap A) + \lambda(L^c \cap A) = \lambda(A)$$

Secondly, we define an outer measure, which is positive, increasing and countably subadditive, where this last is the following property:

$$\forall (E_n).\ \mu\Big(\bigcup_n E_n\Big) \leq \sum_n \mu(E_n)$$

We next prove Carathéodory's lemma, which states that given an outer measure λ on the σ-algebra $\mathcal{F}$, the sets of the λ-system of $\mathcal{F}$ form a σ-algebra on which λ is countably additive. This was formalized in the Mizar development of Białas (1992).[9]

Carathéodory's extension theorem follows from the lemma by taking the outer measure to be

$$\lambda(A) = \inf\Big\{\sum_n \mu_0(A_n) \;\Big|\; (A_n) \subset \mathcal{A} \;\wedge\; A \subset \bigcup_n A_n\Big\}$$

on the σ-algebra $\mathcal{F}$ of all subsets, and then showing both that $\mathcal{A}$ is a subset of the generated λ-system and that $\lambda = \mu_0$ on $\mathcal{A}$.

The HOL formalization of Carathédory's extension theorem took around two weeks, and the proof script file for the whole of measure theory is about 2000 lines long. Much of the effort was spent on various theorems of sets and real analysis that were needed to support the measure theory proofs.

2.2.3 Functions between Measure Spaces

The measurability of a function is an important theoretical concept, and also a useful practical tool for proving that sets are measurable.

Definition 9 *Measurability*

$$\begin{aligned}\vdash\ & \forall\, \mathcal{F}, \mathcal{G}. \\ & \mathsf{measurable}\ \mathcal{F}\ \mathcal{G} = \{f \mid \forall\, G.\ G \in \mathcal{G} \ \Rightarrow\ \mathsf{preimage}\ f\ G \in \mathcal{F}\}\end{aligned} \tag{2.23}$$

Informally, if $f \in \mathsf{measurable}\ \mathcal{F}\ \mathcal{G}$, where $\mathcal{F}$ has type $\mathcal{P}(\alpha)$ and $\mathcal{G}$ has type $\mathcal{P}(\beta)$, then

$$\begin{array}{rcccc} f & : & \alpha & \to & \beta \\ \mathsf{preimage}\ f & \in & \mathcal{G} & \dot{\to} & \mathcal{F} \end{array}$$

However, when $\mathcal{F}$ and $\mathcal{G}$ are σ-algebras, then it may not be easy to prove that a particular function f is in $\mathsf{measurable}\ \mathcal{F}\ \mathcal{G}$. Again, the problem is that usually there is no general representation of σ-algebra sets, so a direct proof is out of the question. Instead, if $\mathcal{G} = \sigma(\mathcal{C})$ for some set $\mathcal{C} \subset \mathcal{G}$, then we can use a very neat argument. Define the set

$$\mathcal{X} = \{G \in \mathcal{G} \mid \mathsf{preimage}\ f\ G \in \mathcal{F}\}$$

Clearly $\mathcal{X} \subset \mathcal{G}$. If we can show that $\mathcal{X}$ is a σ-algebra that contains $\mathcal{C}$, then by the definition of $\sigma(\mathcal{C})$ we must have that $\sigma(\mathcal{C}) \subset \mathcal{X}$, and so $\sigma(\mathcal{C}) = \mathcal{X}$. Showing that $\mathcal{X}$ is a σ-algebra turns out to be a consequence of the definition of $\mathsf{preimage}$ and is independent of f, so we can prove the general theorem:

$$\begin{aligned}\vdash\ & \forall\, f, \mathcal{F}, \mathcal{C}. \\ & \mathsf{sigma\_algebra}\ \mathcal{F} \ \wedge\ f \in \mathsf{measurable}\ \mathcal{F}\ \mathcal{C} \ \Rightarrow \\ & f \in \mathsf{measurable}\ \mathcal{F}\ (\mathsf{sigma}\ \mathcal{C})\end{aligned} \tag{2.24}$$

[9] In the Mizar formalization of Białas (1992), his Caratheodor's[sic] measures are our outer measures, and what he calls Caratheodor's theorem is what we call Carathéodory's lemma.

In other words, instead of showing that the preimage of every set in the σ-algebra is measurable, we need only show this for a collection of sets that generate the σ-algebra.

We next define a useful extension of measurability, the property of a function being measure-preserving.

Definition 10 *Measure Preservation*

$$\begin{aligned}\vdash\ & \forall \mathcal{M}_1, \mathcal{M}_2. \\ & \quad \mathsf{measure\_preserving}\ \mathcal{M}_1\ \mathcal{M}_2 = \\ & \quad \{f \mid \\ & \qquad f \in \mathsf{measurable}\ (\mathsf{measurable\_sets}\ \mathcal{M}_1)\ (\mathsf{measurable\_sets}\ \mathcal{M}_2)\ \wedge \\ & \qquad \forall A. \\ & \qquad\quad A \in \mathsf{measurable\_sets}\ \mathcal{M}_2\ \Rightarrow \\ & \qquad\quad (\mathsf{measure}\ \mathcal{M}_1\ (\mathsf{preimage}\ f\ A) = \mathsf{measure}\ \mathcal{M}_2\ A)\}\end{aligned} \tag{2.25}$$

A function f is $\mathsf{measure\_preserving}\ \mathcal{M}_1\ \mathcal{M}_2$ if for each measurable set M_2 of $\mathcal{M}_2$, the preimage M_1 under f of M_2 satisfies:

- M_1 is a measurable set of $\mathcal{M}_1$ (so f is measurable between the measurable sets of $\mathcal{M}_1$ and $\mathcal{M}_2$);
- and the measure of M_1 in $\mathcal{M}_1$ is the same as the measure of M_2 in $\mathcal{M}_2$.

Analogously to the measurability property, we would like to show that

$$f \in \mathsf{measure\_preserving}\ \mathcal{M}\ (\sigma(\mathcal{C}), \mu)$$

reduces to

$$f \in \mathsf{measure\_preserving}\ \mathcal{M}\ (\mathcal{C}, \mu)$$

by proving that

$$\{A \in \sigma(\mathcal{C}) \mid \mathsf{measure}\ \mathcal{M}\ (\mathsf{preimage}\ f\ A) = \mu(A)\}$$

is a σ-algebra containing $\mathcal{C}$.

Unfortunately, whereas in the case of measurability this last step was an easy consequence of the definition of preimage, for measure preservation it is not quite so straightforward. The problem is showing closure under finite unions. Just because f preserves the measure of A and B does not necessarily imply that f preserves the measure of $A \cup B$. If A and B are disjoint or one is a subset of the other then the result follows, but otherwise it may not.

Thankfully, a standard result of measure theory comes to the rescue. A neat theorem gives an alternative characterization of $\sigma(\mathcal{A})$ when $\mathcal{A}$ is an algebra.

Theorem 11 *If $\mathcal{A}$ is an algebra, then $\sigma(\mathcal{A})$ is the intersection of all sets $\mathcal{F}$ satisfying the following conditions:*

1. $\mathcal{A} \subset \mathcal{F}$
2. *$\mathcal{F}$ is closed under complements.*

3. *$\mathcal{F}$ is closed under countable disjoint unions.*

4. *$\mathcal{F}$ is closed under countable increasing unions (where (E_n) is increasing if $E_n \subset E_{n+1}$ for every n).*

Proof: Let $\mathcal{G}$ be the intersection of all $\mathcal{F}$ satisfying the above properties. First note that $\mathcal{G} \subset \sigma(\mathcal{A})$, since $\sigma(\mathcal{A})$ satisfies the properties. To show that $\sigma(\mathcal{A}) \subset \mathcal{G}$, we must show that $\mathcal{G}$ is a σ-algebra containing $\mathcal{A}$. Since one characterization of a σ-algebra is an algebra closed under countable disjoint unions, the only difficulty is to show closure under unions.

Let $\mathcal{X}$ be the subset of $\mathcal{G}$ that is closed under unions with sets of the algebra $\mathcal{A}$. But the algebra $\mathcal{A}$ is closed under unions, and so $\mathcal{X}$ satisfies the properties in the theorem, and so $\mathcal{G} \subset \mathcal{X}$. Therefore $\mathcal{G} = \mathcal{X}$, and so $\mathcal{G}$ is closed under unions with sets in the algebra $\mathcal{A}$.

Now let $\mathcal{Y}$ be the subset of $\mathcal{G}$ that is closed under unions with all of the sets in $\mathcal{G}$. As we have just proved, all of the sets in the algebra $\mathcal{A}$ are sets in $\mathcal{Y}$, and so $\mathcal{Y}$ satisfies the properties of the theorem, thus $\mathcal{G} \subset \mathcal{Y}$. Therefore $\mathcal{G} = \mathcal{Y}$, and so $\mathcal{G}$ is closed under unions, as required. □

It is easy to prove that the property of measure-preservation satisfies all of the conditions of Theorem 11, and so we can deduce the following theorem:

$$\begin{aligned} \vdash\ & \forall \mathcal{M}_1, \mathcal{M}_2, \mathcal{A}, f. \\ & \mathsf{measure\_space}\ \mathcal{M}_1\ \wedge\ \mathsf{measure\_space}\ \mathcal{M}_2\ \wedge\ \mathsf{algebra}\ \mathcal{A}\ \wedge \\ & (\mathsf{measurable\_sets}\ \mathcal{M}_2 = \mathsf{sigma}\ \mathcal{A})\ \wedge \\ & f \in \mathsf{measure\_preserving}\ \mathcal{M}_1\ (\mathcal{A}, \mathsf{measure}\ \mathcal{M}_2)\ \Rightarrow \\ & f \in \mathsf{measure\_preserving}\ \mathcal{M}_1\ \mathcal{M}_2 \end{aligned} \tag{2.26}$$

Therefore, instead of showing that f preserves the measure of every set in the σ-algebra, we need only show this for an algebra that generates the σ-algebra.

2.2.4 Probability Spaces

A probability space is a measure space where the measure of the whole space is 1. In a probability space, the measurable sets are called events, and the measure is called a probability.

Definition 12 *Probability Spaces*

$$\begin{aligned} \vdash\ & \forall \mathcal{M}. \\ & \mathsf{prob\_space}\ \mathcal{M} = \mathsf{measure\_space}\ \mathcal{M}\ \wedge\ \mathsf{measure}\ \mathcal{M}\ \mathcal{U} = 1 \end{aligned} \tag{2.27}$$

$$\vdash\ \mathsf{events} = \mathsf{measurable\_sets} \tag{2.28}$$

$$\vdash\ \mathsf{prob} = \mathsf{measure} \tag{2.29}$$

$$\vdash\ \mathsf{prob\_preserving} = \mathsf{measure\_preserving} \tag{2.30}$$

These definitions of events, prob and prob_preserving indicate our technique for dealing with the change in terminology between measure theory and probability theory. Rather than conventionally re-using the existing constants, which would lead to a rather confusing

theory of probability where we referred to measure instead of prob, we create a completely new theory for our probability results. We then import the important results from measure theory into this theory, in the process changing the statements of the theorems to use the new constants.

This approach works surprisingly well. If a result is true for all measure spaces, then we prove it in measure theory and then lift it to probability theory. If, on the other hand, it is special to probability spaces, then we can prove it in the probability theory and leave the measure theory alone. At all times the terminology is consistent and appropriate to the application. In addition, our probability theory in not cluttered with the many lemmas that were needed to prove the measure theory results of the previous subsections.

The first (and most important) concept special to probability spaces is that of independence.

Definition 13 *Independence*

$$\begin{aligned}
&\vdash \ \forall \mathcal{M}, A, B. && (2.31)\\
&\quad \mathsf{indep}\ \mathcal{M}\ A\ B = \\
&\quad A \in \mathsf{events}\ \mathcal{M}\ \wedge\ B \in \mathsf{events}\ \mathcal{M}\ \wedge \\
&\quad \mathsf{prob}\ \mathcal{M}\ (A \cap B) = (\mathsf{prob}\ \mathcal{M}\ A)(\mathsf{prob}\ \mathcal{M}\ B) \\
&\vdash \ \forall \mathcal{M}, \mathcal{F}, \mathcal{G}. && (2.32)\\
&\quad \mathsf{indep_families}\ \mathcal{M}\ \mathcal{F}\ \mathcal{G} = \\
&\quad \forall F, G.\ F \in \mathcal{F}\ \wedge\ G \in \mathcal{G}\ \Rightarrow\ \mathsf{indep}\ \mathcal{M}\ F\ G
\end{aligned}$$

The indep_families property of two families of sets $\mathcal{F}$ and $\mathcal{G}$ merely says that every set $F \in \mathcal{F}$ is independent of every set $G \in \mathcal{G}$. In our later verifications of probabilistic programs, independence is the mechanism that will allow us to decompose terms of the form $\mathbb{P}(A \cap B)$.

The following properties of independence will be useful in our verifications:

$$\begin{aligned}
&\vdash \ \forall \mathcal{M}, A.\ \mathsf{prob_space}\ \mathcal{M}\ \wedge\ A \in \mathsf{events}\ \mathcal{M}\ \Rightarrow\ \mathsf{indep}\ \mathcal{M}\ \emptyset\ A && (2.33)\\
&\vdash \ \forall \mathcal{M}, A.\ \mathsf{prob_space}\ \mathcal{M}\ \wedge\ A \in \mathsf{events}\ \mathcal{M}\ \Rightarrow\ \mathsf{indep}\ \mathcal{M}\ \mathcal{U}\ A && (2.34)\\
&\vdash \ \forall \mathcal{M}, A, B. && (2.35)\\
&\quad \mathsf{prob_space}\ \mathcal{M}\ \wedge\ \mathsf{indep}\ \mathcal{M}\ A\ B\ \Rightarrow\ \mathsf{indep}\ \mathcal{M}\ B\ A \\
&\vdash \ \forall \mathcal{M}, A. && (2.36)\\
&\quad \mathsf{prob_space}\ \mathcal{M}\ \wedge\ A \in \mathsf{events}\ \mathcal{M}\ \Rightarrow \\
&\quad (\mathsf{indep}\ \mathcal{M}\ A\ A\ \Longleftrightarrow\ \mathsf{prob}\ \mathcal{M}\ A = 0\ \vee\ \mathsf{prob}\ \mathcal{M}\ A = 1)
\end{aligned}$$

Nędzusiak (1990) has formalized more properties of independence in a neat Mizar version of basic probability theory, but we could also prove these in our theory if they were needed.

2.3 Bernoulli($\frac{1}{2}$) Sequences: Algebra

As we described in Section 2.1.2, the first step in defining a probability space is to define some basic sets, and then close these sets under complements and finite unions to form an algebra. The second step is to then define a measure on sets in the algebra, satisfying the

conditions of positivity and countable additivity. At this point, we are in a position to apply Carathéodory's extension theorem to this measure and yield the final probability space. In this section we develop the necessary theory to define an algebra of sets of sequences, and a measure function on these sets that respects our intuition that "every sequence is equally likely to occur".

However, the relationship of this theory to the eventual theory of probability is not merely a bootstrapping one, as for example the theory of positive reals is to the theory of reals. Rather, we intend to use the theorems of Section 2.2.3 to solve problems of measurability and measure-preservation of probabilistic programs. This will involve reducing problems on the σ-algebra to equivalent problems on the algebra,[10] and so we must ensure that our algebra theory is equipped with the useful lemmas needed for practical verification.

Although the previous section was a direct formalization of measure theory from the textbook of Williams (1991), at this point our formalization diverges from his treatment. In the theorem-prover we must be more concerned with the representation of sets of infinite bit sequences, so that later we can use this theory to directly prove properties of probabilistic programs. This work therefore represents a novel way to develop the probability space of an infinite sequence of coin-flips: in textbooks this is normally made fully rigorous by defining the space as an infinite product of a single coin-flip.[11]

2.3.1 Infinite Sequence Theory

As usual in higher-order logic modelling, the initial decisions are concerned with choosing types to model our objects. In this case we represent infinite bit sequences with functions $\mathbb{N} \to \mathbb{B}$. The intuition is that if s models a bit sequence, then $s(n) = \top$ means that the nth bit in the sequence is 1, and $s(n) = \bot$ means that the nth bit is 0.

Generalizing slightly, we develop a theory of infinite α-sequences modelled by the type $\mathbb{N} \to \alpha$, where α is any higher-order logic type. Our guide here is the theory of lists, and we define sequence analogues of the list functions hd, tl, cons, take and drop.[12] We also define sdest which gives the head and tail of the sequence as a pair; this will be a surprisingly useful probabilistic program!

Definition 14 *Basic Sequence Operations*

$$\vdash \forall s.\ \mathsf{shd}\ s = s\ 0 \tag{2.37}$$

$$\vdash \forall s.\ \mathsf{stl}\ s = s \circ \mathsf{suc} \tag{2.38}$$

$$\vdash \forall a, s, n.\ \mathsf{scons}\ a\ s\ 0 = a\ \wedge\ \mathsf{scons}\ a\ s\ (\mathsf{suc}\ n) = s\ n \tag{2.39}$$

$$\vdash \forall n, s. \tag{2.40}$$
$$\mathsf{stake}\ 0\ s = [\,]\ \wedge\ \mathsf{stake}\ (\mathsf{suc}\ n)\ s = \mathsf{cons}\ (\mathsf{shd}\ s)\ (\mathsf{stake}\ n\ (\mathsf{stl}\ s))$$

$$\vdash \forall n, s.\ \mathsf{sdrop}\ 0\ s = s\ \wedge\ \mathsf{sdrop}\ (\mathsf{suc}\ n)\ s = \mathsf{sdrop}\ n\ (\mathsf{stl}\ s) \tag{2.41}$$

$$\vdash \forall s.\ \mathsf{sdest}\ s = (\mathsf{shd}\ s, \mathsf{stl}\ s) \tag{2.42}$$

[10] This is how the measurability theorems in Section 2.4.2 were proved.

[11] We could also have used infinite products to define the probability space, but this path would have required formalizing many more theorems of measure theory.

[12] Of course, there is no analogue of [] or length, since these are *infinite* sequences.

Another useful generic construct is the following definition, which creates a sequence from an observation and an iteration function.

Definition 15 *Constructing a Sequence from an Iteration Function*

$$\vdash \quad \forall h, t, x.\ \mathsf{siter}\ h\ t\ x = \mathsf{scons}\ (h(x))\ (\mathsf{siter}\ h\ t\ (t(x))) \tag{2.43}$$

So for example, $\mathsf{siter}\ \mathsf{I}\ \mathsf{suc}\ 0$ *is the sequence* $(0, 1, 2, 3, \ldots)$.

We also define a mirror sequence operation specific to boolean sequences: this maps the sequence $(x_0, x_1, x_2, \ldots)$ to $(\neg x_0, x_1, x_2, \ldots)$.

Definition 16 *The* mirror *Sequence Operation*

$$\vdash \quad \forall s.\ \mathsf{mirror}\ s = \mathsf{scons}\ (\neg(\mathsf{shd}\ s))\ (\mathsf{stl}\ s) \tag{2.44}$$

Finally, we prove some basic properties of sequences, to expedite proofs and to avoid using their underlying representation as functions.

$$\vdash \quad \forall h, t.\ \mathsf{shd}\ (\mathsf{scons}\ h\ t) = h \tag{2.45}$$
$$\vdash \quad \forall h, t.\ \mathsf{stl}\ (\mathsf{scons}\ h\ t) = t \tag{2.46}$$
$$\vdash \quad \forall s.\ \exists h, t.\ s = \mathsf{scons}\ h\ t \tag{2.47}$$
$$\vdash \quad \forall s.\ \mathsf{scons}\ (\mathsf{shd}\ s)\ (\mathsf{stl}\ s) = s \tag{2.48}$$
$$\vdash \quad \forall h, h', t, t'.\ \mathsf{scons}\ h\ t = \mathsf{scons}\ h'\ t' \iff h = h' \ \wedge\ t = t' \tag{2.49}$$
$$\vdash \quad \forall s.\ \mathsf{mirror}\ (\mathsf{mirror}\ s) = s \tag{2.50}$$
$$\vdash \quad \forall s.\ \mathsf{stl}\ (\mathsf{mirror}\ s) = \mathsf{stl}\ s \tag{2.51}$$

In the succeeding theory, all of the infinite sequences we meet will be infinite boolean sequences.

2.3.2 The Algebra Generated by Prefix Sets

We set the prefix sets to be the generating sets of an algebra $\mathcal{A}$.

Definition 17 *Prefix Sets*

$$\vdash \quad \forall l.\ \mathsf{prefix\_set}\ l = \{s \mid \mathsf{stake}\ (\mathsf{length}\ l)\ s = l\} \tag{2.52}$$
$$\vdash \quad \forall h, t.\ \mathsf{prefix\_seq}\ (\mathsf{cons}\ h\ t) = \mathsf{scons}\ h\ (\mathsf{prefix\_seq}\ t) \tag{2.53}$$

So $\mathsf{prefix\_set}\ l$ *is the set of all sequences that have initial segment* l*, and* $\mathsf{prefix\_seq}\ l$ *is the canonical sequence of* $\mathsf{prefix\_set}\ l$.[13]

Since the prefix sets do not naturally form an algebra (for example, the empty set cannot be represented as a prefix set), it is necessary to close the prefix sets under complements and finite unions. For this we shall use an embedding function from boolean list lists to sets of sequences.

[13]Note the use of a pattern match definition to fix only the cons case, allowing the [] case to assume an arbitrary value.

Definition 18 *An Embedding Function*

$$\vdash \quad \forall l.\ \mathsf{embed}\ [l_0, \ldots, l_{n-1}] = \bigcup_{0 \leq i < n} \mathsf{prefix\_set}\ l_i \tag{2.54}$$

We may now define what it means to be a set in $\mathcal{A}$.

Definition 19 *The* Bernoulli($\frac{1}{2}$) *Algebra*

$$\vdash \quad \mathcal{A} = \{A \mid \exists l.\ \mathsf{embed}\ l = A\} \tag{2.55}$$

At the moment we do not have all the machinery we will use to prove that $\mathcal{A}$ is an algebra. Showing closure under complements is not trivial, and it will be the first application of our induction principle on canonical forms, described in Section 2.3.3.

The second step in our probability space construction is to define a measure on the sets in $\mathcal{A}$. We begin with the intuition that all sequences are equally likely, so given two lists l, l' of the same length we would expect the measure of $\mathsf{prefix\_set}\ l$ to be the same as the measure of $\mathsf{prefix\_set}\ l'$. Pursuing the consequences of this, there are 2^n boolean lists of length n, and each sequence falls into exactly one of the prefix sets resulting from these lists, so we therefore define the measure of $\mathsf{prefix\_set}\ l$ to be $2^{-(\mathsf{length}\ l)}$.

We can extend this to a measure μ on a general set in $\mathcal{A}$.

Definition 20 *A Measure on Sets in* $\mathcal{A}$

$$\vdash \quad \forall l.\ \mu_0[l_0, \ldots, l_{n-1}] = \sum_{0 \leq i < n} 2^{-(\mathsf{length}\ l_i)} \tag{2.56}$$

$$\vdash \quad \forall A.\ \mu(A) = \inf\{\mu_0(l) \mid \mathsf{embed}\ l = A\} \tag{2.57}$$

We have to be rather careful in this definition, because for a given $A \in \mathcal{A}$ there may be many $l : \mathbb{B}^{**}$ with $\mathsf{embed}\ l = A$. As an example, both $[[\top]]$ and $[[\top, \top], [\top, \bot], [\top, \top]]$ embed to the set of all sequences beginning with $\top$. The definition works around this by picking the infimum, and since for every $l : \mathbb{B}^{**}$ we have that $0 \leq \mu_0(l)$ this infimum is always well-defined for sets in $\mathcal{A}$.[14]

2.3.3 Canonical Forms

In the previous subsection we set out all the definitions for the algebra $\mathcal{A}$ of boolean sequence sets, and the measure μ from sets in the algebra to real numbers. Following our program for applying measure theory, the next step is to prove that $\mathcal{A}$ is an algebra and that $(\mathcal{A}, \mu)$ is positive and countably additive. We will then be ready to apply the extension theorem.

Unfortunately, the definitions of $\mathcal{A}$ and μ do not directly lend themselves to routine proofs in the HOL theorem prover, most of which are performed using an induction scheme of one kind or another.[15] In particular, the definition of μ as the infimum of a set of real

[14] This follows immediately from the fact that we are working in the real numbers, though it will later turn out that for every $A \in \mathcal{A}$ there is always an $l : \mathbb{B}^{**}$ with $\mathsf{embed}\ l = A$ and $\mu_0(l) = \mu(A)$.

[15] One sizeable exception to this generalization is the theory of real analysis, where there is also no induction scheme available.

numbers is particularly difficult to work with, since at least at first sight we have no idea which boolean list lists that embed to a particular set in the algebra are resulting in the small measures. We cannot even easily calculate $\mu(\mathcal{U})$, the measure of the whole algebra. Even though $\mathsf{embed}\ [[\,]] = \mathcal{U}$ and $\mu_0([[\,]]) = 2^0 = 1$, there may also be another $l : \mathbb{B}^{**}$ with $\mathsf{embed}\ l = \mathcal{U}$ and $\mu_0(l) < 1$.

To resolve this, we introduce a (computable) canonicalization function $\mathsf{canon} : \mathbb{B}^{**} \to \mathbb{B}^{**}$.

Definition 21 *Canonical Forms*

$$\begin{aligned} \vdash\ & \forall l. \\ & (\forall l'.\ \mathsf{embed}\ l' = \mathsf{embed}\ l \iff \mathsf{canon}\ l' = \mathsf{canon}\ l)\ \wedge \\ & \mu_0(\mathsf{canon}\ l) \leq \mu_0(l) \end{aligned} \tag{2.58}$$

$$\vdash\ \forall l.\ \mathsf{canonical}\ l \iff \mathsf{canon}\ l = l \tag{2.59}$$

In other words, the canonical form $\mathsf{canon}\ l$ *is a representative of all boolean list lists that embed to* $\mathsf{embed}\ l$*, and the representative chosen has minimal measure.*

This gives us a more tractable characterization of the measure μ:

$$\vdash\ \forall l.\ \mu(\mathsf{embed}\ l) = \mu_0(\mathsf{canon}\ l) \tag{2.60}$$

So to find the probability of a set $A \in \mathcal{A}$, it suffices to find any $l : \mathbb{B}^{**}$ with $\mathsf{embed}\ l = A$, evaluate $\mathsf{canon}\ l$ in the logic, and then deduce that $\mu(A) = \mu_0(\mathsf{canon}\ l)$.

It is not necessary to describe in detail the HOL definition of canon and the subsequent derivation of the characterizing property (2.58), but it may provide some insight to explain the three steps of the canonicalization function canon:

1. The input list l is sorted according to an order defined on $\mathbb{B}^*$.

2. If there is an element x of l that is a prefix of another element y of l, then y is removed from l. This is a valid step (i.e., it preserves $\mathsf{embed}\ l$) because $\mathsf{embed}\ y \subset \mathsf{embed}\ x$ (as can be seen from Definition 17).

3. Elements are merged with their twins, where the lists x and y are twins with parent z if $x = \mathsf{append}\ z\ [\top]$ and $y = \mathsf{append}\ z\ [\bot]$. If elements x and y of l are twins with parent z, then x is removed and y is replaced by z. This is also a valid step, since $\mathsf{embed}\ z = \mathsf{embed}\ x \cup \mathsf{embed}\ y$.

The only difficulty is choosing the order function in step 1 so that the list is kept in order after an application of step 3.[16]

The canonicalization algorithm is useful not only for evaluating measures, it also gives us an induction principle on elements of the algebra $\mathcal{A}$. Since elements of $\mathcal{A}$ correspond to the underlying list type $\mathbb{B}^{**}$, the standard list induction principle[17] is also applicable, but this does not fit well with the natural structure of our sequence space $\mathbb{B}^{\infty}$, where we would like to use the sequence operators shd and stl to reduce a goal to the induction hypothesis. Here is the induction principle for lists in canonical form:

[16]Note that using sets instead of lists does not obviate the need for canonicalization, because elements would still need to be merged with their twins.

[17]Captured by the theorem: $\vdash\ \forall Q.\ Q([\,])\ \wedge\ (\forall h, t.\ Q(t) \Rightarrow Q(\mathsf{cons}\ h\ t))\ \Rightarrow\ \forall l.\ Q(l)$.

Theorem 22 *Canonical Form Induction Principle*

$$\begin{aligned}\vdash\ & \forall Q. \\ & Q([\,]) \ \wedge\ Q([[\,]]) \ \wedge \\ & (\forall l, l'. \\ & \quad \mathsf{canonical}\ l \ \wedge\ \mathsf{canonical}\ l' \ \wedge\ Q(l) \ \wedge\ Q(l') \ \wedge \\ & \quad \mathsf{canonical}\ (\mathsf{append}\ (\mathsf{map}\ (\mathsf{cons}\ \top)\ l)\ (\mathsf{map}\ (\mathsf{cons}\ \bot)\ l')) \ \Rightarrow \\ & \quad Q(\mathsf{append}\ (\mathsf{map}\ (\mathsf{cons}\ \top)\ l)\ (\mathsf{map}\ (\mathsf{cons}\ \bot)\ l'))) \ \Rightarrow \\ & \forall l.\ \mathsf{canonical}\ l \ \Rightarrow\ Q(l)\end{aligned} \tag{2.61}$$

The base cases are the lists [] (embedding to the empty set) and [[]] (embedding to the whole space $\mathcal{U}$); the step case builds a list in canonical form from two others l, l'. For many goals we can prove the step case with a case split on the head of some sequence, followed by some rewriting with l or l' as appropriate.

2.3.4 Properties of $(\mathcal{A}, \mu)$

The first application of the canonical form induction principle is to show that $\mathcal{A}$ is an algebra, satisfying the HOL definition of an algebra we made in Definition 4. The proof of this in HOL is instructive in its use of the induction principle, and so we give a sketch.

Theorem 23 *$\mathcal{A}$ is an Algebra*

$$\vdash\ \mathsf{algebra}\ \mathcal{A} \tag{2.62}$$

Proof: It is clear that $\emptyset \in \mathcal{A}$ (from the embedding of []), and given two sets $A, B \in \mathcal{A}$ which were embedded from the lists l_A, l_B, it is easy to see that the list $\mathsf{append}\ l_A\ l_B$ embeds to the union $A \cup B$. The only difficulty is to show that $\mathcal{A}$ is closed under complements. Suppose we have $A \in \mathcal{A}$ which embeds from the list l_A. Since A is also the embedding of $\mathsf{canon}\ l_A$, we can assume that l_A is in canonical form. If it is either [] or [[]] then the other embeds to A^c, so we may assume that there exist l, l' with

$$l_A = \mathsf{append}\ (\mathsf{map}\ (\mathsf{cons}\ \top)\ l)\ (\mathsf{map}\ (\mathsf{cons}\ \bot)\ l')$$

By the induction hypothesis the embedding of l has a complement in $\mathcal{A}$, say it was embedded from the list l_c. Similarly there exists l'_c for l'. It is now not difficult to show that

$$\mathsf{append}\ (\mathsf{map}\ (\mathsf{cons}\ \top)\ l_c)\ (\mathsf{map}\ (\mathsf{cons}\ \bot)\ l'_c)$$

embeds to A^c, as required. $\square$

Using similar arguments, we prove that $(\mathcal{A}, \mu)$ is positive, additive, and even countably additive. This last follows from a neat property of the algebra $\mathcal{A}$:

$$\begin{aligned}\vdash\ & \forall f. \\ & f \in (\mathcal{U}_{\mathbb{N}} \dot{\rightarrow} \mathcal{A}) \ \wedge\ (\forall m, n.\ m \neq n \Rightarrow f(m) \cap f(n) = \emptyset) \ \wedge \\ & \Big(\bigcup_{n \in \mathcal{U}_{\mathbb{N}}} f(n)\Big) \in \mathcal{A} \ \Rightarrow \\ & \exists N.\ \forall n.\ N \leq n \Rightarrow f(n) = \emptyset\end{aligned} \tag{2.63}$$

Any countable sequence of disjoint sets in $\mathcal{A}$ that union to a set in $\mathcal{A}$ contains only a finite number of non-empty sets. Thus in the case of $(\mathcal{A}, \mu)$, countable additivity collapses to additivity.

An interesting property we prove is that the measure μ has an upper bound of 1:

$$\vdash \quad \forall A.\ A \in \mathcal{A} \Rightarrow \mu(A) \leq 1 \tag{2.64}$$

This is a special case of Kraft's Inequality (Goldie and Pinch, 1991):

$$\forall C.\ \sum_{w \in C} a^{-(\mathsf{length}\ w)} \leq 1 \tag{2.65}$$

which holds for all prefix-free codes C where the alphabet has a symbols, and exactly formulates the balance between keeping the code prefix-free and having many short code words. In our situation we have 'code words' of boolean lists (so $a = 2$), and for the purpose of calculating μ we can assume that our 'code' is in canonical form (and hence prefix-free).

2.4 Bernoulli($\frac{1}{2}$) Sequences: Probability Space

2.4.1 The Need for σ-algebras

In the previous section we defined an algebra $\mathcal{A}$ with associated measure μ, established an induction principle to expedite proofs in the algebra, and proved that $(\mathcal{A}, \mu)$ satisfies the basic properties of measures.

For a probabilistic program $f : \mathbb{B}^\infty \to \alpha \times \mathbb{B}^\infty$ *that is guaranteed to terminate*, all of the interesting behaviour of f can be captured using sets of $\mathcal{A}$.[18] In a previous version of our probability theory (Hurd, 2000), we were able to verify some simple probabilistic programs using only $(\mathcal{A}, \mu)$.

However, probabilistic programs that are guaranteed to terminate are a relatively small class, and there are many interesting programs that cannot be expressed with this restriction. An example is the program f that scans the input sequence for the first occurrence of a $\bot$, and returns the index (counting from 0) of this $\bot$ within the sequence.[19] Although the probability that f terminates is 1, termination is nevertheless not guaranteed: the sequence $(\top, \top, \ldots)$ results in divergence.[20] Now consider the set S of input sequences that result in f returning an even number:

$$S = \mathsf{prefix\_set}\ [\bot]\ \cup\ \mathsf{prefix\_set}\ [\top, \top, \bot]\ \cup\ \mathsf{prefix\_set}\ [\top, \top, \top, \top, \bot]\ \cup\ \cdots$$

Although S is a countable union of sets in $\mathcal{A}$, it is not itself a set in $\mathcal{A}$. Therefore, since μ is only defined on sets in $\mathcal{A}$, we cannot speak of the probability $\mu(S)$ that f returns an even number. On the other hand, σ-algebras *are* closed under countable unions, so if we

[18] If f is guaranteed to terminate then there is a number n such that whatever sequence s is input to f, no more than the initial n bits of s will be used by f to calculate a result.

[19] This is the Geometric($\frac{1}{2}$) distribution, and we study this in detail in Section 4.4.

[20] Using the argument of Footnote 18, the fact that the set of possible results forms an infinite set also proves that f cannot be guaranteed to terminate.

extend the measure μ to be defined on $\sigma(\mathcal{A})$, then S is an event and the probability of S becomes meaningful.

The conclusion is that $\mathcal{A}$ allows us to reason about probabilistic programs that are guaranteed to terminate, but $\sigma(\mathcal{A})$ is required to reason about programs that terminate with probability 1.

2.4.2 Definition of the Probability Space

The properties of $(\mathcal{A}, \mu)$ established in Section 2.3.4 are precisely the conditions required by Carathéodory's extension theorem, which we formalized in Section 2.2.2. Applying the theorem, we can define the probability space $(\mathcal{E}, \mathbb{P})$ and prove the following characterizing theorem.

Definition 24 *The Probability Space of* Bernoulli($\frac{1}{2}$) *Sequences*

$$\begin{aligned} \vdash \quad & \mathsf{prob\_space}\ (\mathcal{E}, \mathbb{P})\ \wedge\ (\mathcal{E} = \mathsf{sigma}\ \mathcal{A})\ \wedge \\ & \forall A.\ A \in \mathcal{A}\ \Rightarrow\ \mathbb{P}(A) = \mu(A) \end{aligned} \tag{2.66}$$

Now using the properties of probability spaces, some standard theorems follow, including the result that any countable set of sequences has probability 0:

$$\vdash \quad \forall A.\ \mathsf{countable}\ A\ \Rightarrow\ A \in \mathcal{E}\ \wedge\ \mathbb{P}(A) = 0 \tag{2.67}$$

Therefore, for any event $E \in \mathcal{E}$, adding or taking away a countable set of sequences does not change the probability of E. In particular, if we start with the whole space $\mathcal{U}$, and for every list $l : \mathbb{B}^*$ we remove the sequence with initial segment l and continuing $(\top, \top, \ldots)$, then the set Θ we are left with still has probability 1.[21]

This result justifies the intuition we gain by the identification of bit sequences with real numbers via the binary expansion

$$(x_0, x_1, x_2, \ldots) \xrightarrow{\phi} 0.\, x_0\, x_1\, x_2\, \cdots$$

even though there are some distinct sequences (such as $(0, 1, 1, 1, \ldots)$ and $(1, 0, 0, 0, \ldots)$) that map to the same real number.[22] The set Θ, an event of probability 1, is in one-to-one correspondence with the real numbers in the interval $[0, 1)$.[23] Though we do not pursue the consequences of this here, it is possible to define Lebesgue integration for a set $S \subset [0, 1]$ of reals by

$$\mathrm{Leb}(S) = \mathbb{P}(\mathsf{preimage}\ \phi\ S)$$

We now prove some important properties of the sequence operations, showing how they map events and change their probabilities. The proof technique is always the same:

[21] This result is the intuitively correct one. In our previous version of probability theory (Hurd, 2000) where we were limited to $(\mathcal{A}, \mu)$, we defined the probability of a set B to be the supremum of all $\mu(A)$ where $A \in \mathcal{A}$ and $A \subset B$. Unfortunately, the only set in $\mathcal{A}$ that is a subset of Θ is the empty set, and so the probability of Θ was set to 0.

[22] This same complication rears its ugly head in the proof of Cantor's theorem that the real numbers are uncountable, which is why proofs normally use decimal expansions!

[23] This is from (Williams, 1991, page 42): every real number in $(0, 1)$ has one bit sequence mapping to it, except for the dyadic rationals which have two. The sequences missing from the set Θ are easily seen to be in one-to-one correspondence with the dyadic rationals.

use the theorems of Section 2.2.3 to reduce a property over $\mathcal{E}$ to a property over $\mathcal{A}$, and from there use the canonical form induction principle of Section 2.3.3.

Firstly, we tackle measurability and stronger properties of eventhood:

$$\vdash \ \forall b.\ \{s \mid \mathsf{shd}\ s = b\} \in \mathcal{E} \tag{2.68}$$
$$\vdash \ \forall E.\ \mathsf{preimage\ stl}\ E \in \mathcal{E} \iff E \in \mathcal{E} \tag{2.69}$$
$$\vdash \ \forall E.\ E \in \mathcal{E} \Rightarrow \mathsf{image\ stl}\ E \in \mathcal{E} \tag{2.70}$$
$$\vdash \ \forall E, n.\ \mathsf{preimage}\ (\mathsf{sdrop}\ n)\ E \in \mathcal{E} \iff E \in \mathcal{E} \tag{2.71}$$
$$\vdash \ \forall E, n.\ E \in \mathcal{E} \Rightarrow \mathsf{image}\ (\mathsf{sdrop}\ n)\ E \in \mathcal{E} \tag{2.72}$$
$$\vdash \ \forall b.\ \mathsf{scons}\ b \in \mathsf{measurable}\ \mathcal{E}\ \mathcal{E} \tag{2.73}$$
$$\vdash \ \forall E, b.\ \mathsf{image}\ (\mathsf{scons}\ b)\ E \in \mathcal{E} \iff E \in \mathcal{E} \tag{2.74}$$
$$\vdash \ \forall E.\ \mathsf{preimage\ mirror}\ E \in \mathcal{E} \iff E \in \mathcal{E} \tag{2.75}$$

These theorems are all as strong as possible, counter-intuitive though this may appear. For example, the plausible-looking statement

$$\mathsf{image\ stl}\ E \in \mathcal{E} \Rightarrow E \in \mathcal{E}$$

fails because of the counterexample

$$E = \mathsf{image}\ (\mathsf{scons}\ \top)\ \mathcal{U} \cup \mathsf{image}\ (\mathsf{scons}\ \bot)\ \Omega$$

where Ω is any non-measurable set (we shall explicitly construct a non-measurable set in Section 2.4.3). The image of E under stl is $\mathcal{U} \in \mathcal{E}$, but if E were measurable then using theorems (2.68) and (2.74) it would be possible to show that Ω was measurable.

