A Formal Approach to Probabilistic Termination

Joe Hurd[*]

Computer Laboratory
University of Cambridge
`joe.hurd@cl.cam.ac.uk`

Abstract. We present a probabilistic version of the while loop, in the context of our mechanised framework for verifying probabilistic programs. The while loop preserves useful program properties of measurability and independence, provided a certain condition is met. This condition is naturally interpreted as *"from every starting state, the while loop will terminate with probability 1"*, and we compare it to other probabilistic termination conditions in the literature. For illustration, we verify in HOL two example probabilistic algorithms that necessarily rely on probabilistic termination: an algorithm to sample the Bernoulli(p) distribution using coin-flips; and the symmetric simple random walk.

1 Introduction

Probabilistic algorithms are used in many areas, from discrete mathematics to physics. There are many examples of simple probabilistic algorithms that cannot be matched in performance (or sometimes even complexity) by deterministic alternatives [12]. It is our goal to specify and verify probabilistic algorithms in a theorem-prover. Formal verification is particularly attractive for probabilistic algorithms, because black-box testing is limited to statistical error reports of the form: *"With confidence 90%, the algorithm is broken."* Additionally, even small probabilistic algorithms can be difficult to implement correctly. A whole new class of errors becomes possible and one has to be mathematically sophisticated to avoid them.

In Section 2 we show how probabilistic algorithms can be specified and verified in the HOL[1] theorem prover[2] [2], by thinking of them as deterministic functions having access to an infinite sequence of coin-flips. This approach is general enough to verify many probabilistic algorithms in HOL (including the Miller-Rabin primality test [5]), but (in its raw form) it is limited to algorithms that are guaranteed to terminate.

Example 1. A sampling algorithm simulates a probability distribution ρ by generating on demand a value x with probability $\rho(x)$. In this spirit, an algorithm

[*] Supported by EPSRC project GR/R27105/01.

[1] As will be seen, higher-order logic is essential for our approach, since many of our results rely on quantification over predicates and functions.

[2] hol98 is available from `http://www.cl.cam.ac.uk/Research/HVG/FTP/`.

to sample the Geometric($\frac{1}{2}$) distribution will return the natural number n with probability $(\frac{1}{2})^{n+1}$. A simple way to implement this is to return the index of the first coin-flip in the sequence that is 'heads'. However, this is not guaranteed to terminate on every possible input sequences of coin-flips, the counter-example being the 'all-tails' sequence.[3] However, the algorithm does satisfies probabilistic termination, meaning that the probability that it terminates is 1 (*"in all practical situations it must terminate"*).

In fact, there is a large class of probabilistic algorithms that cannot be defined without using probabilistic termination. In Section 3 we present our approach to overcoming this limitation: a probabilistic version of the 'while' loop that slots into our HOL framework and supports probabilistic termination. If a certain probabilistic termination condition is satisfied, then an algorithm defined in terms of a probabilistic while loop automatically satisfies useful properties of measurability and independence. In Section 4 we examine the relationship between our probabilistic termination condition and others in the literature.

In Sections 5 and 6 we use probabilistic termination to define two algorithms in HOL. The first uses coin-flips to sample from the Bernoulli(p) distribution, where p can be any real number between 0 and 1. The second is the symmetric simple random walk, a classic example from probability theory that requires a subtle termination argument to even define.

The contributions of this paper are as follows:

– an overview of our formal framework for verifying probabilistic programs in the HOL theorem prover;
– the formal definition of a probabilistic while loop, preserving compositional properties of measurability and independence;
– a comparison of our naturally occurring probabilistic termination condition with others in the literature;
– the verification in HOL of two probabilistic algorithms requiring probabilistic termination: an algorithm to sample the Bernoulli(p) distribution using coin-flips; and the symmetric simple random walk.

2 Verifying Probabilistic Algorithms in HOL

In this section we provide an overview of our framework for verifying probabilistic algorithms in HOL. Although novel this is not the main focus of this paper, and so the section is necessarily brief. For the complete explanation please refer to my Ph.D. thesis [6].

