Proving Properties of Randomized Algorithms

paper for Programming Language Semantics Research Seminar

Martin Pettai

November 25, 2009

Abstract

Formal verification can do much in improving the security of algorithm implementations. While it is not an easy task even for deterministic algorithms, randomized algorithms add to it another degree of complexity. This paper investigates some ways of dealing with this complexity.

We describe a method of formally modeling randomized algorithms without having to formalize the whole probability theory. We give an overview of the techniques used to prove partial and total correctness of randomized algorithms modeled using this method. We also make some comparisons to the techniques used in the deterministic case to show how they are generalized in the randomized case.

Contents

1 Introduction

To improve the security of an algorithm implementation, it is often useful to prove that the implementation is correct with respect to some specification. If we want to prove that a deterministic algoritm is implemented correctly, we can define a denotational semantics of this algorithm as a function and prove that it gives a correct output for every input, i.e. for a function $f : A \to B$ we want to prove that $\forall x \in A.\ P\ (x, f\ x)$ where P is a predicate on $A \times B$. If the algorithm is not deterministic but probabilistic then the output is no longer uniquely defined by the input.

In this paper, we will see how to prove correctness in this more general case, where the algorithms can use randomness. This overview is based mostly on the paper by Audebaud and Paulin-Mohring [1]. Differently from that paper, here we will not consider the aspects specific to the Coq proof assistant. Rather we will give an overview of the techniques involved in modeling randomized algorithms and reasoning about them to prove partial or total correctness. These techniques can be used to mechanize the correctness proofs in Coq or in other proof assistants but they can also be used independently of any proof assistant. Thus the reader is not required to be familiar with proof assistants but familiarity with functional programming, monads, and denotational semantics is assumed. Some elementary knowledge of probability theory would also be helpful.

This paper contains two main sections. In section 2, we will start by describing how to remove randomness from randomized algorithms by modeling them as mathematical functions. These functions can be used as denotations of the randomized algorithms. Next we will introduce a monad that captures

the modeled randomness. We will finish the section by describing a simple randomized language and its denotational semantics.

In section 3, we will investigate several examples to illustrate the main techniques of proving correctness of randomized algorithms. We will also give a more theoretical description of the techniques for general-recursive functions in the subsection 3.4.

2 Semantics of randomized algorithms

2.1 Interpreting probabilistic algorithms as functions

We first consider deterministic functions. Let $f : \alpha \to \beta$ be a deterministic function. Then we can specify a predicate $P : \alpha \to \beta \to \mathbb{B}$ that defines partial correctness for this function. Here $\mathbb{B} = \{\texttt{false}, \texttt{true}\}$ is the set of booleans. To prove partial correctness, we then have to prove that whenever $f\ x$ terminates and returns y, also $P\ x\ y$ must hold. We define the proposition $\mathrm{I}_P(f, x)$:

$$\mathrm{I}_P(f, x) \equiv \text{if } f\ x \text{ terminates then the returned value } y \text{ satisfies } P\ x\ y$$

Thus partial correctness of f is equivalent to the proposition $\forall x.\ \mathrm{I}_P(f, x)$. To prove total correctness, we also need to prove that $f\ x$ terminates for all x.

Now let $\phi : \alpha \to \mathrm{distr}\,\beta$ be a probabilistic algorithm. Here $\mathrm{distr}\,\beta$ is the type of probability distributions over β. In this case we can still specify a predicate $P : \alpha \to \beta \to \mathbb{B}$, but here it defines partial correctness of a concrete run of the algorithm. Here $\phi\ x$ can behave differently on different runs, e.g. on some runs it terminates and returns a correct output, on some runs it terminates and returns an incorrect output, and on some runs it does not terminate. Each of these three cases happens with a certain probability. Instead of the proposition $\mathrm{I}_P(f, x)$, we define a random event $\mathrm{E}_P(\phi, x)$ characterizing a single run of $\phi\ x$:

$$\mathrm{E}_P(\phi, x) = \{\text{if } \phi\ x \text{ terminates then the returned value } y \text{ satisfies } P\ x\ y\}$$

Thus the event happens if the run either does not terminate or terminates returning a correct answer.

Let the argument x be fixed. We consider the probability of the event $\mathrm{E}_P(\phi, x)$. This probability is a real number in $[0, 1]$ that is uniquely determined by $\phi\ x : \mathrm{distr}\,\beta$ and $P\ x : \beta \to \mathbb{B}$. Thus, we have a function $\rho : \mathrm{distr}\,\beta \to (\beta \to \mathbb{B}) \to [0, 1]$.