Note that we do not need a theorem about images of mirror, because

$$\vdash \ \mathsf{image\ mirror} = \mathsf{preimage\ mirror} \tag{2.76}$$

We also prove some probability preservation properties:

$$\vdash \ \forall b.\ \mathbb{P}\{s \mid \mathsf{shd}\ s = b\} = \tfrac{1}{2} \tag{2.77}$$
$$\vdash \ \mathsf{stl} \in \mathsf{prob\_preserving}\ \mathcal{E}\ \mathcal{E} \tag{2.78}$$
$$\vdash \ \forall n.\ \mathsf{sdrop}\ n \in \mathsf{prob\_preserving}\ \mathcal{E}\ \mathcal{E} \tag{2.79}$$
$$\vdash \ \mathsf{mirror} \in \mathsf{prob\_preserving}\ \mathcal{E}\ \mathcal{E} \tag{2.80}$$
$$\vdash \ \forall E. \tag{2.81}$$
$$E \in \mathcal{E} \ \wedge\ \mathsf{image\ mirror}\ E = E \Rightarrow \mathbb{P}(\mathsf{image\ stl}\ E) = \mathbb{P}(E)$$

Finally, once we have the basic properties of the sequence operations, we can prove the following two theorems:

$$\vdash \ \forall n.\ \mathbb{P}\{s \mid \mathsf{shd}\ (\mathsf{sdrop}\ n\ s)\} = \tfrac{1}{2} \tag{2.82}$$
$$\vdash \ \forall m, n. \tag{2.83}$$
$$\mathbb{P}\{s \mid \mathsf{shd}\ (\mathsf{sdrop}\ m\ s) = \mathsf{shd}\ (\mathsf{sdrop}\ n\ s)\} = \mathsf{if}\ m = n\ \mathsf{then}\ 1\ \mathsf{else}\ \tfrac{1}{2}$$

These say that the probability of any random bit being 1 is $\frac{1}{2}$ (2.82), and the probability of two distinct random bits being equal is also $\frac{1}{2}$ (2.83). Since these are the results we would expect for a sequence of coin flips, this gives us confidence that the probability space $(\mathcal{E}, \mathbb{P})$ we have defined in this section is a good model for a random bit generator.

2.4.3 Construction of a Non-Measurable Set

The purpose of this section is to define a set Ω of sequences, and prove the theorem

$$\vdash \quad \Omega \notin \mathcal{E} \tag{2.84}$$

that Ω is not an event of our probability space.

This represents a digression from the focus of this chapter, which is to formalize theory that will support verifications of probabilistic programs. It is, however, a theoretically important result, justifying the route through measure theory that our theory has taken, and providing a tool for generating counter-examples. Ironically, since σ-algebras are so successful at capturing the events of interest to probability theory, it is not easy to construct a set that is provably not an event.

We adapt a construction in Williams (1991, page 192) of a non-measurable set $A \subset S^1$, where S^1 is the unit circle. The argument begins as follows:

> ... we use the Axiom of Choice to show that
>
> $$S^1 = \bigcup_{q \in \mathbb{Q}} A_q$$
>
> where the A_q are disjoint sets, each of which may be obtained from any of the others by rotation. If the set $A = A_0$ has a 'length', then it is intuitively clear that [the above result] would force
>
> $$2\pi = \infty \times \text{length}(A),$$
>
> an impossibility.

The important parts of this are:

1. The Axiom of Choice is used to construct the non-measurable set A.
2. There is a mapping that deterministically changes the measure of sets (in this case rotations preserve length).
3. The whole space is written as a disjoint union of countably many rotations of A.

We now transfer these ideas to sets of boolean sequences. For step 1 of the above outline we use the eventually relation between sequences.

Definition 25 *The 'Eventually' Sequence Relation*

$$\vdash \quad \forall x, y.\ \mathsf{eventually}\ x\ y = \exists n.\ \mathsf{sdrop}\ n\ x = \mathsf{sdrop}\ n\ y \tag{2.85}$$

The sequence x is eventually the sequence y if they disagree only on some initial segment; eventually *can be shown to be an equivalence relation.*

Definition 26 *A Non-Measurable Set*

$$\vdash \quad \forall x.\ \exists! y.\ \mathsf{eventually}\ x\ y\ \wedge\ y \in \Omega \tag{2.86}$$

The set Ω consists of equivalence class representatives of the eventually *relation.*[24]

For step 2, we consider the sequence of sets

$$\Omega_n = \mathsf{preimage}\ (\mathsf{sdrop}\ n)\ (\mathsf{image}\ (\mathsf{sdrop}\ n)\ \Omega)$$

Assuming Ω is an event, and using theorems (2.79) and (2.81), it can be shown that

$$\mathbb{P}(\Omega_n) = 2^n \mathbb{P}(\Omega)$$

Since probabilities are always bounded above by 1, we must have that

$$\mathbb{P}(\Omega) = 0$$

For step 3, we first prove a lemma that the sequence Ω_n monotonically converges to the whole space $\mathcal{U}$, or

$$\Omega_n \subset \Omega_{n+1} \ \wedge\ \mathcal{U} = \bigcup_{n \in \mathcal{U}_{\mathbb{N}}} \Omega_n$$

Now the Monotone Convergence Theorem may be applied, allowing

$$0 = \lim_{n\to\infty} 0 = \lim_{n\to\infty} 2^n \mathbb{P}(\Omega) = \lim_{n\to\infty} \mathbb{P}(\Omega_n) = \mathbb{P}(\mathcal{U}) = 1$$

and we may conclude that Ω is not an event of the probability space.

2.4.4 Probabilistic Quantifiers

In probability textbooks, it is common to find many theorems with the qualifier 'almost surely', 'with probability 1' or just 'w.p. 1'. Intuitively, this means that the set of points for which the theorem is true has probability 1 (the set and the probability space usually being implied by context). We can define probabilistic versions of the $\forall$ and $\exists$ quantifiers that make this notation precise.[25]

Definition 27 *Probabilistic Quantifiers*

$$\vdash\ \forall \phi.\ (\forall^* s.\ \phi(s)) = \{s \mid \phi(s)\} \in \mathcal{E}\ \wedge\ \mathbb{P}\{s \mid \phi(s)\} = 1 \tag{2.88}$$

$$\vdash\ \forall \phi.\ (\exists^* s.\ \phi(s)) = \{s \mid \phi(s)\} \in \mathcal{E}\ \wedge\ \mathbb{P}\{s \mid \phi(s)\} \neq 0 \tag{2.89}$$

As noted by Harrison (1996a) on the subject of variable binding in mathematics, "[formalization] really can be valuable, confronting us with awkward constructs where the everyday notation confuses free and bound variables or a function with its value." Here we have an opportunity of making precise a (potentially) confusing everyday notation, but the familiar use of quantifiers for binding avoids the necessity of an awkward construct.

[24] Theorem (2.86) is a characterizing theorem; the actual definition of Ω explicitly invokes the Axiom of Choice (in the form of Hilbert's choice operator ε):

$$\vdash\ \Omega = \mathsf{image}\ (\lambda x.\ \varepsilon y.\ \mathsf{eventually}\ x\ y)\ \mathcal{U} \tag{2.87}$$

[25] We pronounce $\forall^*$ as "probably" and $\exists^*$ as "possibly".

It might be hoped that the probabilistic quantifiers can be manipulated and exchanged much like their standard counterparts, but unfortunately the analysis is considerably more complicated. Firstly, since we must explicitly include a condition of measurability in the definition of each quantifier,[26] $\forall^*$ and $\exists^*$ are not dual to each other. Keisler (1985) has defined a simple logic where 'probability quantifiers' replace the standard ones, but duality is preserved because every formula results in a measurable set. As we saw in Section 2.4.3, higher-order logic is more expressive than this.

Secondly, even when duality is not relevant, simple operations can raise deep complications. Consider exchanging two $\forall^*$ quantifiers:

$$\forall^* s.\ \forall^* t.\ \phi(s,t) = \forall^* t.\ \forall^* s.\ \phi(s,t) \tag{2.90}$$

In Keisler's logic, he notes a case where exchanging two probability quantifiers is not a valid operation,[27] but the counterexample he gives involves probabilities strictly between 0 and 1, so it might be hoped that equation (2.90) would still hold. Unfortunately, a result of Sierpinski shows that the continuum hypothesis is equivalent to the existence of a set $S \subset \mathbb{R}^2$ with the following properties:

- For all y, the sets $\{x \mid (x,y) \in S\}$ are countable.
- For all x, the sets $\{y \mid (x,y) \in S\}$ are co-countable.[28]

Therefore, a counterexample to equation (2.90) can be found by defining

$$\phi(s,t) = (\bar{s},\bar{t}) \in S$$

where $\bar{s}$ is the real number with binary expansion s. Since this counterexample relies on the continuum hypothesis, which is independent of ZFC (and therefore HOL), it is not possible to use this to prove in HOL the negation of equation (2.90). However, it most certainly does stop us from proving equation (2.90), and does not rule out the possibility that a proof of the negation exists.

We can at least prove one useful theorem about exchanging probabilistic and standard quantifiers

$$\begin{aligned} \vdash\ & \forall\, \phi, A. \\ & \mathsf{countable}\ A\ \wedge\ (\forall\, x \in A.\ \forall^* y.\ \phi\ x\ y)\ \Rightarrow\ \forall^* y.\ \forall\, x \in A.\ \phi\ x\ y \end{aligned} \tag{2.91}$$

which we will use in Section 3.3.3 as part of a useful proof technique for establishing properties of probabilistic while loops. Note that the countability condition is required, or else we could exchange the quantifiers in

$$\vdash\ \forall\, x \in \mathcal{U}.\ \forall^* s.\ s \neq x \tag{2.92}$$

[26] Or else theorems such as $\forall^* s.\ s \in \Omega$ would be provable, and we could deduce very little from a $\forall^*$-quantified theorem.

[27] "The reader can check that no two of the sentences

$$(Px \geq \tfrac{1}{2})(Py \geq \tfrac{1}{2})R(x,y) \qquad (Py \geq \tfrac{1}{2})(Px \geq \tfrac{1}{2})R(x,y) \qquad (Pxy \geq \tfrac{1}{4})R(x,y)$$

are equivalent. (Consider structures with three elements of measure $\frac{1}{3}$.)"

[28] A set $A \subset \mathbb{R}$ is co-countable if $\mathbb{R} - A$ is countable.

to get

$$\forall^* s.\ \forall x.\ s \neq x$$

which is equivalent to

$$(\forall^* s.\ \bot) \iff \emptyset \in \mathcal{E} \ \wedge\ (\mathbb{P}(\emptyset) = 1) \iff \top \wedge\ (0 = 1) \iff \bot$$

So far we have presented these probabilistic quantifiers as a notational device, a precise and still intuitive version of the tag 'with probability 1'. However, we have discovered one situation where the use of $\forall^*$ represents more than just a useful shorthand: it also speeds up the proof. The technique uses a lifting theorem for $\forall^*$-quantified properties.

Theorem 28 *The $\forall^*$-lifting Theorem.*

$$\begin{aligned} \vdash\ & \forall \phi, E, E'. \\ & E \in \mathcal{E} \ \wedge\ E' \in \mathcal{E} \ \wedge\ \forall^* \phi \ \wedge\ (\forall x.\ \phi(x) \Rightarrow (x \in E \iff x \in E')) \ \Rightarrow \\ & \mathbb{P}(E) = \mathbb{P}(E') \end{aligned} \tag{2.93}$$

This says that that the probability of the set E is equal to the probability of E' if we can transform E to E' by regarding as universally true any *probabilistic* universally quantified property ϕ. In this way a $\forall^*$-theorem gets lifted to a $\forall$-theorem in the argument of a $\mathbb{P}$.[29]

Example: Suppose we wish to prove

$$\mathbb{P}\{s \mid \mathsf{shd}\ s \wedge s \neq y\} = \mathbb{P}\{s \mid \mathsf{shd}\ s\}$$

By specializing the $\forall^*$-lifting theorem to the $\forall^*$-quantified property in theorem (2.92) above, it remains only to prove that

$$\begin{aligned} & \{s \mid \mathsf{shd}\ s \wedge s \neq y\} \in \mathcal{E} \ \wedge\ \{s \mid \mathsf{shd}\ s\} \in \mathcal{E} \ \wedge \\ & \forall x.\ x \neq y \Rightarrow (x \in \{s \mid \mathsf{shd}\ s \wedge s \neq y\} \Longleftrightarrow x \in \{s \mid \mathsf{shd}\ s\}) \end{aligned}$$

which is easily accomplished using the measurability properties of shd. □

Before we proved the $\forall^*$-lifting theorem, we would have to prove these kind of results by explicitly constructing an event S of probability 1, and then use set operations to show

$$\mathbb{P}(E) = \mathbb{P}(E \cap S) = \mathbb{P}(E' \cap S) = \mathbb{P}(E')$$

In our experience, this 'bare-hands' approach generally results in more work.

2.5 Concluding Remarks

The main result of this chapter is the measure-theoretic construction of the probability space $(\mathcal{E}, \mathbb{P})$. Chapter 3 will use the properties of $(\mathcal{E}, \mathbb{P})$ we have formalized to specify and verify some probabilistic algorithms.

[29] Another way of seeing the $\forall^*$-lifting theorem is as a congruence rule for $\mathbb{P}$, though it can't be given to a HOL simplifier in its current form with ϕ left unspecified.

There have been several semantics of probabilistic programs presented in the literature of theoretical computer science. Most of them use measure theory in a more or less direct way, to avoid falling into the logical inconsistencies of the Banach-Tarski paradox. Therefore, to formalize one of these semantics in a theorem prover, measure theory (or a particular specialization) must be formalized also. This is the novelty of this chapter: measure theory as a definitional extension of higher-order logic, intended to support our own simple semantics of probabilistic programs.

Much of the technical work in this chapter was anticipated in the Mizar developments of Nędzusiak (1989) and Białas (1990), but there are many situations (such as Carathéodory's extension theorem) in which we have had to go a little further into the theory to extract the particular theorems that our application requires. The formal construction of a non-measurable set is a novelty, as is the introduction of probabilistic quantifiers into a standard logic.

Our formalized probability theory allows us to express both standard logical truths and probabilistic statements in higher-order logic. There has been some theoretical work on the various techniques of combining various (first-order) logics and probability to achieve this effect: papers of Halpern (1990) and Abadi and Halpern (1994) give some decidability and complexity results. Another approach is taken in the fuzzy logic of Zadeh (1965), where the truth values are extended from the booleans $\{\bot, \top\} \cong \{0, 1\}$ to the whole real interval $[0, 1]$. The logical connectives are also extended to operate over the larger domain: although this extension is necessarily rather arbitrary, there are nevertheless many control systems that internally make use of fuzzy logic to operate robustly in a changing environment. There is also a probabilistic logic due to Nilsson (1986), which is a version of fuzzy logic where the truth values correspond to probabilities. In this case the logical connectives are not simple functions of real numbers, instead the entailment relation must be calculated by multiplying matrices together.

We used the hol98 theorem prover to do this work, creating a purely definitional theory to ensure soundness (in common with most other hol98 theories). Particularly important for this work were the theories of real numbers (created by Harrison (1998)) and predicate sets, although of course we found invaluable the basic theories of arithmetic, lists, well-founded recursion and the like. The built-in proof tools of hol98 sped up the whole development, which makes heavy use of the simplifier, first-order prover and real number decision procedure. However, although these tools make a big difference to the task of 'proving the current subgoal', we should not forget the higher-level issues that also sped up this development:

- Picking the right higher-order logic types to represent our objects. It would perhaps have been more natural to model bit sequences with number sequences instead of the boolean sequences we chose, but this would have resulted in extra well-formedness conditions guaranteeing that only the numbers '1' and '0' were present.

- Higher-order theorems capturing common patterns of reasoning should be actively sought. We 'discovered' the Canonical Form Induction Principle quite late in the development, and when we cleaned up the proofs of the earlier theorems using the new technique, the proof scripts shrunk dramatically and many lemmas were rendered obsolete.

Chapter 3

Verifying Probabilistic Algorithms

We present a simple model of probabilistic algorithms in higher-order logic, in which it is possible to express any program with access to a source of random bits. The formalization of probability theory allows us to make meaningful specifications of these algorithms, and we develop the formal framework to make verification in the theorem prover more straightforward. One novelty is a compositional property of probabilistic algorithms that implies measurability and independence. Finally, we examine various schemes of probabilistic termination with reference to our own probabilistic while loop.

3.1 Introduction

3.1.1 Motivating Probabilistic Algorithms

The study of probabilistic algorithms grew out of the Monte Carlo simulation methods used in numerical analysis and statistical physics.[1] The first formal model of probabilistic computation was proposed in a paper of de Leeuw et al. (1955), followed by the pioneering work of Rabin (1963), Karp (1976) and Gill (1977). Today probabilistic algorithms are well-established in computer science, and there are many examples of simple probabilistic algorithms that outperform all known deterministic alternatives. To illustrate this phenomenon, which best conveys the advantages of probabilistic algorithms, consider the following two problems:

1. Given two sets of polynomials over $\mathbb{Q}$

$$\{P_1, \ldots, P_m\} \text{ and } \{Q_1, \ldots, Q_n\}$$

where each polynomial is represented by a list of the non-zero terms, we wish to know if

$$\prod_{1 \le i \le m} P_i \neq \prod_{1 \le i \le n} Q_i$$

[1] An early example of Monte Carlo methods used in the sciences is the classic experiment of Buffon (1777) where the value of π is approximated by repeatedly dropping a needle onto a surface marked with parallel lines. If the lines are exactly one needle's length apart, then the probability that the needle will intersect a line is $2/\pi$.

2. How can the pivot be picked in quicksort so that the sorting algorithm has a fast *expected* running time?

Problem 1 is known in the complexity literature as PRODUCT POLYNOMIAL INEQUIVALENCE. It has no known polynomial time solution,[2] but it can be solved in *expected* polynomial time:

- pick an x in some appropriately random fashion;
- evaluate each P_i and Q_i at x;
- multiply together the two resulting collections of rationals, giving the values of the product polynomials at x;
- if the values are different, then the product polynomials are different and the algorithm terminates;
- if the values are the same, then we repeat the procedure.

It can be shown that if the product polynomials are inequivalent, then the expected running time of this algorithm is polynomial (Schwartz, 1980).

In terms of complexity theory, PRODUCT POLYNOMIAL INEQUIVALENCE falls into a class of problems called RP for Randomized Polynomial time. The class RP is defined to be the set of languages L for which there is a Deterministic Turing Machine M equipped with a source of random bits, such that M runs in worst-case polynomial time and satisfies:

$$\begin{aligned} x \in L &\Rightarrow \mathbb{P}(M \text{ accepts } x) \geq \tfrac{1}{2} \\ x \notin L &\Rightarrow \mathbb{P}(M \text{ accepts } x) = 0 \end{aligned}$$

The known inclusion relations between P, RP and NP are depicted in Figure 3.1 (taken from Johnson, 1990). If any of these inclusions were known to be strict then it would prove that P $\neq$ NP, and if P = NP then all these classes collapse to P. There are no problems known to be complete for RP, in the manner of NP-completeness.[3]

Problem 2 is deceptively hard to address, as discussed by Rabin (1976). The problem is that the concept of *expected running time* depends on how the input data is distributed, but we may have no knowledge of this. For every $O(1)$[4] deterministic algorithm for picking the pivot[5] there is a small class of input permutations that result in $O(n^2)$ performance. Since the distribution of the input data is not known, for the analysis we must assume that inputs are always chosen from this bad class, and so the expected running time is $O(n^2)$.

However, by introducing randomness into the algorithm, we can eliminate the class of bad inputs by making the expected computation time the same for every input. In randomized quicksort, we introduce the randomness by picking pivots uniformly at random.

[2]Note that directly calculating the product polynomials does not work, since the result may have exponentially more non-zero terms.

[3]Relative to a random oracle, P = RP $\cap$ co-RP = RP, and all are properly contained in NP. However, oracles exist for which P $\neq$ RP $\cap$ co-RP and RP $\cap$ co-RP $\neq$ RP.

[4]The number of comparisons is our complexity measure, as usual for sorting algorithms.

[5]Median-of-three falls into this category.

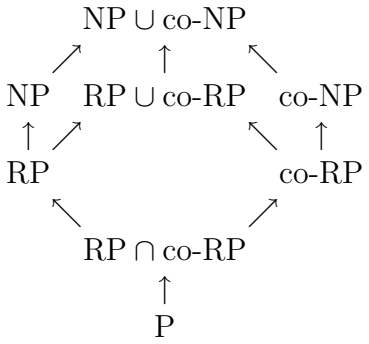


Figure 3.1: The Known Inclusion Relations between P, RP and NP.

Then, as shown in Motwani and Raghavan (1995), the expected number of comparisons for every input permutation of size n is bounded above by $2nH_n$, where

$$H_n = \sum_{i=1}^{n} \frac{1}{i} \sim \ln n + O(1)$$

is the nth Harmonic number. Thus the expected running time of this version of quicksort is $O(n \log n)$. The small class of $O(n^2)$ inputs in the deterministic version has been replaced in the probabilistic version: for any input there is a small probability that $O(n^2)$ comparisons will be required. However, this probability is not large enough to adversely affect the overall expected time, and so for any input the expected number of comparisons is the best-possible $O(n \log n)$.

3.1.2 Verifying Probabilistic Algorithms in Practice

As we saw in Section 1.3, probabilistic algorithms are hard to test using conventional methods. It may therefore not come as much of a surprise that they are also harder to implement correctly. A whole new class of errors become possible and one has to be mathematically sophisticated to avoid them.

For example, consider the example of the randomized quicksort algorithm introduced in the previous section. Suppose we have a faulty SML implementation of the algorithm, in which the pivot index is selected as follows:

```
fun select_pivot_index n =
  if n = 1 then 0
  else if coin_flip () then n - 1
  else select_pivot_index (n - 1)
```

The specification of randomized quicksort says that the pivot for the list $[a_0, \ldots, a_{n-1}]$ should be uniformly distributed (i.e., pick each element with equal probability), but this selection method is much more likely to pick a later element of the list than an earlier one. The effect of this bug is to reinstate a class of 'bad inputs' for the algorithm that result in $O(n^2)$ expected performance: one such bad input being the pre-sorted list.

In the context of a large program, this might be a difficult bug to find, even if it did significantly reduce the overall performance. The purpose of this chapter is to show how

formal methods may be applied to probabilistic algorithms, so that we can verify in a mechanical theorem prover that a program conforms to a probabilistic specification.

Recall from Section 1.4 that we model a probabilistic 'function' $\hat{f} : \alpha \to \beta$ with a HOL function

$$f : \alpha \to \mathbb{B}^\infty \to \beta \times \mathbb{B}^\infty$$

that explicitly passes around the sequence of random bits, using some in the computation and passing back the unused bits. In Chapter 2 we defined the probability space $(\mathcal{E}, \mathbb{P})$ which allows us write specifications for probabilistic algorithms. For example, we can formulate goals in the theorem prover such as

$$\forall\, n.\ \mathbb{P}\,\{s \mid \mathsf{fst}\ (\mathsf{prob\_program}\ n\ s) = \mathsf{failure}\} \leq 2^{-n} \tag{3.1}$$

to specify the operation of a certain prob_program. In a certain theoretical sense, this solves the problem of giving semantics to our modelling of probabilistic programs in higher-order logic. However, we are interested in mechanically verifying real probabilistic algorithms, and what is so far missing is a collection of proof techniques to make this as smooth as possible. Upon attempting a typical verification, two difficulties quickly emerge:

1. Since the probability measure $\mathbb{P}$ is only defined on events (i.e., sets in $\mathcal{E}$), much effort is spent proving that the sets that arise in verification really are events.

2. Many verification goals reduce to the form

$$\mathbb{P}(A_1 \cap A_2 \cap \cdots \cap A_n) = \mathbb{P}(A_1)\mathbb{P}(A_2) \cdots \mathbb{P}(A_n) \tag{3.2}$$

   where the $A_1, A_2, \ldots, A_n$ are events produced by 'probabilistic subroutines'. This result follows from the independence of the events, and so a theory of 'subroutine independence' is required.[6]

In addition, there is another deficiency with the theory up to this point: we have presented no syntactic support for expressing probabilistic programs. So far we have assumed they will be expressed directly in λ-calculus, explicitly passing around the bit sequence. To illustrate why this is not satisfactory, consider the probabilistic program dice which returns a number in the set $\{1, 2, 3, 4, 5, 6\}$ uniformly at random. The probabilistic program two_dice that uses dice to return the sum of two dice can be expressed in λ-calculus as[7]

$$\mathsf{two\_dice} = \lambda\, s.\ \mathsf{let}\ (x, s') \leftarrow \mathsf{dice}\ s\ \mathsf{in}\ \Big(\mathsf{let}\ (y, s'') \leftarrow \mathsf{dice}\ s'\ \mathsf{in}\ (x + y, s'')\Big)$$

This notation, as well as being rather unwieldy, contains opportunities for mistakes by confusing the three different versions of the random bit sequence. The ability to express our probabilistic programs in a cleaner way is not just an aesthetic issue of little logical significance. In fact, it clarifies the theory to the point where we can provide practical solutions to the two main verification problems listed above.

[6] This boils down to ensuring that random bits are never re-used by different subroutines: the side-conditions of the theorems we prove will enforce this property.

[7] It is true that the definition would be shorter if we had not used the syntactic sugar $\mathsf{let}\ v \leftarrow x\ \mathsf{in}\ f\ v$ for $(\lambda\, v.\ f\ v)\ x$. However, this only strengthens our basic point that better notation is required, since the shorter version is even more difficult to comprehend.

3.1.3 A Notation for Probabilistic Programs

A useful observation is that our modelling of probabilistic programs in higher-order logic by 'passing around the random-number generator' is how probabilistic programs are routinely written in pure functional languages such as Haskell[8] (Wadler, 1992; Launchbury and Jones, 1994).[9] In fact, probabilistic programs live in the more general state-transforming monad,[10] where the state that is transformed is the sequence of random bits. Definition 29 gives the monadic operators, which can be used to combine state-transforming programs with a minimum of notational clutter.

Definition 29 *The state-transformer monadic operators* unit *and* bind

$$\vdash \quad \forall a, s.\ \mathsf{unit}\ a\ s = (a, s) \tag{3.3}$$

$$\vdash \quad \forall f, g, s.\ \mathsf{bind}\ f\ g\ s = \mathsf{let}\ (x, s') \leftarrow f(s)\ \mathsf{in}\ g\ x\ s' \tag{3.4}$$

unit *is used to lift values to the monad, and* bind *is the monadic analogue of function application.*

For example, using unit and bind, our two_dice program can be expressed concisely as

$$\mathsf{two\_dice} = \mathsf{bind}\ \mathsf{dice}\ (\lambda x.\ \mathsf{bind}\ \mathsf{dice}\ (\lambda y.\ \mathsf{unit}\ (x + y)))$$

Note that the sequence is never referred to directly, instead the unit and bind operators pass it around behind the scenes.

In addition to providing an elegant notation in which to express probabilistic programs, unit and bind can be used to test whether a property is compositional on probabilistic programs.

Definition 30 *A property* $Q : (\mathbb{B}^\infty \to \alpha \times \mathbb{B}^\infty) \to \mathbb{B}$ *is compositional for probabilistic programs if*

1. *For every* $a : \alpha$,
$$Q(\mathsf{unit}\ a)$$
2. *For every* $f : \mathbb{B}^\infty \to \alpha \times \mathbb{B}^\infty$ *and* $g : \alpha \to \mathbb{B}^\infty \to \beta \times \mathbb{B}^\infty$,
$$Q(f)\ \wedge\ (\forall a.\ Q(g(a)))\ \Rightarrow\ Q(\mathsf{bind}\ f\ g)$$

Therefore, by writing our probabilistic programs in state-transformer notation, we can automatically break down compositional properties on the program to properties on the components. For example, if Q is a compositional property and $Q(\mathsf{dice})$ holds, then so does $Q(\mathsf{two\_dice})$.

In Section 3.2 we investigate the compositionality of the measurability and independence properties for probabilistic programs. Measurability allows us to deduce that sets

[8] `http://www.haskell.org`

[9] With hindsight, it is not surprising that we developed the same model independently, since it is a natural way to model state in an environment with no global variables.

[10] Monads are a notion from category theory, but happily no knowledge of category theory is required to use the notation to write probabilistic programs.

of sequences defined by probabilistic programs really are events. An example of this is the set

$$\{s \mid \mathsf{fst}\ (\mathsf{prob\_program}\ n\ s) = \mathsf{failure}\}$$

that was part of the example specification (3.1). This set of sequences is guaranteed to be an event if prob_program is measurable. Independence allows us to decompose goals containing the probability of an intersection (such as (3.2) above). The end result is a compositional property of probabilistic programs called indep_fn that implies measurability and independence.

3.1.4 Probabilistic Termination

Many probabilistic algorithms cannot be guaranteed to terminate on any input sequence of bits. For example, as we shall see in Section 4.3, no terminating algorithm exists to generate uniform random numbers in the range $\{0, 1, \ldots, n-1\}$ unless n is a power of 2. Thus if we insist on guaranteed termination, then even the dice program cannot be implemented. In practice then, the restriction of guaranteed termination is usually relaxed to allow termination with probability 1. This means that the set of sequences that cause a probabilistic program to terminate is an event and has probability 1.

In Section 3.3 we define a probabilistic while operator called prob_while, with characterizing theorem as shown in Definition 31.

Definition 31 *A Probabilistic While Loop*

$$\begin{aligned}\vdash\ & \forall\, c, b. \\ & (\forall\, a.\ b(a) \in \mathsf{indep\_fn})\ \wedge\ \mathsf{prob\_while\_terminates}\ c\ b\ \Rightarrow \\ & \forall\, a. \\ & \quad \mathsf{prob\_while}\ c\ b\ a \in \mathsf{indep\_fn}\ \wedge \\ & \quad \mathsf{prob\_while}\ c\ b\ a = \mathsf{if}\ c(a)\ \mathsf{then}\ \mathsf{bind}\ (b(a))\ (\mathsf{prob\_while}\ c\ b)\ \mathsf{else}\ \mathsf{unit}\ a\end{aligned} \tag{3.5}$$

The b parameter is a probabilistic program to advance the state, and has type $\alpha \to \mathbb{B}^\infty \to \alpha \times \mathbb{B}^\infty$. *The c parameter is a deterministic condition on the state that decides whether to perform another iteration, and has type* $\alpha \to \mathbb{B}$.

We also define a prob_while_terminates condition requiring that for every initial state a, the set of sequences that result in the condition c eventually being false is an event of probability 1. As Definition 31 shows, this termination condition holding means that prob_while $c\ b$ preserves the important indep_fn property of b.

The prob_while operator allows us to define in higher-order logic a large class of probabilistic algorithms, and the propagation of the indep_fn property allows us to assume useful results of measurability and independence for each program in the class. However, in practice it is difficult to verify that a particular probabilistic while loop satisfies its specification. This is because the usual proof techniques of program correctness rely on induction over the termination relation, but a probabilistic while loop is explicitly allowed to run forever on some input sequences (as long as the set of those input sequences has probability 0).

We remedy this in Section 3.3.3 by using a version of prob_while $c\ b$ called prob_while_cut $c\ b\ n$ containing an extra cut-off parameter n. In the cut-off version, if the condition c is still

true after n iterations of b then the loop terminates anyway. It is then possible to prove the theorem that if a property holds for $\mathsf{prob\_while\_cut}\ c\ b\ n$ with probability 1 (for every n), then it also holds for $\mathsf{prob\_while}\ c\ b$ with probability 1. The benefit of this to verification is that $\mathsf{prob\_while\_cut}\ c\ b\ n$ is a program that is guaranteed to terminate (in at most n iterations), so the standard program correctness techniques apply, most usefully induction on n. In this way some tricky probabilistic reasoning is reduced to a proof by induction. An example of this reduction can be found in Section 4.4, in the verification of a sampling algorithm for the Geometric($\frac{1}{2}$) distribution.

3.2 Measurability and Independence

3.2.1 Measurability

Recall from Section 2.2.3 that a function f is $\mathsf{measurable}\ \mathcal{F}\ \mathcal{G}$ if $\mathsf{preimage}\ f$ maps sets in $\mathcal{G}$ to sets in $\mathcal{F}$. To demonstrate the practical use of measurability, suppose that $f : \mathbb{B}^\infty \to \alpha \times \mathbb{B}^\infty$ models a probabilistic function, and that

$$A = \{x \mid \mathsf{snd}\ (f\ x) \in B\}$$

is a set that appears in the verification, where B is an event (i.e., is a member of $\mathcal{E}$: the set of events of the probability space). If we have that

$$(\mathsf{snd} \circ f) \in \mathsf{measurable}\ \mathcal{E}\ \mathcal{E}$$

then we can immediately deduce that $A \in \mathcal{E}$, since

$$A = \mathsf{preimage}\ (\mathsf{snd} \circ f)\ B$$

In practice, we would like to know that[11]

$$(\mathsf{fst} \circ f) \in \mathsf{measurable}\ \mathcal{E}\ \mathcal{U}\ \wedge\ (\mathsf{snd} \circ f) \in \mathsf{measurable}\ \mathcal{E}\ \mathcal{E} \tag{3.6}$$

so that we can additionally deduce that the preimage of *any* set of results of (the first component of) f is an event. We extend the definition of measurability to probabilistic programs by saying that f is measurable if it satisfies property (3.6).

It turns out that property (3.6) is compositional for probabilistic programs, in the sense of Definition 30. It is trivial that $\mathsf{unit}\ a$ is measurable for all a; the difficulty is showing that $\mathsf{bind}\ f\ g$ is measurable, given that $f : \mathbb{B}^\infty \to \alpha \times \mathbb{B}^\infty$ is measurable and $g(a) : \mathbb{B}^\infty \to \beta \times \mathbb{B}^\infty$ is measurable for every $a : \alpha$.

The task is illustrated in Figure 3.2. We aim to show that

$$\mathsf{fst} \circ \mathsf{bind}\ f\ g \in \mathsf{measurable}\ \mathcal{E}\ \mathcal{U}_{\mathcal{P}(\beta)}$$

[11] This is the same as requiring that

$$f \in \mathsf{measurable}\ \mathcal{E}\ (\mathcal{U} \otimes \mathcal{E})$$

where $\mathcal{U} \otimes \mathcal{E}$ is the σ-algebra generated by the product sets

$$\{A \times B \mid A \in \mathcal{U} \wedge B \in \mathcal{E}\}$$

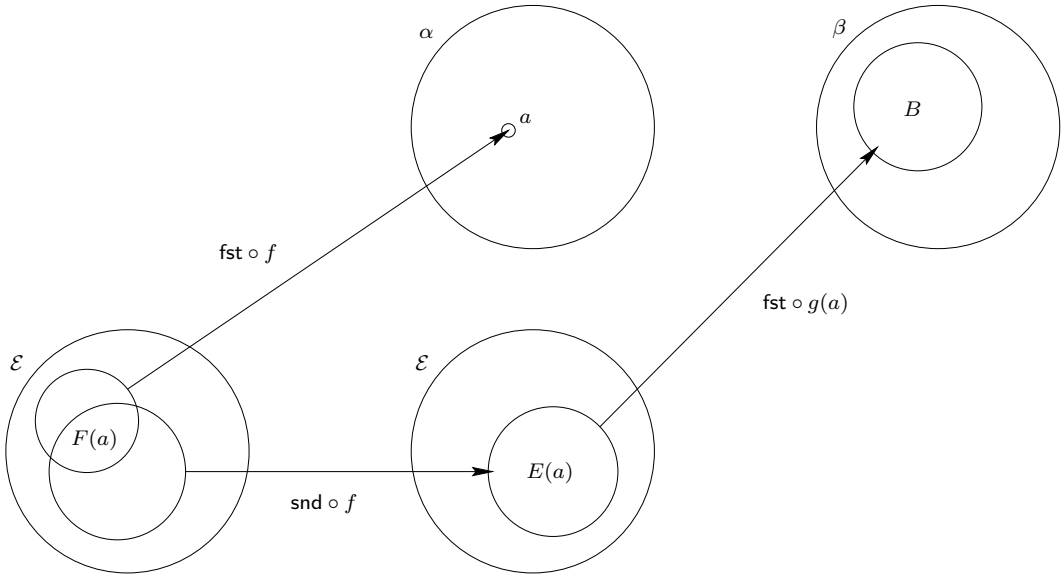


Figure 3.2: Is Measurability Compositional?

and so we take an arbitrary set $B \in \mathcal{U}_{\mathcal{P}(\beta)}$, and the goal is now to show that

$$\mathsf{preimage}\ (\mathsf{fst} \circ \mathsf{bind}\ f\ g)\ B$$

is an event (i.e., a set in $\mathcal{E}$). For each $a : \alpha$ the set

$$E(a) = \mathsf{preimage}\ (\mathsf{fst} \circ g(a))\ B$$

is an event, by the measurability of $g(a)$. Also, the set

$$F(a) = \mathsf{preimage}\ (\mathsf{fst} \circ f)\ \{a\} \cap \mathsf{preimage}\ (\mathsf{snd} \circ f)\ (E(a))$$

is an event, by the measurability of f. Finally, the desired preimage can be written as

$$\mathsf{preimage}\ (\mathsf{fst} \circ \mathsf{bind}\ f\ g)\ B = \bigcup_{a \in \mathcal{U}_\alpha} F(a)$$

Now we would like to deduce that any union of events is an event, and be done. Unfortunately, this property of a σ-algebra is only true for *countable* unions. A sufficient condition to carry the result then, is for the range of $\mathsf{fst} \circ f$ to be countable, and this follows from f being measurable by applying a result of Hansell (1971) in the theory of non-separable analytic sets. However, this result would be a difficult one to formalize, and so we instead make progress by strengthening the measurability property.

Definition 32 *A function $f : \mathbb{B}^\infty \to \alpha \times \mathbb{B}^\infty$ is strongly measurable if*

$$\begin{aligned} &\mathsf{countable}\ (\mathsf{range}\ (\mathsf{fst} \circ f))\ \wedge \\ &(\mathsf{fst} \circ f) \in \mathsf{measurable}\ \mathcal{E}\ \mathcal{U}\ \wedge\ (\mathsf{snd} \circ f) \in \mathsf{measurable}\ \mathcal{E}\ \mathcal{E} \end{aligned} \tag{3.7}$$

Since countability of range is easily shown to be a compositional property, the above argument shows that property (3.7) is compositional for probabilistic programs.

3.2.2 Function Independence

Recalling Definition 13 in Section 2.2.4 where we defined independence of both sets and families of sets, we now extend this to independence of a function.

Definition 33 *The set of independent functions* $f : \mathbb{B}^\infty \to \alpha \times \mathbb{B}^\infty$

$$\begin{aligned} \vdash\ & \mathsf{indep\_function} = \\ & \{f \mid \\ & \quad \mathsf{indep\_families} \\ & \quad (\mathsf{image}\ (\lambda\, A.\ \mathsf{preimage}\ (\mathsf{fst} \circ f)\ A)\ \mathcal{U}_{\mathcal{P}(\alpha)}) \\ & \quad (\mathsf{image}\ (\lambda\, E.\ \mathsf{preimage}\ (\mathsf{snd} \circ f)\ E)\ \mathcal{E})\} \end{aligned} \tag{3.8}$$

Whereas the measurability of a function $f : \mathbb{B}^\infty \to \alpha \times \mathbb{B}^\infty$ is really only a sanity check, independence corresponds to a very natural property of probabilistic programs. Consider once more the dice probabilistic program, which returns an element of $\{1, 2, 3, 4, 5, 6\}$ uniformly at random. Define the function pair_dice as follows:

$$\mathsf{pair\_dice} = \mathsf{bind}\ \mathsf{dice}\ (\lambda\, x.\ \mathsf{bind}\ \mathsf{dice}\ (\lambda\, y.\ \mathsf{unit}\ (x, y))) \tag{3.9}$$

Intuitively, we would expect that each of the 36 possible results

$$\{(1,1),\ (1,2),\ (1,3),\ (1,4),\ (1,5),\ (1,6),\ (2,1),\ \ldots,\ (6,6)\}$$

of pair_dice would be equally likely. But this reasoning relies on a subtle assumption of dice that was not explicitly stated: the returned result (i.e., dice roll) is independent of the returned sequence. If this is true then our reasoning is valid, since the second call to dice is not biased by the result of the first call. But consider this broken version of dice

$$\mathsf{broken\_dice} = \lambda\, s.\ (\mathsf{fst}\ (\mathsf{dice}\ s),\ s)$$

that always returns the sequence that is passed in. If pair_dice had called this instead of dice, then the independence assumption is not valid, and in fact, only 6 results out of the possible 36 can occur, namely

$$\{(1,1),\ (2,2),\ (3,3),\ (4,4),\ (5,5),\ (6,6)\}$$

each one having probability 1/6.