2.1 Modelling Probabilistic Programs in HOL

Probabilistic algorithms can be modelled in HOL by thinking of them as deterministic algorithms with access to an infinite sequence of coin-flips. The infinite

[3] For a literary example of a coin that always lands on the same side, see the beginning of the Tom Stoppard play: *Rosencrantz & Guildenstern Are Dead.*

sequence of coin flips is modelled by an element of the type $\mathbb{B}^\infty$ of infinite boolean sequences, and serves as an 'oracle'. The oracle provides an inexhaustible source of values 'heads' and 'tails', encoded by $\top$ and $\bot$. Every time a coin is flipped, the random result of the coin flip is popped and consumed from the oracle. A probabilistic algorithm takes, besides the usual parameters, another oracle parameter from which it may pop the random values it needs. In addition to its result, it returns the rest of the oracle for consumption by someone else.

A simple example of a probabilistic algorithm is a random variable. Consider the random variable V ranging over values of type α, which we represent in HOL by the function

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

Since random variables do not take any parameters, the only parameter of v is the oracle: an element of the type $\mathbb{B}^\infty$ of infinite boolean sequences. It returns the value of the random variable (an element of type α) and the rest of the oracle (another element of $\mathbb{B}^\infty$).

Example 2. If shd and stl are the sequence equivalents of the list operations 'head' and 'tail', then the function

$$\vdash \mathsf{bit} = \lambda s.\ (\mathsf{if\ shd}\ s\ \mathsf{then}\ 1\ \mathsf{else}\ 0,\ \mathsf{stl}\ s) \qquad (1)$$

models a Bernoulli($\frac{1}{2}$) random variable that returns 1 with probability $\frac{1}{2}$, and 0 with probability $\frac{1}{2}$. For example,

$$\mathsf{bit}\ (\top, \bot, \top, \bot, \ldots) = (1, (\bot, \top, \bot, \ldots))$$

shows the result of applying bit to one particular infinite boolean sequence.

It is possible to combine random variables by 'passing around' the sequence of coin-flips.

Example 3. We define a bin n function that combines several applications of bit to calculate the number of 'heads' in the first n flips.

$$\begin{aligned} &\vdash \mathsf{bin}\ 0\ s = (0, s)\ \wedge && (2)\\ &\quad \forall n.\ \mathsf{bin}\ (\mathsf{suc}\ n)\ s = \mathsf{let}\ (x, s') \leftarrow \mathsf{bin}\ n\ s\ \mathsf{in}\ \Big(\mathsf{let}\ (y, s'') \leftarrow \mathsf{bit}\ s'\ \mathsf{in}\ (x + y, s'')\Big) \end{aligned}$$

The HOL function bin n models a Binomial($n, \frac{1}{2}$) random variable.

Concentrating on an infinite sequence of coin-flips as the only source of randomness for our programs is a boon to formalisation in HOL, since only one probability space needs to be formalised in the logic. It also has a practical significance, since we can extract our HOL implementations of probabilistic programs to ML, and execute them on a sequence of high quality random bits from the operating system. These random bits are derived from system noise, and are so designed that a sequence of them should have the same probability distribution as a sequence of coin-flips. An example of this extraction process is given in Section 6 for the random walk, and a more detailed examination of the issues can be found in a previous case study of the Miller-Rabin primality test [5].

2.2 Monadic Operator Notation

The above representation is also used in Haskell[4] and other pure functional languages to write probabilistic programs [13,9]. In fact, these programs live in the more general state-transforming monad: in this case the state that is transformed is the sequence of coin-flips. The following monadic operators can be used to reduce notational clutter when combining state-transforming programs.

Definition 1. *The state-transformer monadic operators* unit *and* bind*.*

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

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

Example 4. Our bin n function can now be defined more concisely:

$$\begin{aligned}&\vdash \mathsf{bin}\ 0 = \mathsf{unit}\ 0\ \wedge \\ &\quad \forall\, n.\ \mathsf{bin}\ (\mathsf{suc}\ n) = \mathsf{bind}\ (\mathsf{bin}\ n)\ (\lambda\, x.\ \mathsf{bind}\ \mathsf{bit}\ (\lambda\, y.\ \mathsf{unit}\ (x + y)))\end{aligned} \tag{3}$$

Observe that the sequence of coin-flips is never referred to directly, instead the unit and bind operators pass it around behind the scenes.

2.3 Formalised Probability Theory

By formalising some mathematical measure theory in HOL, it is possible to define a probability function

$$\mathbb{P} : \mathcal{P}(\mathbb{B}^{\infty}) \to \mathbb{R}$$

from sets of sequences to real numbers between 0 and 1.