Independence then, allows us to decompose our probabilistic program, calculate the distribution of results for each individual subroutine, and then combine them with multiplication to give the distribution of results for the whole program.

Given this intended application for independence to reasoning about probabilistic programs, it is therefore a pity that the independence property of Definition 33 is not compositional. It is true that unit a is independent for every a, and also the sequence operations[12] sdest and scons b can be shown to be independent. But the following counterexample[13] proves that bind does not preserve independence:

$$\begin{aligned} f &= \mathsf{sdest} \\ g &= \lambda\, x.\ \mathsf{if}\ x\ \mathsf{then}\ (\lambda\, s.\ (\top, \mathsf{scons}\ \top\ s)\ \mathsf{else}\ (\lambda\, s.\ (\bot, \mathsf{scons}\ \bot\ s)) \end{aligned}$$

[12] Recall from Section 2.3.1 that shd, stl and scons are the sequence analogues of the list operations hd, tl and cons, and also sdest $s = (\mathsf{shd}\ s, \mathsf{stl}\ s)$.

[13] Thanks to David Preiss for this counterexample.

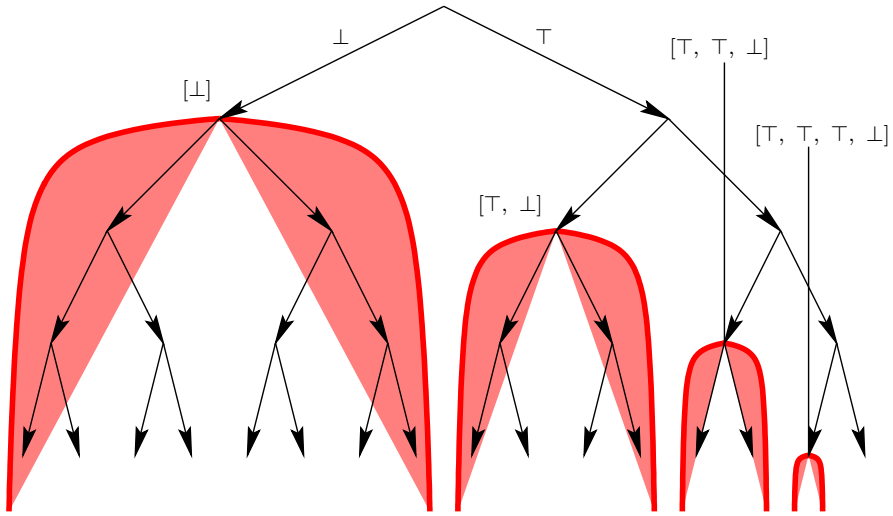


Figure 3.3: The Prefix Cover $\{[\bot],\ [\top,\ \bot],\ [\top,\ \top,\ \bot],\ \ldots\}$.

We can even prove the HOL theorem

$$\vdash\ f \in \mathsf{indep\_function}\ \wedge\ (\forall x.\ g(x) \in \mathsf{indep\_function}\ \wedge\ \mathsf{bind}\ f\ g \notin \mathsf{indep\_function} \tag{3.10}$$

In the next section we recover from this setback by demonstrating a compositional property that implies both function independence and measurability.

3.2.3 Strong Function Independence

We begin with the notion of a prefix cover: a set of non-overlapping prefixes that match almost every boolean sequence.

Definition 34 *Prefix Covers*

$$\vdash\ \forall C. \tag{3.11}$$
$$\mathsf{prefix\_cover}\ C =$$
$$(\forall l_1, l_2.\ l_1 \in C \wedge l_2 \in C \wedge l_1 \neq l_2 \Rightarrow \neg(\mathsf{is\_prefix}\ l_1\ l_2)\ \wedge$$
$$\mathbb{P}\Big(\bigcup_{l \in C} \mathsf{prefix\_set}\ l\Big) = 1 \tag{3.12}$$

Some examples of prefix covers are

$$\{[]\},\ \{[\top]\},\ [\bot]\},\ \{[\bot],\ [\top,\ \bot],\ [\top,\ \top,\ \bot],\ \ldots\}$$

(the last one being depicted in Figure 3.3). Some non-examples are

$$\{\},\ \{[\top]\},\ \{[],\ [\top]\},\ \{[\top],\ [\bot],\ [\top,\ \top]\}$$

We now define a property called indep_fn, subsuming both measurability and independence of a probabilistic function.

Definition 35 *Strong Function Independence*

$$
\begin{aligned}
\vdash\ & \mathsf{indep\_fn} = \\
& \{f \mid \\
& \quad \mathsf{countable}\ (\mathsf{range}\ (\mathsf{fst} \circ f))\ \wedge \\
& \quad (\mathsf{fst} \circ f) \in \mathsf{measurable}\ \mathcal{E}\ \mathcal{U}\ \wedge\ (\mathsf{snd} \circ f) \in \mathsf{measurable}\ \mathcal{E}\ \mathcal{E}\ \wedge \\
& \quad \exists C. \\
& \qquad \mathsf{prefix\_cover}\ C\ \wedge \\
& \qquad \forall l, s. \\
& \qquad\quad l \in C\ \wedge\ s \in \mathsf{prefix\_set}\ l\ \Rightarrow \\
& \qquad\quad f(s) = (\mathsf{fst}\ (f(\mathsf{prefix\_seq}\ l)), \mathsf{sdrop}\ (\mathsf{length}\ l)\ s)\}
\end{aligned}
\tag{3.13}
$$

Since strong function independence is one of the most important concepts of the thesis, let us take some time to understand its meaning. Firstly, it incorporates the measurability property discussed in Section 3.2.1, which we showed to be compositional. In addition, it says that there exists a prefix cover C, such that for each $l \in C$ the function $\mathsf{fst} \circ f$ is constant on $\mathsf{prefix\_set}\ l$, and the function $\mathsf{snd} \circ f$ is $\mathsf{sdrop}\ (\mathsf{length}\ l)$. This has a natural interpretation when we consider f as a probabilistic program: if f uses some bits from the sequence to compute its result, then *these bits must be removed from the sequence* (using sdrop) before the sequence is passed back.

In an imperative language (such as C) where the random number generator is implemented using global state, this property is automatically enforced. Whenever a random bit is asked for, the global state in the random number generator is advanced at the same time. However, modelling global state with state-transforming monads gives us the extra ability to roll back the state to an earlier time. For some notions of state this may be useful. But in the case where the state is a random number generator, the ability to roll back the state is nothing more than an opportunity to shoot oneself in the foot. Probabilistic programs are already hard to reason about, and re-using random numbers introduces dependencies that tremendously complicate the analysis. An example of this bad behaviour is the broken_dice program of Section 3.2.2: in this case the rolling-back of the random number generator produced a dice that always returned the same value!

Insisting that once random bits are used they are always thrown away would seem to imply function independence, because the result sequence can never contain any information about the result value. In fact, this can be made rigorous, yielding the first property of strong function independence:

$$\vdash\ \mathsf{indep\_fn} \subset \mathsf{indep\_function} \tag{3.14}$$

Note there is some subtlety here: a function that is indep_fn might behave badly on sequences that are not in the prefix cover. However, by the definition of prefix cover, the bad behaviour can only occur inside an event of probability 0, so does not affect the value of any probabilities.

The second property of strong function independence is that it is not the same as function independence. Setting

$$f = \lambda s.\ (\mathsf{shd}\ s = \mathsf{shd}\ (\mathsf{stl}\ s),\ \mathsf{stl}\ s)$$

we can prove the HOL theorems

$$\vdash \quad f \in \mathsf{indep\_function} \tag{3.15}$$

$$\vdash \quad f \notin \mathsf{indep\_fn} \tag{3.16}$$

This f is 'functionally equivalent' to sdest, returning a (coin-flip) boolean and a sequence that are independent of each other. The difference is that sdest satisfies strong function independence, but f uses two bits from the sequence and only throws one away, violating the property.

In practice, restricting ourselves to strongly independent functions does not cause any problems. The independent functions that we exclude (such as f) are pathological examples; by definition they cannot be coded in an imperative language with a global random number generator. In addition, as we showed in Section 3.2.2, function independence is not compositional, so we should expect that some independent functions must be sacrificed to make the property compositional. This is the third property of strong function independence.

Theorem 36 *Strong Function Independence is Compositional*

$$\vdash \quad \mathsf{sdest} \in \mathsf{indep\_fn} \tag{3.17}$$

$$\vdash \quad \forall a.\ \mathsf{unit}\ a \in \mathsf{indep\_fn} \tag{3.18}$$

$$\vdash \quad \forall f, g. \tag{3.19}$$
$$f \in \mathsf{indep\_fn} \ \wedge\ (\forall a.\ g(a) \in \mathsf{indep\_fn}) \ \Rightarrow\ \mathsf{bind}\ f\ g \in \mathsf{indep\_fn}$$

Proof: (Sketch) It is easily seen that the prefix cover $\{[\top],\ [\bot]\}$ provides the witness for sdest, and similarly $\{[\,]\}$ works for every unit a.

Turning our attention to bind: by hypothesis we have a prefix cover F for f, and for every a we have a prefix cover $G(a)$ for $g(a)$. It can be shown that the set

$$\bigcup_{l \in F} \bigcup_{l' \in G(\mathsf{fst}\ (f(\mathsf{prefix\_seq}\ l)))} \mathsf{append}\ l\ l'$$

is a prefix cover for bind f g. $\square$

This says that if we define a function f that accesses the random bit sequence using only the sdest, unit and bind primitives, then f is guaranteed to satisfy strong function independence. This already includes a significant number of real probabilistic programs, and we shall see in Chapter 5 that it is both possible and desirable to fit the Miller-Rabin probabilistic primality test into this class.

In addition, there are many properties that are satisfied by every strongly independent function, and if proved at this level of abstraction then we can instantiate them to any strongly independent function that appears in the verification. An example of such a property is the following theorem:

$$\begin{aligned}\vdash \quad & \forall f_1, f_2, g_1, g_2, B. \\ & f_1 \in \mathsf{indep\_fn} \ \wedge\ f_2 \in \mathsf{indep\_fn} \ \wedge \\ & (\forall a.\ g_1(a) \in \mathsf{indep\_fn}) \ \wedge\ (\forall a.\ g_2(a) \in \mathsf{indep\_fn}) \ \wedge \\ & (\forall a.\ \mathbb{P}\{s \mid \mathsf{fst}\ (f_1(s)) = a\} = \mathbb{P}\{s \mid \mathsf{fst}\ (f_2(s)) = a\}) \ \wedge \\ & (\forall a.\ \mathbb{P}\{s \mid \mathsf{fst}\ (g_1\ a\ s) \in B\} = \mathbb{P}\{s \mid \mathsf{fst}\ (g_2\ a\ s) \in B\}) \ \Rightarrow \\ & \mathbb{P}\{s \mid \mathsf{fst}\ (\mathsf{bind}\ f_1\ g_1\ s) \in B\} = \mathbb{P}\{s \mid \mathsf{fst}\ (\mathsf{bind}\ f_2\ g_2\ s) \in B\}\end{aligned} \tag{3.20}$$

From the conditions it is plain that strong function independence is essential to apply the theorem, which allows us to show that the probability of two bind events can be reduced to the probabilities of their component events. This came in useful during the verification of the sampling algorithm for the Binomial($n, \frac{1}{2}$) distribution (described in Section 4.2).

3.3 Probabilistic Termination

3.3.1 Probabilistic 'While' Loops

We now have all the machinery we need to define a probabilistic 'while' loop, where the body of the while loop is a probabilistic function

$$b : \alpha \to \mathbb{B}^\infty \to \alpha \times \mathbb{B}^\infty$$

that non-deterministically advances a state of type α, and the condition of the while loop is a deterministic state predicate

$$c : \alpha \to \mathbb{B}$$

We first define a version of the probabilistic while loop with a cut-off parameter n: if the condition is still true after n iterations, the loop terminates anyway.

Definition 37 *Probabilistic While with a Cut-off*

$$\begin{aligned} \vdash\ & \forall c, b, n, a. \\ & \quad \mathsf{prob\_while\_cut}\ c\ b\ 0\ a = \mathsf{unit}\ a\ \wedge \\ & \quad \mathsf{prob\_while\_cut}\ c\ b\ (\mathsf{suc}\ n)\ a = \\ & \quad \mathsf{if}\ c(a)\ \mathsf{then}\ \mathsf{bind}\ (b(a))\ (\mathsf{prob\_while\_cut}\ c\ b\ n)\ \mathsf{else}\ \mathsf{unit}\ a \end{aligned} \tag{3.21}$$

This cut-off version of probabilistic while does not employ probabilistic recursion. Rather it uses standard recursion on the cut-off parameter n, and consequently the following properties are easily established by induction on n:

$$\begin{aligned} \vdash\ & \forall c, b, n, a. \\ & \quad (\forall a.\ b(a) \in \mathsf{indep\_fn})\ \Rightarrow\ \mathsf{prob\_while\_cut}\ c\ b\ n\ a \in \mathsf{indep\_fn} \end{aligned} \tag{3.22}$$

$$\begin{aligned} \vdash\ & \forall c, b, n, a, p. \\ & \quad (\forall a.\ b(a) \in \mathsf{indep\_fn})\ \wedge\ (\forall a.\ \mathbb{P}\{s \mid c(\mathsf{fst}\ (b\ a\ s))\} \leq p)\ \Rightarrow \\ & \quad \mathbb{P}\{s \mid c(\mathsf{fst}\ (\mathsf{prob\_while\_cut}\ c\ b\ n\ a\ s))\} \leq p^n \end{aligned} \tag{3.23}$$

$$\begin{aligned} \vdash\ & \forall c, b, n, a. \\ & \quad \mathsf{prob\_while\_cut}\ c\ b\ (\mathsf{suc}\ n)\ a = \\ & \quad \mathsf{bind}\ (\mathsf{prob\_while\_cut}\ c\ b\ n\ a)\ (\lambda a'.\ \mathsf{if}\ c(a')\ \mathsf{then}\ b(a')\ \mathsf{else}\ \mathsf{unit}\ a') \end{aligned} \tag{3.24}$$

We now use prob_while_cut to make the 'raw definition' of prob_while

$$\begin{aligned} \vdash\ & \forall c, b, a, s. \\ & \quad \mathsf{prob\_while}\ c\ b\ a\ s = \\ & \quad \mathsf{if}\ \exists n.\ \neg c(\mathsf{fst}\ (\mathsf{prob\_while\_cut}\ c\ b\ n\ a\ s))\ \mathsf{then} \\ & \qquad \mathsf{prob\_while\_cut}\ c\ b \\ & \qquad (\mathsf{minimal}\ (\lambda n.\ \neg c(\mathsf{fst}\ (\mathsf{prob\_while\_cut}\ c\ b\ n\ a\ s))))\ a\ s \\ & \quad \mathsf{else}\ \mathsf{arb} \end{aligned} \tag{3.25}$$

where arb is an arbitrary fixed value. Our goal now is to prove the characterizing theorem of prob_while (given in Definition 31).

The first part of the characterizing theorem follows immediately.

Theorem 38

$$\vdash \ \forall c, b, a. \qquad \text{prob\_while } c\ b\ a = \text{if } c(a) \text{ then bind } (b(a)) \ (\text{prob\_while } c\ b) \text{ else unit } a \tag{3.26}$$

Proof: For given values of c, b, a and s, if there is some number of iterations of b (starting in state a with sequence s) that would lead to the condition c becoming false, then prob_while performs the minimum number of iterations that are necessary for this to occur, otherwise it returns arb. The proof now splits into the following cases:

- The condition eventually becomes false:
  - The condition is false to start with: in this case the minimum number of iterations for the condition to become false will be zero.
  - The condition is not false to start with: in this case the minimum number of iterations for the condition to become false will be greater than zero, and so we can safely perform an iteration and then ask the question again.
- The condition will always be true: therefore, after performing one iteration the condition will still always be true. So both LHS and RHS are equal to arb.

In each case theorem (3.26) follows. □

Suppose that for every state a the function $b(a)$ is strongly independent. Therefore, for each a there exists a prefix cover $B(a)$ for the function $b(a)$. Intending $C(a)$ to be a prefix cover for prob_while $c\ b\ a$, the family $C(a)$ is defined to be the least fixed point of the following rule induction:

$$\frac{\neg c(a)}{[\,] \in C(a)} \qquad \frac{c(a) \ \wedge \ l \in B(a) \ \wedge \ l' \in C(\text{fst } (b\ a\ (\text{prefix\_seq } l)))}{\text{append } l\ l' \in C(a)}$$

Example: Let the parameters of a probabilistic while loop be

$$\begin{aligned} c(a) &= a \\ b(a) &= \text{sdest} \\ B(a) &= \{[\top],\ [\bot]\} \end{aligned}$$

Noting that in our example we have

$$c(\top) \ \wedge \ \neg c(\bot) \ \wedge \ \text{fst } (b\ a\ (\text{prefix\_seq } l)) = \text{hd } l$$

let us use the first rule to assign

$$C(\top) = \emptyset, \quad C(\bot) = \{[\,]\}$$

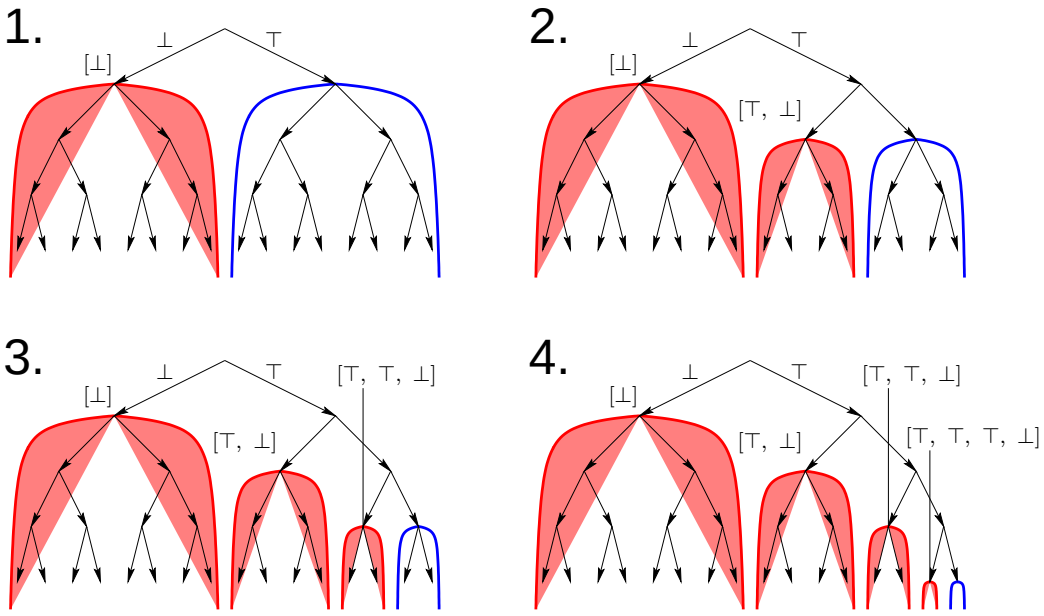


Figure 3.4: Iterations in the Rule Induction Example.

and try to find a fixed-point by using the second rule to add elements to these sets. After one iteration of rule induction, we have

$$C(\top) = \{[\bot]\}, \quad C(\bot) = \{[\,]\}$$

since $c(\top)$, $[\bot] \in B(\top)$ and $[\,] \in C(\mathsf{hd}\ [\bot])$. After two iterations we have

$$C(\top) = \{[\bot],\ [\top,\ \bot]\}, \quad C(\bot) = \{[\,]\}$$

since $c(\top)$, $[\top] \in B(\top)$ and $[\bot] \in C(\mathsf{hd}\ [\top])$. After three iterations we have

$$C(\top) = \{[\bot],\ [\top,\ \bot],\ [\top,\ \top,\ \bot]\}, \quad C(\bot) = \{[\,]\}$$

since $c(\top)$, $[\top] \in B(\top)$ and $[\top,\ \bot] \in C(\mathsf{hd}\ [\top])$. Continuing in this way, we end up with the prefix covers

$$C(\top) = \{[\bot],\ [\top,\ \bot],\ [\top,\ \top,\ \bot],\ \ldots\}, \quad C(\bot) = \{[\,]\}$$

This example is illustrated in Figure 3.4. □

The reader may have noticed that the above rule induction may result in sets $C(a)$ that are not prefix covers. An example of this occurs if the condition c is $\lambda\, a.\ \top$, when the least fixed point works out to be $C(a) = \emptyset$ for every a. It is essential for the union of sets in a prefix cover to be an event of probability 1, and for this to be true the following condition of c and b must hold:[14]

$$\forall\, a.\ \mathbb{P}\,\{s \mid \exists\, n.\ \neg c(\mathsf{fst}\ (\mathsf{prob\_while\_cut}\ c\ b\ n\ a\ s))\} = 1$$

[14]The set in this probability is guaranteed to be an event, since prob_while_cut c b n a satisfies strong function independence whenever b does.

This is plainly a (probabilistic) termination condition for the while loop, and so we wrap up the property into the following definition, which uses our probabilistic quantifiers (Section 2.4.4).

Definition 39

$$\begin{aligned} \vdash\ & \forall c, b. \\ & \quad \mathsf{prob\_while\_terminates}\ c\ b = \\ & \quad \forall a.\ \forall^* s.\ \exists n.\ \neg c(\mathsf{fst}\ (\mathsf{prob\_while\_cut}\ c\ b\ n\ a\ s)) \end{aligned} \tag{3.27}$$

This condition ensures that a prefix cover exists, and therefore that probabilistic while loops satisfy strong function independence:

$$\begin{aligned} \vdash\ & \forall c, b. \\ & \quad (\forall a.\ b(a) \in \mathsf{indep\_fn})\ \wedge\ \mathsf{prob\_while\_terminates}\ c\ b\ \Rightarrow \\ & \quad \forall a.\ \mathsf{prob\_while}\ c\ b\ a \in \mathsf{indep\_fn} \end{aligned} \tag{3.28}$$

and we have now justified the characterizing theorem of prob_while that we gave in Definition 31 (Section 3.1.4).

3.3.2 Probabilistic Termination Conditions

In the previous section we saw that a probabilistic termination condition was needed to prove that a probabilistic while loop satisfied strong function independence. In this section we take a closer look at this condition, in the context of related work on termination.

First, let us consider the deterministic case. As described by Slind (1999), we can use the HOL function definition package (called TFL) to define a while operator as follows:

$$\mathsf{while}\ c\ b\ a = \mathsf{if}\ c(a)\ \mathsf{then}\ \mathsf{while}\ c\ b\ (b(a))\ \mathsf{else}\ a$$

TFL automatically extracts the termination condition[15]

$$\exists R.\ \mathsf{WF}\ R\ \wedge\ \forall a.\ c(a) \Rightarrow R\ (b(a))\ a$$

where WF R means that the relation R is well-founded. This is equivalent to[16]

$$\forall a.\ \exists n.\ \neg c(\mathsf{funpow}\ b\ n\ a) \tag{3.29}$$

which is a deterministic version of our prob_while_terminates $c\ b$ condition (Definition 39). It can be easily seen that condition (3.29) is both necessary and sufficient for while $c\ b$ to terminate on every input a, and a corresponding theorem is true in the probabilistic case.

Theorem 40 *The condition* prob_while_terminates $c\ b$ *is both necessary and sufficient for each* prob_while $c\ b\ a$ *to terminate on a set of probability 1.*

[15]Note that TFL must be told to treat this definition as schematic in c and b, or else the extracted termination condition will be unprovable.

[16]To see the equivalence, the number n corresponds to the 'height' of a in R (well-defined since R is well-founded).

Although it is proper for our definition of probabilistic termination to be necessary and sufficient, in practice it is often easier to show the following simpler condition:

$$\begin{array}{l} \vdash \ \forall\, c, b. \\ \quad (\forall\, a.\ b(a) \in \mathsf{indep\_fn}) \ \wedge \ (\exists^* s.\ \forall\, a.\ \neg c(\mathsf{fst}\ (b\ a\ s))) \ \Rightarrow \\ \quad \mathsf{prob\_while\_terminates}\ c\ b \end{array} \tag{3.30}$$

This requires an event E with non-zero probability, such that for every state a, if the input sequence s comes from E then the loop will terminate after one more iteration.

It may seem plausible that we could instead use the condition

$$\forall\, a.\ \exists^* s.\ \neg c(\mathsf{fst}\ (b\ a\ s)))$$

so that for each state a we could use a different event $E(a)$ (each with non-zero probability). Unfortunately this does not quite work. A counterexample is the probabilistic while loop with parameters

$$\begin{array}{rcl} c(n,b) & = & b \\ b(n,b) & = & \mathsf{bind}\ (\mathsf{uniform}\ 2^n)\ (\lambda\, x.\ (n+1, 0 < x)) \end{array}$$

where $\mathsf{uniform}\ n$ returns a number from the set $\{0, \ldots, n-1\}$ uniformly at random (Section 4.3). From the start state (n, b), the (non-zero) probability that the loop will terminate after the next iteration is 2^{-n}, but the probability that the loop will ever terminate is

$$\mathsf{if}\ b\ \mathsf{then}\ \left(\sum_{m=n}^{\infty} 2^{-m}\right)\ \mathsf{else}\ 1$$

which works out to $\frac{1}{2}$ for the start state $(2, \top)$.

In the context of probabilistic concurrent systems, the following 0-1 law was proved by Hart et al. (1983):[17]

> Let process P be defined over a state space S, and suppose that from every state in some subset S' of S the probability of P's eventual escape from S' is at least p, for some fixed $0 < p$.
>
> Then P's escape from S' is certain, occurring with probability 1.

Identifying P with $\mathsf{prob\_while}\ c\ b$ and S' with the set of states a for which $c(a)$ holds, we can formulate the 0-1 law as an equivalent condition for probabilistic termination:

Theorem 41 *The 0-1 Law of Probabilistic Termination*

$$\begin{array}{l} \vdash \ \forall\, c, b. \\ \quad (\forall\, a.\ b(a) \in \mathsf{indep\_fn}) \ \Rightarrow \\ \quad (\mathsf{prob\_while\_terminates}\ c\ b \iff \\ \quad \exists\, p.\ 0 < p \ \wedge \ \forall\, a.\ p \leq \mathbb{P}\,\{s \mid \exists\, n.\ \neg c(\mathsf{fst}\ (\mathsf{prob\_while\_cut}\ c\ b\ n\ a\ s))\}) \end{array} \tag{3.31}$$

[17] This paraphrasing comes from Morgan (1996).

This interesting result implies that over the whole state space, the infimum of all the termination probabilities is either 0 or 1, it cannot lie properly in between. To see this, recall that prob_while_cut $c\ b$ means that all states terminate with probability 1, but the negation of the RHS means that there is a state that terminates with probability 0. An example of its use for proving probabilistic termination can be found in our verification of a sampling algorithm for the Bernoulli(p) distribution (Section 4.5).

Hart et al. (1983) also established a sufficient condition for probabilistic termination called the probabilistic variant rule. This has been used by Morgan (1996) to show the termination of a probabilistic self-stabilization algorithm of Herman (1990). We can formalize this as a sufficient condition for termination of our probabilistic while loops; the proof is relatively easy from the 0-1 law.

Theorem 42 *The Probabilistic Variant Condition*

$$\begin{aligned}\vdash\ & \forall c, b. \\ & (\forall a.\ b(a) \in \mathsf{indep\_fn})\ \wedge \\ & (\exists f, N, p. \\ & \quad 0 < p\ \wedge \\ & \quad \forall a.\ c(a)\ \Rightarrow\ f(a) < N\ \wedge\ p \leq \mathbb{P}\{s \mid f(\mathsf{fst}\ (b\ a\ s)) < f(a)\})\ \Rightarrow \\ & \mathsf{prob\_while\_terminates}\ c\ b\end{aligned} \tag{3.32}$$

As its name suggests, the probabilistic variant condition is a probabilistic analogue of the variant method used to prove termination of deterministic while loops. If we can assign to each state a a natural number measure from a finite set, and if each iteration of the loop has probability at least p of decreasing the measure, then probabilistic termination is assured. In addition, when $\{a \mid c(a)\}$ is finite, condition (3.32) has been shown to be necessary as well as sufficient.

3.3.3 Proof Techniques for Probabilistic While Loops

In this section we consider how to prove some properties of probabilistic while loops that typically arise in verification. From the way we defined prob_while we know that it satisfies strong function independence when it can be shown to terminate, and the mechanics of doing that were covered in the previous section. However, it turns out to be difficult in practice to show basic properties that are not immediate consequences of strong function independence.

Consider the following program G that lies at the heart of the Geometric($\frac{1}{2}$) sampling algorithm (Section 4.4):

$$\begin{aligned} C &= \lambda\,(a, m).\ a \\ B &= \lambda\,(a, m).\ \mathsf{bind}\ \mathsf{sdest}\ (\lambda\,a'.\ \mathsf{unit}\ (a', m+1)) \\ G &= \mathsf{prob\_while}\ C\ B \end{aligned}$$

The plausible property that G never decreases the second component of the state

$$\forall\, a, m, s.\ m \leq \mathsf{snd}\ (G\ (a, m)\ s)$$

is not provable. The reason is that there may be sequences (in this case $(\top, \top, \ldots)$ is the only example) on which G does not terminate, and so nothing can be said of its result. The correct property is

$$\forall\, a, m.\ \forall^* s.\ m \leq \mathsf{snd}\ (G\ (a, m)\ s)$$

but even this turns out to be hard to prove. Often when showing program correctness it is desirable to induct on the termination relation. However, this is not well-founded for our probabilistic while loops, which are explicitly allowed to run forever (so long as the set of sequences on which they terminate has probability 1). Continuing this train of thought, it sounds reasonable that we might be able to use induction on a termination relation to show properties that are true with probability 1, and indeed this is what the following theorem allows us to do.

Theorem 43 *Reduction Theorem for Probabilistic While Loops*

$$\begin{array}{ll} \vdash & \forall\, \phi, c, b, a. \\ & \quad (\forall\, a.\ b(a) \in \mathsf{indep\_fn})\ \wedge\ \mathsf{prob\_while\_terminates}\ c\ b\ \wedge \\ & \quad (\forall^* s.\ \forall\, n. \\ & \qquad \neg c(\mathsf{fst}\ (\mathsf{prob\_while\_cut}\ c\ b\ n\ a\ s)) \Rightarrow \\ & \qquad \phi(\mathsf{fst}\ (\mathsf{prob\_while\_cut}\ c\ b\ n\ a\ s)))\ \Rightarrow \\ & \quad \forall^* s.\ \phi(\mathsf{fst}\ (\mathsf{prob\_while}\ c\ b\ a\ s)) \end{array} \tag{3.33}$$

Proof: The proof uses the following chain of equalities:

$$\begin{array}{rcl} 1 & = & \mathbb{P}\,\{s \mid \exists\, n.\ \neg c(\mathsf{fst}\ (\mathsf{prob\_while\_cut}\ c\ b\ n\ a\ s))\} \\ & = & \mathbb{P}\left\{ s \;\middle|\; \exists\, n.\ \begin{array}{l} \neg c(\mathsf{fst}\ (\mathsf{prob\_while\_cut}\ c\ b\ n\ a\ s))\ \wedge \\ \phi(\mathsf{fst}\ (\mathsf{prob\_while\_cut}\ c\ b\ n\ a\ s)) \end{array} \right\} \\ & = & \mathbb{P}\left\{ s \;\middle|\; \exists\, n.\ \begin{array}{l} \phi(\mathsf{fst}\ (\mathsf{prob\_while}\ c\ b\ a\ s))\ \wedge \\ \phi(\mathsf{fst}\ (\mathsf{prob\_while\_cut}\ c\ b\ n\ a\ s))\ \wedge \\ \neg c(\mathsf{fst}\ (\mathsf{prob\_while\_cut}\ c\ b\ n\ a\ s)) \end{array} \right\} \\ & = & \mathbb{P}\left\{ s \;\middle|\; \exists\, n.\ \begin{array}{l} \phi(\mathsf{fst}\ (\mathsf{prob\_while}\ c\ b\ a\ s))\ \wedge \\ \neg c(\mathsf{fst}\ (\mathsf{prob\_while\_cut}\ c\ b\ n\ a\ s)) \end{array} \right\} \\ & = & \mathbb{P}\,\{s \mid \phi(\mathsf{fst}\ (\mathsf{prob\_while}\ c\ b\ a\ s))\} \end{array}$$

The first two and the last two probabilities are proved equal using the $\forall^*$ lifting theorem of Section 2.4.4. $\Box$

Now we may complete our example by applying the above reduction theorem, since the generated subgoal

$$\begin{array}{l} \forall^* s.\ \forall\, n. \\ \quad \neg c(\mathsf{fst}\ (\mathsf{prob\_while\_cut}\ C\ B\ n\ (a, m)\ s)) \Rightarrow \\ \quad m \leq (\mathsf{fst}\ (\mathsf{prob\_while\_cut}\ C\ B\ n\ (a, m)\ s)) \end{array}$$

is further broken down with an application of the quantifier exchange theorem (2.91) of

Section 2.4.4. This yields

$$\begin{array}{l}\mathsf{countable}\,\mathcal{U}_{\mathbb{N}}\ \wedge\\ \forall\, n.\ \forall^* s.\\ \quad \neg c(\mathsf{fst}\ (\mathsf{prob\_while\_cut}\ C\ B\ n\ (a,m)\ s)) \Rightarrow\\ \quad m \leq (\mathsf{fst}\ (\mathsf{prob\_while\_cut}\ C\ B\ n\ (a,m)\ s))\end{array}$$

which is dispatched by induction on n and standard techniques.

This example completely illustrates the technique of proving non-trivial properties of probabilistic while loops. The important point is that a difficult proof to reduce a property of unbounded iterations of prob_while to the bounded prob_while_cut need only be performed once, and packaged up as a higher-order theorem for future application.

One noteworthy special case of Theorem 43 is the 'post-condition' result of probabilistic while loops.

$$\begin{array}{ll}\vdash & \forall\, c, b, a. \\ & \quad (\forall\, a.\ b(a) \in \mathsf{indep\_fn})\ \wedge\ \mathsf{prob\_while\_terminates}\ c\ b\ \Rightarrow\\ & \quad \forall^* s.\ \neg c(\mathsf{fst}\ (\mathsf{prob\_while}\ c\ b\ a\ s))\end{array} \tag{3.34}$$

Finally, no formal definition of a new while loop would be complete without a Hoare-style while rule, and the following can also be proved from Theorem 43.

$$\begin{array}{ll}\vdash & \forall\, \phi, c, b, a. \\ & \quad (\forall\, a.\ b(a) \in \mathsf{indep\_fn})\ \wedge\ \mathsf{prob\_while\_terminates}\ c\ b\ \wedge\\ & \quad \phi(a)\ \wedge\ (\forall\, a.\ \forall^*\, s.\ \phi(a) \wedge c(a) \Rightarrow \phi(\mathsf{fst}\ (b\ a\ s)))\ \Rightarrow\\ & \quad \forall^* s.\ \phi(\mathsf{fst}\ (\mathsf{prob\_while}\ c\ b\ a\ s))\end{array} \tag{3.35}$$

3.3.4 Probabilistic 'Until' Loops

To complete this section on probabilistic looping, we shall show how a useful probabilistic *repeat... until* construct may be defined in terms of probabilistic while loops, and the properties that follow as consequences of the general theorems we have proved.

Definition 44 *Probabilistic Until Loop*

$$\vdash\ \forall\, b, c.\ \mathsf{prob\_until}\ b\ c = \mathsf{bind}\ b\ (\mathsf{prob\_while}\ (\neg \circ c)\ (\mathsf{K}\ b)) \tag{3.36}$$

For example, the program

$$\begin{array}{ll}\vdash & \mathsf{trichotomy} = \\ & \mathsf{prob\_until}\ (\mathsf{bind}\ \mathsf{sdest}\ (\lambda\, x.\ \mathsf{bind}\ \mathsf{sdest}\ (\lambda\, y.\ \mathsf{unit}\ (x,y))))\ (\lambda(x,y).\ x \vee y)\end{array} \tag{3.37}$$

will repeatedly extract pairs of bits from the random sequence until at least one of the pair is $\top$ (i.e., it will keep rejecting $(\bot, \bot)$).

The right termination condition for a prob_until program is not difficult to find. Since it has no notion of state, to ensure termination there must be a non-zero probability that each iteration of b will produce a result that is accepted by c. This is precisely the

sufficient condition (3.30) for probabilistic termination (Section 3.3.2). Using this, the following theorems fall out as special cases of probabilistic while:

$$\vdash \ \forall b, c. \quad b \in \mathsf{indep\_fn} \ \wedge \ (\exists^* s.\ c(\mathsf{fst}\ (b(s)))) \ \Rightarrow \ \mathsf{prob\_until}\ b\ c \in \mathsf{indep\_fn} \tag{3.38}$$

$$\vdash \ \forall b, c. \quad b \in \mathsf{indep\_fn} \ \wedge \ (\exists^* s.\ c(\mathsf{fst}\ (b(s)))) \ \Rightarrow \ \mathsf{prob\_until}\ b\ c = \mathsf{bind}\ b\ (\lambda x.\ \mathsf{if}\ c(x)\ \mathsf{then}\ \mathsf{unit}\ x\ \mathsf{else}\ \mathsf{prob\_until}\ b\ c) \tag{3.39}$$

$$\vdash \ \forall b, c. \quad b \in \mathsf{indep\_fn} \ \wedge \ (\exists^* s.\ c(\mathsf{fst}\ (b(s)))) \ \Rightarrow \ \forall^* s.\ c(\mathsf{fst}\ (\mathsf{prob\_until}\ b\ c\ s)) \tag{3.40}$$

In addition, we apply the reduction theorem for probabilistic while loops (Theorem 43) to calculate the result probabilities for probabilistic until loops.

Theorem 45 *Result Probability for Probabilistic Until Loops*

$$\vdash \ \forall A, b, c. \quad b \in \mathsf{indep\_fn} \ \wedge \ (\exists^* s.\ c(\mathsf{fst}\ (b(s)))) \ \Rightarrow \ \mathbb{P}\{s \mid \mathsf{fst}\ (\mathsf{prob\_until}\ b\ c\ s) \in A\} = \left(\frac{\mathbb{P}\{s \mid \mathsf{fst}\ (\mathsf{prob\_until}\ b\ c\ s) \in A \cap \{x \mid c(x)\}\}}{\mathbb{P}\{s \mid \mathsf{fst}\ (\mathsf{prob\_until}\ b\ c\ s) \in \{x \mid c(x)\}\}}\right) \tag{3.41}$$

If we specialize these general theorems to our simple trichotomy example, then its probability distribution is easily calculated:

$$\vdash \ \forall k.\ \mathbb{P}\{s \mid \mathsf{fst}\ (\mathsf{trichotomy}\ s) = k\} = \mathsf{if}\ k = (\bot, \bot)\ \mathsf{then}\ 0\ \mathsf{else}\ \tfrac{1}{3}$$

This intuitive result formally shows trichotomy to be a true sampling algorithm for the Uniform(3) distribution: an impossible task without using probabilistic termination (as we show in Section 4.3).

3.4 Execution in the Logic of Probabilistic Programs

3.4.1 Introduction to Execution in the Logic

The execution of programs within the logic is an essential component of many automatic proof procedures, and can also be useful to check that a new program behaves correctly on some simple examples. To finish off this chapter, we demonstrate how the formalization of a pseudo-random number generator allows us to execute probabilistic algorithms in the logic.