Since the Banach-Tarski paradox prevents us from assigning a well-defined probability to *every* set of sequences, it is helpful to think of $\mathbb{P}$ as a partial function. The domain of $\mathbb{P}$ is the set

$$\mathcal{E} : \mathcal{P}(\mathcal{P}(\mathbb{B}^{\infty}))$$

of events of the probability.[5] Our current version of formalised measure theory is powerful enough that any practically occurring set of sequences is an event. Specifically, we define $\mathbb{P}$ and $\mathcal{E}$ using Carathéodory's Extension Theorem, which ensures that $\mathcal{E}$ is a σ-algebra: closed under complements and countable unions.

Once we have formally defined $\mathbb{P}$ and $\mathcal{E}$ in HOL, we can derive the usual laws of probability from their definitions. One such law is the following, which says that the probability of two disjoint events is the sum of their probabilities:

$$\vdash \forall\, A, B.\ A \in \mathcal{E} \wedge B \in \mathcal{E} \wedge A \cap B = \emptyset\ \Rightarrow\ \mathbb{P}(A \cup B) = \mathbb{P}(A) + \mathbb{P}(B)$$

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

[5] Of course, $\mathbb{P}$ must be a total function in HOL, but the values of $\mathbb{P}$ outside $\mathcal{E}$ are never logically significant.

Example 5. Our formalised probability theory allows us to prove results such as

$$\vdash \forall n, r.\ \mathbb{P}\{s \mid \mathsf{fst}\ (\mathsf{bin}\ n\ s) = r\} = \binom{n}{r} \left(\tfrac{1}{2}\right)^n \qquad (4)$$

making explicit the Binomial$(n, \frac{1}{2})$ probability distribution of the $\mathsf{bin}\ n$ function. The fst function selects the first component of a pair, in this case the $\mathbb{N}$ from $\mathbb{N} \times \mathbb{B}^\infty$. The proof proceeds by induction on n, followed by a case split on the first coin-flip in the sequence (a probability weight of $\frac{1}{2}$ is assigned to each case). At this point the goal may be massaged (using real analysis and the laws of probability) to match the inductive hypothesis.

2.4 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 (which probability space is usually clear from context). We can define probabilistic versions of the $\forall$ and $\exists$ quantifiers that make this notation precise.[6]

Definition 2. *Probabilistic Quantifiers*

$$\vdash \forall \phi.\ (\forall^* s.\ \phi(s)) = \{s \mid \phi(s)\} \in \mathcal{E}\ \wedge\ \mathbb{P}\{s \mid \phi(s)\} = 1$$
$$\vdash \forall \phi.\ (\exists^* s.\ \phi(s)) = \{s \mid \phi(s)\} \in \mathcal{E}\ \wedge\ \mathbb{P}\{s \mid \phi(s)\} \neq 0$$

Observe that these quantifiers come specialised to the probability space $(\mathcal{E}, \mathbb{P})$ of infinite sequences of coin-flips: this cuts down on notational clutter.

2.5 Measurability and Independence

Recall that we model probabilistic programs with HOL functions of type

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

However, not all functions f of this HOL type correspond to reasonable probabilistic programs. Some are not measurable, and hence a set of sequences

$$S = \{s \mid P(\mathsf{fst}\ (f(s)))\}$$

that satisfy some property P of the result is not an event of the probability (i.e., $S \notin \mathcal{E}$). Alternatively, f might not be independent, and hence it may use some coin-flips to compute a result and also return those 'used' coin-flips, like this:

$$\mathsf{broken\_bit} = \lambda s.\ (\mathsf{fst}\ (\mathsf{bit}\ s),\ s)$$

We therefore introduce a property indep called strong function independence. If $f \in \mathsf{indep}$, then f will be both measurable and independent. All reasonable probabilistic programs satisfy strong function independence, and the extra properties are a great aid to verification.

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

Definition 3. *Strong Function Independence*

$$
\begin{aligned}
&\vdash \mathsf{indep} = \\
&\quad \{f \mid \\
&\qquad (\mathsf{fst} \circ f) \in \mathsf{measurable}\ \mathcal{E}\ \mathcal{U}\ \wedge\ (\mathsf{snd} \circ f) \in \mathsf{measurable}\ \mathcal{E}\ \mathcal{E}\ \wedge \\
&\qquad \mathsf{countable}\ (\mathsf{range}\ (\mathsf{fst} \circ f))\ \wedge\ \exists C.\ \mathsf{is\_prefix\_cover}\ f\ C\}
\end{aligned}
$$

In Definition 3 we give the HOL definition of indep. Strongly independent functions must be measurable, and satisfy a compositional form of independence that is enforced by their range being countable and having a 'prefix cover' of probability 1.

Strong function independence fits in neatly with the monadic operator notation we introduced earlier, as the following theorem shows.

Theorem 1. *Strong Function Independence is Compositional*

$$
\begin{aligned}
&\vdash \forall a.\ \mathsf{unit}\ a \in \mathsf{indep} \\
&\vdash \forall f, g.\ f \in \mathsf{indep} \wedge (\forall a.\ g(a) \in \mathsf{indep}) \Rightarrow \mathsf{bind}\ f\ g \in \mathsf{indep} \\
&\vdash \forall f, g.\ f \in \mathsf{indep} \wedge g \in \mathsf{indep} \Rightarrow \mathsf{coin\_flip}\ f\ g \in \mathsf{indep}
\end{aligned}
$$

Proof (sketch). The proof of each statement begins by expanding the definition of indep. *The measurability conditions are proved by lifting results from the underlying algebra to $\mathcal{E}$, and the countability condition is easily established. Finally, in each case the required prefix cover is explicitly constructed.*

The coin_flip operator flips a coin to decide whether to execute f or g, and is defined as

$$\vdash \mathsf{coin\_flip}\ f\ g = \lambda s.\ (\mathsf{if}\ \mathsf{shd}\ s\ \mathsf{then}\ f\ \mathsf{else}\ g)\ (\mathsf{stl}\ s) \tag{5}$$

The compositional nature of strong function independence means that it will be satisfied by any probabilistic program that accesses the underlying sequence of coin-flips using only the operators $\{\mathsf{unit}, \mathsf{bind}, \mathsf{coin\_flip}\}$.

3 Probabilistic While Loop

In the previous section we laid out our verification framework for probabilistic programs, emphasising the monadic operator style which ensures that strong function independence holds. The programs have access to a source of randomness in the form of an infinite sequence of coin-flips, and this allows us to easily extract programs and execute them. As we saw with the example program bin n that sampled from the Binomial$(n, \frac{1}{2})$ distribution, it is no problem to define probabilistic programs using well-founded recursion.

However, well-founded recursion is limited to probabilistic programs that compute a finite number of values, each having a probability of the form $m/2^n$.[7]

[7] This follows from the fact that any well-founded function of type $\mathbb{B}^\infty \to \alpha \times \mathbb{B}^\infty$ can only read a finite number of booleans from the input sequence.

Example 6. The limitations of well-founded recursion prevent the definition of the following probabilistic programs:

– an algorithm to sample the Uniform(3) distribution (the probability of each result is 1/3, which cannot be expressed in the form $m/2^n$);
– and an algorithm to sample the Geometric($\frac{1}{2}$) distribution (there are an infinite number of possible results).

In this section we go further, and show how to define probabilistic programs that are not strictly well-founded, but terminate with probability 1. Every probabilistic program in the literature falls into this enlarged definition, and so (in principle, at least) can be modelled in HOL.

3.1 Definition of the Probabilistic While Loop

We aim to define a probabilistic version of the while loop, where the body

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

of the while loop probabilistically advances a state of type α, and the condition

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

is a deterministic state predicate.

We first define a bounded 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 4. *Bounded Probabilistic While Loop*

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

The bounded version of probabilistic while does not employ probabilistic recursion. Rather it uses standard recursion on the cut-off parameter n, and consequently many useful properties follow by induction on n.

We now use while_cut to make a 'raw definition' of an unbounded probabilistic while loop.

Definition 5. *Probabilistic While Loop*

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

where arb *is an arbitrary fixed value, and* minimal ϕ *is specified to be the smallest natural number* n *satisfying* $\phi(n)$.

3.2 Characterisation of the Probabilistic While Loop

There are two characterising theorems that we would like to prove about probabilistic while loops. The first demonstrates that it executes as we might expect: check the condition, if true then perform an iteration and repeat, if false then halt and return the current state. Note the bind and unit in the theorem: this is a probabilistic while loop, not a standard one!

Theorem 2. *Iterating the Probabilistic While Loop*

$$\vdash \forall c, b, a.\ \mathsf{while}\ c\ b\ a = \mathsf{if}\ \ c(a)\ \mathsf{then}\ \mathsf{bind}\ (b(a))\ (\mathsf{while}\ c\ b)\ \mathsf{else}\ \mathsf{unit}\ a$$

Proof. For a given c, b, a, 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 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.