Execution within the logic takes place by making many passes of rewriting with the definitions of logical constants. For example, by rewriting with the following definition of list append

$$\vdash \forall h, t, l.\ \mathsf{append}\ [\,]\ l = l \ \wedge \ \mathsf{append}\ (h :: t)\ l = h :: (\mathsf{append}\ t\ l)$$

we can execute the 'program'

$$\mathsf{append}\ [1,2]\ [3,4,5]$$

which results in the theorem

$$\vdash \mathsf{append}\ [1,2]\ [3,4,5] = [1,2,3,4,5]$$

The ACL2 theorem prover (Kaufmann et al., 2000b) uses the underlying LISP interpreter to perform execution in the logic; this is efficient enough to simulate many cycles of a microprocessor (Kaufmann et al., 2000a). Execution in HOL can be carried out by using either the standard rewriting tools, or the `computeLib` tool of Barras (2000). This latter is guaranteed to match the complexity of execution in ML, though there is a large constant factor difference.[18]

3.4.2 Formalizing a Pseudo-random Bit Sequence

To execute probabilistic programs in the logic, we need a formal 'sequence of random bits' to feed in. Although there exist logics—such as the probabilistic logic of Nilsson (1986)—in which truly random objects can be formally defined, this is not supported by higher-order logic. We therefore use a pseudo-random bit sequence for this purpose instead, based on the linear congruence method of generating pseudo-random numbers (Knuth, 1997):

Definition 46 *A Pseudo-random Bit Sequence*

$$\vdash \quad \mathsf{pseudo}\ A\ B\ N = \mathsf{siter}\ \mathsf{even}\ (\lambda\, n.\ An + B \bmod (2N+1))$$

The following two properties are all we need for executing probabilistic programs in the logic, and these follow directly from the definition of siter (in Section 2.3.1).

$$\vdash \quad \forall\, A, B, N, n.\ \mathsf{shd}\ (\mathsf{pseudo}\ A\ B\ N\ n) = \mathsf{even}\ n \tag{3.42}$$

$$\vdash \quad \forall\, A, B, N, n. \tag{3.43}$$
$$\mathsf{stl}\ (\mathsf{pseudo}\ A\ B\ N\ n) = \mathsf{pseudo}\ A\ B\ N\ (An + B \bmod (2N+1))$$

Our method is this: after selecting appropriate parameters A, B, N and a starting seed i, we pass the sequence pseudo $A\ B\ N\ n$ as the argument of a probabilistic program. During execution, as random bits are required, there will appear terms of the form

$$\mathsf{shd}\ (\mathsf{pseudo}\ A\ B\ N\ n)$$
$$\mathsf{stl}\ (\mathsf{pseudo}\ A\ B\ N\ n)$$

These are immediately reduced using theorems (3.42) and (3.43). In this way the purely functional pseudo closely simulates an imperative pseudo-random number generator by carrying around its current state.[19]

[18] Barras calculates the constant factor to be about 1000 for his merge-sort example.

[19] This avoids the efficiency disaster in which to calculate the 100th iteration of the state we must calculate the 99th, the 98th, etc., all the way back to the beginning.

Example: Choosing the parameters

$$A = 103 \qquad B = 95 \qquad N = 79$$

and initial seed 0, we can execute trichotomy in the logic (defined in Section 3.3.4). This is performed by using `computeLib` to rewrite with: the definitions of bind, unit and sdest; the reduction theorems (3.42) and (3.43) above; various boolean simplifications; and the following theorem for trichotomy:

$$\begin{aligned} \vdash\ & \forall s. \\ & \mathsf{trichotomy}\ s = \\ & \mathsf{bind}\ \mathsf{sdest} \\ & \quad (\lambda x.\ \mathsf{bind}\ \mathsf{sdest}\ (\lambda y.\ \mathsf{if}\ x \vee y\ \mathsf{then}\ \mathsf{unit}\ (x, y)\ \mathsf{else}\ \mathsf{trichotomy}))\ s \end{aligned} \tag{3.44}$$

By threading the pseudo-random sequence through many iterations of trichotomy, we can produce the following chain of theorems:

$$\begin{aligned}
\vdash\ & \mathsf{trichotomy}\ (\mathsf{pseudo}\ 103\ 95\ 75\ 0) &&= ((\top, \bot),\ \mathsf{pseudo}\ 103\ 95\ 75\ 65) \\
\vdash\ & \mathsf{trichotomy}\ (\mathsf{pseudo}\ 103\ 95\ 75\ 65) &&= ((\bot, \top),\ \mathsf{pseudo}\ 103\ 95\ 75\ 33) \\
\vdash\ & \mathsf{trichotomy}\ (\mathsf{pseudo}\ 103\ 95\ 75\ 33) &&= ((\top, \bot),\ \mathsf{pseudo}\ 103\ 95\ 75\ 94) \\
\vdash\ & \mathsf{trichotomy}\ (\mathsf{pseudo}\ 103\ 95\ 75\ 94) &&= ((\top, \bot),\ \mathsf{pseudo}\ 103\ 95\ 75\ 107) \\
\vdash\ & \mathsf{trichotomy}\ (\mathsf{pseudo}\ 103\ 95\ 75\ 107) &&= ((\top, \top),\ \mathsf{pseudo}\ 103\ 95\ 75\ 2) \\
\vdash\ & \mathsf{trichotomy}\ (\mathsf{pseudo}\ 103\ 95\ 75\ 2) &&= ((\top, \top),\ \mathsf{pseudo}\ 103\ 95\ 75\ 143) \\
\vdash\ & \mathsf{trichotomy}\ (\mathsf{pseudo}\ 103\ 95\ 75\ 143) &&= ((\bot, \top),\ \mathsf{pseudo}\ 103\ 95\ 75\ 55) \\
\vdash\ & \mathsf{trichotomy}\ (\mathsf{pseudo}\ 103\ 95\ 75\ 55) &&= ((\bot, \top),\ \mathsf{pseudo}\ 103\ 95\ 75\ 96) \\
\vdash\ & \mathsf{trichotomy}\ (\mathsf{pseudo}\ 103\ 95\ 75\ 96) &&= ((\top, \bot),\ \mathsf{pseudo}\ 103\ 95\ 75\ 34) \\
\vdash\ & \mathsf{trichotomy}\ (\mathsf{pseudo}\ 103\ 95\ 75\ 34) &&= ((\top, \top),\ \mathsf{pseudo}\ 103\ 95\ 75\ 32) \\
& \text{etc...}
\end{aligned}$$

□

We emphasize that linear congruential pseudo-random bit sequences are merely a convenient way to generate superficially unpredictable bits, and are of course completely deterministic. We might equally well test our probabilistic programs on the sequence where every element is $\top$, but this is not even superficially unpredictable. The point is well made by von Neumann (1963):

> Any one who considers arithmetical methods of producing random digits is, of course, in a state of sin. For, as has been pointed out several times, there is no such thing as a random number—there are only methods to produce random numbers, and a strict arithmetical procedure is of course not such a method.

3.4.3 Execution as an Automatic Proof Procedure

If a deterministic program terminates, the result of the computation will be unique. However, for non-deterministic programs—including probabilistic programs—there may be many possible results depending on how the internal choices were resolved during the computation. Therefore, to get useful results from execution in the logic, we must make use of theorems about probabilistic programs that hold for *every* input sequence of bits.

By inspection of the theorems we have proved about probabilistic programs, it looks to be true that programs using probabilistic termination generally have properties that are quantified by $\forall^*$ instead of the stronger $\forall$.[20] Here we see a practical effect of this: $\forall$-quantified theorems apply to every input sequence (and so include our pseudo-random bit sequence), but $\forall^*$-quantified theorems apply only to an unspecified set of sequences having probability 1 (and so might not include our pseudo-random bit sequence). Therefore, we are more likely to be able to deduce useful properties by executing programs in the logic that do not use probabilistic termination.

For example, consider the following properties of a hypothetical probabilistic program:

$$\forall n.\ \forall^* s.\ \phi(n) \ \Rightarrow\ \psi(\mathsf{fst}\ (\mathsf{prob\_program}\ n\ s)) \tag{3.45}$$

$$\forall n.\ \forall s.\ \phi(n) \ \Rightarrow\ \psi(\mathsf{fst}\ (\mathsf{prob\_program}\ n\ s)) \tag{3.46}$$

Property (3.45) would typically be all that we can prove if prob_program employs probabilistic termination, whereas the stronger property (3.46) might be provable if prob_program does not. Suppose we execute

$$\mathsf{prob\_program}\ 10\ \mathsf{pseudorandom}$$

in the logic for some pseudo-random bit sequence pseudorandom, and in doing so discover that

$$\neg\psi(\mathsf{fst}\ (\mathsf{prob\_program}\ 10\ \mathsf{pseudorandom}))$$

With the stronger property (3.46) we can apply the law of contrapositives to deduce the potentially useful result $\neg\phi(10)$, but this does not follow from the weaker property (3.45).

3.5 Concluding Remarks

The main result of this chapter is the development of an infrastructure for the practical verification of probabilistic algorithms. This builds upon our formalization of probability theory in Chapter 2, and will be applied in Chapters 4 and 5 to verify some example probabilistic programs.

The semantics of probabilistic programs was first tackled by Kozen (1979), and developed by Jones (1990), He et al. (1997) and Morgan et al. (1995). This line of research extends the predicate transformer idea of Dijkstra (1976) in which programs are regarded as functions: they take a set of desired end results to the set of initial states from which the program is guaranteed to produce one of these final states. With the addition of probabilistic choice, the 'sets of states' must be generalized to functions from states to the real interval $[0, 1]$. Jones applies the semantics to proving completeness of a Hoare logic defined on a little while language equipped with probabilistic choice. It would be interesting to embed this language in higher-order logic, and use our probability theory to derive the same proof rules.

The probabilistic dynamic logic of Feldman and Harel (1984) is another extension of Kozen's initial work. The essential concepts of their system are the same as ours; programs operate on a sequence of random variables; the analysis is measure-theoretic,

[20] This is observed to be the case in Section 4.3, where we compare implementations of Uniform(n) sampling algorithms with and without probabilistic termination.

and cylinders (our prefix sets) play an important role. One difference from our model is that random variables can come from arbitrary probability distributions, but we provide only the Bernoulli($\frac{1}{2}$) distribution as primitive (others may derived by using sampling functions). In addition, they support reasoning about partial functions, but lack our probabilistic quantifiers. Their resulting logic is a two-level extension of first-order logic, with an axiomatized proof system. Our logic is simpler, by virtue of being a definitional extension of higher-order logic.

There are many application domains in which it is natural to use continuous distributions, and the approach laid out in this chapter adapts very well to algorithms that access the continuous distributions using infinite precision real arithmetic. In this way of looking at things, our infinite sequence of samples from Bernoulli($\frac{1}{2}$) is a single sample from Uniform$[0, 1]$. Knuth and Yao (1976) show how to use a sequence of random bits to efficiently generate bits of many more continuous distributions, and it would be interesting to verify some simple algorithms along these lines.

Finally, we examine our underlying assumption that probabilistic programs have access to a source of Bernoulli($\frac{1}{2}$) random bits. In terms of whether this is sufficient to model all probabilistic programs, Gill (1977) showed that the computational power of a language with probabilistic choice $\vee_p$ is the same as a language with $\vee_{\frac{1}{2}}$, so long as p is computable. In the other direction, do infinite sequences of IID Bernoulli($\frac{1}{2}$) random variables exist in the real world? Leaving aside the more difficult independence issues, von Neumann (1963) gives a neat trick to obtain a precisely fair coin (sequence of IID Bernoulli($\frac{1}{2}$) random variables) from any biased coin (sequence of IID Bernoulli(p) random variables[21]). Flip the biased coin twice, and if the result is HT then output H, if the result is TH then output T, and if the result is HH or TT then output nothing. Independent of how the coin is biased, we know that the probability of HT must be equal to the probability of TH, so if we repeat this procedure long enough then an infinite sequence of fair coin flips will emerge.

[21] We must have $0 < p < 1$ for this to work.

Chapter 4

Example Probabilistic Programs

To illustrate our framework for verifying probabilistic algorithms in a theorem prover, we prove the correctness of several example programs. Most of these are sampling algorithms for various probability distributions, which support the development of more sophisticated probabilistic programs. We also define the symmetric simple random walk, since it represents a complicated case of probabilistic termination, and verify the 'optimal dice' from a paper by Knuth and Yao (1976).

4.1 Introduction

In Chapter 2 we formalized probability theory in higher-order logic, to give meaning to specifications of probabilistic programs. In Chapter 3 we introduced a convenient notation in which to express implementations of probabilistic programs in higher-order logic. In addition, we defined some program constructs (such as prob_while) and properties (such as indep_fn) and proved a body of theorems about them. On the strength of all this, we claimed to have developed a framework in which it was practical to verify probabilistic programs in a theorem prover. This chapter constitutes evidence substantiating our claim. We implement in higher-order logic several useful probabilistic programs, and formally verify their correctness.

The probabilistic algorithms that we initially choose to verify are used to generate samples from different probability distributions, given a source of random bits. These sampling algorithms are low-level probabilistic programs, useful in many programming contexts where a certain kind of probabilistic behaviour is required. For example, some hardware simulation software may need a particular distribution of inputs to test a circuit, but only has access to random bits from a pseudo-random number generator or from the operating system.[1] Typically, sampling algorithms can be implemented as small probabilistic programs, but involve some interesting reasoning about probability to prove their correctness. They thus represent ideal examples to test the verification framework we have laid in place. In addition, once we have verified a suite of sampling algorithms for different distributions, we can use them as subroutines in higher-level textbook probabilistic algorithms. Chapter 5 contains a case study where we do just this: using a Uniform(n)

[1]The latest Verilog standard (IEEE Standards Department, 2001) includes some continuous distribution sampling algorithms for just such a purpose. To further motivate our own work, in one draft of the standard the χ^2 distribution sampling algorithm was found to be wrong!

sampling algorithm in an implementation of the Miller-Rabin probabilistic primality test. The proof of correctness of the Uniform(n) sampling algorithm is then instrumental in the proof of correctness of the Miller-Rabin primality test.

Despite being essential subroutines in the implementation of probabilistic algorithms, sampling algorithms often lie beneath the considerations of textbook presentations. The following excerpt from *Randomized Algorithms* (Motwani and Raghavan, 1995, page 6) assumes a typical position:

> We define a randomized algorithm as an algorithm that is allowed access to a source of independent, unbiased, random bits; it is then permitted to use these random bits to influence its computation. ... While we will usually not worry about the conversion of random bits to the required distribution, the reader should keep in mind that random bits are a resource whose use involves a non-trivial cost. Moreover, there is sometimes a non-trivial computational overhead associated with sampling from a seemingly well-behaved distribution. For example, consider the problem of using a source of unbiased bits to sample uniformly from a set S whose cardinality is not a power of 2.

The formal consideration of this type of problem is the main focus of the present chapter.

The first sampling algorithm we verify, in Section 4.2, generates samples from the Binomial($n, \frac{1}{2}$) distribution. This does not use probabilistic termination, and so we are able to demonstrate the application of our formal verification framework in the absence of this complicating factor.

In Section 4.3 we verify two algorithms to sample from the Uniform(n) distribution. The first uses probabilistic termination and returns each of n different results with probability exactly $1/n$. The second has the property of guaranteed termination, but is only an approximate algorithm. An additional natural number parameter t must be passed in, and the probability p_i of the result $0 \leq i \leq n-1$ satisfies

$$|p_i - 1/n| \leq 2^{-t}$$

Implementing essentially the same algorithm twice—with and without probabilistic termination—allows us to compare the effect of using probabilistic termination on the formal properties of the algorithm.

In Section 4.4 we give a sampling algorithm for the Geometric($\frac{1}{2}$) distribution. This example stimulated the development of the probabilistic while loop, and demonstrates that our method is expressive enough to model discrete distributions over an infinite set. For each natural number n, the probability that the Geometric($\frac{1}{2}$) sampling algorithm returns n is $(\frac{1}{2})^{n+1}$.

The last sampling algorithm that we verify, in Section 4.5, samples from the Bernoulli(p) distribution. Since p can be any real number between 0 and 1, both the (probabilistic) termination and correctness of this program represents a challenge.

In addition to these sampling algorithms, we verify two more probabilistic algorithms that illustrate different aspects of our formal framework. Firstly, in Section 4.6, we verify two algorithms given by Knuth and Yao (1976) for generating dice rolls and sums of two dice rolls. These algorithms are optimal in the number of random bits that they are expected to consume, but their correctness is not a simple matter of inspection. The algorithms are interesting because they are expressed as finite state machines with probabilistic transitions. Properties of such machines are routinely established by probabilistic

model checkers (such as Prism[2]), and showing how the verification proceeds in our framework sheds light on the connection between the two different methods.

Finally, in Section 4.7, we define a variant of the symmetric simple random walk. Starting at level n, at each step the random walk either moves up a level or down a level with probability $\frac{1}{2}$ each. If it ever hits level 0, then the random walk finishes and returns the numbers of steps taken. This probabilistic program is interesting because the proof that it terminates with probability 1 is not at all obvious.

4.2 The Binomial($n, \frac{1}{2}$) Distribution

The first sampling algorithm we will present samples from the Binomial($n, \frac{1}{2}$) distribution, where a Binomial(n, p) random variable is the sum of n independent Bernoulli(p) random variables (DeGroot, 1989, page 100). We can thus define a simple sampling algorithm for the Binomial($n, \frac{1}{2}$) distribution by counting the number of $\top$'s in the first n bits of our sequence.

Definition 47 *The* Binomial($n, \frac{1}{2}$) *Sampling Algorithm*

$$\begin{aligned}\vdash\ & \mathsf{prob\_binomial}\ 0 = \mathsf{unit}\ 0\ \wedge \\ & \forall n. \\ & \quad \mathsf{prob\_binomial}\ (\mathsf{suc}\ n) = \\ & \quad \mathsf{bind}\ (\mathsf{prob\_binomial}\ n) \\ & \quad (\lambda m.\ \mathsf{bind}\ \mathsf{sdest}\ (\lambda b.\ \mathsf{unit}\ (\mathsf{if}\ b\ \mathsf{then}\ \mathsf{suc}\ m\ \mathsf{else}\ m)))\end{aligned} \tag{4.1}$$

By induction on n, it is easy to show that this sampling algorithm satisfies strong function independence:

$$\vdash\ \forall n.\ \mathsf{prob\_binomial}\ n \in \mathsf{indep\_fn} \tag{4.2}$$

However, due to an interesting aspect of our definition of prob_binomial, it is not quite so easy to show that it has the correct probability distribution. If we had phrased the definition as

$$\ldots \mathsf{prob\_binomial}\ (\mathsf{suc}\ n) = \mathsf{bind}\ (\mathsf{sdest}\ (\lambda b.\ \mathsf{bind}\ (\mathsf{prob\_binomial}\ n) \ldots$$

so that we extract an element from the sequence and then make the recursive call (instead of the other way around), then the probability distribution theorem follows by induction on n. Therefore we proceed using the following bind commutativity theorem to show that the two definitions result in events of the same probability:

$$\begin{aligned}\vdash\ & \forall f, g, h, A. \\ & \quad f \in \mathsf{indep\_fn}\ \wedge\ g \in \mathsf{indep\_fn}\ \wedge\ (\forall x, y.\ h\ x\ y \in \mathsf{indep\_fn})\ \Rightarrow \\ & \quad \mathbb{P}(\mathsf{preimage}\ (\mathsf{fst} \circ \mathsf{bind}\ f\ (\lambda x.\ \mathsf{bind}\ g\ (\lambda y.\ h\ x\ y)))\ A) = \\ & \quad \mathbb{P}(\mathsf{preimage}\ (\mathsf{fst} \circ \mathsf{bind}\ g\ (\lambda y.\ \mathsf{bind}\ f\ (\lambda x.\ h\ x\ y)))\ A)\end{aligned} \tag{4.3}$$

[2] `http://www.cs.bham.ac.uk/~dxp/prism/`

This is proved using the associativity of bind[3] and theorem (3.20) relating the probability of two bind events (Section 3.2.3).

Finally, we may deduce that prob_binomial has the expected distribution:

$$\vdash \quad \forall n, r.\ \mathbb{P}\{s \mid \mathsf{fst}\ (\mathsf{prob\_binomial}\ n\ s) = r\} = \binom{n}{r}\left(\tfrac{1}{2}\right)^n \tag{4.5}$$

Incidentally, although the Binomial$(n, \frac{1}{2})$ example is particularly easy to sample with a source of random bits, it is not without intrinsic interest. The central limit theorem (DeGroot, 1989, page 275) implies that for large n, the sampling algorithm

$$\mathsf{bind}\ (\mathsf{prob\_binomial}\ n)\ \left(\lambda m.\ \mathsf{unit}\ \left(\frac{2m-n}{\sqrt{n}}\right)\right)$$

is a good approximation to the normal distribution with mean 0 and standard deviation 1: the most important distribution in statistics.

4.3 The Uniform(n) Distribution

In this section we are interested in sampling from the Uniform(n) distribution, which assigns equal probability to each element in the set $\{0, \ldots, n-1\}$. Supposing $n-1$ has k bits, we approach the problem with an algorithm that returns the number represented by the first k random bits.

Definition 48 *A Sampling Algorithm for* Uniform(2^k)

$$\begin{aligned}\vdash \quad & \forall n. \\ & \mathsf{prob\_unif}\ n = \\ & \mathsf{if}\ n = 0\ \mathsf{then}\ \mathsf{unit}\ 0\ \mathsf{else} \\ & \quad \mathsf{bind}\ (\mathsf{prob\_unif}\ (n\ \mathsf{div}\ 2)) \\ & \quad (\lambda m.\ \mathsf{bind}\ \mathsf{sdest}\ (\lambda b.\ \mathsf{unit}\ (\mathsf{if}\ b\ \mathsf{then}\ 2m+1\ \mathsf{else}\ 2m)))\end{aligned} \tag{4.6}$$

Since prob_unif makes no use of probabilistic termination, the following properties may be proved by (complete) induction on its argument.

$$\vdash \quad \forall n.\ \mathsf{prob\_unif}\ n \in \mathsf{indep\_fn} \tag{4.7}$$

$$\begin{aligned}\vdash \quad & \forall n, k. \\ & (n = 0 \wedge k = 0) \vee 2^{k-1} \leq n < 2^k \Rightarrow \\ & \forall m.\ \mathbb{P}\{s \mid \mathsf{fst}\ (\mathsf{prob\_unif}\ n\ s) = m\} = \mathsf{if}\ m < 2^k\ \mathsf{then}\ 2^{-k}\ \mathsf{else}\ 0\end{aligned} \tag{4.8}$$

This says that prob_unif n samples from a Uniform(m) distribution, but that m can be up to twice as large as n. Before proceeding to the correct sampling algorithm for Uniform(n),

[3] This is one of the laws that must be satisfied by all monads:

$$\vdash \quad \forall f, g, h.\ \mathsf{bind}\ f\ (\lambda x.\ \mathsf{bind}\ (g(x))\ h) = \mathsf{bind}\ (\mathsf{bind}\ f\ g)\ h \tag{4.4}$$

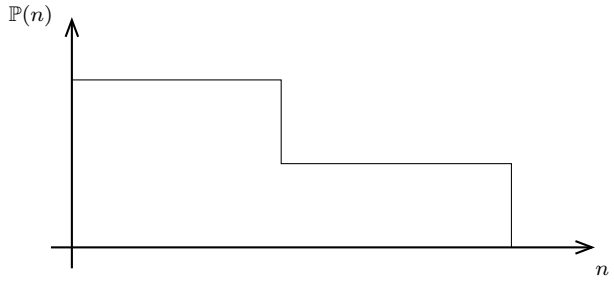


Figure 4.1: The Distribution Resulting from broken_prob_uniform.

it is instructive to consider the following broken version:

$$\begin{aligned}&\mathsf{broken\_prob\_uniform}\ n = \\ &\mathsf{bind}\ (\mathsf{prob\_unif}\ (n-1))\ (\lambda\, m.\ \mathsf{unit}\ (\mathsf{if}\ n \leq m\ \mathsf{then}\ m-n\ \mathsf{else}\ m))\end{aligned}$$

This produces the distribution as shown in Figure 4.1, whereas the ideal distribution is completely flat. The numbers close to zero have twice the probability of numbers close to $n-1$, because two results of prob_unif n map to the smaller numbers, but only one to the larger numbers. Now we give the correct version.

Definition 49 *A Sampling Algorithm for the* Uniform(n) *Distribution*

$$\begin{aligned}\vdash\ & \forall\, n. \\ & \mathsf{prob\_uniform}\ (\mathsf{suc}\ n) = \mathsf{prob\_until}\ (\mathsf{prob\_unif}\ n)\ (\lambda\, x.\ x < \mathsf{suc}\ n)\end{aligned} \tag{4.9}$$

This prob_uniform function uses a probabilistic until loop to repeatedly evaluate prob_unif, stopping when prob_unif returns a number in the correct range. It is deliberately left undefined on an argument of zero, since the distribution Uniform(0) is an ill-formed concept. Showing termination is easy, since the probability that prob_unif returns 0 is 2^{-k} for some k, and so there is a non-zero probability that the loop will terminate on each iteration. We may therefore apply the general prob_until theorems of Section 3.3.4 to deduce the following properties of prob_uniform:

$$\vdash\ \forall\, n.\ \mathsf{prob\_uniform}\ (\mathsf{suc}\ n) \in \mathsf{indep\_fn} \tag{4.10}$$

$$\vdash\ \forall\, n.\ \forall^{*} s.\ \mathsf{fst}\ (\mathsf{prob\_uniform}\ (\mathsf{suc}\ n)\ s) < \mathsf{suc}\ n \tag{4.11}$$

$$\begin{aligned}\vdash\ & \forall\, n, m. \\ & m < n\ \Rightarrow\ \mathbb{P}\,\{s \mid \mathsf{fst}\ (\mathsf{prob\_uniform}\ n\ s) = m\} = 1/n\end{aligned} \tag{4.12}$$

We have proved that prob_uniform is a correct sampling algorithm for Uniform(n). Its use of probabilistic termination is necessary, since an algorithm that is guaranteed to terminate within N steps can only access N random bits. Therefore if a particular result is produced in q different configurations of these N bits, then it will have probability $q/2^N$. Since $1/n$ can be written as $q/2^N$ only when n is a power of 2, a sampling algorithm for Uniform(n) that is guaranteed to terminate can only exist when n is a power of 2.

However, the downside of using probabilistic termination is that the number of random bits that will be required cannot be bounded in advance. Since random bits are

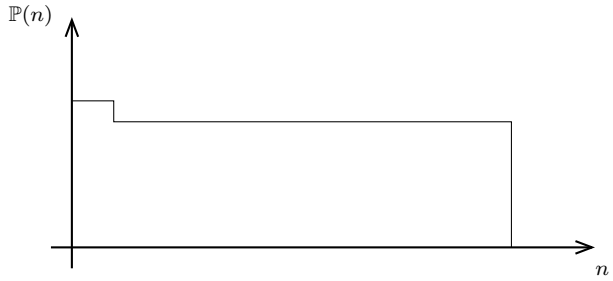


Figure 4.2: The Distribution Resulting from prob_uniform_cut.

not a free resource, this may introduce unacceptable overhead into a calculation. Consequently, most random number libraries in programming languages do not generate completely uniform random numbers (see Knuth (1997)), but instead try to reduce any bias to an acceptable level.[4] For these reasons, we also give a sampling algorithm for an 'approximately-Uniform(n)' distribution that is guaranteed to terminate.

Definition 50 *An Approximately-*Uniform(n) *Sampling Algorithm*

$$\begin{aligned}\vdash\ & \forall t, n. \\ & \quad \mathsf{prob\_uniform\_cut}\ 0\ (\mathsf{suc}\ n) = \mathsf{unit}\ 0\ \wedge \\ & \quad \mathsf{prob\_uniform\_cut}\ (\mathsf{suc}\ t)\ (\mathsf{suc}\ n) = \\ & \qquad \mathsf{bind}\ (\mathsf{prob\_unif}\ n) \\ & \qquad (\lambda m.\ \mathsf{if}\ n \leq m\ \mathsf{then}\ \mathsf{prob\_uniform\_cut}\ t\ (\mathsf{suc}\ n)\ \mathsf{else}\ \mathsf{unit}\ m)\end{aligned} \tag{4.13}$$

This prob_uniform_cut function takes an additional parameter t, and evaluates prob_unif at most t times to get a number in the correct range. If this never happens, then it returns 0. It therefore results in an approximately-uniform distribution that slightly favours 0, depicted in Figure 4.2.

Noting that each of the t independent calls to prob_unif gives a better than $\frac{1}{2}$ probability of returning a number in the correct range, it is possible to prove the following theorems about the behaviour of prob_uniform_cut:

$$\vdash\ \forall t, n.\ \mathsf{prob\_uniform\_cut}\ t\ (\mathsf{suc}\ n) \in \mathsf{indep\_fn} \tag{4.14}$$

$$\vdash\ \forall t, n, s.\ \mathsf{fst}\ (\mathsf{prob\_uniform\_cut}\ t\ (\mathsf{suc}\ n)\ s) < \mathsf{suc}\ n \tag{4.15}$$

$$\begin{aligned}\vdash\ & \forall t, n, m. \\ & \quad m < n\ \Rightarrow\ |\mathbb{P}\,\{s \mid \mathsf{fst}\ (\mathsf{prob\_uniform\_cut}\ n\ s) = m\} - 1/n| \leq 2^{-t}\end{aligned} \tag{4.16}$$

It is interesting to compare the two sets of theorems that we can prove about prob_uniform and prob_uniform_cut. Both satisfy strong function independence, a basic result of probabilistic programs that allows them to be composed in a tractable way. As we discussed, the version with probabilistic termination can yield precisely correct probabilities, whereas the version without can only get arbitrarily close. Theorems (4.11) and (4.15) show the

[4]When the bias is significantly less than the probability of a machine error, this is certainly acceptable for all practical purposes.

ranges of the two algorithms, but where the $\forall^*$ quantifier in the theorem for prob_uniform says that the range being correct is an event of probability 1, the corresponding theorem for prob_uniform_cut is much stronger: every input sequence gives a result in the correct range.

The conclusion is that if we can implement an algorithm satisfying a specification without using probabilistic termination, then we can expect it to satisfy stronger properties. In Chapter 5 we exploit this logical consequence of probabilistic termination, slightly tweaking the usual textbook Miller-Rabin algorithm so that it satisfies the same specification, but calls prob_uniform_cut instead of prob_uniform. Our implementation is then guaranteed to terminate, requires a bounded number of random bits, and satisfies strong versions of some key properties.

4.4 The Geometric($\frac{1}{2}$) Distribution

In any reasonable model of computation with access to a sequence of random bits, only a finite number of bits from the sequence can be read in finite time. Thus in no reasonable model of computation will we be able to calculate the function: *return the index of the last* $\top$ *in the sequence, or* 0 *if there is no last* $\top$. Therefore if a distribution has non-zero probability on an infinite number of points, it cannot be modelled without using probabilistic termination. Such is the case with the Geometric($\frac{1}{2}$) distribution, where a Geometric(p) random variable is defined to be the index of the first success in an infinite sequence of Bernoulli(p) trials (DeGroot, 1989, page 260). In our model, the Geometric($\frac{1}{2}$) distribution may be sampled by extracting random bits from the sequence, stopping as soon as the first $\bot$ is encountered, and returning the number of $\top$'s extracted.

Definition 51 *A Sampling Algorithm for the* Geometric($\frac{1}{2}$) *Distribution*

$$\vdash \forall b, n. \quad \mathsf{prob\_geometric\_iter}\ (b, n) = \mathsf{bind}\ \mathsf{sdest}\ (\lambda b'.\ \mathsf{unit}\ (b', \mathsf{suc}\ n)) \tag{4.17}$$

$$\vdash \mathsf{prob\_geometric} = \mathsf{bind}\ (\mathsf{prob\_while}\ \mathsf{fst}\ \mathsf{prob\_geometric\_iter}\ (\top, 0))\ (\lambda\,(b, n).\ \mathsf{unit}\ (n-1)) \tag{4.18}$$

Here the state is a pair, the first component containing the last random bit, and the second the number of bits seen so far. This is initialized to $(\top, 0)$ and updated by the probabilistic while loop until the first component becomes $\bot$, at which point the result is the second component (subtracting one because we do not count the final $\bot$).

Termination again follows by using the sufficient condition (3.30) established in Section 3.3.2, and so we may deduce strong function independence:

$$\vdash \mathsf{prob\_geometric} \in \mathsf{indep\_fn} \tag{4.19}$$

An application of the reduction theorem for while loops (Theorem 43 in Section 3.3.3) allows us to show that prob_geometric samples from the correct distribution:

$$\vdash \forall n.\ \mathbb{P}\{s \mid \mathsf{fst}\ (\mathsf{prob\_geometric}\ s) = n\} = (\tfrac{1}{2})^{n+1} \tag{4.20}$$

In contrast with the theorems we proved about prob_uniform, we do not need a theorem about the range of prob_geometric, since the correct range is the entirety of $\mathbb{N}$.

4.5 The Bernoulli(p) Distribution

The final distribution that we sample from is the Bernoulli(p) distribution. This distribution is over the boolean values $\{\top, \bot\}$, and models a test where $\top$ is picked with probability p and $\bot$ with probability $1 - p$. Our sequence of random bits each come from a Bernoulli($\frac{1}{2}$) distribution, and the present goal is to use these to sample from the Bernoulli(p) distribution, where p is any real number between 0 and 1.[5]

The sampling algorithm we use is based on a simple intuition. Suppose the binary expansion of p is $0.p_0p_1p_2\cdots$, and consider the bits of the random sequence s as forming a binary expansion $0.s_0s_1s_2\cdots$: in this way s can also be regarded as a real number between 0 and 1. Since the 'number' s is uniformly distributed between 0 and 1, the probability that s is less than p is p. Therefore, an algorithm that returns the result $\top$ if s is less than p, and $\bot$ otherwise, will be sampling from the Bernoulli(p) distribution. This question can be easily decided by looking at the binary expansions, and the matter is further simplified since we can ignore awkward cases (such as $s = p$) that occur with probability 0.

Definition 52 *A Sampling Algorithm for the* Bernoulli(p) *Distribution*

$$
\begin{array}{ll}
\vdash & \forall p. \\
& \quad \mathsf{prob\_bernoulli\_iter}\ p = \\
& \quad\quad \mathsf{bind\ sdest} \\
& \quad\quad (\lambda\, b. \\
& \quad\quad\quad \mathsf{unit} \\
& \quad\quad\quad (\mathsf{if}\ p < \tfrac{1}{2}\ \mathsf{then\ if}\ b\ \mathsf{then\ inl}\ (2p)\ \mathsf{else\ inr}\ \bot \\
& \quad\quad\quad\ \mathsf{else\ if}\ b\ \mathsf{then\ inr}\ \top\ \mathsf{else\ inl}\ (2p-1))) \\
\vdash & \forall p. \\
& \quad \mathsf{prob\_bernoulli}\ p = \\
& \quad \mathsf{bind}\ (\mathsf{prob\_while\ is\_inr}\ (\mathsf{prob\_bernoulli\_iter} \circ \mathsf{outl})\ (\mathsf{inl}\ p))\ (\mathsf{unit} \circ \mathsf{outr})
\end{array}
$$

So that the sampling algorithm implementation fits neatly into a probabilistic while loop, it makes heavy use of the HOL sum type $\alpha + \beta$, having constructors $\mathsf{inl}, \mathsf{inr}$, destructors $\mathsf{outl}, \mathsf{outr}$ and predicates $\mathsf{is\_inl}, \mathsf{is\_inr}$. However, the intent of the probabilistic while loop is simply to evaluate $s \leq p$ by iteration on the bits of s:

- if $\mathsf{shd}\ s = \bot$ and $\frac{1}{2} \leq p$, then return $\top$;
- if $\mathsf{shd}\ s = \top$ and $p \leq \frac{1}{2}$, then return $\bot$;
- if $\mathsf{shd}\ s = \bot$ and $p \leq \frac{1}{2}$, then repeat with $s := \mathsf{stl}\ s$ and $p := 2p$;
- if $\mathsf{shd}\ s = \top$ and $\frac{1}{2} \leq p$, then repeat with $s := \mathsf{stl}\ s$ and $p := 2p - 1$.

[5] The problem for reals is harder than for rationals, since if $p = m/n$ then the following sampling algorithm trivially works:

$$\mathsf{bind}\ (\mathsf{prob\_uniform}\ n)\ (\lambda\, k.\ \mathsf{unit}\ (k < m))$$

This method of evaluation has two important properties: firstly, it is obviously correct since the scaling operations on p just have the effect of removing its leading bit; secondly, probabilistic termination holds, since every iteration has a probability $\frac{1}{2}$ of terminating the loop. Indeed, Hart's 0-1 law of termination (Theorem 41) provides a convenient method of showing probabilistic termination:

$$\vdash \quad \mathsf{prob\_while\_terminates\ isl\ (prob\_bernoulli\_iter} \circ \mathsf{outl)} \tag{4.21}$$

From this follows strong function independence

$$\vdash \quad \forall p.\ \mathsf{prob\_bernoulli}\ p \in \mathsf{indep\_fn} \tag{4.22}$$

and we can then formulate an alternative definition of prob_bernoulli:

$$\begin{aligned}
&\vdash \quad \forall p. \\
&\quad \mathsf{prob_bernoulli}\ p = \\
&\quad \mathsf{bind\ sdest} \\
&\quad (\lambda\, b. \\
&\qquad \mathsf{if}\ b\ \mathsf{then}\ (\mathsf{if}\ p \leq \tfrac{1}{2}\ \mathsf{then\ unit}\ \bot\ \mathsf{else\ prob_bernoulli}\ (2p-1)) \\
&\qquad \mathsf{else}\ (\mathsf{if}\ p \leq \tfrac{1}{2}\ \mathsf{then\ prob_bernoulli}\ (2p)\ \mathsf{else\ unit}\ \top))
\end{aligned} \tag{4.23}$$

This definition of prob_bernoulli is more readable, closer to the intuitive version, and easier to use in proofs. We use this to prove the correctness theorem:

$$\vdash \quad \forall p.\ 0 \leq p\ \wedge\ p \leq 1\ \Rightarrow\ \mathbb{P}\{s \mid \mathsf{prob\_bernoulli}\ p\ s\} = p \tag{4.24}$$

The proof of this is quite simple, once the right idea is found. The idea is to show that the probability gets within $(\frac{1}{2})^n$ of p, for an arbitrary natural number n. This occurs after n iterations, as can be shown by induction.

It may be slightly counter-intuitive that for any real number $0 \leq p \leq 1$ we can generate an event of probability p, using a probabilistic program that with probability 1 will only look at a finite number of random bits. However, it might be helpful to look at this in the following way: for any two real numbers $0 \leq p, q \leq 1$, the probability that prob_bernoulli p and prob_bernoulli q follow the same execution path is $|p - q|$. Therefore, for p and q that differ only after position n in their binary expansions, prob_bernoulli p and prob_bernoulli q will differ with probability less $(\frac{1}{2})^n$.

Note that now we have samples from the Bernoulli(p) distribution, it would be simple to define other standard distributions parameterized by real p. For example, just as we based our Binomial($n, \frac{1}{2}$) and Geometric($\frac{1}{2}$) sampling algorithms on samples from a Bernoulli($\frac{1}{2}$) distribution, we could similarly define Binomial(n, p) and Geometric(p) sampling algorithms based on Bernoulli(p) samples.

4.6 Optimal Dice

A paper of Knuth and Yao (1976) shows that many probability distributions can be sampled using only a sequence of random bits. One of these is a dice program that requires on average $3\frac{2}{3}$ coin flips to produce a dice roll, and this is proved to be optimal.[6] The procedure is depicted in Figure 4.3.

[6]This optimality is very strong: if p_n is the probability that the program requires more than n rolls to produce a result, then each p_n is minimal over all dice programs.