Note that up to this point, the definitions and theorems have not mentioned the underlying state, and so generalise to any state-transforming while loop. The second theorem that we would like to prove is specific to probabilistic while loops, and states that while preserves strong function independence. This allows us to add while to our set of monadic operators for safely constructing probabilistic programs. However, for while $c\ b$ to satisfy strong function independence, the following 'termination' condition is placed on b and c.[8]

Definition 6. *Probabilistic Termination Condition*

$$\vdash \forall c, b.\ \mathsf{while\_terminates}\ c\ b = \forall a.\ \forall^* s.\ \exists n.\ \neg c(\mathsf{fst}\ (\mathsf{while\_cut}\ c\ b\ n\ a\ s))$$

This extra condition says that for every state a, there is an event of probability 1 that leads to the termination of the probabilistic while loop. This additional condition ensures that probabilistic while loops preserve strong function independence.

Theorem 3. *Probabilistic While Loops Preserve Strong Function Independence*

$$\vdash \forall c, b.\ (\forall a.\ b(a) \in \mathsf{indep}) \land \mathsf{while\_terminates}\ c\ b \Rightarrow \forall a.\ \mathsf{while}\ c\ b\ a \in \mathsf{indep}$$

[8] The $\forall^*$ in the definition is a probabilistic universal quantifier (see Section 2.4).

Proof (sketch). Countability of range and measurability follow easily from the strong function independence of $b(a)$, for every a. The prefix cover is again explicitly constructed, by stitching together the prefix covers of $b(a)$ for all reachable states a that terminate the while loop (i.e., that satisfy $\neg c(a)$).

At this point the definition of a probabilistic while loop is finished. We export Theorems 2 and 3 and Definition 6, and these totally characterise the while operator: users never need to work with (or even see) the raw definition.

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

Theorem 4. *Probabilistic While Rule*

$$\begin{aligned}
&\vdash \forall\, \phi, c, b, a.\\
&\qquad (\forall\, a.\ b(a) \in \mathsf{indep})\ \wedge\ \mathsf{while\_terminates}\ c\ b\ \wedge\\
&\qquad \phi(a)\ \wedge\ (\forall\, a.\ \forall^*\, s.\ \phi(a) \wedge c(a) \Rightarrow \phi(\mathsf{fst}\ (b\ a\ s)))\ \Rightarrow\\
&\qquad \forall^* s.\ \phi(\mathsf{fst}\ (\mathsf{while}\ c\ b\ a\ s))
\end{aligned}$$

"For a well-behaved probabilistic while loop, if a property is true of the initial state and with probability 1 is preserved by each iteration, then with probability 1 the property will be true of the final state."

4 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.

Let us begin by observing that our termination condition while_terminates $c\ b$ is both necessary and sufficient for each while $c\ b\ a$ to terminate on a set of probability 1. Therefore, the other termination conditions we survey are either equivalent to ours or logically imply it.

In the context of probabilistic concurrent systems, the following 0-1 law was proved by Hart, Sharir, and Pnueli [3]:[9]

> 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 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:

[9] This paraphrasing comes from Morgan [10].

Theorem 5. *The 0-1 Law of Probabilistic Termination*

$$\begin{aligned}
&\vdash \forall\, c, b.\\
&\quad (\forall\, a.\ b(a) \in \mathsf{indep}) \ \Rightarrow\\
&\quad (\mathsf{while\_terminates}\ c\ b \iff\\
&\quad\ \exists\, p.\ 0 < p\ \wedge\ \forall\, a.\ p \le \mathbb{P}\{s \mid \exists\, n.\ \neg c(\mathsf{fst}\ (\mathsf{while\_cut}\ c\ b\ n\ a\ s))\})
\end{aligned}$$

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. An example of its use for proving probabilistic termination will be seen in our verification of a sampling algorithm for the Bernoulli(p) distribution.

Hart, Sharir, and Pnueli [3] also established a sufficient condition for probabilistic termination called the probabilistic variant rule. We can formalise this as a sufficient condition for termination of our probabilistic while loops; the proof is relatively easy from the 0-1 law.

Theorem 6. *The Probabilistic Variant Condition*

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

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, the probabilistic variant condition has been shown to be necessary as well as sufficient.