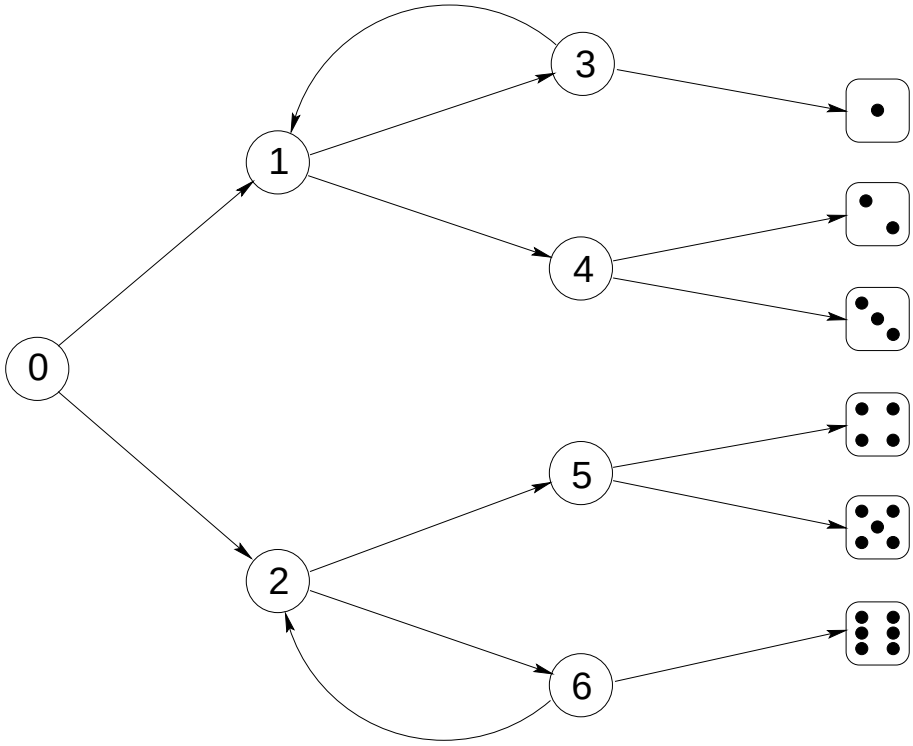


Figure 4.3: An Optimal Way to Sample Dice Rolls Using Coin Flips.

The idea is that one starts at the root node and repeatedly tosses a coin, taking a upper branch whenever the coin produces H, and a lower branch whenever the coin produces T. When a box is reached the procedure is finished, the result being the contents of the box. This way of representing probabilistic programs is called a DDG-tree,[7] and any probabilistic program using a source of random bits can be written as a (possibly infinite) DDG-tree.

We can model finite DDG-trees as HOL probabilistic programs, and then verify that they produce the correct probability distribution. The procedure is illustrated on the dice DDG-tree. Starting at the root node (node 0 in Figure 4.3), we make a coin_flip to decide whether to move to node 1 or node 2. Suppose the coin produces a H and so we move to node 1. Since there is an arrow that returns to node 1 from later in the tree, we declare this loop with a prob_repeat at this point. This means that the nodes after this point must return an option type, and the prob_repeat construct will repeatedly try the subtree until a some x is returned (it strips off the some using the to create its result). Node 1 also contains a coin_flip, suppose that another H is produced so that we move to node 3. There is another coin_flip here: if it is a H then we return a none to force another try from node 1; and if it is a T then we return some 1. This is the complete model we use, except for the following piece of notational convenience. If we have reached node 4, then we know we will never go back to node 1, and so we insert a mmap some so that whatever we return will automatically be wrapped with a some before it is passed further up the tree.[8] We now give the precise definitions of these constructs and the HOL version of the

[7] DDG stands for Discrete Distribution Generating.

[8] It may be helpful to think of prob_repeat as 'opening a bracket' that must be closed on every succeeding

dice DDG-tree.

Definition 53 *Constructs for Modelling DDG-Trees*

$$\vdash \forall a, b.\ \mathsf{coin\_flip}\ a\ b = \mathsf{bind\ sdest}\ (\lambda x.\ \mathsf{if}\ x\ \mathsf{then}\ a\ \mathsf{else}\ b) \tag{4.25}$$

$$\vdash \forall a.\ \mathsf{prob\_repeat}\ a = \mathsf{mmap\ the}\ (\mathsf{prob\_until}\ a\ \mathsf{is\_some}) \tag{4.26}$$

$$\vdash \forall f, m.\ \mathsf{mmap}\ f\ m = \mathsf{bind}\ m\ (\mathsf{unit} \circ f) \tag{4.27}$$

Definition 54 *An Optimal* dice *Program*

$$\begin{aligned}
\vdash\ & \mathsf{dice} = \\
& \mathsf{coin_flip} \\
& (\mathsf{prob_repeat} \\
& \quad (\mathsf{coin_flip} \\
& \quad\quad (\mathsf{coin_flip}\ (\mathsf{unit\ none})\ (\mathsf{unit}\ (\mathsf{some}\ 1))) \\
& \quad\quad (\mathsf{mmap\ some}\ (\mathsf{coin_flip}\ (\mathsf{unit}\ 2)\ (\mathsf{unit}\ 3)))) \\
& \quad (\mathsf{coin_flip} \\
& \quad\quad (\mathsf{mmap\ some}\ (\mathsf{coin_flip}\ (\mathsf{unit}\ 4)\ (\mathsf{unit}\ 5))) \\
& \quad\quad (\mathsf{coin_flip}\ (\mathsf{unit}\ (\mathsf{some}\ 6))\ (\mathsf{unit\ none}))))
\end{aligned} \tag{4.28}$$

Using the techniques we have introduced for verifying probabilistic programs, it is possible to prove that the dice program returns the right probability distribution:

$$\begin{aligned}
\vdash\ & \forall n. \\
& \mathbb{P}\,\{s \mid \mathsf{fst}\ (\mathsf{dice}\ s) = n\} = \mathsf{if}\ 1 \leq n \wedge n \leq 6\ \mathsf{then}\ \tfrac{1}{6}\ \mathsf{else}\ 0
\end{aligned} \tag{4.29}$$

We now turn our attention to the problem of generating the sum of two dice. The probabilistic program

$$\vdash \mathsf{two\_dice} = \mathsf{bind\ dice}\ (\lambda a.\ \mathsf{bind\ dice}\ (\lambda b.\ \mathsf{unit}\ (a + b))) \tag{4.30}$$

will do this, and we can even prove that it returns the right distribution:

$$\begin{aligned}
\vdash\ & \forall n. \\
& \mathbb{P}\,\{s \mid \mathsf{fst}\ (\mathsf{two_dice}\ s) = n\} = \\
& \mathsf{if}\ n = 2 \vee n = 12\ \mathsf{then}\ \tfrac{1}{36}\ \mathsf{else\ if}\ n = 3 \vee n = 11\ \mathsf{then}\ \tfrac{2}{36} \\
& \mathsf{else\ if}\ n = 4 \vee n = 10\ \mathsf{then}\ \tfrac{3}{36}\ \mathsf{else\ if}\ n = 5 \vee n = 9\ \mathsf{then}\ \tfrac{4}{36} \\
& \mathsf{else\ if}\ n = 6 \vee n = 8\ \mathsf{then}\ \tfrac{5}{36}\ \mathsf{else\ if}\ n = 7\ \mathsf{then}\ \tfrac{6}{36}\ \mathsf{else}\ 0
\end{aligned} \tag{4.31}$$

Since dice requires on average $3\frac{2}{3}$ coin flips, it follows that two_dice will require $7\frac{1}{3}$. It might also be thought that since dice was the optimal way to generate a single dice roll, two_dice would be the best way to generate the sum of two dice. However, Knuth and Yao (1976) show that this is not so, and in fact the optimal DDG-tree depicted in Figure 4.4 requires only $4\frac{7}{18}$ coin flips on the average.

branch, either by returning an option type at a leaf node or by inserting an mmap some inside a branch.

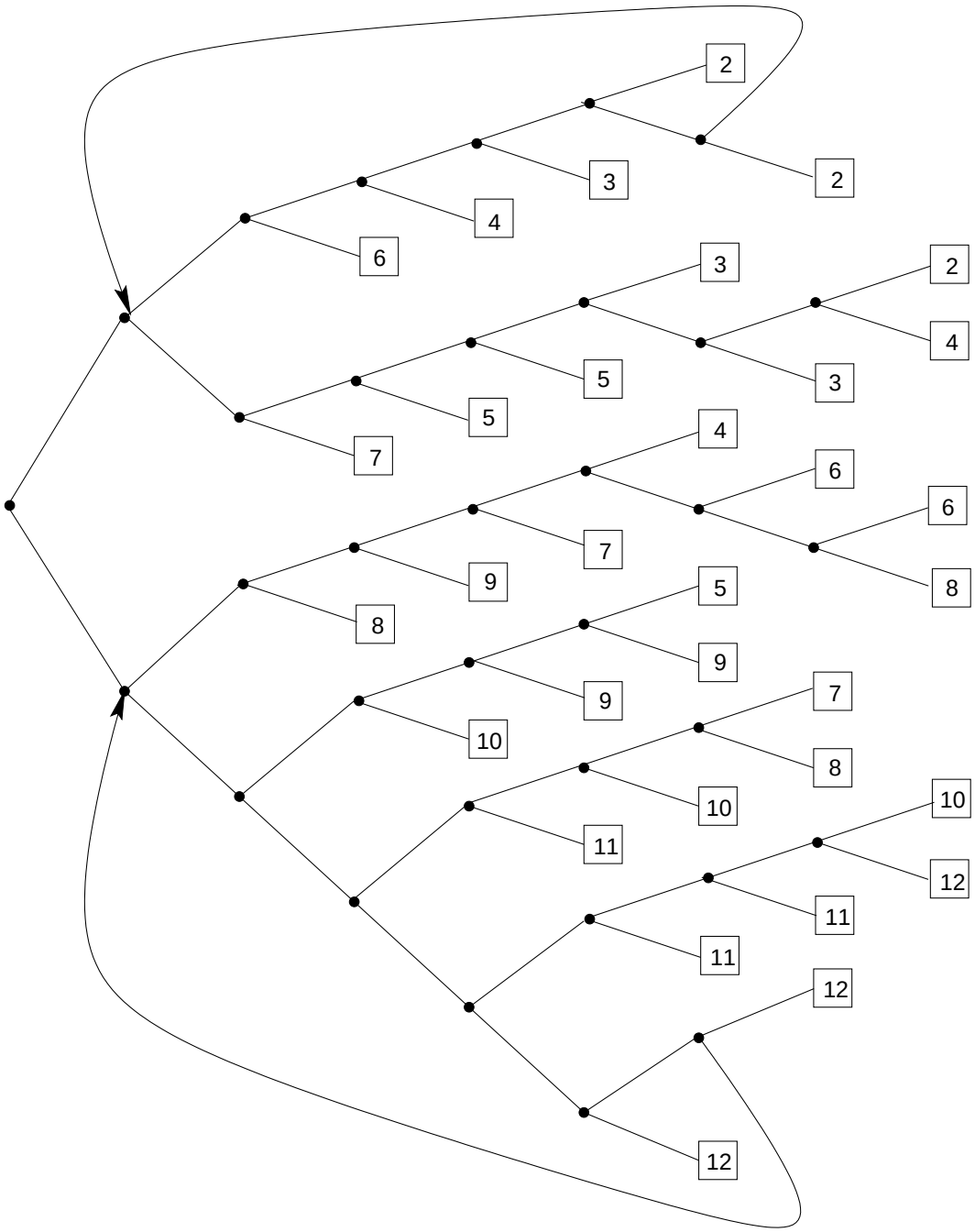


Figure 4.4: An Optimal Way to Sample the Sum of Two Dice Rolls.

Using our constructs for modelling DDG-trees, we can formalize the optimal DDG-tree for generating the sum of two dice, and even prove that it yields the same probability distribution as the straightforward version two_dice we gave above:

$$\vdash \forall n. \quad \mathbb{P}\{s \mid \mathsf{fst}\ (\mathsf{optimal\_two\_dice}\ s) = n\} = \mathbb{P}\{s \mid \mathsf{fst}\ (\mathsf{two\_dice}\ s) = n\} \tag{4.32}$$

The HOL version of this theory (including the full 77-line definition of optimal_two_dice) can be found in Appendix C.3.

4.7 The Symmetric Simple Random Walk

For all of the probabilistic while loops we have seen so far, it was fairly easy to prove that they terminated with probability 1. All followed from some general scheme of probabilistic termination, such as Hart's 0-1 law. The purpose of this section is to show an example of a probabilistic program with a more difficult proof of probabilistic termination.

The symmetric simple random walk is a probabilistic process with a compelling intuitive interpretation (for a standard probability perspective, see Stirzaker (1994, page 145)). A drunk starts at point n (the pub) and is trying to get to point 0 (home). Unfortunately, every step he makes from point i is equally likely to take him to point $i+1$ as it is to take him to point $i-1$. The following program simulates the drunk's passage home; returning the number of steps that were taken.

Definition 55 *A Simulation of the Symmetric Simple Random Walk*

$$\vdash \forall n. \quad \mathsf{random\_lurch}\ n = \mathsf{bind}\ \mathsf{sdest}\ (\lambda b.\ \mathsf{unit}\ (\mathsf{if}\ b\ \mathsf{then}\ n+1\ \mathsf{else}\ n-1)) \tag{4.33}$$

$$\vdash \forall f, b, a, k. \quad \mathsf{prob\_cost}\ f\ b\ (a,k) = \mathsf{bind}\ (b(a))\ (\lambda a'.\ \mathsf{unit}\ (a', f(k))) \tag{4.34}$$

$$\vdash \forall n, k. \quad \mathsf{random\_walk}\ n\ k = \mathsf{bind}\ (\mathsf{prob\_while}\ (\lambda (n, \_).\ 0 < n)\ (\mathsf{prob\_cost\ suc\ random\_lurch})\ (n,k))\ (\lambda (\_, k).\ \mathsf{unit}\ k) \tag{4.35}$$

Why should the random walk always terminate? Firstly, let $\pi_{i,j}$ the probability that starting at point i, the drunk will eventually reach point j. We first formalize the two lemmas $\pi_{p+i,p} = \pi_{i,0}$ and $\pi_{i,0} = \pi_{1,0}^i$. Therefore if the drunk is guaranteed to reach home from a pub at point 1, he will be guaranteed to reach home from a pub at any point. By examining a single iteration of the random walk we have

$$\pi_{1,0} = \tfrac{1}{2}\pi_{2,0} + \tfrac{1}{2} = \tfrac{1}{2}\pi_{1,0}^2 + \tfrac{1}{2}$$

which rewrites to

$$(\pi_{1,0} - 1)^2 = 0$$

This completes the proof of probabilistic termination, and as usual strong independence immediately follows.

$$\vdash \quad \forall n, k.\ \mathsf{random\_walk}\ n\ k \in \mathsf{indep\_fn} \tag{4.36}$$

At this point, we may formulate the definition of random_walk in a more natural way:

$$\begin{aligned} \vdash \quad & \forall n, k. \\ & \mathsf{random\_walk}\ n\ k = \\ & \mathsf{if}\ n = 0\ \mathsf{then}\ \mathsf{unit}\ k\ \mathsf{else} \\ & \quad \mathsf{coin\_flip}\ (\mathsf{random\_walk}\ (n{+}1)\ (k{+}1))\ (\mathsf{random\_walk}\ (n{-}1)\ (k{+}1)) \end{aligned} \tag{4.37}$$

We have now finished the hard work of defining the random walk as a probabilistically terminating program. To demonstrate that once defined it is just as easy to reason about as any of our probabilistic programs, we prove a basic property of the random walk.

$$\vdash \quad \forall n, k.\ \forall^* s.\ \mathsf{even}\ (\mathsf{fst}\ (\mathsf{random\_walk}\ n\ k\ s)) = \mathsf{even}\ (n + k) \tag{4.38}$$

Therefore for a pub at point 1001, the drunk is guaranteed to get home eventually, but he must take an odd number of steps to do so!

We can extract this probabilistic program to ML, and repeatedly simulate it using high-quality random bits from the operating system.[9] Here is a typical sequence of results from random walks starting at level 1:

$$57, 1, 7, 173, 5, 49, 1, 3, 1, 11, 9, 9, 1, 1, 1547, 27, 3, 1, 1, 1, \ldots$$

As can be seen, the number of steps that are required for the random walk to hit zero is usually less than 100. But sometimes, the number can be much larger. Continuing the above sequence of simulations, the 34th simulation sets a new record of 2645 steps, and the next record-breakers are the 135th simulation with 603787 steps and the 664th simulation with 1605511 steps. Such large records early on are understandable, since the theoretical expected number of steps for the random walk is actually infinite![10]

4.8 Concluding Remarks

In this chapter we have demonstrated the practical effectiveness of our framework for verifying probabilistic programs in a theorem prover, set up in Chapters 2 and 3. For this

[9] See Section 5.4 for more details about such extractions.

[10] In case it is difficult to see how an algorithm could have infinite expected running time but terminate with probability 1, consider an algorithm where the probability of termination after n steps is $\frac{6}{\pi^2 n^2}$. The probability of termination is then

$$\sum_n \frac{6}{\pi^2 n^2} = \frac{6}{\pi^2} \sum_n \frac{1}{n^2} = \frac{6}{\pi^2} \cdot \frac{\pi^2}{6} = 1$$

and the expected running time is

$$\sum_n n \frac{6}{\pi^2 n^2} = \frac{6}{\pi^2} \sum_n \frac{1}{n} = \infty$$

purpose we verified a collection of sampling algorithms upon which more sophisticated probabilistic programs can be built (such as the Miller-Rabin primality test in Chapter 5), and another two interesting examples of probabilistic programs.

Following our formalization, the correct probabilities for the optimal dice examples have also been verified by the Prism probabilistic model checker (Kwiatkowska et al., 2001). The optimal dice programs are naturally expressed as probabilistic finite state automata, and Prism automatically evaluates the final probabilities. It would be interesting to link up a theorem prover and a probabilistic model checker; perhaps some analogue of bounded model checking could be used to find bugs in probabilistic programs?

Finally, in Section 3.5 we quoted a result of Gill (1977) that "the computational power of a language with probabilistic choice $\vee_p$ is the same as a language with $\vee_{\frac{1}{2}}$, so long as p is computable." From our verification of the Bernoulli(p) sampling algorithm, it may appear that using a source of random bits we can simulate probabilistic choice $\vee_p$ *for any* p. However, by inspecting the prob_bernoulli algorithm, it is only possible to evaluate the algorithm for a real p that supports the following operations: multiplying by 2; subtracting by 1; comparing with $\frac{1}{2}$. Using these operations, we can compute each bit of p, and so p must be a computable real. Additionally, for every computable p and non-computable q the execution paths of prob_bernoulli p and prob_bernoulli q will differ with probability $|p - q|$. Therefore—in line with the result of Gill—prob_bernoulli allows us to simulate probabilistic choice $\vee_p$ precisely when p is computable.

Chapter 5

Verification of the Miller-Rabin Primality Test

We formally verify a HOL implementation of the Miller-Rabin primality test, a well-known and commercially used probabilistic algorithm. Our fundamental perspective allows us to define a version with strong properties, which we can execute in the logic to prove compositeness of numbers. Finally, we manually extract the HOL Miller-Rabin test to Standard ML, and its performance is evaluated on a large number of candidate primes.[1]

5.1 Introduction

5.1.1 The Miller-Rabin Probabilistic Primality Test

In the 1970s a handful of probabilistic algorithms were introduced that demonstrated two practical advantages over deterministic alternatives: simplicity of expression and efficiency of execution. An algorithm of Berlekamp (1970) uses randomization to factor polynomials; Solovay and Strassen (1977) introduced a probabilistic primality test based on the Jacobi symbol; and Rabin (1976) presented two probabilistic algorithms: the first finds the nearest neighbours of a set $S \subset \mathbb{R}^n$, and the second uses a number theory result of Miller (1975) to test numbers for primality.

This last algorithm has subsequently become known as the Miller-Rabin probabilistic primality test, and is the subject of the present chapter. We implement a version of the Miller-Rabin algorithm in HOL, and use our theory of probability to express its probabilistic specification. A formalization of Miller's result is required to show that the algorithm satisfies its specification, and the verification is completed by using some of the proof techniques for probabilistic programs we developed in Chapter 3.

In his 1976 paper, Rabin evaluated the algorithm by finding the largest prime less than 2^{400} (it took less than a minute to return the result $2^{400} - 593$), and reports that "the algorithm was also used to find twin primes by far larger than any hitherto known pair." Today the Miller-Rabin algorithm is used for primality testing in computer algebra systems such as Mathematica. It is also relevant to public key cryptography software, since the RSA algorithm uses a modulus of the form $n = pq$ where p and q are large primes chosen at random. In practice p and q are found by randomly choosing large

[1] This chapter is a revision of Hurd (2001b).

numbers, and then checking them for primality using a convenient primality test such as Miller-Rabin.[2]

5.1.2 The HOL Verification

The end result of this chapter is a HOL version of the Miller-Rabin probabilistic primality test, satisfying the following specification.

Theorem 56 *Correctness of the Miller-Rabin Primality Test*

$$\vdash \forall n, t, s.\ \mathsf{prime}\ n \Rightarrow \mathsf{fst}\ (\mathsf{miller\_rabin}\ n\ t\ s) = \top \tag{5.1}$$

$$\vdash \forall n, t. \tag{5.2}$$
$$\neg\mathsf{prime}\ n \Rightarrow 1 - 2^{-t} \leq \mathbb{P}\{s \mid \mathsf{fst}\ (\mathsf{miller\_rabin}\ n\ t\ s) = \bot\}$$

The miller_rabin test takes two natural number parameters n and t (in addition to a source s of random bits), where n is the number that we wish to test for primality and t determines the amount of computation that the test is allowed to perform. If n is prime then the test is guaranteed to return $\top$; if n is composite then it will return $\bot$ with probability at least $1 - 2^{-t}$ and $\top$ with probability at most 2^{-t}. Thus for a given value of n if miller_rabin $n\ t\ s$ returns $\bot$ then n is definitely composite, but if it returns $\top$ then all we know is that n is probably prime.[3] However, setting $t = 50$ we see that the probability of the algorithm returning $\top$ for an n that is actually composite is $\leq 2^{-50} < 10^{-15}$.

The novelty of this verification lies in the fact that this is an algorithm with a probabilistic specification used in commercial software. In addition, we present in Appendix D a version of the verified algorithm that we have manually extracted to ML. This is not quite identical to the version commonly found in algorithm textbooks, since they assume a generator for the Uniform(n) distribution, while our version works directly from a sequence of random bits. Practically speaking, this makes the textbook version harder to implement properly, because 'genuine' randomness from the operating system is usually presented to the user as a sequence of random bits. To obtain numbers from a Uniform(n) distribution, a suitable sampling algorithm must be implemented, and we saw in Section 4.3 that this is a slightly delicate matter.[4] In the version we present, a bound can be made on the number of random bits that the algorithm will require.

Completing the proof of correctness requires a significant body of group theory and computational number theory to be formalized in the theorem prover, and Section 5.2 shows how the classical results fit together in the verification. This formalization actually constituted the bulk of the effort and provided the testing ground for a new automatic proof procedure; we briefly report on this experience. Section 5.3 describes the somewhat easier task of interfacing the number theory with the probability theory to produce the

[2]Surprisingly, the popular email encryption program PGP (and the Gnu version GPG) use the Fermat test to check numbers for primality, although the Miller-Rabin test is stronger and involves no extra computation.

[3]Quantifying that 'probably' is a hard problem: the probability that n is prime given that fst (miller_rabin $n\ t\ s$) returned $\top$ depends on the set S from which n was chosen and the distribution of primes in S.

[4]An inevitable consequence is the loss of guaranteed termination, replaced with probabilistic termination.

result, and we also give an immediate application in the form of a procedure for formally proving that numbers are composite. Finally, we highlight the software engineering benefit of this formal methods research by manually extracting our Miller-Rabin primality test to the ML programming language. In Section 5.4 we examine the correctness issues in extracting the algorithm to ML, and profile its performance.

5.2 Computational Number Theory

5.2.1 Definitions

Our HOL implementation of the Miller-Rabin algorithm is (almost) a functional translation of the version presented in Cormen, Leiserson, and Rivest (1990). To prepare, we define functions to factor out powers of 2 and perform modular exponentiation. Here are the correctness theorems for these functions:

$$\vdash \forall n, r, s. \quad 0 < n \Rightarrow (\mathsf{factor\_twos}\ n = (r, s) \iff \mathsf{odd}\ s \wedge 2^r s = n) \tag{5.3}$$

$$\vdash \forall n, a, b.\ 1 < n \Rightarrow \mathsf{modexp}\ n\ a\ b = (a^b \bmod n) \tag{5.4}$$

Next we define a function $\mathsf{witness}\ a\ n$ that is completely deterministic, returning $\top$ if the base a can be used to provide a quick proof that n is composite, and $\bot$ otherwise. We assume that a and n satisfy $0 < a < n$. The witness function uses a helper function witness_tail which is defined using pattern-matching.

Definition 57 *The Miller-Rabin Witness Function*

$$\begin{aligned}\vdash \forall n, a, r. \quad & \mathsf{witness\_tail}\ n\ a\ 0 = a \neq 1 \wedge \\ & \mathsf{witness\_tail}\ n\ a\ (\mathsf{suc}\ r) = \\ & \quad \mathsf{let}\ a' \leftarrow a^2 \bmod n \\ & \quad \mathsf{in\ if}\ a' = 1\ \mathsf{then}\ a \neq 1 \wedge a \neq n - 1\ \mathsf{else\ witness\_tail}\ n\ a'\ r\end{aligned} \tag{5.5}$$

$$\begin{aligned}\vdash \forall n, a. \quad & \mathsf{witness}\ n\ a = \\ & \mathsf{let}\ (r, s) \leftarrow \mathsf{factor\_twos}\ (n - 1)\ \mathsf{in\ witness\_tail}\ n\ (\mathsf{modexp}\ n\ a\ s)\ r\end{aligned} \tag{5.6}$$

The witness function calls factor_twos to find r, s such that s is odd and $2^r s = n - 1$, then uses modexp and witness_tail to calculate the sequence

$$(a^{2^0 s} \bmod n, a^{2^1 s} \bmod n, \ldots, a^{2^r s} \bmod n)$$

This sequence provides two primality tests for n:

1. $a^{2^r s} \bmod n = 1$.
2. If $a^{2^j s} \bmod n = 1$ for some $0 < j \leq r$, then either $a^{2^{j-1} s} \bmod n = 1$ or $a^{2^{j-1} s} \bmod n = n - 1$.

If n is a prime then we wish to show that both these tests will always be true. Test 1 is equivalent to $a^{\phi(n)} \bmod n = 1$ (since $2^r s = n - 1 = \phi(n)$ for n prime), and this is exactly Fermat's little theorem. For this reason this test for primality is called the Fermat test. Test 2 is true since for every x, if $0 = (x^2 - 1) \bmod n = (x+1)(x-1) \bmod n$, then if n is prime we must have that either $(x+1) \bmod n = 0$ or $(x-1) \bmod n = 0$. We thus obtain the following correctness theorem for witness:

$$\vdash \quad \forall n, a.\ 0 < a < n \ \wedge\ \mathsf{witness}\ n\ a \ \Rightarrow\ \neg\mathsf{prime}\ n \tag{5.7}$$

5.2.2 Underlying Mathematics

A composite number n that passes a primality test for some base a is called a pseudoprime. In the case of the Fermat test, there exist numbers n that are pseudoprimes for all bases a coprime to n. These numbers are called Carmichael numbers, and the two smallest examples are 561 and 1729.[5] Testing Carmichael numbers for primality using the Fermat test is just as hard as factorizing them, since the only bases that fail the test are multiples of divisors. Miller's insight was that by also performing Test 2, the number of bases that are witnesses for any composite n will be at least $(n-1)/2$, as formalized in the following theorem.[6]

Theorem 58 *Cardinality of Miller-Rabin Witnesses*

$$\begin{aligned} \vdash \quad & \forall n. \\ & 1 < n \ \wedge\ \mathsf{odd}\ n \ \wedge\ \neg(\mathsf{prime}\ n) \ \Rightarrow \\ & n - 1 \leq 2\,|\{a : 0 < a < n \ \wedge\ \mathsf{witness\ n\ a}\}| \end{aligned} \tag{5.8}$$

Therefore there are no Carmichael numbers for the Miller-Rabin test, and in fact just picking bases at random will quickly find a witness. This is the basis for the Miller-Rabin probabilistic primality test.

We now give a brief sketch of how Theorem 58 is proved, stating which classical results of number theory are necessary for the result.

The proof aims to find a proper subgroup B of the multiplicative group $\mathbb{Z}_n^*$ which contains all the nonwitnesses. This will then imply the result, since by Lagrange's theorem the size of a subgroup must divide the size of the group, and so $|B| \leq |\mathbb{Z}_n^*|/2 = \phi(n)/2 \leq (n-1)/2$.

Firstly, assume that there exists an $x \in \mathbb{Z}_n^*$ such that $x^{n-1} \bmod n \neq 1$. In this case we choose $B = \{x \in \mathbb{Z}_n^* : x^{n-1} \bmod n = 1\}$. The Fermat test ensures that all nonwitnesses are members of B, and since B is closed under multiplication it is a proper subgroup of $\mathbb{Z}_n^*$. Therefore in this case the proof is finished.

[5] 1729 is also famous as the Hardy-Ramanujan number, explained by C. P. Snow in the foreword to *A Mathematician's Apology* (Hardy, 1993): "Once, in the taxi from London, Hardy noticed its number, 1729. He must have thought about it a little because he entered the room where Ramanujan lay in bed and, with scarcely a hello, blurted out his disappointment with it. It was, he declared, 'rather a dull number,' adding that he hoped that wasn't a bad omen. 'No, Hardy,' said Ramanujan, 'it is a very interesting number. It is the smallest number expressible as the sum of two cubes in two different ways.'" ($10^3 + 9^3 = 1729 = 12^3 + 1^3$)

[6] In fact, it is possible to prove a stronger result that the number of nonwitnesses must be at most $\phi(n)/4$, and furthermore this bound can be attained (an example is the Carmichael number 8911).

Secondly, assume that for every $x \in \mathbb{Z}_n^*$ we have that $x^{n-1} \bmod n = 1$. We next show by contradiction that n cannot be a prime power. If $n = p^a$ (with p prime and $a > 1$), then $\mathbb{Z}_n^*$ is cyclic, and so there exists an element $g \in \mathbb{Z}_n^*$ with order $\phi(n) = \phi(p^a) = p^{a-1}(p-1)$. But $g^{n-1} \bmod n = 1$, and so $p^{a-1}(p-1) \mid p^a - 1$. This is a contradiction, since $p \mid p^{a-1}(p-1)$ but $p \not\mid p^a - 1$.

Since n is composite but not a prime power, we can find two numbers $1 < a, b$ with $\mathsf{gcd}(a, b) = 1$ and $ab = n$. At this point we require the unique r, s such that $n - 1 = 2^r s$ and s odd. Next we find a maximal $j \in \{0, \ldots, r\}$ such that there exists a $v \in \mathbb{Z}_n^*$ with $v^{2^j s} \bmod n = n - 1$. Such a j must exist, because since s is odd we can set $j = 0$ and $v = n - 1$. Now choose

$$B = \{x \in \mathbb{Z}_n^* : x^{2^j s} \bmod n = 1 \ \vee \ x^{2^j s} \bmod n = n - 1\}$$

B is closed under multiplication and so is a subgroup of $\mathbb{Z}_n^*$; also the maximality of j ensures that B must contain all nonwitnesses. It remains only to show that $B \neq \mathbb{Z}_n^*$. By the Chinese remainder theorem there exists $w \in \mathbb{Z}_n^*$ such that

$$w \bmod a = v \bmod a \ \wedge \ w \bmod b = 1$$

and so

$$w^{2^j s} \bmod a = a - 1 \ \wedge \ w^{2^j s} \bmod b = 1$$

Hence by the Chinese remainder theorem $w^{2^j s} \bmod n$ cannot be equal to either 1 or $n-1$, so $w \notin B$ and the proof is complete.

5.2.3 Formalization

Formalizing this proof in HOL was a long but mostly routine task, resulting in the theories depicted in Figure 5.1. The most time-consuming activity was a thorough development of group theory, from the initial axioms through to classical results such as Lagrange's theorem (5.9), Fermat's little theorem for groups (5.10) and the structure theorem for Abelian groups (5.11):

$$\vdash \ \forall G \in \mathsf{finite\_group}.\ \forall H \in \mathsf{subgroup}\ G.\ |\mathsf{set}\ H| \,||\, |\mathsf{set}\ G| \tag{5.9}$$

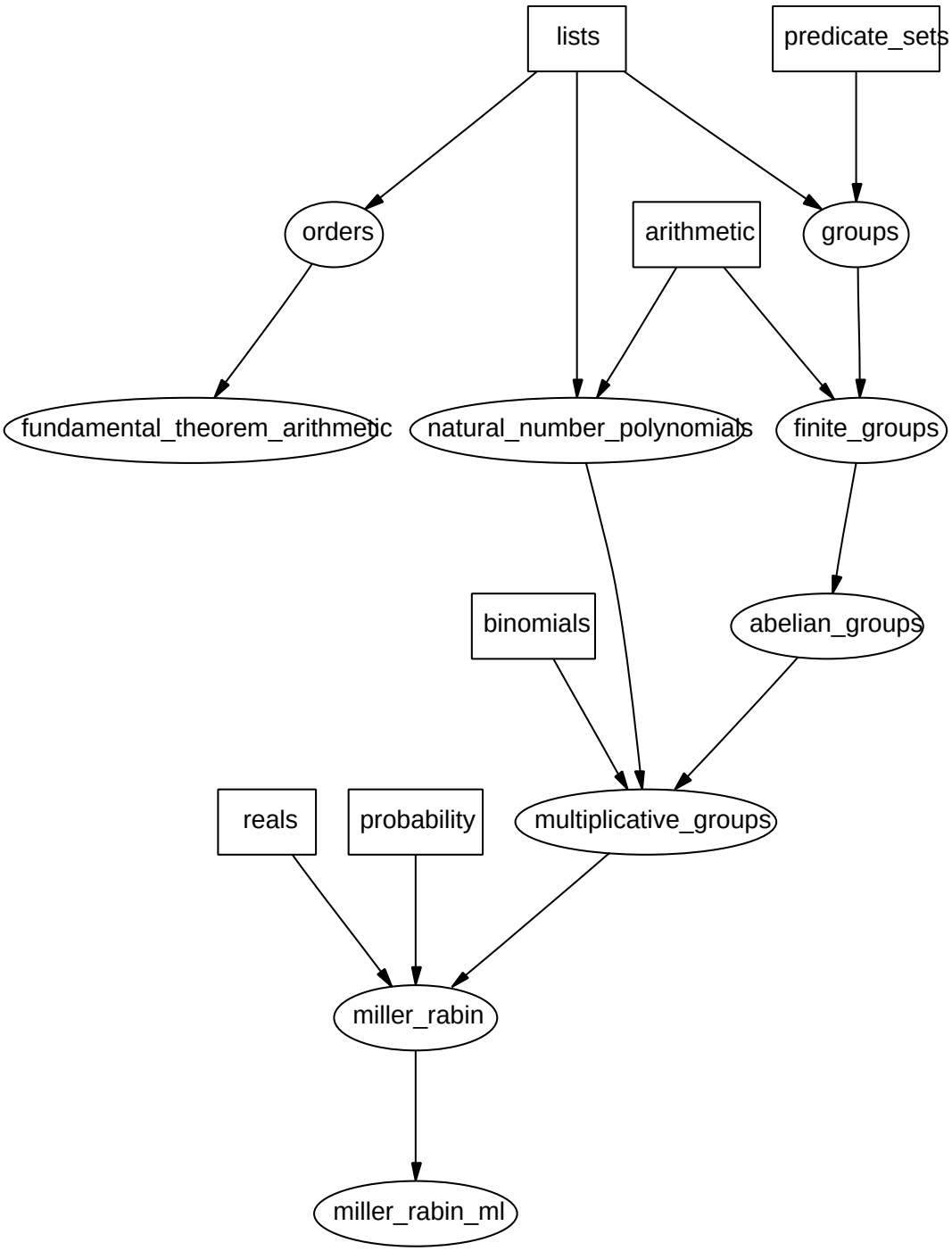
$$\vdash \ \forall G \in \mathsf{finite\_group}.\ \forall g \in \mathsf{set}\ G.\ g^{|\mathsf{set}\ G|} = e \tag{5.10}$$

$$\vdash \ \forall G \in \mathsf{finite\_group}. \tag{5.11}$$
$$\mathsf{abelian}\ G \Rightarrow \exists g \in \mathsf{set}\ G.\ \forall h \in \mathsf{set}\ G.\ h^{|g|} = e$$

This development also allowed some classical arithmetic theorems to be rendered in the language of groups, including the Chinese remainder theorem (5.12) and the existence of primitive roots (5.13):

$$\vdash \ \forall p, q. \tag{5.12}$$
$$1 < p \ \wedge \ 1 < q \ \wedge \ \mathsf{gcd}\ p\ q = 1 \ \Rightarrow$$
$$(\lambda x.\ (x \bmod p, x \bmod q)) \in$$
$$\mathsf{group\_iso}\ (\mathsf{mult\_group}\ pq)$$
$$(\mathsf{prod\_group}\ (\mathsf{mult\_group}\ p)\ (\mathsf{mult\_group}\ q))$$

$$\vdash \ \forall p, a. \tag{5.13}$$
$$\mathsf{odd}\ p \ \wedge \ \mathsf{prime}\ p \ \wedge \ 0 < a \ \Rightarrow \ \mathsf{cyclic}\ (\mathsf{mult\_group}\ p^a)$$



Boxes indicate pre-existing HOL theories, and ellipses are theories created for this development.

Figure 5.1: The Dependency Relation between the Theories of the HOL Formalization.

As well as making the arithmetic theorems more concise, this rendering also allowed the main proof to proceed entirely in the language of groups, eliminating the burden of switching mathematical context in the middle of a mechanical proof and incidentally mirroring the informal proof in Section 5.2.2.

The most difficult part of the whole formalization was theorem (5.13), guaranteeing the existence of primitive roots. This required creating new HOL theories of natural number polynomials and Abelian groups for the $a = 1$ case, and a subtle argument from Baker (1984) for the step case.

One surprising difference between the informal mathematics and the formalization involved the use of the fundamental theorem of arithmetic. This states that every natural number can be uniquely factorized into primes, and many informal mathematical proofs begin by applying this to some variable mentioned in the goal (e.g., by saying "let $p_1^{a_1} \cdots p_k^{a_k}$ be the prime factorization of n.") However, although we had previously formalized the fundamental theorem and it was ready to be applied, in the mechanical proofs we always chose what seemed to be an easier proof direction and so never needed it. Two examples of this phenomenon occur in the structure theorem for Abelian groups (5.11), and the cardinality of the witness set (Theorem 58): the former theorem we formalized using least common multiples which proves the goal more directly; and in the latter all we needed was a case split between n being a prime power or being a product of coprime p, q, so we separately proved this lemma.

Finally, this development provided a testing ground for our predicate set prover described in Appendix B. This automatic proof procedure takes a term t, and derives particular sets S such that $t \in S$ can be proved in the current logical context. The procedure works by first recursively deriving sets for the subterms t_i of t, and then using this information to derive sets for the term t. For example, consider the set $P = \{m : \mathbb{N} \mid 0 < m\}$, where the following theorems allow membership $t \in P$ to be deduced from knowledge of $t_i \in P$:

$$\begin{aligned} \vdash \quad & t_1 \in P \ \vee \ t_2 \in P \ \Rightarrow \ t_1 + t_2 \in P \\ \vdash \quad & t_1 \in P \ \wedge \ t_2 \in P \ \Rightarrow \ t_1 t_2 \in P \end{aligned}$$

So, for instance,

$$m \in P \ \vee \ n \in P \ \vee \ p \in P \ \Rightarrow \ m + mn + mnp \in P$$

The set membership $t \in P$ is clearly equivalent to the property $0 < t$, and there are many more properties that can be phrased more or less directly as set memberships. These include group membership (e.g., $g *_G h \in \mathsf{set}\ G$) and nonemptiness properties of lists and sets (e.g., $l \neq [\,]$). The predicate set prover can be used to robustly prove all of these simple properties, and these come up time and again as side-conditions that must be proved during term rewriting. As a consequence, this new automatic proof procedure lent itself to more efficient development of the theories that needed to be formalized in this verification, particularly the group theory where almost every theorem has one or more group membership side-conditions.

If the predicate set prover had not been available, it would have been possible to use the first-order prover to show most of the side-conditions, but there are three reasons why this is a less attractive proposition: firstly it would have required effort to find the

right 'property propagation' theorems needed for the each goal; secondly the explicit invocations would have led to more complicated tactics; and thirdly some of the goals that can be proved using our specialized tool would simply have been out of range of a more general first-order prover.

5.3 Probability Theory

5.3.1 Guaranteed Termination

In most algorithm textbooks this is how the Miller-Rabin test is defined:

> Given an odd integer n greater than 1, we pick a base a at random from the set $\{1, \ldots, n-1\}$ and call witness n a. Suppose n is composite: since at least $(n-1)/2$ of the bases in the set are guaranteed to be witnesses, the probability that the procedure errs is at most $((n-1)/2)\,/\,(n-1) = 1/2$.