5 Example: Sampling the Bernoulli(p) Distribution

The Bernoulli(p) 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 coin-flips can be considered as sampling a Bernoulli($\frac{1}{2}$) distribution, and the present goal is to use these to produce samples from a Bernoulli(p) distribution, where p is any real number between 0 and 1.

The sampling algorithm we use is based on the following simple idea. Suppose the binary expansion of p is $0.p_0p_1p_2\cdots$; consider the coin-flips of the sequence s as forming a binary expansion $0.s_0s_1s_2\cdots$.[10] In this way s can also be regarded

[10] We can conveniently ignore the fact that some numbers have two binary expansions (e.g., $\frac{1}{2} = 0.1000\cdots = 0.0111\cdots$), since the set of these 'dyadic rationals' is countable and therefore has probability 0.

as a real number between 0 and 1. Since the 'number' s is uniformly distributed between 0 and 1, we (informally) have

$$\text{Probability}(s < p = \left\{ \begin{matrix} \top \\ \bot \end{matrix} \right\}) = \left\{ \begin{matrix} p \\ 1-p \end{matrix} \right\}$$

Therefore, an algorithm that evaluates the comparison $s < p$ will be sampling from the Bernoulli(p) distribution, and this comparison can easily be decided by looking at the binary expansions. The matter is further simplified since we can ignore awkward cases (such as $s = p$) that occur with probability 0.

Definition 7. *A Sampling Algorithm for the* Bernoulli(p) *Distribution*

$$\begin{aligned}
&\vdash \forall p. \\
&\qquad \mathsf{bern\_iter}\ p = \\
&\qquad \mathsf{if}\ p < \tfrac{1}{2}\ \mathsf{then\ coin\_flip\ (unit\ (inr}\ \bot))\ (\mathsf{unit\ (inl}\ (2p))) \\
&\qquad \mathsf{else\ coin\_flip\ (unit\ (inl}\ (2p-1)))\ (\mathsf{unit\ (inr}\ \top)) \\
&\vdash \forall p.\ \mathsf{bernoulli}\ p = \mathsf{bind\ (while\ is\_inl\ (bern\_iter} \circ \mathsf{outl})\ (\mathsf{inl}\ p))\ (\mathsf{unit} \circ \mathsf{outr})
\end{aligned}$$

To make the sampling algorithm fit into a probabilistic while loop, the definition makes heavy use of the HOL sum type $\alpha + \beta$, which has 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 $s < p$ by iteration on the bits of s:

- if $\mathsf{shd}\ s = \bot$ and $\frac{1}{2} \le p$, then return $\top$;
- if $\mathsf{shd}\ s = \top$ and $p < \frac{1}{2}$, then return $\bot$;
- if $\mathsf{shd}\ s = \bot$ and $p < \frac{1}{2}$, then repeat with $s := \mathsf{stl}\ s$ and $p := 2p$;
- if $\mathsf{shd}\ s = \top$ and $\frac{1}{2} \le p$, then repeat with $s := \mathsf{stl}\ s$ and $p := 2p - 1$.

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 5) provides a convenient method of showing probabilistic termination:

$$\vdash \mathsf{while\_terminates\ is\_inl\ (bern\_iter} \circ \mathsf{outl}) \tag{6}$$

From this follows strong function independence

$$\vdash \forall p.\ \mathsf{bernoulli}\ p \in \mathsf{indep} \tag{7}$$

and we can then prove that bernoulli satisfies an alternative definition:

$$\begin{aligned}
&\vdash \forall p. \\
&\qquad \mathsf{bernoulli}\ p = \\
&\qquad \mathsf{if}\ p < \tfrac{1}{2}\ \mathsf{then\ coin\_flip\ (unit}\ \bot)\ (\mathsf{bernoulli}\ (2p)) \\
&\qquad \mathsf{else\ coin\_flip\ (bernoulli}\ (2p-1))\ (\mathsf{unit}\ \top)
\end{aligned} \tag{8}$$

This definition of bernoulli is more readable, closer to the intuitive version, and easier to use in proofs. We use this to prove the correctness theorem:

$$\vdash \forall p.\ 0 \le p \wedge p \le 1 \Rightarrow \mathbb{P}\{s \mid \mathsf{bernoulli}\ p\ s\} = p \tag{9}$$

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. As can be shown by induction, this will occur after n iterations.

It is perhaps surprising that the uncountable set $\{\mathsf{bernoulli}\ p \mid 0 \le p \le 1\}$ of programs are all distinct, even though each one examines only a finite number of bits (with probability 1).