This abstract view implicitly assumes a generator of uniform random numbers in the range $\{0, \ldots, n-1\}$, and would appear to be a natural application for our prob_uniform sampling algorithm for the Uniform(n) distribution that we verified in Section 4.3. However, since prob_uniform employs probabilistic termination, it is prudent to first consider the implications of this decision. When input a prime the Miller-Rabin test is *always* meant to return $\top$ (the only error that is permitted is the return of $\top$ for a composite). If an implementation of Miller-Rabin employs probabilistic termination then this property may be shown to hold with probability 1, but if not then we can prove it for every input sequence of random bits: a much stronger statement. As well as being a more satisfying solution for theoretical reasons, it also confers a practical benefit. We can use our formalization of a pseudo-random number generator (Section 3.4) to execute our Miller-Rabin test in the logic: if it returns $\bot$ then we can immediately deduce the HOL theorem that the input number is composite.

In fact, we can use a single observation to relax the requirement from perfectly uniform random numbers to approximately-uniform random numbers, and we may then use the sampling algorithm prob_uniform_cut that is guaranteed to terminate. The observation is that the base 1 is always going to be a nonwitness for every n, so to find witnesses we can pick bases from the subset $\{2, \ldots, n-1\}$. Now if we can guarantee that the probability of picking each element from this subset is at least $1/(n-1)$, then the probability that we pick a witness is still at least $(1/(n-1))((n-1)/2) = 1/2$.

Recall from Chapter 3 that we model probabilistic algorithms as state-transforming functions $\mathbb{B}^\infty \to \alpha \times \mathbb{B}^\infty$, where the state models the generator of random bits and has the type $\mathbb{B}^\infty$ of infinite boolean sequences. The probabilistic algorithm prob_uniform_cut has an extra parameter t which allows us to specify how close the resulting distribution should be to the ideal Uniform(n). Its correctness theorem is as follows:

$$\vdash \ \forall t, n, k.\ k < n \ \Rightarrow\ |\mathbb{P}\{s \mid \mathsf{fst}\ (\mathsf{prob\_uniform\_cut}\ t\ n\ s) = k\} - 1/n| \leq 2^{-t}$$

Thus if we use the natural number function log2 that is related to calculating logarithms to the base 2

$$\vdash \ \forall n.\ \mathsf{log2}\ n = \mathsf{if}\ n = 0\ \mathsf{then}\ 0\ \mathsf{else}\ \mathsf{suc}\ (\mathsf{log2}\ (n \operatorname{div} 2)) \tag{5.14}$$

$$\vdash \ \forall n, t. \quad 0 < n \ \wedge \ 2(\mathsf{log2}\ (n+1)) \leq t \ \Rightarrow \ 2^{-t} \leq 1/n - 1/(n+1) \tag{5.15}$$

then the following theorem holds:

$$\begin{aligned} \vdash \ & \forall t, n, k. \\ & k < n \ \wedge \ 2(\mathsf{log2}\ (n+1)) \leq t \ \Rightarrow \\ & 1/(n+1) \leq \mathbb{P}\{s \mid \mathsf{fst}\ (\mathsf{prob\_uniform\_cut}\ t\ n\ s) = k\} \end{aligned} \tag{5.16}$$

Therefore, if we set the threshold t to be greater than $2(\mathsf{log2}(n+1))$, then for each $k \in \{0, \ldots, n-1\}$ the probability that $\mathsf{uniform}\ t\ n\ s$ yields the result k is greater than $1/(n+1)$. Coupled with the observation above, this approximation to the uniform distribution is sufficient to implement a version of Miller-Rabin that is guaranteed to terminate.

5.3.2 Definition of the Miller-Rabin Test

Having established that guaranteed termination is possible, we are now in a position to define (one iteration of) the Miller-Rabin probabilistic primality test.

Definition 59 *A Single Iteration of the Miller-Rabin Primality Test*

$$\begin{aligned} \vdash \ & \forall n. \\ & \mathsf{miller\_rabin\_1}\ n = \\ & \mathsf{if}\ n = 2\ \mathsf{then}\ \mathsf{unit}\ \top \\ & \mathsf{else\ if}\ n = 1 \ \vee \ \mathsf{even}\ n\ \mathsf{then}\ \mathsf{unit}\ \bot \\ & \mathsf{else} \\ & \quad \mathsf{bind}\ (\mathsf{prob\_uniform\_cut}\ (2(\mathsf{log2}\ (n-1)))\ (n-2)) \\ & \quad (\lambda a.\ \mathsf{unit}\ (\neg\mathsf{witness}\ n\ (a+2))) \end{aligned} \tag{5.17}$$

This satisfies the correctness theorems

$$\vdash \ \forall n, s.\ \mathsf{prime}\ n \ \Rightarrow \ \mathsf{fst}\ (\mathsf{miller\_rabin\_1}\ n\ s) \tag{5.18}$$

$$\vdash \ \forall n.\ \neg\mathsf{prime}\ n \ \Rightarrow \ 1/2 \leq \mathbb{P}\{s \mid \neg\mathsf{fst}\ (\mathsf{miller\_rabin\_1}\ n\ s)\} \tag{5.19}$$

$$\vdash \ \forall n.\ \mathsf{miller\_rabin\_1}\ n \in \mathsf{indep\_fn} \tag{5.20}$$

In order to define the full Miller-Rabin which tests several bases, we create a new (state-transformer) monadic operator many. The intention of $\mathsf{many}\ p\ n$ is a test that repeats n times the test p using different parts of the random bit stream, returning true if and only if each evaluation of p returned true. For instance, sdest is the destructor function for a stream, and so the function many sdest 10 tests that the next 10 booleans in the random stream are all $\top$. Here is the definition of many and some basic properties:

$$\begin{aligned} \vdash \ & \forall f, n. \\ & \mathsf{many}\ f\ 0 = \mathsf{unit}\ \top \ \wedge \\ & \mathsf{many}\ f\ (\mathsf{suc}\ n) = \mathsf{bind}\ f\ (\lambda x.\ \mathsf{if}\ x\ \mathsf{then}\ \mathsf{many}\ f\ n\ \mathsf{else}\ \mathsf{unit}\ \bot) \end{aligned} \tag{5.21}$$

$$\begin{aligned} \vdash \ & \forall f, n. \\ & f \in \mathsf{indep\_fn} \ \Rightarrow \ \mathbb{P}\{s \mid \mathsf{fst}\ (\mathsf{many}\ f\ n\ s)\} = (\mathbb{P}\{s \mid \mathsf{fst}\ (f(s))\})^n \end{aligned} \tag{5.22}$$

$$\vdash \ \forall f, n.\ f \in \mathsf{indep\_fn} \ \Rightarrow \ \mathsf{many}\ f\ n \in \mathsf{indep\_fn} \tag{5.23}$$

n	Run time	GC time
$2^{2^5}+1$	53.080	7.170
$2^{2^6}+1$	370.210	53.530
$2^{2^7}+1$	2842.920	409.620
$2^{2^8}+1$	22095.770	3170.780

Table 5.1: Testing the Composite Prover

Using the new many operator it is simple to define the multiple iteration Miller-Rabin test miller_rabin.

Definition 60 *The Miller-Rabin Primality Test*

$$\vdash \quad \forall n, t.\ \mathsf{miller\_rabin}\ n\ t = \mathsf{many}\ (\mathsf{miller\_rabin\_1}\ n)\ t \tag{5.24}$$

Finally, the correctness of the Miller-Rabin primality test (Theorem 56) follows from (5.18)–(5.24).

5.3.3 A Compositeness Prover

For any input prime p, Theorem 56 gives us a very strong guarantee: for any t and any sequence s, our implementation of the Miller-Rabin test will always output $\top$. Therefore, given a number n and a sequence s, if execution of miller_rabin n 1 s gives the result $\bot$, then we can immediately deduce the HOL theorem $\vdash \neg\mathsf{prime}\ n$.

Section 3.4 gives us all the machinery we need to execute the Miller-Rabin test inside the HOL logic.[7] Table 5.1 gives the run times in seconds (including garbage collection (GC) time) that was taken to prove the compositeness of some Fermat numbers.

The last number in this table has 78 digits, and demonstrates the possibility of formally proving numbers to be composite with no knowledge of their factors. Since this proof procedure is not the main focus of this chapter (rather an interesting digression), we shall not do any more profiling than this. However, one point that can be made from the existing results is that from each line of the table to the next, the number of digits in n roughly doubles while the run time increases by a factor of 8. This cubic growth is indeed what we would expect if we executed the algorithm in ML, contributing more empirical evidence to the theoretical result that the efficiency of `computeLib` is a constant factor away from ML.

5.4 Extraction to Standard ML

The advantage of extracting the algorithm to a standard programming language such as ML is twofold: firstly execution is more efficient, and so the algorithm can be applied to usefully large numbers; and secondly it can be packaged up as a module and used as a reliable component of larger programs.

[7] We use the linear congruence pseudo-random bit generator, with parameters $A = 103$, $B = 95$ and $N = 79$.

However, there is a danger that the properties that have been verified in the theorem prover are no longer true in the new context. In this section we make a detailed examination of the following places where the change in context might potentially lead to problems: the source of random bits, the arbitrarily large natural numbers, and the manual translation of the Miller-Rabin functions to ML. Finally we test the algorithm on some examples, to check again that nothing has gone amiss and also to get some idea of the performance and computational complexity of the code.

5.4.1 Random Bits

Our theorems are founded on the assumption that our algorithms have access to a generator of perfectly random bits: each bit has probability of exactly $\frac{1}{2}$of being either 1 or 0, and is completely independent of every other bit. In the real world this idealized generator cannot exist, and we must necessarily select an approximation.

The first idea that might be considered is to use a pseudo-random number generator, such as the linear congruence method we used to execute the Miller-Rabin test in the logic. These have been extensively analysed (for example by Knuth (1997)) and pass many statistical tests for randomness, but their determinism makes them unsuitable for applications that require genuine unpredictability. For instance, when generating cryptographic keys it is not sufficient that the bits appear random, they must be truly unpredictable even by an adversary intent on exploiting the random number generator used.

Rejecting determinism, we must turn to the operating system for help. Many modern operating systems can utilize genuine non-determinism in the hardware to provide a higher quality of random bits. For example, here is a description of how random bits are derived and made available in Linux, excerpted from the relevant `man` page:

> "The random number generator gathers environmental noise from device drivers and other sources into an entropy pool. The generator also keeps an estimate of the number of bit[s] of the noise in the entropy pool. From this entropy pool random numbers are created.
>
> When read, the `/dev/random` device will only return random bytes within the estimated number of bits of noise in the entropy pool. `/dev/random` should be suitable for uses that need very high quality randomness such as one-time pad or key generation. When the entropy pool is empty, reads to `/dev/random` will block until additional environmental noise is gathered.
>
> When read, `/dev/urandom` device will return as many bytes as are requested. As a result, if there is not sufficient entropy in the entropy pool, the returned values are theoretically vulnerable to a cryptographic attack on the algorithms used by the driver. Knowledge of how to do this is not available in the current non-classified literature, but it is theoretically possible that such an attack may exist. If this is a concern in your application, use `/dev/random` instead."

These devices represent the highest quality source of randomness to which we have easy access, and so we have packaged them up as ML boolean streams for use in our extracted program.

5.4.2 Arbitrarily Large Natural Numbers

Another place where there is a potential disparity between HOL and ML regards their treatment of numbers. The HOL algorithm operates on the natural numbers $\{0, 1, 2, \ldots\}$, while in ML the primitive type `int` contains signed numbers in a range depending on the machine architecture.

We resolved this incompatibility by creating the ML module `HolNum`, which implements an equality type `num` of arbitrarily large natural numbers. The Miller-Rabin functions may then use this type of numbers, and the arithmetic operations will behave exactly as in HOL.

We first implemented our own large number module, written in the purely functional subset of ML (and using a `word vector` representation of numbers). However, this was found to be about 100 times slower than the Moscow ML interface to the GNU Multi-Precision library,[8] so we switched to this instead.

5.4.3 Extracting from HOL to ML

A further place where errors could creep in is the manual extraction of the Miller-Rabin functions from HOL to ML. Consequently we did this in two steps, the first of which was creating a new HOL theory containing a version of all the functions we wish to extract. For example, in theorem (5.21) the monadic operator many was defined like so

$$\begin{aligned}\vdash\ & \forall f, n.\\ & \mathsf{many}\ f\ 0 = \mathsf{unit}\ \top\ \wedge\\ & \mathsf{many}\ f\ (\mathsf{suc}\ n) = \mathsf{bind}\ f\ (\lambda x.\ \mathsf{if}\ x\ \mathsf{then}\ \mathsf{many}\ f\ n\ \mathsf{else}\ \mathsf{unit}\ \bot)\end{aligned}$$

and in this new HOL theory we prove it is equivalent to

$$\begin{aligned}\vdash\ & \forall f, n.\\ & \mathsf{many}\ f\ n =\\ & \mathsf{if}\ n = 0\ \mathsf{then}\ \mathsf{unit}\ \top\\ & \mathsf{else}\ \mathsf{bind}\ f\ (\lambda x.\ \mathsf{if}\ x\ \mathsf{then}\ \mathsf{many}\ f\ (n-1)\ \mathsf{else}\ \mathsf{unit}\ \bot)\end{aligned}$$

so that we may export it to ML as

```
fun MANY f n =
  if n = ZERO then UNIT true
  else
    BIND f (fn x => if x then MANY f (n -- ONE) else UNIT false);
```

As can be seen here the ML version involves some lexical changes,[9] but has precisely the same parse tree as the intermediate HOL version. This reduces the chance of errors introduced by the cut-and-paste operation. To compare the two versions of all the functions we use, refer to the HOL theory `miller_rabin_ml` and the ML program file `Miller.sml` in Appendix D.

[8] `http://www.swox.com/gmp/`

[9] For example, `ZERO`, `true` and `--` are the ML versions of the HOL terms 0, $\top$ and $-$.

An intellectually interesting problem in the extraction is the question of how to handle partial functions. Consider the HOL function prob_uniform_cut that generates (approximations to) uniform random numbers:

$$\vdash \quad \forall t, n.\ \mathsf{prob\_uniform\_cut}\ t\ (\mathsf{suc}\ n) = \mathsf{if}\ t = 0\ \mathsf{then}\ \mathsf{unit}\ 0\ \mathsf{else}\ \ldots$$

The function is deliberately underspecified: there is no case where the second argument takes the value 0 because it does not make sense to talk of random numbers uniformly distributed over the empty set. HOL allows us to define functions like this, but there is no immediate ML equivalent. In the intermediate HOL version, we prove it to be equivalent to

$$\begin{aligned}\vdash \quad & \forall t, n. \\ & \mathsf{prob\_uniform\_cut}\ t\ n = \\ & \mathsf{if}\ n = 0\ \mathsf{then}\ \mathsf{prob\_uniform\_cut}\ t\ n\ \mathsf{else}\ \mathsf{if}\ t = 0\ \mathsf{then}\ \mathsf{unit}\ 0\ \mathsf{else}\ \ldots\end{aligned}$$

This rather strange theorem represents an explicitly total version of the prob_uniform_cut function. Of course if we simply extract this to ML it will loop forever when called with second argument 0, so we extract in the following way:

```
fun prob_uniform_cut t n =
  if n = ZERO then raise Fail "prob_uniform_cut: out of domain"
  else if t = ZERO then UNIT ZERO
  else ...
```

This device allows us to faithfully extract partial functions with as much confidence as for total functions.

5.4.4 Testing

It would be pleasant to say that since the function had been mechanically verified, no testing was necessary. But the preceding sections have shown that this would be naive. Even if we are prepared to trust the generation of random bits, the operations of our arbitrarily large number module and the manual extraction of the algorithms, testing would still be prudent to catch bugs at the interface between these components.

The first quick test was an ML version of Rabin's 2^{400} test mentioned in the introduction: with the number of tries (the t parameter) set to 50 the program took 15 seconds to confirm that $2^{400} - 593$ is indeed the smallest (probable) prime below 2^{400}.

The main test proceeded in the following way: for various values of l, generate n odd candidate numbers of length l bits. Perform a quick compositeness test on each by checking for divisibility by the first l primes, and also run Miller-Rabin with the maximum number of bases fixed at 50. The results are displayed in Table 5.2. $\mathbb{E}_{l,n}(\text{composite})$ is equal to $n(1-\mathbb{P}_l(\text{prime}))$ and mathematically estimates the number of composites that the above algorithm will consider as candidates.[10] QC is the number of candidates that were

[10]Using the prime number theorem, $\pi(n) \sim n/\log n$, we can derive:

$$\mathbb{P}_l(\text{prime}) = \frac{\pi(2^l) - \pi(2^{l-1})}{2^{l-2}} \sim \frac{2}{\log 2}\left(\frac{2}{l} - \frac{1}{l-1}\right)$$

l	n	$\mathbb{E}_{l,n}$	QC	MR	MR_+
10	100000	74352	70262	70683	520
15	100000	82138	72438	80448	85
20	100000	86332	74311	85338	5
50	100000	94347	79480	94172	0
100	100000	97144	82258	97134	0
150	100000	98089	83401	98077	0
200	100000	98565	84370	98557	0
500	100000	99424	86262	99458	0
1000	100000	99712	87377	99716	0
1500	100000	99808	87935	99798	0
2000	100000	99856	88342	99852	0

Table 5.2: Testing the Extracted Miller-Rabin Algorithm

l	Gen time	QC time	MR_1 time
10	0.0004	0.0014	0.0028
15	0.0007	0.0017	0.0041
20	0.0009	0.0019	0.0054
50	0.0023	0.0034	0.0136
100	0.0068	0.0075	0.0370
150	0.0107	0.0112	0.0584
200	0.0157	0.0156	0.0844
500	0.0443	0.0416	0.2498
1000	0.0881	0.0976	0.7284
1500	0.1543	0.2164	1.7691
2000	0.3999	0.2843	4.2910

Table 5.3: Profiling the Extracted Miller-Rabin Algorithm

found to be divisible by the quick compositeness test, MR is the number that the Miller-Rabin algorithm found to be composite, and finally MR_+ is the number of candidates that the Miller-Rabin algorithm needed more than one iteration to determine that it was composite.

The most important property for testing purposes cannot be deduced from the table: for each number that the quick compositeness test found to be composite, the Miller-Rabin test also returned this result (and as can be seen, this almost always required only one iteration). In the other direction, using the $\mathbb{E}_{l,n}$ column as a guide, we can see that the Miller-Rabin algorithm did not find many more composites than expected.

In Table 5.3 we compare for each l the average time in seconds[11] taken to generate a random odd number, subject it to the quick composite test, and perform one iteration of the Miller-Rabin algorithm.

[11]These results were produced using the Moscow ML 2.00 interpreter and RedHat Linux 6.2, running on a computer with a 200MHz Pentium Pro processor and 128Mb of RAM.

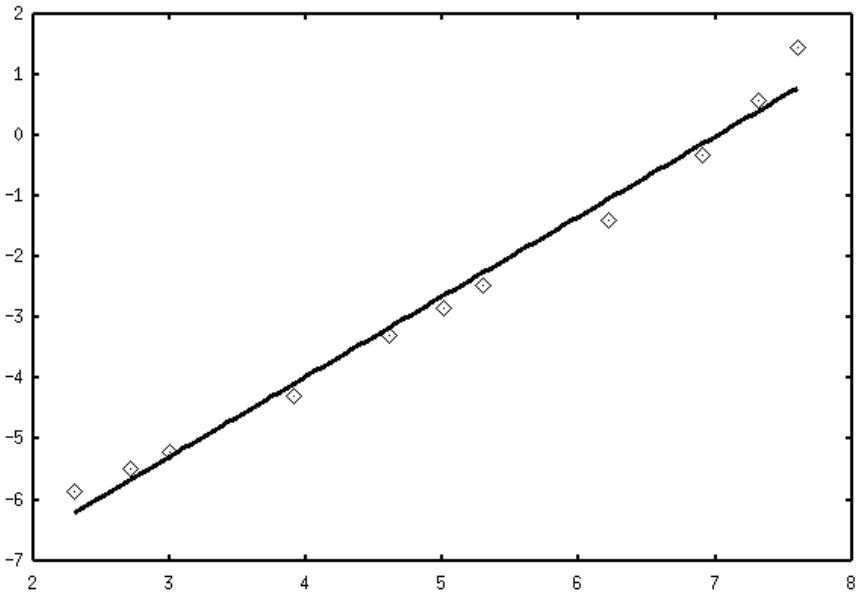


Figure 5.2: Graph of $\log(\mathrm{MR}_1$ time) against $\log l$.

The complexity of (one iteration of) the Miller-Rabin algorithm is around $O(l^2 \log l)$, since it uses asymptotically the same number of operations as modular exponentiation (Cormen et al., 1990). However, performing linear regression on the log-log graph in Figure 5.2 gives a good fit with degree 1.32, implying a complexity of $O(l^{1.32})$. We can only conclude that the GNU Multi-Precision library is heavily optimized for the 'small' numbers in the range we were using, and so we cannot expect an asymptotically valid result.[12]

5.5 Concluding Remarks

The main result of this chapter is that our framework for verifying probabilistic programs is powerful enough to formally verify the Miller-Rabin primality test in the HOL theorem prover, a well-known and commercially used probabilistic algorithm.

The predicate set prover helped to make the proof development more efficient; it was particularly useful for proving group membership conditions and simple but ubiquitous arithmetic conditions. Our evaluation is that it is a useful tool for reasoning about term properties that naturally propagate up terms, and a useful condition prover for contextual rewriters.

The difference between formal and informal proofs in their use of the fundamental theorem of arithmetic was pointed out in Section 5.2.3. This is the most striking example of many small differences in the style of informal and formal proofs, stemming from the

[12]Contrast this with the same experiment performed using our own purely functional implementation of arbitrarily large numbers. Linear regression on the log-log graph now gives a good fit with degree 2.98, confirming the theoretical result since we used the simple $O(l^2)$ algorithm for multiplication. Incidentally, garbage collection was not the bottleneck in this implementation, typically accounting for less than 5% of the time taken.

different proof consumers in each case. Machines make it easier to formalize principles of induction such as dividing out a prime or prime power factor of a number, whereas humans would seem to be better at reasoning with the multisets that contain the prime factors.

There has been a long history of number theory formalizations, most relevantly for us: a proof of Wilson's theorem in the Boyer-Moore theorem prover by Russinoff (1985); a correctness proof of the RSA algorithm in ACL2 by Boyer and Moore (1984); and a correctness proof of RSA in three different theorem provers by Théry (1999). This last work was especially useful, since one of the theorem provers was hol98, and we were able to use his proof of the binomial theorem in our own development.

Though lacking any probabilistic algorithms, the closest work in spirit to this chapter is some recent work on primality proving in Coq by Caprotti and Oostdijk (2001). They formalize a similar computational number theory development, and utilize a computer algebra system to prove numbers prime. Seeing this work improved the organization of theories in our own formalism. Harrison has also implemented a primality prover in HOL (Light), using Pratt's criterion instead of Pocklington's.

Our own development of group theory benefitted from the developments of Gunter (1989), Kammüller and Paulson (1999) and Zammit (1999); the theory of groups has been formalized in many other theorem provers as well.

Chapter 6

Summary

The aim of this thesis has been to show that probabilistic algorithms can be formally verified using a mechanical theorem prover, 'just like deterministic algorithms.'

We began in Chapter 2 with an extensive foundational development of probability, creating a higher-order logic formalization of mathematical measure theory. This allowed the definition of the probability space we use to model a random bit generator, which informally is a stream of coin-flips, or technically an infinite sequence of IID Bernoulli($\frac{1}{2}$) random variables. In Chapter 3 we introduced a simple higher-order logic formalism of probabilistic programs, and introduced a probabilistic while loop that allowed us to define probabilistic programs that terminate with probability 1. Using our formalized theory of probability we were able to write specifications for our probabilistic programs, and special emphasis was placed on developing technical support to expedite practical verification in a mechanical theorem prover. Finally, in Chapter 4 we demonstrated the formal framework with some example probabilistic programs: sampling algorithms for four probability distributions; some optimal procedures for generating dice rolls from coin flips; the symmetric simple random walk; and in Chapter 5 the Miller-Rabin primality test.

This work has made a number of original contributions to knowledge; these were pointed out throughout the text, and here we collect them together in categories and summarize them:

Probabilistic Algorithms: Our formal framework has allowed us to connect work on sampling probability distributions to textbook presentations of random algorithms. In addition to the theoretical benefits of seeing how the underlying reliance on a source of random bits manifests itself at the level of the random algorithm, we also extracted some practical mileage from this connection. Seeing that strong properties can be proved of probabilistic programs that avoid probabilistic termination, we deliberately created a version of the Miller-Rabin primality test that is guaranteed to terminate, and thus for each input an upper bound exists on how many random bits it will require.

Formal Methods: This work has demonstrated that probabilistic algorithms can be verified in a standard theorem prover, and since we have developed our theory as a definitional extension of higher-order logic, we can have complete confidence in the properties we prove. Our most significant contributions in this area are: the compositional property indep_fn, showing that all probabilistic programs we construct

using our primitives satisfy useful properties of measurability and independence; and the probabilistic while reduction theorem, showing how potentially difficult proofs about probabilistic termination reduce to standard induction techniques. In addition, our extraction of the Miller-Rabin primality test gave an example of software engineering of probabilistic algorithms when a high assurance of correctness is required.

Formalized Mathematics: The formalization of mathematics into logic with the aid of mechanical theorem provers is already well-established; indeed the process is virtually industrialized in the Mizar theorem prover. In this thesis we have focussed on showing the utility of such endeavours, by applying formalized theories to the practical task of verifying probabilistic algorithms. Nevertheless, this has necessitated the formalization of some mathematics that has not been previously attempted, such as Carathéodory's extension theorem and the construction of a non-measurable set in probability theory, and the existence of primitive roots and the cardinality of the Miller-Rabin witness set in computational number theory. For interest, we present here some approximate values for the 'de Bruijn factor' (dB)

$$\mathrm{dB}\,(\text{theorem}) \;=\; \frac{\text{size of textbook proof}}{\text{size of machine proof}}$$

of three textbooks we used:

$$\mathrm{dB}\left(\begin{matrix}\text{Carathéodory's extension theorem}\\ \text{assuming real analysis}\end{matrix}\right) \;=\; \frac{4\frac{1}{2}\text{ pages of Williams (1991)}}{1200\text{ proof script lines}}$$

$$\mathrm{dB}\left(\begin{matrix}\text{Miller-Rabin witness theorem}\\ \text{assuming number theory}\end{matrix}\right) \;=\; \frac{1\frac{1}{2}\text{ pages of Cormen et al. (1990)}}{600\text{ proof script lines}}$$

$$\mathrm{dB}\left(\begin{matrix}\text{Multiplicative group of } p^a \text{ is cyclic}\\ \text{assuming group theory}\end{matrix}\right) \;=\; \frac{\frac{1}{2}\text{ page of Baker (1984)}}{400\text{ proof script lines}}$$

Theorem Proving: The investigation (in Appendix B) of PVS predicate subtyping—and showing the extent to which it can be simulated in HOL with predicate sets—provides an interesting comparison of the two theorem provers. One advantage of working in an LCF theorem prover is that the ML programming language is provided for the user to create new proof tools in a secure fashion. Our predicate set prover is a typical example of developing such a tool: automating a particular pattern of reasoning and then immediately deploying it in our formalization proofs; without needing to worry that soundness may have been violated in some way.

6.1 Future Work

The concluding remarks of each chapter suggested possible improvements and extensions that could be made, and so here we consider larger additions to this work.

- Now that we have developed a verification infrastructure, we intend to deploy it in application areas that rely on probabilistic algorithms. Examples include web anonymizing services (such as Crowds (Reiter and Rubin, 1999)) which use probabilistic algorithms to prevent web servers from learning information that could

identify their visitors; financial applications which calculate prices of complex bonds using Monte Carlo integration algorithms (Fishman, 1996); and hardware simulators which generate test vectors probabilistically to avoid any bias that may lead to missing a fault (Bardell et al., 1987).

- Currently the probabilistic programs we verify are expressed as higher-order logic functions. It would be possible to formalize in HOL a little while-language containing a probabilistic primitive for obtaining a random bit, and use this to express our probabilistic programs. Such a deep embedding would even allow us to capture a class of strongly independent probabilistic programs in a new HOL type, removing many side-conditions of theorems.

- Verifying more examples of probabilistic algorithms will inevitably necessitate more formalization; in particular we already can see that a theory of expectation will be required to prove the correctness of probabilistic quicksort. If we can continue our policy of formalizing standard theorems of mathematics to aid verifications, then this will provide long-term benefits to many users of the HOL theorem prover.

- Proof tools to aid formalization can never be too powerful, and we certainly had the impression during our work so far that more could be done to help the user. Although it is difficult to predict exactly where the real benefits will lie in this area, we consider the development of general proof tools to be an important complement to the development of theories.

- Finally, the extraction to ML was an interesting part of the Miller-Rabin verification, and we built from scratch the infrastructure that enabled us to do this. It would be interesting to push this further, perhaps aiming for a whole library of verified functions built upon some trusted primitives.

Appendix A

Higher-order Logic and the HOL Theorem Prover

This appendix contains a very brief introduction to higher-order logic and its implementation in the HOL theorem prover, containing no more than is necessary for a reader wishing to understand the basis on which this thesis is built. A detailed introduction is given by Gordon and Melham (1993) in the canonical reference: *Introduction to HOL (A theorem proving environment for higher order logic)*.

A.1 Terms and Types

Terms in higher-order logic are typed, and we write $t : \tau$ to mean that term t has type τ. It is helpful to identify types with the set of their elements, so for example the type of booleans is $\mathbb{B} = \{\top, \bot\}$, while the type of natural numbers is $\mathbb{N} = \{0, 1, 2, \ldots\}$. Compound types may be created by using type operators; an important example is the function space type operator $\cdot \to \cdot$, so the type $\mathbb{N} \to \mathbb{B}$ is the set of functions mapping natural numbers to either $\top$ or $\bot$. Another example is the list type operator $\cdot^*$, allowing us to create types such as

$$\mathbb{B}^* = \{[\,],\ [\top],\ [\bot],\ [\top, \top],\ \ldots\}$$

Finally, we allow an infinite set $\{\alpha,\ \beta,\ \ldots\}$ of type variables, which may instantiate to any higher-order logic type. An example of this is the type α^* of lists containing elements of type α. This mechanism of polymorphism allows us to prove theorems once about the type α^* of lists containing elements of type α, and then instantiate α multiple times to give us the same theorems about $\mathbb{B}^*$, $(\mathbb{N} \to \mathbb{B})^*$, β^{**} etc.

The terms of higher-order logic are terms of λ-calculus (Church, 1940), where the syntax $\lambda\, x.\ t[x]$ should be read as 'the function that takes an x and returns $t[x]$.' The λ binds x, so for example the application of $\lambda\, x.\ \lambda\, x.\ x$ to y yields the result $\lambda\, x.\ x$ (and *not* $\lambda\, x.\ y$). The restriction to *typed* terms of λ-calculus means that every term must be well-typed (or inconsistencies emerge in the form of the Russell paradox), where the type relation is calculated as follows:

Variables: All variables $v : \tau$ are well-typed.

Constants: Constants are given a type when they are defined, for example $\top : \mathbb{B}$, $[\,] : \alpha^*$, $= : \alpha \to \alpha \to \mathbb{B}$ and $\mathsf{prime} : \mathbb{N} \to \mathbb{B}$. Any type variables that they contain

may be specialized when they are used in terms, so for example the term $[\,] : \mathbb{N}^*$ is well-typed.

Function applications: A function term $f : \alpha \to \beta$ is well-typed when applied to any argument term $x : \alpha$, and the result is $f(x) : \beta$.

λ-abstractions: Given a variable $v : \alpha$ and a term $t[v] : \beta$, the λ-abstraction $(\lambda\, v.\ t[v]) : \alpha \to \beta$ is well-typed.

Using a type-inference algorithm of Milner (1978), it is possible to take a term t that does not contain any type information at all, and deduce a most general type for the term (or raise an error if it is not well-typed). Therefore, when we write terms $t : \tau$ of higher-order logic we can safely leave off the type annotation $: \tau$ without causing ambiguity. We generally only include type annotations to make things clearer to the reader.

A.2 Theorems

Theorems in higher-order logic are *sequents* $\Gamma \vdash t$ where the term t is the *conclusion* of the theorem, and the set of terms Γ are the *hypotheses* of the theorem. Typically of LCF theorem provers, HOL allows the creation of theorems only in the *logical kernel*, which faithfully executes the primitive rules of inference of higher-order logic. For example:

$$\frac{}{\vdash t = t}\mathsf{REFL} \qquad \frac{\Gamma \vdash s = t \quad \Delta \vdash t = u}{\Gamma \cup \Delta \vdash s = u}\mathsf{TRANS}$$

$$\frac{\Gamma[\alpha] \vdash t[\alpha]}{\Gamma[\sigma] \vdash t[\sigma]}\mathsf{INST\_TYPE} \qquad \frac{}{\vdash (\lambda\, x.\ t[x])\ u = t[u]}\mathsf{BETA\_CONV}$$

These are four of the primitive rules of inference, asserting that the equality relation is both reflexive and transitive, that type variables in a theorem may be instantiated to any higher-order logic type, and that the usual β-conversion of λ-calculus is valid in higher-order logic. These primitive rules of inference appear in the ML signature of the logical kernel as follows:

$$\begin{aligned} \mathsf{REFL} &: \mathsf{term} \to \mathsf{thm} \\ \mathsf{TRANS} &: \mathsf{thm} \to \mathsf{thm} \to \mathsf{thm} \\ \mathsf{INST\_TYPE} &: \mathsf{hol\_type} \times \mathsf{hol\_type} \to \mathsf{thm} \to \mathsf{thm} \\ \mathsf{BETA\_CONV} &: \mathsf{term} \to \mathsf{thm} \end{aligned}$$

Since the ML type thm is abstract, the type security of ML ensures that the functions in the logical kernel represent the only way that theorems may be created.

A principle of definition implemented as a function in the logical kernel allows new constants to be defined. Given a theorem[1] $\vdash \exists\, c : \alpha.\ P(c)$, it makes a new constant $c : \alpha$ and returns the characterizing theorem $\vdash P(c)$. Standard definitions of the form $\vdash c = t$ are covered in this scheme, since the theorem $\vdash \exists\, c.\ c = t$ follows trivially by taking the witness to be t.

To help the reader with the HOL theories in Appendices C and D, in Table A.1 we tabulate the HOL versions of some mathematical symbols we have used.

[1]For this to be valid, we must insist that $P(c)$ contains no free variables other than c and no type variables other than those in α.

HOL	Mathematical	English
`\`	λ	Lambda abstraction
`!`	$\forall$	Universal quantification
`?`	$\exists$	Existential quantification
`/\`	$\wedge$	Conjunction
`\/`	$\vee$	Disjunction
`==>`	$\Rightarrow$	Implication
`~`	$\neg$ and $-$	Logical and numerical negation
`=`	$=$ and $\Longleftrightarrow$	Equality and 'if and only if'
`{}`	$\emptyset$	Empty set
`UNIV`	$\mathcal{U}_\alpha$	Universe set (type α implicit)
`!*`	$\forall^*$	'Probably' probabilistic quantifier
`?*`	$\exists^*$	'Possibly' probabilistic quantifier

Table A.1: HOL Versions of Mathematical Symbols

Appendix B

Predicate Set Prover

This appendix contains an investigation of predicate subtyping available in the PVS theorem prover, and show how it may be simulated in HOL using predicate sets. This leads to the development of a new proof procedure called the predicate set prover, which employs this reasoning to prove a class of theorems that frequently appear as side-conditions.[1]

B.1 Introduction

B.1.1 An Introduction to Predicate Subtyping

PVS (Owre et al., 1999) is an interactive theorem prover implementing a similar logic to HOL; both logics are extensions of the simple type theory of Church (1940). In HOL the extension is Hindley-Milner polymorphism (Milner, 1978), and in PVS it is parametric theories[2] and predicate subtyping (Owre and Shankar, 1997).

Predicate subtyping allows the creation of a new type corresponding to an arbitrary predicate, where elements of the new type are also elements of the containing type. As a simple illustration of this, the type of real division ($/$) in HOL is $\mathbb{R} \to \mathbb{R} \to \mathbb{R}$, and in PVS is $\mathbb{R} \to \mathbb{R}^{\neq 0} \to \mathbb{R}$, where $\mathbb{R}$ is the type of real numbers and $\mathbb{R}^{\neq 0}$ is the predicate subtype of non-zero real numbers, expressed by the predicate $\lambda\, x.\ x \neq 0$.

This extension of the type system allows more information to be encoded in types, leading to benefits for specification and verification such as:

- The ability to express dependent types, for example a type for natural number subtraction that prevents the second argument from being larger than the first argument, or a type representing lists of a fixed length in order to model arrays.
- Greater use of types to express side-conditions of theorems, for example the PVS rewrite rule

$$\vdash_{\text{PVS}} \quad \forall\, x : \mathbb{R}^{\neq 0}.\ x/x = 1 \tag{B.1}$$

  In HOL this would have to be expressed

$$\vdash \quad \forall\, x : \mathbb{R}.\ (x \neq 0) \Rightarrow (x/x = 1) \tag{B.2}$$

[1] This appendix is a revision of Hurd (2001a).

[2] Parametric theories allow polymorphism at the granularity of the theory (think C++ templates), whereas Hindley-Milner polymorphism operates at the granularity of the formula (think ML functions).

and the extra condition must be proved each time the rule is applied.[3]

- More errors can be found in specifications during type-checking, giving greater confidence that the goal is correct before a verification is embarked upon. Mokkedem and Leonard (2000) observed this to be very effective in a large network protocol verification performed in PVS. Many specifications are not verified at all, and in that case the extra confidence is especially valuable.

However, there are also some costs:

- Type-checking becomes undecidable, so (potentially human) effort must be expended to allow terms to be accepted into the system.
- Type-correctness depends on the current logical context, imposing an extra burden on term operations to keep careful track of what can be assumed at each subterm. In the case of users wishing to program their own tactics this merely steepens the learning curve; for term operations in the logical kernel faulty implementations have produced a string of soundness bugs.[4]

In the literature there are papers arguing for and against predicate subtyping. On the one hand Shankar and Owre (1999) give many examples of their utility, while on the other Lamport and Paulson (1999) give an example where the costs must be paid without much benefit, in this case because predicate subtypes cannot naturally encode the desired invariant.

B.1.2 Simulating Predicate Subtyping in HOL

In Section B.2 we describe an approach to gain the functionality of predicate subtypes in HOL, without altering the logic in any way. Instead of creating a first-class type associated with a particular predicate P, we reason with the subset of elements that satisfy P. For example, the PVS predicate subtype $\mathbb{R}^{\neq 0}$ can be modelled with the HOL predicate set

$$\mathsf{nzreal} = \{x : \mathbb{R} \mid x \neq 0\}$$

With this substitution, it is possible to assign subtypes to HOL constants, where the PVS subtype judgement is modelling with a HOL theorem. For example, we assign a subtype to the real multiplicative inverse (reciprocal) function inv by proving the theorem

$$\vdash \quad \mathsf{inv} \in \mathsf{nzreal} \dot{\rightarrow} \mathsf{nzreal}$$

We can even extend this technique to simulate PVS predicate subtype-checking, and present an algorithm to derive subtypes for HOL terms. If a term contains violations of the subtypes we have assigned, then the algorithm will not be able to derive any subtypes for the term, and the user is issued a warning about the potential problem. Thus by

[3] Analogously, typed logics such as HOL and PVS enjoy this advantage over an untyped logic such as ZF set theory, in which the theorem $\vdash \forall n : \mathbb{N}.\ n + 0 = n$ must be expressed $\vdash \forall n.\ n \in \mathbb{N} \Rightarrow (n + 0 = n)$.

[4] Shankar and Owre (1999) write "these bugs stem largely from minor coding errors rather than foundational issues or complexities". However, the extra complexity generated by predicate subtyping does play a part in this: there have been very few soundness bugs found in HOL over the years.

using this method, we can catch terms that contain partial functions applied outside their domain, as in $\mathsf{inv}\ 0$ (although as we shall see, higher-order logic imposes certain limitations on our ability to find *all* violations).

The subtype derivation algorithm also gives us the extra reasoning power needed to automatically prove side-conditions of HOL theorems that would be expressed in PVS using predicate subtypes (such as the condition of theorem (B.2) above). We package this up as a proof procedure called a predicate set prover, and integrate it as a condition prover into a conditional rewriter. The resulting tool is evaluated by using it in a formalization case-study of a body of computational number theory, and it is found to be effective on a certain class of conditions that occur frequently and are not well-handled by existing methods. Group theory is a fertile field for such conditions, where virtually all theorems contain some group membership conditions, for example

$$\vdash \quad \forall G \in \mathsf{group}.\ \forall g \in \mathsf{set}\ G.\ g *_G \mathsf{id}_G = g$$

If this theorem was being used as a conditional rewrite, then after $g *_G \mathsf{id}_G$ was matched in the term we would have to prove that $G \in \mathsf{group}$ and $g \in G$ before rewriting it to g. These are the kind of goals that the predicate set prover is designed to solve.

B.2 The Formalism

B.2.1 Subtypes

We model PVS subtypes with HOL predicate sets, where a predicate set is a function $P : \alpha \to \mathbb{B}$ which represents the set of elements x that satisfy $P(x) = \top$. Note that P is parameterized by the type α, and we shall use the terminology 'α-set' when we wish to make this dependency explicit. Predicate sets are a standard modelling of sets in higher-order logic; all the usual set operations can be defined including universe sets $\mathcal{U}_\alpha$ for each type α, and the notation $\{x \mid P(x)\}$ refers to the predicate set $\lambda\, x.\ P(x)$. The following definition gives several examples of predicate sets we use to model subtypes.

Definition 61 *Examples of Predicate Sets Modelling Subtypes*

$$\vdash \quad \mathsf{nzreal} = \{x : \mathbb{R} \mid x \neq 0\} \tag{B.3}$$
$$\vdash \quad \mathsf{posreal} = \{x : \mathbb{R} \mid 0 < x\} \tag{B.4}$$
$$\vdash \quad \mathsf{nnegreal} = \{x : \mathbb{R} \mid 0 \leq x\} \tag{B.5}$$
$$\vdash \quad \forall n.\ \mathsf{lenum}\ n = \{m : \mathbb{N} \mid m \leq n\} \tag{B.6}$$
$$\vdash \quad \forall n.\ \mathsf{nlist}\ n = \{l : \alpha^* \mid \mathsf{length}\ l = n\} \tag{B.7}$$
$$\vdash \quad \forall P.\ \mathsf{list}\ P = \{l : \alpha^* \mid \forall x.\ \mathsf{mem}\ x\ l \Rightarrow x \in P\} \tag{B.8}$$
$$\vdash \quad \forall P, Q.\ \mathsf{pair}\ P\ Q = \{x : \alpha \times \beta \mid \mathsf{fst}\ x \in P \wedge \mathsf{snd}\ x \in Q\} \tag{B.9}$$

The predicate sets nzreal, posreal and nnegreal defined by (B.3)–(B.5) are straightforward, each representing the set of real numbers that are mapped to $\top$ by the predicate on the right hand side of the definition. The predicate sets (B.6)–(B.9) are all parameterized by various terms: in the case of lenum n by a natural number n so that $\mathsf{lenum}\ n = \{0, 1, \ldots, n\}$. The sets (B.7)–(B.9) are polymorphic as well as parametric,

which of course is just another way of saying they are parameterized by types as well as terms. The set $\mathsf{nlist}\ n$ contains all α-lists having length n; the set $\mathsf{list}\ P$ contains all α-lists satisfying the condition that each member must lie in the parameter set P; and finally the set $\mathsf{pair}\ P\ Q$ contains all (α, β)-pairs (x, y) where x lies in the first parameter set P and y lies in the second parameter set Q.

B.2.2 Subtype Constructors

The definitions of list (B.8) and pair (B.9) in the previous subsection illustrated 'lifting': an α-set is lifted to an α-list set using list, and an α-set and a β-set are lifted to a $(\alpha \times \beta)$-set using pair. We might thus call list and pair subtype constructors, and we can similarly define subtype constructors for every datatype. The automatic generation of these constructors might be a worthwhile addition to the HOL datatype package.

We can also define a subtype constructor for the function space $\alpha \to \beta$.

Definition 62 *The Function Space Subtype Constructor*

$$\vdash \quad \forall P, Q.\ P \dot{\to} Q = \{f : \alpha \to \beta \mid \forall x \in P.\ f(x) \in Q\} \tag{B.10}$$

Given sets P and Q, f is in the function space $P \dot{\to} Q$ if it maps every element of P to an element of Q.[5]

We can illustrate the function space subtype constructor with the following theorems that follow from the definitions so far:[6]

$$\vdash (\lambda x.\ x^2) \in \mathsf{nzreal} \dot{\to} \mathsf{nzreal} \tag{B.11}$$

$$\vdash (\lambda x.\ x^2) \in \mathcal{U}_{\mathbb{R}} \dot{\to} \mathsf{nnegreal} \tag{B.12}$$

$$\vdash \forall f, p.\ f \in p \dot{\to} \mathcal{U}_{\mathbb{R}} \tag{B.13}$$

$$\vdash \forall f, q.\ f \in \emptyset \dot{\to} q \tag{B.14}$$

$$\vdash \forall f, p.\ f \in p \dot{\to} \emptyset \iff p = \emptyset \tag{B.15}$$

There is an alternative subtype constructor for function spaces, where the result set is *dependent* on the input.

Definition 63 *The Dependent Function Space Subtype Constructor*

$$\vdash \quad \forall P, Q.\ P \overset{\star}{\to} Q = \{f \mid \forall x \in P.\ f(x) \in Q(x)\} \tag{B.16}$$

The difference here is that Q is a parameterized set, where the parameter comes from the set P. This allows us to model dependent predicate subtypes with predicate sets, such as the following for natural number subtraction:

$$\mathcal{U}_{\mathbb{N}} \overset{\star}{\to} (\lambda n.\ \mathsf{lenum}\ n \dot{\to} \mathsf{lenum}\ n) \tag{B.17}$$

[5] This definition makes use of the restricted quantifier notation of Wong (1993), and $\forall x \in p.\ M(x)$ expands to $\forall x.\ x \in p \Rightarrow M(x)$.

[6] The infix operator $\dot{\to}$ associates to the right and has tighter binding than $\in$, so $f \in p \dot{\to} q \dot{\to} r$ means the same as $f \in (p \dot{\to} (q \dot{\to} r))$. Also x^2 here means x squared.

Recall that $\mathcal{U}_{\mathbb{N}}$ is the universe set for the type $\mathbb{N}$ of natural numbers. We should therefore read subtype (B.17) above as the set of functions that: given any natural number n return a function from $\{0, \ldots, n\}$ to $\{0, \ldots, n\}$. One point to note: if the parameterized set Q has the form $\lambda\, x.\ Q'$ where Q' does not contain any occurrences of the bound variable x, then $P \xrightarrow{\star} Q = P \xrightarrow{\cdot} Q'$: this shows that Definition 63 is more general than Definition 62. Dependent types are familiar from type theory, and are incorporated into the LEGO theorem prover.[7]

PVS also has two function space subtypes, covering the dependent and non-dependent cases. It also allows dependent products, which we do not use. However, analogously to $\xrightarrow{\star}$ it is simple to define a dependent pair constructor dpair, taking a set P and a parameterized set Q:

$$\vdash \quad \forall P, Q.\ \mathsf{dpair}\ P\ Q = \{(x, y) \mid x \in P \wedge y \in Q(x)\} \tag{B.18}$$

B.2.3 Subtype Rules

Now that we have defined the form of subtype sets, we shall show how to derive subtypes of a HOL term. Given a term t, we say that P is a subtype of t if we can prove the theorem $\vdash t \in P$. In our model, these HOL theorems are analogous to PVS subtype judgements.

The type inference algorithm of Milner (1978) for simply-typed λ-calculus is structural: a single bottom-up pass of a well-formed term suffices to establish its most general type.[8] Subtype inference also proceeds by making a single bottom-up pass to derive subtypes, though the new situation is complicated by two factors:

- two rules to break down terms (covering function applications and λ-abstractions) are no longer sufficient, since we also need to keep track of logical context;
- there is no concept of a 'most general set of subtype theorems',[9] so instead we perform proof search up to some fixed depth and return all the subtype theorems that we can prove.

Given a term t, the subtype inference algorithm finds sets P that satisfy $t \in P$. Note that like the simply-typed λ-calculus, there is essentially no computational difference between type inference (i.e., finding P such that $t \in P$ holds) and type checking (i.e., proving $t \in P$ for a given P).

To keep track of logical context, we create subtype rules similar to the congruence rules of a contextual rewriter. Here are some examples:

$$\vdash \quad \forall c, a, b, P. \qquad (B.19)$$
$$(c \in \mathcal{U}_{\mathbb{B}}) \wedge (c \Rightarrow a \in P) \wedge (\neg c \Rightarrow b \in P) \Rightarrow (\mathsf{if}\ c\ \mathsf{then}\ a\ \mathsf{else}\ b) \in P$$

$$\vdash \quad \forall a, b.\ (b \Rightarrow a \in \mathcal{U}_{\mathbb{B}}) \wedge (a \Rightarrow b \in \mathcal{U}_{\mathbb{B}}) \Rightarrow (a \wedge b) \in \mathcal{U}_{\mathbb{B}} \tag{B.20}$$

$$\vdash \quad \forall f, a, P, Q.\ (f \in P \xrightarrow{\star} Q) \wedge (a \in P) \Rightarrow f(a) \in Q(a) \tag{B.21}$$

$$\vdash \quad \forall f, P.\ (\forall x.\ f(x) \in P(x)) \Rightarrow (\lambda\, x.\ f(x)) \in (\mathcal{U}_\alpha \xrightarrow{\star} P) \tag{B.22}$$

[7] `http://www.dcs.ed.ac.uk/home/lego/`

[8] Though not completely avoiding all difficulty: Mairson (1990) has shown that the most general simple type of a term can be exponentially large in the size of the term.

[9] Theoretically we could intersect all subtypes that a term t satisfies, but then we would just end up with $\{t\}$ if the logical context was consistent, or $\emptyset$ if it was not!

These rules are rather long, but can be read easily from left to right. For example the conditional subtype rule (B.19) reads: "if we can show c to be in $\mathcal{U}_{\mathbb{B}}$; and assuming c we can show a to be in a subtype P; and assuming $\neg c$ we can show b to be in the same subtype P; then the combined term $\mathsf{if}\ c\ \mathsf{then}\ a\ \mathsf{else}\ b$ must also be an element of P." In this way we can build up logical context. Note that c is trivially in the universe set $\mathcal{U}_{\mathbb{B}}$, the only purpose of retaining this condition is to force the subtype checker to recurse into c and check all its subterms. The conjunction rule (B.20) similarly ensures that subterms are covered by the subtype checker, while building the correct logical context.[10] Also shown are the subtype rules for function application (B.21) and abstraction (B.22), the main point to note is that they both use the more general dependent version $\xrightarrow{\star}$ of the subtype constructor for function spaces.

For each boolean constant , we need a subtype rule of the above form. Therefore the set of subtype rules used is not fixed; we allow the user to add rules for new constants.

Subtype rules tell us how to derive subtypes for a term by combining the subtypes of smaller terms, but they leave two questions unanswered: how to calculate the subtypes of base terms (variables and constants); and how to unify the (possibly higher-order) subtypes of the smaller terms, for example to match the two occurrences of P in the antecedent of the conditional subtype rule (B.19). These questions are addressed in the next two sections.

B.2.4 Subtypes of Constants

To calculate subtypes of a base term $t : \alpha$, we maintain a dictionary of constant subtypes.[11] If the term we are focussed on is a constant that appears in the dictionary, we return the subtype theorem listed there. If the term is a variable or a constant that is not in the dictionary, we return the default subtype theorem $\vdash t \in \mathcal{U}_\alpha$.[12]

Here are some miscellaneous entries in the dictionary:

$$\vdash \mathsf{sqrt} \in (\mathsf{nnegreal} \xrightarrow{\cdot} \mathsf{nnegreal} \cap \mathsf{posreal} \xrightarrow{\cdot} \mathsf{posreal}) \tag{B.23}$$

$$\vdash \mathsf{inv} \in (\mathsf{nzreal} \xrightarrow{\cdot} \mathsf{nzreal} \cap \mathsf{posreal} \xrightarrow{\cdot} \mathsf{posreal} \cap \mathsf{negreal} \xrightarrow{\cdot} \mathsf{negreal}) \tag{B.24}$$

$$\vdash \forall n.\ - \in \mathcal{U}_{\mathbb{N}} \xrightarrow{\star} (\lambda n.\ \mathsf{lenum}\ n \xrightarrow{\cdot} \mathsf{lenum}\ n) \tag{B.25}$$

$$\vdash \forall P.\ \mathsf{funpow} \in (P \xrightarrow{\cdot} P) \xrightarrow{\cdot} \mathcal{U}_{\mathbb{N}} \xrightarrow{\cdot} P \xrightarrow{\cdot} P \tag{B.26}$$

$$\vdash \forall P.\ [\,] \in (\mathsf{list}\ P \cap \mathsf{nlist}\ 0) \tag{B.27}$$

$$\vdash \forall P, n.\ \mathsf{cons} \in P \xrightarrow{\cdot} (\mathsf{list}\ P \cap \mathsf{nlist}\ n) \xrightarrow{\cdot} (\mathsf{list}\ P \cap \mathsf{nlist}\ (\mathsf{suc}\ n)) \tag{B.28}$$

[10] A version of the conjunction rule that does not always return the universe set $\mathcal{U}_{\mathbb{B}}$ is as follows:

$$\vdash \forall a, b, P, Q.\ (b \Rightarrow a \in P) \wedge (a \Rightarrow b \in Q) \Rightarrow (a \wedge b) \in (\{\bot\} \cup (P \mathbin{\dot\wedge} Q))$$

where $\dot\wedge$ is a lifted version of $\wedge$ that operates on sets of booleans instead of booleans. However, the version we present is much simpler to work with and usually all that is required in practice.

[11] It is up to the user to add constant subtypes to the dictionary: as yet there is no mechanism to automatically generate these for newly defined constants, though this further work is briefly discussed in Section B.5.

[12] Note that when we come to use the subtype of t later on, other subtypes may also be deduced from the logical context.

$$\vdash \ \forall f, P, Q, n. \qquad \text{(B.29)}$$
$$\mathsf{map} \in (P \dot{\to} Q) \dot{\to} (\mathsf{list}\ P \cap \mathsf{nlist}\ n) \dot{\to} (\mathsf{list}\ Q \cap \mathsf{nlist}\ n)$$
$$\vdash \ \forall P, Q, n. \qquad \text{(B.30)}$$
$$\mathsf{zip} \in$$
$$(\mathsf{nlist}\ n \cap \mathsf{list}\ P) \dot{\to} (\mathsf{nlist}\ n \cap \mathsf{list}\ Q) \dot{\to} (\mathsf{nlist}\ n \cap \mathsf{list}\ (\mathsf{pair}\ P\ Q))$$

The universal quantification allows variables in the types of constants, and exactly like 'forall types' in functional programming these generate fresh variables at every instance of the constant. The intersections that occur in the subtypes are a crude mechanism to model full intersection types in type theory (Compagnoni, 1995).

This dictionary corresponds to the constant judgement mechanism of PVS, whereby the type-checker can be told that for the purpose of calculating type correctness conditions, particular constants are also elements of more specific subtypes than their principal subtype.[13]

B.2.5 Subtype Judgements

Suppose we have a subtype rule that we are committed to using, and we have recursively derived subtype theorems for the terms in the antecedent of the rule. We must now deduce[14] from these subtype theorems, aiming to find a consistent set of subtype theorems that is matched by the antecedent of the rule.

Example: Suppose our term is $f(a)$ (where f has simple type $\mathbb{R} \to \mathbb{R}$); we are using the function application subtype rule (B.21); and we have recursively shown that $\vdash f \in \mathsf{nzreal} \dot{\to} \mathsf{nzreal}$ and $\vdash a \in \mathsf{posreal}$. However, in order to apply the rule we must find instantiations of the variables P and Q such that

$$(f \in P \xrightarrow{\star} Q) \ \wedge \ (a \in P)$$

is a theorem. We present this goal to our prover, which performs bounded proof search and returns some instantiations, one of which corresponds to the following theorem:

$$\vdash \ \ (f \in \mathsf{nzreal} \xrightarrow{\star} (\lambda\, x.\ \mathsf{nzreal})) \ \wedge \ (a \in \mathsf{nzreal})$$

Now we can apply the rule to conclude that $\vdash f(a) \in (\lambda\, x.\ \mathsf{nzreal})\ a$, which can in turn be simplified to $\vdash f(a) \in \mathsf{nzreal}$. $\square$

In this example, the prover needed to show $a \in \mathsf{posreal} \Rightarrow a \in \mathsf{nzreal}$. Steps like these are achieved using subtype judgements:[15] theorems that are manually added to the

[13] The principal subtype is the one asserted when the constant is defined.

[14] Deciding the logical context in which we should perform this deduction is quite delicate. It is sound to use the current logical context, but not complete. A more careful approach is to use the (possibly larger) logical context of a subterm whenever we manipulate the subtypes of that subterm. In this way if we can deduce $\vdash 1 \in \mathsf{nzreal}$ and $\vdash \neg\top \Rightarrow 0 \in \mathsf{nzreal}$ then we will be able to deduce $\vdash (\mathsf{if}\ \top\ \mathsf{then}\ 1\ \mathsf{else}\ 0) \in \mathsf{nzreal}$ using the conditional rule.

[15] The name 'subtype judgements' was borrowed from PVS, which contains declarations used for exactly the same purpose.

top-level logical context, and are available for use in deriving subtypes. These will be theory specific, and can be extended by the user at any time. Examples are:

$$\vdash \quad \mathsf{posreal} \subset \mathsf{nzreal} \tag{B.31}$$

$$\vdash \quad \forall P, Q.\ P \subset Q \ \Rightarrow\ \mathsf{list}\ P \subset \mathsf{list}\ Q \tag{B.32}$$

From the example we can see that a suitable prover must be: higher-order to deal with parameterized types; able to find multiple instantiations of a goal ('prolog-style'); and able to perform bounded proof search. Any prover that satisfies these criteria will be able to plug in at this point and enable subtype derivation.

However, since there are not many provers available that can satisfy all these requirements, we have implemented one to test our subtype derivation algorithm. Robinson (1970) presents an approach to higher-order proving by converting all terms to combinatory form.[16] Together with translation to CNF this conversion leaves terms in a normal form that simplifies the writing of automatic proof search tools. For our application we implement a version of model elimination (mostly following the presentation of Harrison (1996b), with some higher-order extensions), since that is able to return multiple instantiations of goals and we can use a simple depth-bound to limit the search. More powerful normalization means that it can compete with the HOL first-order prover `MESON_TAC` on some first-order problems, and results on higher-order problems are basic but promising.

B.2.6 Subtype Derivation Algorithm

To summarize this section, we present the complete algorithm to derive subtypes of a term.

Inputs: A term t having simple type α; a logical context C initialized with a set of assumptions and the current subtype judgements; a set R of subtype rules; and a dictionary D of constant subtypes.

Outputs: A set S of subtype theorems.

1. If t is a variable, return $S = \{\vdash t \in \mathcal{U}_\alpha\}$.

2. If t is a constant, look in the dictionary D to see if there is an entry for t. If so, return the entry $S = \{\vdash t \in P\}$, otherwise return $S = \{\vdash t \in \mathcal{U}_\alpha\}$.

3. Otherwise find a subtype rule in R matching t.[17] The rule will have the form

$$\vdash \forall \vec{v}.\ \left(\bigwedge_{1 \le i \le n} \forall \vec{v_i}.\ a_i[\vec{v_i}] \Rightarrow t_i[\vec{v_i}] \in P_i[\vec{v_i}, \vec{v}] \right) \Rightarrow t \in P[\vec{v}] \tag{B.33}$$

4. For each $1 \le i \le n$, create the logical context C_i by adding the assumption $a_i[\vec{v_i}]$ to C and recursively apply the algorithm using C_i to t_i to find a set of subtypes

$$S_i = \{\vdash t_i[\vec{v_i}] \in P_{i0}[\vec{v_i}],\ \ldots,\ \vdash t_i[\vec{v_i}] \in P_{in_i}[\vec{v_i}]\}$$

[16] Many thanks to John Harrison for drawing my attention to this paper.

[17] If there is more than one rule that matches, return the rule that was most recently added to R: this is almost always the most specific rule too.

5. Find consistent sets of subtypes by calling the following search function with counter $i \leftarrow 1$ and instantiation $\sigma \leftarrow id$.

   (a) If $i > n$ then return σ.

   (b) Using the subtype theorems in S_i and the logical context C_i, use the higher-order prover to find theorems matched by the form

   $$\vdash \quad t_i[\vec{v_i}] \in \sigma(P_i[\vec{v_i}, \vec{v}])$$

   (c) For each theorem returned, let σ_i be the returned instantiation. Recursively call the depth-first search function with counter $i \leftarrow i + 1$ and instantiation $\sigma \leftarrow (\sigma_i \circ \sigma)$.

6. Each instantiation σ returned by the above search corresponds to a specialization of the subtype rule (B.33) for which we have proven the antecedent. We may thus deduce the consequent by modus ponens, and we add this to the result set S of subtype theorems.

B.3 Subtype-checking in HOL

B.3.1 Debugging Specifications

We use our subtype derivation algorithm to find errors in a specification s, by invoking the algorithm on s, and generating an exception whenever the algorithm would return an empty set of subtypes for a subterm.

We illustrate this with the following family of specifications:

$$(\mathsf{inv}\ x) * x = 1 \tag{B.34}$$

$$x \in \mathsf{nzreal} \ \Rightarrow \ (\mathsf{inv}\ x) * x = 1 \tag{B.35}$$

$$\mathsf{inv}\ x \in \mathsf{nzreal} \ \Rightarrow \ (\mathsf{inv}\ x) * x = 1 \tag{B.36}$$

$$\mathsf{inv} \in \mathcal{U}_{\mathbb{R}} \xrightarrow{\cdot} \mathsf{nzreal} \ \Rightarrow \ (\mathsf{inv}\ x) * x = 1 \tag{B.37}$$

$$\mathsf{inv} \in \mathcal{U}_{\mathbb{R}} \xrightarrow{\cdot} \mathcal{U}_{\mathbb{R}} \ \Rightarrow \ (\mathsf{inv}\ x) * x = 1 \tag{B.38}$$

The subtype checker will generate an exception for specification (B.34), complaining that it could not derive a type for the subterm $\mathsf{inv}\ x$. An exception was raised because the algorithm could not find a consistent set of subtypes for the subterms inv and x, when using the subtype rule (B.21) for function application. And this we see to be true, because without any additional knowledge of x we cannot show it to be in any of the sets nzreal, $\mathsf{posreal}$ or $\mathsf{negreal}$ that the subtype (B.24) of the constant inv demands.

Specification (B.35) shows the right solution: add a guard to stop x from taking the illegal value of 0. And this solves the problem, the subtype checker can now derive a subtype of nzreal for the subterm $\mathsf{inv}\ x$ (and a subtype of $\mathcal{U}_{\mathbb{B}}$ for the whole term).

This is how we would expect the subtype checker to be used in practice. A specification is entered and subtype checked, the errors are corrected by adding the necessary guards, and only then is verification started. This could potentially save much wasted effort and act as a valuable teaching tool. In addition, the derived theorems that show subtype-correctness could be provided to the user for later use.

Specifications (B.36)–(B.38) represent various attempts to subvert the subtype checker. Specification (B.36) is a silly attempt: now the $\mathsf{inv}\ x$ in the antecedent fails to subtype check! However, even if the antecedent were added unchecked to the logical context, the consequent would still not subtype check: an extra subtype for $\mathsf{inv}\ x$ does not help at all in the search to find consistent subtypes for inv and x using the rule for function application. Specification (B.37) steps up a level in the arms race by assuming a subtype for inv, and now this term does subtype check since the higher-order prover just needs to show that $x \in \mathcal{U}_{\mathbb{R}}$: a triviality. However, we may take consolation in the fact that this antecedent is unprovable. Finally Specification (B.38) is the most worrying attack: the term subtype checks, and we can use theorem (B.13) to prove the condition.

B.3.2 Logical Limits

In the previous section we showed by example how the subtype checker that we implement may be duped into passing a term that contains a violation of subtype constraints. Unfortunately, this is not just an inadequacy of the subtype checker, but rather an inescapable consequence of reasoning with predicate sets in the HOL logic. Since HOL is a logic of total functions, given any function $f : \alpha \to \beta$ we can prove the theorem

$$\vdash \quad f \in \mathcal{U}_\alpha \dot{\to} \mathcal{U}_\beta \tag{B.39}$$

since this just expands to

$$\vdash \forall\, x \in \mathcal{U}_\alpha.\ f(x) \in \mathcal{U}_\beta$$

which is true by the definition of the universal set $\mathcal{U}_\beta$.

In particular, theorem (B.39) is true for all function constants f that we would like to have a restricted domain (such as inv). But the subtype $\mathcal{U}_\alpha \to \mathcal{U}_\beta$ allows f to be applied to any element of type α. This means that enforceable predicate subtyping using predicate sets cannot exist as a layer on top of the existing HOL kernel.

Example: Even if the user is not trying to subvert the system, it might happen accidentally. If we are subtype checking the specification

$$P \Rightarrow Q\ (\mathsf{inv}\ 0)$$

then when we subtype check the consequent $Q\ (\mathsf{inv}\ 0)$ we add the antecedent P to the logical context, and it might transpire that P somehow causes the higher-order prover to deduce $\mathsf{inv} \in \mathcal{U}_{\mathbb{R}} \dot{\to} \mathcal{U}_{\mathbb{R}}$,[18] which then allows $Q\ (\mathsf{inv}\ 0)$ to be successfully subtype checked. □

To summarize the situation, if a specification fails to subtype-check[19] then there's definitely a problem with it, if subtype-checking succeeds then there could be a hidden problem with it. It's like testing: subtype-checking in this context can only show the presence of errors.

PVS is also a logic of total functions, but the ability to make a first-class type of nonzero reals means that if inv is declared to have type $\mathbb{R}^{\neq 0} \to \mathbb{R}^{\neq 0}$ then the type-system can

[18] After all, it is a theorem!

[19] with sufficiently generous search bounds

stop the function from being 'lifted' to a larger type. Essentially the PVS logic implements a logic of partial functions, but by insisting that a type is available for every function's domain can avoid awkward questions of definedness.

B.4 Predicate Set Prover and Applications

B.4.1 Predicate Set Prover

An obvious application for the subtype derivation algorithm is to prove set membership goals. Supposing the current goal is $t \in P$, we can derive a set S of subtype theorems for t, and then invoke the higher-order prover once again with the top-level context and the set S to tackle the goal directly. We package up this proof procedure into a HOL tactic called the predicate set prover.

Example: To illustrate the two steps of the predicate set prover, consider the goal $3 \in \mathsf{nzreal}$.[20] Subtypes are derived for the term 3 (of type $\mathbb{R}$), and the following set of subtype theorems are returned:

$$\{\vdash 3 \in \mathsf{K}\ \mathsf{posreal}\ 3,\ \vdash 3 \in \mathsf{K}\ \mathsf{nnegreal}\ 3\}$$

(where $\mathsf{K} = \lambda x.\, \lambda y.\, x$). Next these two theorems are passed to the higher-order prover along with the top-level logical context containing type-judgements, and it quickly proves the goal $\vdash 3 \in \mathsf{nzreal}$. $\Box$

We can prove some interesting goals with the predicate set prover:

$$\vdash\ \mathsf{map\ inv\ (cons}\ (-1)\ \mathsf{(map\ sqrt}\ [3,1])) \in \mathsf{list\ nzreal}$$
$$\vdash\ (\lambda x \in \mathsf{negreal.\ funpow\ inv}\ n\ x) \in \mathsf{negreal} \dot{\rightarrow} \mathsf{negreal}$$

One optimization that is effective with this tactic is to maintain a cache of the subtypes that have already been derived for HOL constants.[21] For example, the innocuous-looking term '3' used in the above example is actually composed of 4 nested function applications! Re-deriving subtypes for constants is unnecessary and inefficient.

Another optimization arises naturally from certain subtype rules, such as the conjunction rule (B.20), repeated here:

$$\forall a, b.\ (b \Rightarrow a \in \mathcal{U}_{\mathbb{B}}) \wedge (a \Rightarrow b \in \mathcal{U}_{\mathbb{B}}) \Rightarrow (a \wedge b) \in \mathcal{U}_{\mathbb{B}}$$

The set $\mathcal{U}_{\mathbb{B}}$ is the universe set $\mathcal{U}_{\mathbb{B}}$ of booleans, so we can immediately prove $\vdash b \Rightarrow a \in \mathcal{U}_{\mathbb{B}}$ and $\vdash a \Rightarrow b \in \mathcal{U}_{\mathbb{B}}$ without recursing into the structure of the subterms a and b. Note that if we were deriving subtypes in order to check the term for subtype correctness then we would be obliged to carry out this recursion step to check a and b, but if our goal is proving set membership then we can safely skip this.

[20] This is not quite as trivial as it looks, since the real number '3' in HOL is really the complex term real_of_num (numeral (bit1 (bit1 0))).

[21] Here 'constant' means any term having no free variables.

The predicate set prover allows us to gain much of the proving benefit that PVS predicate subtyping provides,[22] and in some ways it does even better: the subtype derivation algorithm is not restricted to calculating with principal subtypes, but rather with any subtype that the term can be shown to satisfy using the derivation rules.

B.4.2 Proving Conditions During Rewriting

We can use the predicate set prover as a condition prover in a contextual rewriter, and there are several reasons why it is useful to integrate these tools:

- There is a trend to incorporate tools into contextual rewriters because of the automatic subterm traversal and context accumulation. The logical context built up by the contextual rewriter is easily transferred to the predicate set prover, and the subterm traversal allows us to attempt a proof of all occurrences of $t \in P$ in the goal term.[23]
- Many rewrites have side conditions that can be expressed very naturally using restricted quantifiers, and these generate goals for the predicate set prover when the rewrite is applied.
- Subtype judgements, rules and constants can be stored with the simplification set of a theory, thus reducing the administration burden of theory-specific rules.

Here are some miscellaneous rewrite rules that make use of subtype conditions:

$$\vdash \forall x \in \mathsf{nzreal}.\ x/x = 1 \tag{B.40}$$
$$\vdash \forall n.\ \forall m \in \mathsf{lenum}\ n.\ m + (n - m) = n \tag{B.41}$$
$$\vdash \forall n \in \mathsf{nznum}.\ n\ \mathsf{mod}\ n = 0 \tag{B.42}$$
$$\vdash \forall s \in \mathsf{finite}.\ \forall f \in \mathsf{injection}\ s.\ |\mathsf{image}\ f\ s| = |s| \tag{B.43}$$
$$\vdash \forall G \in \mathsf{group}.\ \forall g \in \mathsf{set}\ G.\ \mathsf{id}_G *_G g = g \tag{B.44}$$
$$\vdash \forall G \in \mathsf{group}.\ \forall g, h \in \mathsf{set}\ G.\ (g *_G h = h) = (g = \mathsf{id}_G) \tag{B.45}$$

Using rule (B.40), a term like $5/5 = 3/3$ is straightforwardly rewritten to $\top$. The last two examples come from a body of computational number theory that we formalized to verify the Miller-Rabin probabilistic primality test (Chapter 5).

An effective optimization for this tool is to make adding assumptions into the subtype logical context a lazy operation. This delays their conversion to combinatory form and CNF normalization until the predicate set prover is invoked on a goal, which might not happen at all.

In theory, the architecture laid out in this chapter can establish much more exotic properties than these, but the predicate subtype prover was found to be most useful

[22] Although almost certainly the performance using our model is worse than implementing predicate subtyping as part of the logic: keeping subtypes with the terms so they never have to be re-derived must provide a boost. We also experimented with this kind of architecture, but eventually dropped it to simplify the design and increase interoperability with existing tools.

[23] When subtype derivation is applied to a subterm it accumulates context in much the same way as a contextual rewriter. However, despite this similarity, the two tools are orthogonal and are best implemented separately: we tried both approaches.

and robust on these relatively simple properties that come up again and again during conditional rewriting. These properties naturally propagate upwards through a term, being preserved by most of the basic operations, and in such situations the predicate set prover can be relied upon to show the desired condition (albeit sometimes rather slowly). This tool lent itself to more efficient development of the required theories, particularly the group theory where almost every theorem has one or more group membership side-conditions. Our evaluation is that it is a useful tool for reasoning about term properties that naturally propagate up terms.

B.5 Concluding Remarks

We have demonstrated that predicate subtyping can be simulated in HOL using predicate sets, and presented a new automatic proof procedure that takes advantage of this new style of reasoning. This proved to be an effective proof tool on a case study of formalized computational number theory.

A comparison of HOL and PVS was made by Gordon (1996) from the perspectives of logic, automatic proof and usability. The model of predicate subtyping using predicate sets builds upon HOL88 restricted quantifier library of Wong (1993), and the exact details of the predicate subtyping in our model attempts to follow the PVS architecture. For the full details of the semantics and implementation of subtyping in PVS, refer to Owre and Shankar (1997).

Previous work in this area has been done by Jones (1997), who built a tool in HOL to specify the subtype of constants and subtype check terms with respect to the subtypes. Given a term t, the tool sets up HOL goals that, if proved, would correspond to the term being subtype-correct. The user is then free to use these extra theorems during verification. Our model extends Jones' by the introduction of subtype rules for generating type-correctness conditions, the higher-order prover to automatically prove conditions, and the integration of the tool into a rewriter to aid interactive proof.

ACL2 uses guards (Kaufmann and Moore, 1997) to stop functions from being applied outside their domain; these generate proof obligations when new functions are defined in terms of guarded functions. When the proof obligations have been satisfied the new function is given a 'gold' status, and can be safely executed without causing run-time type errors. This is very similar to the way PVS type-checks terms before admitting them into the system.

A system of guard formulas in Z/EVES has also been implemented by Saaltink (1997); similarly to our work these guards are used to aid formal Z proofs and find errors in Z specifications. The Z logic allows terms to be 'undefined', but the system of guard formulas imposed will flag the situations that can result in undefinedness, allowing classical reasoning on the partial logic. Since Z is based on set theory, this use of guards does not suffer from the logical limitations we outlined in Section B.3.2, and can provide strong guarantees about a checked specification. However, whereas our subtype rules propagate all available logical information around the term, Saaltink chooses a "left-to-right system of interpretation" that is not complete, but works well in most practical situations and simplifies guard conditions.

There has also been a huge amount of work on subtyping and polymorphism in various λ-calculi, used to model object-oriented programming languages. Some concepts from this

field are related to our work, in particular the notion of intersection types corresponds to finding multiple subtypes of a term. The thesis of Compagnoni (1995) provides a good introduction to this area.

Since we now have a preliminary evaluation of the predicate subtype prover, we must decide the direction in which to advance it. On the one hand it may be possible to widen its effective scope by including rules for building up more complicated predicate sets, or on the other hand it may be most useful to speed up the performance and make it more robust on the scope we have identified here. The path we take may depend on how far we can push the underlying higher-order prover.

Another interesting extension to this research would be to use subtype checking to perform subtype inference of new constants. This would be extremely useful: currently for each constant which we would like to add to the constant subtype dictionary, we must prove a result of the form of theorems (B.24)–(B.30). One way to approach this task would be to initially set the type of the new constant c to be $c \in P$ where P is a variable; during subtype checking collect constraints on P; and finally at the top-level try to solve these constraints: the solutions being subtypes for c.