6 Example: The Symmetric Simple Random Walk

The (1-dimensional) symmetric simple random walk is 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 $i-1$. The following program simulates the drunk's passage home, and upon arrival returns the total number of steps taken.

Definition 8. *A Simulation of the Symmetric Simple Random Walk*

$$\vdash \forall n.\ \mathsf{lurch}\ n = \mathsf{coin\_flip}\ (\mathsf{unit}\ (n+1))\ (\mathsf{unit}\ (n-1))$$
$$\vdash \forall f, b, a, k.\ \mathsf{cost}\ f\ b\ (a,k) = \mathsf{bind}\ (b(a))\ (\lambda\, a'.\ \mathsf{unit}\ (a', f(k)))$$
$$\vdash \forall n, k.$$
$$\mathsf{walk}\ n\ k =$$
$$\mathsf{bind}\ (\mathsf{while}\ (\lambda\,(n,\_).\ 0 < n)\ (\mathsf{cost\ suc\ lurch})\ (n,k))\ (\lambda\,(\_,k).\ \mathsf{unit}\ k)$$

Theorem 7. *The Random Walk Terminates with Probability 1*

$$\vdash \forall n, k.\ \mathsf{while\_terminates}\ (\lambda\,(n,\_).\ 0 < n)\ (\mathsf{cost\ suc\ lurch})$$

Proof. Let $\pi_{i,j}$ be the probability that starting at point i, the drunk will eventually reach point j.

We first formalise the two lemmas $\pi_{p+i,p} = \pi_{i,0}$ and $\pi_{i,0} = \pi_{1,0}^i$. Therefore, if with probability 1 the drunk eventually gets home from a pub at point 1, with probability 1 he will eventually get 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

$$(\pi_{1,0} - 1)^2 = 0$$

and therefore

$$\pi_{1,0} = 1$$

Once probabilistic termination is established, strong independence easily follows:

$$\vdash \forall n, k.\ \mathsf{walk}\ n\ k \in \mathsf{indep} \tag{10}$$

At this point, we may formulate the definition of walk in a more natural way:

$$\begin{aligned}
&\vdash \forall n, k. && (11)\\
&\quad \mathsf{walk}\ n\ k = \\
&\quad \mathsf{if}\ n = 0\ \mathsf{then}\ \mathsf{unit}\ k\ \mathsf{else} \\
&\qquad \mathsf{coin\_flip}\ (\mathsf{walk}\ (n{+}1)\ (k{+}1))\ (\mathsf{walk}\ (n{-}1)\ (k{+}1))
\end{aligned}$$

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 the following basic property of the random walk:[11]

$$\vdash \forall n, k.\ \forall^* s.\ \mathsf{even}\ (\mathsf{fst}\ (\mathsf{walk}\ n\ k\ s)) = \mathsf{even}\ (n + k) \tag{12}$$

For a pub at point 1001, the drunk must get home eventually, but he will take an odd number of steps to do so!

It is possible to extract this probabilistic program to ML, and repeatedly simulate it using high-quality random bits from the operating system. 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!

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

$$\sum_n n \frac{6}{\pi^2 n^2} = \frac{6}{\pi^2} \sum_n \frac{1}{n} = \infty$$

[11] Note the use of the probabilistic universal quantifier $\forall^* s$. This allows us to ignore the set of sequences that cause the drunk to walk forever, since it has probability 0.

7 Conclusions

In this paper we have described how probabilistic programs can be verified in the HOL theorem prover, and then shown how programs that terminate with probability 1 can be defined in the model. Finally, we applied the technology to verify two example programs that necessarily rely on probabilistic termination. In principle, our HOL framework is powerful enough to verify any probabilistic program that terminates with probability 1. However, the labour-intensive nature of theorem proving means that it is only practical to verify particularly important probabilistic algorithms.

Fixing a sequence of coin-flips as the primitive source of randomness creates a distinction between probabilistic programs that are guaranteed to terminate on every possible sequence of coin-flips, and programs that terminate on a set of sequences having probability 1. Probabilistic programs that are guaranteed to terminate can place an upper bound on the number of random bits they will require for a computation, but programs defined using probabilistic termination may consume an unbounded number of bits. In application areas where random bits are expensive to generate, or where tight bounds are required on execution time, probabilistic termination must be viewed with a certain amount of suspicion.

There is also a logical distinction between guaranteed termination and probabilistic termination. Typically, a program p defined using probabilistic termination generally has properties that are quantified by $\forall^*$ instead of the stronger $\forall$. This is because 'all bets are off' on the set of sequences where p doesn't terminate, and so universal quantification over all sequences usually results in an unprovable property. In our verification of the Miller-Rabin primality test [5], we deliberately avoided using probabilistic termination to get stronger theorems, and the added power meant that we were able to implement a 'composite prover' for natural numbers.

In our random walk example, the proof of probabilistic termination is quite subtle. The random walk is therefore not likely to fit into a standard scheme of programs satisfying probabilistic termination. For this reason it is important that the definition of probabilistic programs in our formal framework is not tied to any particular program scheme. Instead, we can define an arbitrary probabilistic program, then prove it satisfies probabilistic termination, and finally go on to verify it. In the future, it may be useful to define program schemes that automatically satisfy probabilistic termination: these can be implemented by reduction to our current method followed by an automatic termination proof. However, it is important to retain the general method, or unusual programs such as the random walk could not be modelled.

Finally, in both the Bernoulli(p) and random walk examples, we defined a function in terms of probabilistic while loops and then went to great pains to prove that it was equivalent to a simpler version using straightforward recursion. It might reasonably be asked why we don't directly support recursive definitions of probabilistic programs, and the answer is that it's harder to extract the probabilistic termination condition. One possible approach to this, building on the present work, would be to reduce the definition to a probabilistic while loop and then read off the termination condition from that.

8 Related Work

The semantics of probabilistic programs was first tackled by Kozen [8], and developed by Jones [7], He et al. [4] and Morgan et al. [11]. This line of research extends the predicate transformer idea of [1] 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 generalised to functions from states to the real interval $[0, 1]$.

Jones defines a Hoare-style logic for total correctness, in which termination with probability 1 is covered by using upper continuous functions as pre- and post-conditions. In this model there is no distinction between guaranteed termination and probabilistic termination. The verification of a sampling algorithm for the Geometric($\frac{1}{2}$) distribution provides an instructive proof of probabilistic termination, but (from the perspective of mechanisation) the method appears to be more complicated than the approach presented in this paper. Also in the context of probabilistic predicate transformers, Morgan [10] explicitly looks at "proof rules for probabilistic loops", applying the probabilistic variant condition of Hart, Sharir, and Pnueli [3] to verify a probabilistic self-stabilisation algorithm.

Our semantics of probabilistic programs is very different from the predicate transformer framework. Being concerned with mechanisation, our aim was to minimise the amount of necessary formalisation. This led to a simple view of probabilistic programs in terms of a sequence of coin-flips, and this bears no obvious correspondence to the predicate transformer view. Proofs in the two settings plainly use the same high-level arguments, but soon diverge to match the low-level details of the semantics. However, it may be that our 'shallow embedding' of probabilistic programs as HOL functions is obscuring similarities. An interesting direction for future work would be to formalise the syntax of a simple while language including a probabilistic choice operator, and then derive the rules of the predicate transformer semantics in terms of our own.

Acknowledgements

The research in this paper is drawn from my Ph.D. thesis, which was supervised by Mike Gordon. The work has greatly benefitted from his guidance, as well as numerous discussions with Konrad Slind, Michael Norrish, Mark Staples, Judita Preiss, and numerous past and present members of the Cambridge Automated Reasoning Group. Finally, I am grateful to the TPHOLs referees for many suggestions on how to improve the presentation.

References

1. E.W. Dijkstra. *A Discipline of Programming.* Prentice-Hall, 1976.
2. M.J.C. Gordon and T.F. Melham. *Introduction to HOL (A theorem-proving environment for higher order logic).* Cambridge University Press, 1993.
3. 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.
4. 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.
5. 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 2001.
6. Joe Hurd. *Formal Verification of Probabilistic Algorithms.* PhD thesis, University of Cambridge, 2002.
7. Claire Jones. *Probabilistic Non-Determinism.* PhD thesis, University of Edinburgh, 1990.
8. 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.
9. 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.
10. Carroll Morgan. Proof rules for probabilistic loops. In *Proceedings of the BCS-FACS 7th Refinement Workshop*, 1996.
11. 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.
12. Rajeev Motwani and Prabhakar Raghavan. *Randomized Algorithms.* Cambridge University Press, Cambridge, England, June 1995.
13. Philip Wadler. The essence of functional programming. In *19th Symposium on Principles of Programming Languages.* ACM Press, January 1992.