Appendix C

The HOL Probability Theories (Abridged)

C.1 measure Theory

```
[additive_def]
|- !m.
     additive m =
     !s t.
       s IN measurable_sets m /\ t IN measurable_sets m /\
       DISJOINT s t ==>
       (measure m (s UNION t) = measure m s + measure m t)

[algebra_def]
|- !a.
     algebra a =
     {} IN a /\ (!s. s IN a ==> COMPL s IN a) /\
     !s t. s IN a /\ t IN a ==> s UNION t IN a

[countably_additive_def]
|- !m.
     countably_additive m =
     !f.
       f IN (UNIV -> measurable_sets m) /\
       (!m n. ~(m = n) ==> DISJOINT (f m) (f n)) /\
       BIGUNION (IMAGE f UNIV) IN measurable_sets m ==>
       measure m o f sums measure m (BIGUNION (IMAGE f UNIV))

[countably_subadditive_def]
|- !m.
     countably_subadditive m =
     !f.
       f IN (UNIV -> measurable_sets m) /\
       BIGUNION (IMAGE f UNIV) IN measurable_sets m /\
       summable (measure m o f) ==>
       measure m (BIGUNION (IMAGE f UNIV)) <= suminf (measure m o f)

[increasing_def]
|- !m.
```

```
     increasing m =
     !s t.
       s IN measurable_sets m /\ t IN measurable_sets m /\
       s SUBSET t ==>
       measure m s <= measure m t

[inf_measure_def]
|- !m s.
     inf_measure m s =
     inf
       {r
        |
        ?f.
          f IN (UNIV -> measurable_sets m) /\
          (!m n. ~(m = n) ==> DISJOINT (f m) (f n)) /\
          s SUBSET BIGUNION (IMAGE f UNIV) /\ measure m o f sums r}

[lambda_system_def]
|- !g0 lam.
     lambda_system g0 lam =
     {l
      |
      l IN g0 /\
      !g. g IN g0 ==> (lam (l INTER g) + lam (COMPL l INTER g) = lam g)}

[measurable_def]
|- !a b. measurable a b = {f | !s. s IN b ==> PREIMAGE f s IN a}

[measurable_sets_def]
|- !a mu. measurable_sets (a,mu) = a

[measure_def]
!a mu. measure (a,mu) = mu

[measure_preserving_def]
|- !m1 m2.
     measure_preserving m1 m2 =
     {f
      |
      f IN measurable (measurable_sets m1) (measurable_sets m2) /\
      !s.
        s IN measurable_sets m2 ==>
        (measure m1 (PREIMAGE f s) = measure m2 s)}

[measure_space_def]
|- !m.
     measure_space m =
     sigma_algebra (measurable_sets m) /\ positive m /\
     countably_additive m

[outer_measure_space_def]
|- !m.
     outer_measure_space m =
     positive m /\ increasing m /\ countably_subadditive m

[positive_def]
```

```
|- !m.
     positive m =
     (measure m {} = 0) /\
     !s. s IN measurable_sets m ==> 0 <= measure m s

[sigma_algebra_def]
|- !a.
     sigma_algebra a =
     algebra a /\ !c. countable c /\ c SUBSET a ==> BIGUNION c IN a

[sigma_def]
|- !b. sigma b = BIGINTER {a | b SUBSET a /\ sigma_algebra a}

[subadditive_def]
|- !m.
     subadditive m =
     !s t.
       s IN measurable_sets m /\ t IN measurable_sets m ==>
       measure m (s UNION t) <= measure m s + measure m t

[ADDITIVE_INCREASING]
|- !m.
     algebra (measurable_sets m) /\ positive m /\ additive m ==>
     increasing m

[ALGEBRA_COMPL]
|- !a s. algebra a /\ s IN a ==> COMPL s IN a

[ALGEBRA_DIFF]
|- !a s t. algebra a /\ s IN a /\ t IN a ==> s DIFF t IN a

[ALGEBRA_EMPTY]
|- !a. algebra a ==> {} IN a

[ALGEBRA_FINITE_UNION]
|- !a c. algebra a /\ FINITE c /\ c SUBSET a ==> BIGUNION c IN a

[ALGEBRA_INTER]
|- !a s t. algebra a /\ s IN a /\ t IN a ==> s INTER t IN a

[ALGEBRA_SUBSET_LAMBDA_SYSTEM]
|- !m.
     algebra (measurable_sets m) /\ positive m /\ increasing m /\
     additive m ==>
     measurable_sets m SUBSET lambda_system UNIV (inf_measure m)

[ALGEBRA_UNION]
|- !a s t. algebra a /\ s IN a /\ t IN a ==> s UNION t IN a

[ALGEBRA_UNIV]
|- !a. algebra a ==> UNIV IN a

[CARATHEODORY]
|- !m0.
     algebra (measurable_sets m0) /\ positive m0 /\
     countably_additive m0 ==>
```

```
     ?m.
       (!s. s IN measurable_sets m0 ==> (measure m s = measure m0 s)) /\
       (measurable_sets m = sigma (measurable_sets m0)) /\
       measure_space m

[CARATHEODORY_LEMMA]
|- !gsig lam.
     sigma_algebra gsig /\ outer_measure_space (gsig,lam) ==>
     measure_space (lambda_system gsig lam,lam)

[COUNTABLY_ADDITIVE_ADDITIVE]
|- !m.
     algebra (measurable_sets m) /\ positive m /\
     countably_additive m ==>
     additive m

[COUNTABLY_SUBADDITIVE_SUBADDITIVE]
|- !m.
     algebra (measurable_sets m) /\ positive m /\
     countably_subadditive m ==>
     subadditive m

[INCREASING_ADDITIVE_SUMMABLE]
|- !m f.
     algebra (measurable_sets m) /\ positive m /\ increasing m /\
     additive m /\ f IN (UNIV -> measurable_sets m) /\
     (!m n. ~(m = n) ==> DISJOINT (f m) (f n)) ==>
     summable (measure m o f)

[INF_MEASURE_AGREES]
|- !m s.
     algebra (measurable_sets m) /\ positive m /\
     countably_additive m /\ s IN measurable_sets m ==>
     (inf_measure m s = measure m s)

[INF_MEASURE_OUTER]
|- !m.
     algebra (measurable_sets m) /\ positive m /\ increasing m ==>
     outer_measure_space (UNIV,inf_measure m)

[IN_SIGMA]
|- !a x. x IN a ==> x IN sigma a

[LAMBDA_SYSTEM_ADDITIVE]
|- !g0 lam l1 l2.
     algebra g0 /\ positive (g0,lam) ==>
     additive (lambda_system g0 lam,lam)

[LAMBDA_SYSTEM_ALGEBRA]
|- !g0 lam.
     algebra g0 /\ positive (g0,lam) ==> algebra (lambda_system g0 lam)

[LAMBDA_SYSTEM_CARATHEODORY]
|- !gsig lam.
     sigma_algebra gsig /\ outer_measure_space (gsig,lam) ==>
     !f.
```

```
       f IN (UNIV -> lambda_system gsig lam) /\
       (!m n. ~(m = n) ==> DISJOINT (f m) (f n)) ==>
       BIGUNION (IMAGE f UNIV) IN lambda_system gsig lam /\
       lam o f sums lam (BIGUNION (IMAGE f UNIV))

[LAMBDA_SYSTEM_INCREASING]
|- !g0 lam.
     increasing (g0,lam) ==> increasing (lambda_system g0 lam,lam)

[LAMBDA_SYSTEM_POSITIVE]
|- !g0 lam. positive (g0,lam) ==> positive (lambda_system g0 lam,lam)

[MEASURABLE_COMP]
|- !f g a b c.
     f IN measurable a b /\ g IN measurable b c ==>
     g o f IN measurable a c

[MEASURABLE_I]
|- !a. I IN measurable a a

[MEASURABLE_LIFT]
|- !f a b.
     sigma_algebra a /\ f IN measurable a b ==>
     f IN measurable a (sigma b)

[MEASURABLE_PROD_SIGMA]
|- !a a1 a2 f.
     sigma_algebra a /\ FST o f IN measurable a a1 /\
     SND o f IN measurable a a2 ==>
     f IN measurable a (sigma (prod_sets a1 a2))

[MEASURABLE_UP_LIFT]
|- !a b c f. f IN measurable a c /\ a SUBSET b ==> f IN measurable b c

[MEASURE_PRESERVING_LIFT]
|- !m1 m2 a f.
     measure_space m1 /\ measure_space m2 /\ algebra a /\
     (measurable_sets m2 = sigma a) /\
     f IN measure_preserving m1 (a,measure m2) ==>
     f IN measure_preserving m1 m2

[MEASURE_PRESERVING_UP_LIFT]
|- !m1 m2 f.
     f IN measure_preserving (a,measure m1) m2 /\
     a SUBSET measurable_sets m1 ==>
     f IN measure_preserving m1 m2

[MEASURE_SPACE_REDUCE]
|- !m. (measurable_sets m,measure m) = m

[MONOTONE_CONVERGENCE]
|- !m s f.
     measure_space m /\ f IN (UNIV -> measurable_sets m) /\
     (!n. f n SUBSET f (SUC n)) /\ (s = BIGUNION (IMAGE f UNIV)) ==>
     measure m o f --> measure m s
```

```
[SIGMA_ALGEBRA]
|- !p.
     sigma_algebra p =
     {} IN p /\ (!s. s IN p ==> COMPL s IN p) /\
     !c. countable c /\ c SUBSET p ==> BIGUNION c IN p

[SIGMA_ALGEBRA_SIGMA]
|- !b. sigma_algebra (sigma b)

[SIGMA_PROPERTY]
|- !p a.
     {} IN p /\ a SUBSET p /\
     (!s. s IN p INTER sigma a ==> COMPL s IN p) /\
     (!c.
        countable c /\ c SUBSET p INTER sigma a ==> BIGUNION c IN p) ==>
     sigma a SUBSET p

[SIGMA_PROPERTY_DISJOINT]
|- !p a.
     algebra a /\ a SUBSET p /\
     (!s. s IN p INTER sigma a ==> COMPL s IN p) /\
     (!f.
        f IN (UNIV -> p INTER sigma a) /\ (f 0 = {}) /\
        (!n. f n SUBSET f (SUC n)) ==>
        BIGUNION (IMAGE f UNIV) IN p) /\
     (!f.
        f IN (UNIV -> p INTER sigma a) /\
        (!m n. ~(m = n) ==> DISJOINT (f m) (f n)) ==>
        BIGUNION (IMAGE f UNIV) IN p) ==>
     sigma a SUBSET p

[UNIV_SIGMA_ALGEBRA]
|- sigma_algebra UNIV
```

C.2 probability Theory

```
[events_def]
|- events = measurable_sets

[indep_def]
|- !p a b.
     indep p a b =
     a IN events p /\ b IN events p /\
     (prob p (a INTER b) = prob p a * prob p b)

[indep_families_def]
|- !p q r. indep_families p q r = !s t. s IN q /\ t IN r ==> indep p s t

[indep_function_def]
|- !p.
     indep_function p =
     {f |
```

```
      indep_families p (IMAGE (PREIMAGE (FST o f)) UNIV)
        (IMAGE (PREIMAGE (SND o f)) (events p))}

[possibly_def]
|- !p e. possibly p e = e IN events p /\ ~(prob p e = 0)

[prob_def]
|- prob = measure

[prob_preserving_def]
|- prob_preserving = measure_preserving

[prob_space_def]
|- !p. prob_space p = measure_space p /\ (measure p UNIV = 1)

[probably_def]
|- !p e. probably p e = e IN events p /\ (prob p e = 1)

[EVENTS]
|- !a b. events (a,b) = a

[EVENTS_SIGMA_ALGEBRA]
|- !p. prob_space p ==> sigma_algebra (events p)

[INDEP_EMPTY]
|- !p s. prob_space p /\ s IN events p ==> indep p {} s

[INDEP_REFL]
|- !p a.
     prob_space p /\ a IN events p ==>
     (indep p a a = (prob p a = 0) \/ (prob p a = 1))

[INDEP_SYM]
|- !p a b. prob_space p /\ indep p a b ==> indep p b a

[INDEP_UNIV]
|- !p s. prob_space p /\ s IN events p ==> indep p UNIV s

[PROB]
|- !a b. prob (a,b) = b

[PROB_ADDITIVE]
|- !p s t u.
     prob_space p /\ s IN events p /\ t IN events p /\ DISJOINT s t /\
     (u = s UNION t) ==>
     (prob p u = prob p s + prob p t)

[PROB_COMPL]
|- !p s.
     prob_space p /\ s IN events p ==> (prob p (COMPL s) = 1 - prob p s)

[PROB_COUNTABLY_ADDITIVE]
|- !p s f.
     prob_space p /\ f IN (UNIV -> events p) /\
     (!m n. ~(m = n) ==> DISJOINT (f m) (f n)) /\
     (s = BIGUNION (IMAGE f UNIV)) ==>
```

```
     prob p o f sums prob p s

[PROB_EMPTY]
|- !p. prob_space p ==> (prob p {} = 0)

[PROB_FINITELY_ADDITIVE]
|- !p s f n.
     prob_space p /\ f IN (count n -> events p) /\
     (!a b. a < n /\ b < n /\ ~(a = b) ==> DISJOINT (f a) (f b)) /\
     (s = BIGUNION (IMAGE f (count n))) ==>
     (sum (0,n) (prob p o f) = prob p s)

[PROB_INCREASING]
|- !p s t.
     prob_space p /\ s IN events p /\ t IN events p /\ s SUBSET t ==>
     prob p s <= prob p t

[PROB_INCREASING_UNION]
|- !p s f.
     prob_space p /\ f IN (UNIV -> events p) /\
     (!n. f n SUBSET f (SUC n)) /\ (s = BIGUNION (IMAGE f UNIV)) ==>
     prob p o f --> prob p s

[PROB_LE_1]
|- !p s. prob_space p /\ s IN events p ==> prob p s <= 1

[PROB_ONE_INTER]
|- !p s t.
     prob_space p /\ s IN events p /\ t IN events p /\
     (prob p t = 1) ==>
     (prob p (s INTER t) = prob p s)

[PROB_POSITIVE]
|- !p s. prob_space p /\ s IN events p ==> 0 <= prob p s

[PROB_PRESERVING]
|- !p1 p2.
     prob_preserving p1 p2 =
     {f |
      f IN measurable (events p1) (events p2) /\
      !s. s IN events p2 ==> (prob p1 (PREIMAGE f s) = prob p2 s)}

[PROB_PRESERVING_LIFT]
|- !p1 p2 a f.
     prob_space p1 /\ prob_space p2 /\ algebra a /\
     (events p2 = sigma a) /\ f IN prob_preserving p1 (a,prob p2) ==>
     f IN prob_preserving p1 p2

[PROB_PRESERVING_UP_LIFT]
|- !p1 p2 f.
     f IN prob_preserving (a,prob p1) p2 /\ a SUBSET events p1 ==>
     f IN prob_preserving p1 p2

[PROB_SPACE]
|- !p.
     prob_space p =
```

```
     sigma_algebra (events p) /\ positive p /\ countably_additive p /\
     (prob p UNIV = 1)

[PROB_ZERO_UNION]
|- !p s t.
     prob_space p /\ s IN events p /\ t IN events p /\
     (prob p t = 0) ==>
     (prob p (s UNION t) = prob p s)
```

C.3 prob_dice Theory

```
[dice_def]
|- dice =
   coin_flip
   (prob_repeat
    (coin_flip
     (coin_flip
      (UNIT NONE)
      (UNIT (SOME 1)))
     (MMAP SOME
      (coin_flip
       (UNIT 2)
       (UNIT 3)))))
   (prob_repeat
    (coin_flip
     (MMAP SOME
      (coin_flip
       (UNIT 4)
       (UNIT 5)))
     (coin_flip
      (UNIT (SOME 6))
      (UNIT NONE))))

[optimal_two_dice_def]
|- optimal_two_dice =
   coin_flip
   (prob_repeat
    (coin_flip
     (coin_flip
      (coin_flip
       (coin_flip
        (coin_flip
         (UNIT (SOME 2))
         (coin_flip
          (UNIT NONE)
          (UNIT (SOME 2))))
        (UNIT (SOME 3)))
       (UNIT (SOME 4)))
      (UNIT (SOME 6)))
     (MMAP SOME
      (coin_flip
       (coin_flip
```

```
       (coin_flip
        (coin_flip
         (UNIT 3)
         (coin_flip
          (coin_flip
           (UNIT 2)
           (UNIT 4))
          (UNIT 3)))
        (UNIT 5))
       (UNIT 5))
      (UNIT 7)))))
  (prob_repeat
   (coin_flip
    (MMAP SOME
     (coin_flip
      (coin_flip
       (coin_flip
        (coin_flip
         (UNIT 4)
         (coin_flip
          (UNIT 6)
          (coin_flip
           (UNIT 6)
           (UNIT 8))))
        (UNIT 7))
       (UNIT 9))
      (UNIT 8)))
    (coin_flip
     (MMAP SOME
      (coin_flip
       (coin_flip
        (coin_flip
         (UNIT 5)
         (UNIT 9))
        (UNIT 9))
       (UNIT 10)))
     (coin_flip
      (MMAP SOME
       (coin_flip
        (coin_flip
         (coin_flip
          (UNIT 7)
          (UNIT 8))
         (UNIT 10))
        (UNIT 11)))
      (coin_flip
       (MMAP SOME
        (coin_flip
         (coin_flip
          (coin_flip
           (UNIT 10)
           (UNIT 12))
          (UNIT 11))
         (UNIT 11)))
       (coin_flip
        (coin_flip
```

```
          (UNIT (SOME 12))
          (UNIT NONE))
         (UNIT (SOME 12))))))))

[two_dice_def]
|- two_dice = BIND dice (\a. BIND dice (\b. UNIT (a + b)))

[INDEP_FN_DICE]
|- dice IN indep_fn

[INDEP_FN_OPTIMAL_TWO_DICE]
|- optimal_two_dice IN indep_fn

[INDEP_FN_TWO_DICE]
|- two_dice IN indep_fn

[OPTIMAL_TWO_DICE_CORRECT]
|- !n.
     prob bern {s | FST (optimal_two_dice s) = n} =
     prob bern {s | FST (two_dice s) = n}

[PROB_BERN_DICE]
|- !n.
     prob bern {s | FST (dice s) = n} =
     (if 1 <= n /\ n <= 6 then 1 / 6 else 0)

[PROB_BERN_TWO_DICE]
|- !n.
     prob bern {s | FST (two_dice s) = n} =
     (if (n = 2) \/ (n = 12) then 1 / 36
      else if (n = 3) \/ (n = 11) then 2 / 36
      else if (n = 4) \/ (n = 10) then 3 / 36
      else if (n = 5) \/ (n = 9) then 4 / 36
      else if (n = 6) \/ (n = 8) then 5 / 36
      else if n = 7 then 6 / 36
      else 0)
```

Appendix D

Miller-Rabin Primality Test Extracted to Standard ML

D.1 HOL `miller_rabin_ml` Theory

```
[BIND_ML]
|- !f g. BIND f g = UNCURRY g o f

[EVEN_ML]
|- !n. EVEN n = (n MOD 2 = 0)

[FACTOR_TWOS_ML]
|- factor_twos n =
   (if 0 < n /\ EVEN n then
      (let (r,s) = factor_twos (n DIV 2) in (SUC r,s))
    else
      (0,n))

[LOG2_ML]
|- !n. log2 n = (if n = 0 then 0 else SUC (log2 (n DIV 2)))

[MANY_ML]
|- !f n.
     MANY f n =
     (if n = 0 then
        UNIT T
      else
        BIND f (\x. (if x then MANY f (n - 1) else UNIT F)))

[MILLER_RABIN_1_ML]
|- !n.
     miller_rabin_1 n =
     (if n = 2 then
        UNIT T
      else
        (if (n = 1) \/ EVEN n then
           UNIT F
         else
           BIND (prob_uniform_cut (2 * log2 (n - 1)) (n - 2))
```

```
          (\a. ~witness n (a + 2))))

[MILLER_RABIN_ML]
|- !n t. miller_rabin n t = MANY (miller_rabin_1 n) t

[MODEXP_ML]
|- modexp n a b =
   (if b = 0 then
      1
    else
      (let r = modexp n a (b DIV 2) in
         let r2 = (r * r) MOD n in
           (if EVEN b then r2 else (r2 * a) MOD n)))

[ODD_ML]
|- ODD = $~ o EVEN

[PROB_UNIFORM_CUT_ML]
|- !t n.
     prob_uniform_cut t n =
     (if n = 0 then
        prob_uniform_cut t n
      else
        (if t = 0 then
           UNIT 0
         else
           BIND (prob_unif (n - 1))
             (\m. if m < n then UNIT m else prob_uniform_cut (t - 1) n)))

[PROB_UNIF_ML]
|- !n.
     prob_unif n =
     (if n = 0 then
        UNIT 0
      else
        BIND (prob_unif (n DIV 2))
        (\m. BIND sdest (\b. if b then 2 * m + 1 else 2 * m)))

[UNCURRY_ML]
|- !f x y. UNCURRY f (x,y) = f x y

[UNIT_ML]
|- !x. UNIT x = (\s. (x,s))

[WITNESS_ML]
|- !n a.
     witness n a =
     (let (r,s) = factor_twos (n - 1) in
        witness_tail n (modexp n a s) r)

[WITNESS_TAIL_ML]
|- !n a r.
     witness_tail n a r =
     (if r = 0 then
        ~(a = 1)
      else
```

```
        (let a2 = (a * a) MOD n in
           (if a2 = 1 then
              ~(a = 1) /\ ~(a = n - 1)
            else
              witness_tail n a2 (r - 1))))
```

D.2 ML Support Modules

D.2.1 `HolStream` Signature

```
signature HolStream =
sig
  type 'a stream = 'a Stream.stream

  val shd   : 'a stream -> 'a
  val stl   : 'a stream -> 'a stream
  val sdest : 'a stream -> 'a * 'a stream
end
```

This module provides a HOL wrapper to a more comprehensive `Stream` module.

D.2.2 `RandomBits` Signature

```
signature RandomBits =
sig
  val random : unit -> bool Stream.stream
  val urandom : unit -> bool Stream.stream
end
```

In this module, the values `random` and `urandom` provide an interface to the Linux devices `/dev/random` and `/dev/urandom`.

D.2.3 `HolNum` Signature

```
signature HolNum =
sig
  type num

  val num_to_bits : num -> bool list
  val bits_to_num : bool list -> num

  val Num           : string -> num
  val num_to_string : num -> string
  val pp_num        : ppstream -> num -> unit

  val ZERO : num
  val ONE  : num
```

```
  val TWO  : num

  val SUC : num -> num
  val ++  : num * num -> num
  val --  : num * num -> num
  val **  : num * num -> num
  val DIV : num * num -> num
  val MOD : num * num -> num

  val ==  : num * num -> bool
  val <<= : num * num -> bool
  val >>= : num * num -> bool
  val <<  : num * num -> bool
  val >>  : num * num -> bool
end
```

This module provides support for a type `num` of arbitrarily large natural numbers.

D.3 ML Miller-Rabin

D.3.1 `HolMiller.sml` Structure

```
structure HolMiller :> HolMiller =
struct

open HolNum HolStream;

infix 7 ** MOD DIV;
infix 6 ++ --;
infix 4 == <<= << >>= >>;

(* pair *)

fun UNCURRY f (x, y) = f x y;

(* state_transformer *)

fun UNIT x = fn s => (x, s);

fun BIND f g = UNCURRY g o f;

(* num *)

fun EVEN n = (n MOD TWO == ZERO);

val ODD = not o EVEN;

(* prob *)

fun MANY f n =
  if n == ZERO then UNIT true
  else BIND f (fn x => if x then MANY f (n -- ONE) else UNIT false);
```

```
(* prob_uniform *)

fun log2 n = if n == ZERO then ZERO else SUC (log2 (n DIV TWO));

fun prob_unif n =
  if n == ZERO then UNIT ZERO
  else
    BIND (prob_unif (n DIV TWO)) (fn m =>
    BIND S.dest (fn b =>
    if b then TWO ** m ++ ONE else TWO ** m));

fun prob_uniform_cut t n =
  if n == ZERO then raise Fail "prob_uniform_cut: out of domain"
  else if t == ZERO then UNIT ZERO
  else
    BIND (prob_unif (n -- ONE))
    (fn m => if m << n then UNIT m else prob_uniform_cut (t -- ONE) n);

(* miller_rabin *)

fun factor_twos n =
  if ZERO << n andalso EVEN n then
    let val (r, s) = factor_twos (n DIV TWO)
    in (SUC r, s)
    end
  else (ZERO, n);

fun modexp n a b =
  if b == ZERO then ONE
  else
    let
      val r = modexp n a (b DIV TWO)
      val r2 = (r ** r) MOD n
    in
      if EVEN b then r2 else (r2 ** a) MOD n
    end;

fun witness_tail n a r =
  if r == ZERO then not (a == ONE)
  else
    let
      val a2 = (a ** a) MOD n
    in
      if a2 == ONE then not (a == ONE) andalso not (a == n -- ONE)
      else witness_tail n a2 (r -- ONE)
    end;

fun witness n a =
  let val (r, s) = factor_twos (n -- ONE)
  in witness_tail n (modexp n a s) r
  end;

fun miller_rabin_1 n =
  if n == TWO then UNIT true,
  else if n == ONE orelse EVEN n then UNIT false
```

```
  else
    BIND (prob_uniform_cut (TWO ** log2 (n -- ONE)) (n -- TWO))
    (fn a => not (witness n (a ++ TWO)));

fun miller_rabin n t = MANY (miller_rabin_1 n) t;

end;
```

Bibliography

Mark Aagaard and John Harrison, editors. *Theorem Proving in Higher Order Logics, 13th International Conference: TPHOLs 2000*, volume 1869 of *Lecture Notes in Computer Science*, Portland, OR, USA, August 2000. Springer.

Martin Abadi and Joseph Y. Halpern. Decidability and expressiveness for first-order logics of probability. *Information and Computation*, 1994.

Anonymous. The QED manifesto. In Bundy (1994).

Alan Baker. *A Concise Introduction to the Theory of Numbers*. Cambridge University Press, 1984.

Paul H. Bardell, W. H. McAnney, and J. Savir. *Built In Test for VLSI: Pseudorandom Techniques*. John Wiley, October 1987. URL `http://www.wiley.com/Corporate/Website/Objects/Products/0,9049,65654,00%.html`.

Bruno Barras. Programming and computing in HOL. In Aagaard and Harrison (2000), pages 17–37.

E. R. Berlekamp. Factoring polynomials over large finite fields. *Math. Comput.*, 24, 1970.

Józef Białas. The σ-additive measure theory. *Journal of Formalized Mathematics*, 1990. URL `http://mizar.uwb.edu.pl/JFM/Vol2/measure1.html`.

Józef Białas. Properties of Caratheodor's measure. *Journal of Formalized Mathematics*, 1992. URL `http://mizar.uwb.edu.pl/JFM/Vol4/measure4.html`.

P. Billingsley. *Probability and Measure*. John Wiley & Sons, Inc., New York, 2nd edition, 1986.

George Boole. *The Mathematical Analysis of Logic, Being an Essay Toward a Calculus of Deductive Reasoning*. Macmillan, Cambridge, 1847.

Robert S. Boyer, N. G. de Bruijn, Gérard Huet, and Andrzej Trybulec. A mechanically proof-checked encyclopedia of mathematics: Should we build one? can we? In Bundy (1994), pages 237–251.

Robert S. Boyer and J Strother Moore. Proof checking the RSA public key encryption algorithm. *American Mathematical Monthly*, 91(3):181–189, 1984. URL `http://www.cs.utexas.edu/users/boyer/rsa.ps.Z`.

G. Buffon. Essai d'arithmétique morale. *Supplément à l'Histoire Naturelle*, 4, 1777.

Alan Bundy, editor. *12th International Conference on Automated Deduction (CADE-12)*, volume 814 of *Lecture Notes in Artificial Intelligence*, Nancy, France, June 1994. Springer.

Michael Burrows, Martín Abadi, and Roger Needham. A logic of authentication. *ACM Transactions on Computer Systems*, 8(1):18–36, February 1990. URL `http://www.acm.org/pubs/articles/journals/tocs/1990-8-1/p18-burrows/p18%-burrows.pdf`.

O. Caprotti and M. Oostdijk. Formal and efficient primality proofs by use of computer algebra oracles. *Journal of Symbolic Computation*, 32(1–2):55–70, 2001. Special Issue on Computer Algebra and Mechanized Reasoning.

Alonzo Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940.

Bob Colwell and Bob Brennan. Intel's formal verification experience on the Willamette development. In Aagaard and Harrison (2000), pages 106–107.

Adriana B. Compagnoni. *Higher-Order Subtyping with Intersection Types*. PhD thesis, Catholic University, Nigmegen, January 1995.

Thomas H. Cormen, Charles Eric Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. MIT Press/McGraw-Hill, Cambridge, Massachusetts, 1990.

Paul Curzon. The formal verification of the Fairisle ATM switching element: an overview. Technical Report 328, University of Cambridge Computer Laboratory, March 1994.

N. de Bruijn. The mathematical language AUTOMATH, its usage, and some of its extensions. In *Symposium on Automatic Demonstration*, pages 29–61. Lecture Notes in Mathematics, 125, Springer, 1970.

K. de Leeuw, E. F. Moore, C. E. Shannon, and N. Shapiro. Computability by probabilistic machines. In C. E. Shannon and J. McCarthy, editors, *Automata Studies*, pages 183–212. Princeton University Press, Princeton, NJ, 1955.

Morris DeGroot. *Probability and Statistics*. Addison-Wesley, 2nd edition, 1989.

E. W. Dijkstra. *A Discipline of Programming*. Prentice Hall, 1976.

Euclid. Elements, 300B.C. URL `http://www-groups.dcs.st-andrews.ac.uk/~history/Mathematicians/Euclid.h%tml`.

V. A. Feldman and D. Harel. A probabilistic dynamic logic. *Journal of Computer and System Sciences*, 28(2):193–215, 1984.

George S. Fishman. *Monte Carlo: Concepts, Algorithms and Applications*. Springer, 1996.

J. D. Fleuriot. *A Combination of Geometry Theorem Proving and Nonstandard Analysis, with Application to Newton's Principia*. Distinguished Dissertations. Springer, 2001.

J. T. Gill. Computational complexity of probabilistic Turing machines. *SIAM Journal on Computing*, 6(4):675–695, December 1977.

K. Gödel. Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme. *Monatshefte für Mathematik und Physik*, 38:173–198, 1931.

Charles M. Goldie and Richard G. E. Pinch. *Communication theory*, volume 20 of *LMS Student Texts*. Cambridge University Press, 1991.

M. Gordon, R. Milner, and C. Wadsworth. *Edinburgh LCF*, volume 78 of *Lecture Notes in Computer Science*. Springer, 1979.

M. J. C. Gordon. HOL: A proof generating system for higher-order logic. In G. M. Birtwistle and P. A. Subrahmanyam, editors, *VLSI Specification, Verification and Synthesis*, pages 73–128. Kluwer Academic Publishers, Boston, 1988.

M. J. C. Gordon. Notes on PVS from a HOL perspective. URL `http://www.cl.cam.ac.uk/~mjcg/PVS.html`. Available from the author's web page, 1996.

M. J. C. Gordon and T. F. Melham. *Introduction to HOL (A theorem-proving environment for higher order logic)*. Cambridge University Press, 1993.

Elsa Gunter. Doing algebra in simple type theory. Technical Report MS-CIS-89-38, Logic & Computation 09, Department of Computer and Information Science, University of Pennsylvania, 1989.

Joseph Y. Halpern. An analysis of first-order logics of probability. *Artificial Intelligence*, 1990. URL `http://www.cs.cornell.edu/home/halpern/abstract.html#journal25`.

R. W. Hansell. Borel measurable mappings for nonseparable metric spaces. *Transactions of the American Mathematical Society*, 161:145–169, November 1971. URL `http://uk.jstor.org/cgi-bin/jstor/listjournal/00029947/di970179`.

G. H. Hardy. *A Mathematician's Apology, reprinted with a foreword by C. P. Snow*. Cambridge University Press, 1993.

John Harrison. Formalized mathematics. Technical Report 36, Turku Centre for Computer Science (TUCS), Lemminkäisenkatu 14 A, FIN-20520 Turku, Finland, 1996a. URL `http://www.cl.cam.ac.uk/users/jrh/papers/form-math3.html`.

John Harrison. Optimizing proof search in model elimination. In Michael A. McRobbie and John K. Slaney, editors, *13th International Conference on Automated Deduction (CADE-13)*, volume 1104 of *Lecture Notes in Artificial Intelligence*, pages 313–327, New Brunswick, NJ, USA, July 1996b. Springer. URL `http://www.cl.cam.ac.uk/users/jrh/papers/me.html`.

John Harrison. Floating point verification in HOL light: the exponential function. Technical Report 428, University of Cambridge Computer Laboratory, 1997. URL `http://www.cl.cam.ac.uk/users/jrh/papers/tang.html`.

John Harrison. *Theorem Proving with the Real Numbers (Distinguished dissertations).* Springer, 1998.

Sergiu Hart, Micha Sharir, and Amir Pnueli. Termination of probabilistic concurrent programs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 5 (3):356–380, July 1983.

Jifeng He, K. Seidel, and A. McIver. Probabilistic models for the guarded command language. *Science of Computer Programming*, 28(2–3):171–192, April 1997.

Ted Herman. Probabilistic self-stabilization. *Information Processing Letters*, 35(2):63–67, June 1990.

Marieke Huisman. *Reasoning about Java Programs in higher order logic with PVS and Isabelle.* PhD thesis, University of Nijmegen, Holland, February 2001.

Joe Hurd. Lightweight probability theory for verification. In Mark Aagaard, John Harrison, and Tom Schubert, editors, *TPHOLs 2000: Supplemental Proceedings*, number CSE-00-009 in Oregon Graduate Institute Technical Reports, pages 103–113, August 2000. URL `http://www.cl.cam.ac.uk/~jeh1004/research/papers/probability.html`.

Joe Hurd. Predicate subtyping with predicate sets. In Richard J. Boulton and Paul B. Jackson, editors, *14th International Conference on Theorem Proving in Higher Order Logics: TPHOLs 2001*, volume 2152 of *Lecture Notes in Computer Science*, pages 265–280, Edinburgh, Scotland, September 2001a. Springer. URL `http://www.cl.cam.ac.uk/~jeh1004/research/papers/subtypes.html`.

Joe Hurd. Verification of the Miller-Rabin probabilistic primality test. In Richard J. Boulton and Paul B. Jackson, editors, *TPHOLs 2001: Supplemental Proceedings*, number EDI-INF-RR-0046 in University of Edinburgh Informatics Report Series, pages 223–238, September 2001b. URL `http://www.cl.cam.ac.uk/~jeh1004/research/papers/miller.html`.

IEEE Standards Department. *IEEE Standard for Verilog Hardware Description Language.* Number 1364-2001 in IEEE Standards. IEEE, 2001.

D. S. Johnson. A catalog of complexity classes. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume A: Algorithms and Complexity*, chapter 9, pages 67–161. Elsevier and The MIT Press (co-publishers), 1990.

Claire Jones. *Probabilistic Non-Determinism.* PhD thesis, University of Edinburgh, 1990. URL `http://www.lfcs.informatics.ed.ac.uk/reports/90/ECS-LFCS-90-105/`.

Michael D. Jones. Restricted types for HOL. In Elsa L. Gunter, editor, *Supplemental Proceedings of the 10th International Conference on Theorem Proving in Higher Order Logics, TPHOLs '97*, Murray Hill, NJ, USA, August 1997. URL `http://www.cs.utah.edu/~mjones/my.papers.html`.

F. Kammüller and L. C. Paulson. A formal proof of Sylow's first theorem – an experiment in abstract algebra with Isabelle HOL. *Journal of Automated Reasoning*, 23(3-4):235–264, 1999.

R. M. Karp. The probabilistic analysis of some combinatorial search algorithms. In Traub (1976), pages 1–20.

Matt Kaufmann, Panagiotis Manolios, and J Strother Moore, editors. *Computer-Aided Reasoning: ACL2 Case Studies*. Kluwer Academic Publishers, June 2000a.

Matt Kaufmann, Panagiotis Manolios, and J Strother Moore. *Computer-Aided Reasoning: An Approach*. Kluwer Academic Publishers, June 2000b.

Matt Kaufmann and J Strother Moore. An industrial strength theorem prover for a logic based on Common Lisp. *IEEE Transactions on Software Engineering*, 23(4): 203–213, April 1997. URL `http://www.cs.utexas.edu/users/moore/publications/acl2-papers.html#Over%views`.

H. J. Keisler. Probability quantifiers. In J. Barwise and S. Feferman, editors, *Model-Theoretic Logics*, pages 509–556. Springer, New York, 1985.

Donald E. Knuth. *The Art of Computer Programming: Seminumerical Algorithms*. Addison-Wesley, 1997. Third edition.

Donald E. Knuth and Andrew C. Yao. The complexity of nonuniform random number generation. In Traub (1976).

Andrei N. Kolmogorov. *Foundations of the Theory of Probability*. Chelsea, New York, 1950.

Dexter Kozen. Semantics of probabilistic programs. In *20th Annual Symposium on Foundations of Computer Science*, pages 101–114, Long Beach, Ca., USA, October 1979. IEEE Computer Society Press.

M. Kwiatkowska, G. Norman, and D. Parker. Prism: Probabilistic symbolic model checker. In *Proceedings of PAPM/PROBMIV 2001 Tools Session*, September 2001. URL `http://www.cs.bham.ac.uk/~dxp/prism/papers/PROBMIV01Tool.ps.gz`.

Leslie Lamport and Lawrence C. Paulson. Should your specification language be typed? *ACM Transactions on Programming Languages and Systems*, 21(3):502–526, May 1999. URL `http://www.research.compaq.com/SRC/personal/lamport/pubs/pubs.html#lamp%ort-types`.

John Launchbury and Simon L. Peyton Jones. Lazy functional state threads. In *SIGPLAN Symposium on Programming Language Design and Implementation (PLDI'94), Orlando*, pages 24–35, June 1994.

Clarence Irving Lewis. *A Survey of Symbolic Logic*. Univ. of California Press, Berkeley, Berkeley, 1918. Reprint of Chapters I–IV by Dover Publications, 1960, New York.

Harry G. Mairson. Deciding ML typability is complete for deterministic exponential time. In *Conference Record of the Seventeenth Annual ACM Symposium on Principles of Programming Languages*, pages 382–401. ACM SIGACT and SIGPLAN, ACM Press, 1990.

Gary L. Miller. Riemann's hypothesis and tests for primality. In *Conference Record of Seventh Annual ACM Symposium on Theory of Computation*, pages 234–239, Albuquerque, New Mexico, May 1975.

R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, December 1978.

Abdel Mokkedem and Tim Leonard. Formal verification of the Alpha 21364 network protocol. In Aagaard and Harrison (2000), pages 443–461.

David Monniaux. An abstract Monte-Carlo method for the analysis of probabilistic programs (extended abstract). In *28th Symposium on Principles of Programming Languages (POPL '01)*. Association for Computer Machinery, 2001. URL `http://www.di.ens.fr/~monniaux/biblio/David_Monniaux.html`.

Carroll Morgan. Proof rules for probabilistic loops. In *Proceedings of the BCS-FACS 7th Refinement Workshop*, 1996. URL `http://web.comlab.ox.ac.uk/oucl/publications/tr/tr-25-95.html`.

Carroll Morgan, Annabelle McIver, Karen Seidel, and J. W. Sanders. Probabilistic predicate transformers. Technical Report TR-4-95, Oxford University Computing Laboratory Programming Research Group, February 1995. URL `http://web.comlab.ox.ac.uk/oucl/publications/tr/tr-4-95.html`.

Rajeev Motwani and Prabhakar Raghavan. *Randomized Algorithms*. Cambridge University Press, Cambridge, England, June 1995.

Andrzej Nędzusiak. σ-fields and probability. *Journal of Formalized Mathematics*, 1989. URL `http://mizar.uwb.edu.pl/JFM/Vol1/prob_1.html`.

Andrzej Nędzusiak. Probability. *Journal of Formalized Mathematics*, 1990. URL `http://mizar.uwb.edu.pl/JFM/Vol2/prob_2.html`.

Nils J. Nilsson. Probabilistic logic. *Artificial Intelligence*, 28(1):71–87, 1986.

Michael Norrish. *C formalised in HOL*. PhD thesis, University of Cambridge Computer Laboratory, 1998.

S. Owre, N. Shankar, J. M. Rushby, and D. W. J. Stringer-Calvert. *PVS System Guide*. Computer Science Laboratory, SRI International, Menlo Park, CA, September 1999.

Sam Owre and Natarajan Shankar. The formal semantics of PVS. Technical Report SRI-CSL-97-2, Computer Science Laboratory, SRI International, Menlo Park, CA, August 1997. URL `http://pvs.csl.sri.com/manuals.html`.

Lawrence C. Paulson. Inductive analysis of the Internet protocol TLS. *TISSEC*, 2(3): 332–351, August 1999.

M. O. Rabin. Probabilistic automata. *Information and Control*, 6:230–245, 1963.

M. O. Rabin. Probabilistic algorithms. In Traub (1976), pages 21–39.

M. K. Reiter and A. D. Rubin. Anonymous web transactions with crowds. *Communications of the ACM*, 42(2):32–38, February 1999.

J. A. Robinson. A note on mechanizing higher order logic. *Machine Intelligence*, 5: 121–135, 1970.

Bertrand Russell. *The Autobiography of Bertrand Russell*. George Allen & Unwin, London, 1968. 3 volumes.

David M. Russinoff. An experiment with the Boyer-Moore theorem prover: A proof of Wilson's theorem. *Journal of Automated Reasoning*, 1:121–139, 1985.

Mark Saaltink. Domain checking Z specifications. In C. Michael Holloway and Kelly J. Hayhurst, editors, *LFM' 97: Fourth NASA Langley Formal Methods Workshop*, pages 185–192, Hampton, VA, September 1997. URL `http://atb-www.larc.nasa.gov/Lfm97/proceedings`.

J. T. Schwartz. Fast probabilistic algorithms for verification of polynomial identities. *Journal of the ACM*, 27(4):701–717, October 1980. URL `http://www.acm.org/pubs/articles/journals/jacm/1980-27-4/p701-schwartz/%p701-schwartz.pdf`.

Natarajan Shankar and Sam Owre. Principles and pragmatics of subtyping in PVS. In D. Bert, C. Choppy, and P. D. Mosses, editors, *Recent Trends in Algebraic Development Techniques, WADT '99*, volume 1827 of *Lecture Notes in Computer Science*, pages 37–52, Toulouse, France, September 1999. Springer. URL `http://www.csl.sri.com/reports/html/wadt99.html`.

Konrad Slind. *Reasoning about Terminating Functional Programs*. PhD thesis, TU Munich, 1999.

Konrad Slind and Michael Norrish. *The HOL System Tutorial*, February 2001. Part of the documentation included with the hol98 theorem-prover.

R. Solovay and V. Strassen. A fast Monte-Carlo test for primality. *SIAM Journal on Computing*, 6(1):84–85, March 1977.

David Stirzaker. *Elementary Probability*. Cambridge University Press, 1994.

Laurent Théry. A quick overview of HOL and PVS, August 1999. URL `http://www-sop.inria.fr/types-project/lnotes/types99-lnotes.html`. Lecture Notes from the Types Summer School '99: Theory and Practice of Formal Proofs, held in Giens, France.

J. F. Traub, editor. *Algorithms and Complexity: New Directions and Recent Results*. Academic Press, New York, 1976.

A. Trybulec and H. A. Blair. Computer aided reasoning. In Rohit Parikh, editor, *Proceedings of the Conference on Logic of Programs*, volume 193 of *Lecture Notes in Computer Science*, pages 406–412, Brooklyn, NY, June 1985. Springer.

Alan M. Turing. Checking a large routine. In *Report of a Conference on High Speed Automatic Calculating Machines*, pages 67–69, Cambridge, England, June 1949. University Mathematical Laboratory.

John von Neumann. Various techniques for use in connection with random digits. In *von Neumann's Collected Works*, volume 5, pages 768–770. Pergamon, 1963.

Tanja E. J. Vos. *UNITY in Diversity: A Stratified Approach to the Verification of Distributed Algorithms*. PhD thesis, Utrecht University, 2000.

Philip Wadler. The essence of functional programming. In *19th Symposium on Principles of Programming Languages*. ACM Press, January 1992.

Stan Wagon. *The Banach-Tarski Paradox*. Cambridge University Press, 1993.

Keith Wansbrough, Michael Norrish, Peter Sewell, and Andrei Serjantov. Timing UDP: mechanized semantics for sockets, threads and failures. URL `http://www.cl.cam.ac.uk/users/pes20/Netsem/`. Draft, 2001.

A. N. Whitehead. *An Introduction to Mathematics*. Williams and Northgate, London, 1911.

Alfred North Whitehead and Bertrand Russell. *Principia Mathematica*. Cambridge University Press, Cambridge, 1910.

David Williams. *Probability with Martingales*. Cambridge University Press, 1991.

Wai Wong. *The HOL* `res_quan` *library*, 1993. URL `http://www.ftp.cl.cam.ac.uk/ftp/hvg/hol98/libraries/library-docs.html`. HOL88 documentation.

Lotfi A. Zadeh. Fuzzy sets. *Information and Control*, 8:338–353, 1965.

Vincent Zammit. *On the Readability of Machine Checkable Formal Proofs*. PhD thesis, University of Kent at Canterbury, March 1999.