Since we are only interested in the probability that the return value satisfies a certain predicate (and not, e.g. sampling the distribution $\phi\ x$ of the

return value), we can replace $\phi\ x$ by $\rho\ (\phi\ x)$. Instead of the function ϕ we can define the function $f : \alpha \to (\beta \to \mathbb{B}) \to [0,1]$ where $f\ x = \rho\ (\phi\ x)$.

Now we can estimate the three probabilities that we mentioned a few paragraphs ago. The probability that $f\ x$ terminates and returns a correct result is $f\ x\ (P\ x)$. The probability that $f\ x$ terminates at all is $f\ x\ \mathrm{I}$ where $\mathrm{I} : \beta \to \mathbb{B}$ is the maximal predicate ($\mathrm{I}\ y = \texttt{true}$ for all y, this predicate is satisfied whenever the algorithm returns any value, i.e. terminates). The probability that $f\ x$ terminates and returns an incorrect result is $f\ x\ \mathrm{I} - f\ x\ (P\ x)$. The probability that $f\ x$ does not terminate is $1 - f\ x\ \mathrm{I}$.

When estimating the properties of a probabilistic algorithm it might be easier to consider parts of the algorithm separately than to consider the whole algorithm at once. Let $f_1 : \alpha \to (\beta \to \mathbb{B}) \to [0,1]$ and $f_2 : \beta \to (\gamma \to \mathbb{B}) \to [0,1]$ be probabilistic algorithms and $f : \alpha \to (\gamma \to \mathbb{B}) \to [0,1]$ be the probabilistic algorithm that applies f_1 to its argument and when it receives a result y from f_1, it applies f_2 to y, returning its result if there is any.

We would like to estimate the probability that the predicate $P : \alpha \to \gamma \to [0,1]$ holds when f is applied to the argument x. If we have already applied f_1 and it returned y then after applying $f_2\ y$, P will hold with probability $f_2\ y\ (P\ x)$. We can consider this probability as a function of y. Let $h : \beta \to [0,1]$ where $h\ y = f_2\ y\ (P\ x)$. The function h is a generalized predicate on β — instead of assigning a boolean value to each argument, it assigns it a real number in $[0,1]$. This real number can be interpreted as the probability that the predicate P holds if y is fixed (i.e. y does not determine P completely).

Because the return value of $f_1\ x$ does not completely determine whether P holds or not, it was not the best choice to assign to $f_1\ x$ the type $(\beta \to \mathbb{B}) \to [0,1]$. Instead, we assign f_1 a more specific type $\alpha \to (\beta \to [0,1]) \to [0,1]$. For the sake of consistency, we then also reassign the types of f_2 and f:

$$f_2 : \beta \to (\gamma \to [0,1]) \to [0,1]$$
$$f : \alpha \to (\gamma \to [0,1]) \to [0,1]$$

Now the probability that P holds when x is fixed is $f\ x\ (\mathbb{I}_P\ x) = f_1\ x\ h$ where $\mathbb{I}_P$ is the function equivalent to the predicate P, i.e. it returns 1 instead of `true` and 0 instead of `false`. We can also consider h as a function of x where $h : \alpha \to \beta \to [0,1]$, $h\ x\ y = f_2\ y\ (\mathbb{I}_P\ x)$. Then $f\ x\ (\mathbb{I}_P\ x) = f_1\ x\ (h\ x)$.

We now introduce the operator $\cdot$ for two functions a and b:

$$(a \cdot b)\ x = a\ x\ b$$

The function b is the continuation given to the function a. Then

$$f \cdot \mathbb{I}_P\ x = f_1 \cdot (f_2 \cdot \mathbb{I}_P\ x)$$

In general, for any $p : \gamma \to [0,1]$

$$f \cdot p = f_1 \cdot (f_2 \cdot p)$$

2.2 Monadic interpretation

In the previous section we interpreted probability distributions over type α as functions of type $(\alpha \to [0,1]) \to [0,1]$. We define $\mathrm{M}\alpha = (\alpha \to [0,1]) \to [0,1]$. We now turn M into a monad. The monadic operations are defined as

$$\begin{aligned}
\mathrm{return} &: \alpha \to \mathrm{M}\alpha \\
\mathrm{return}\, x &= \mathbf{fun}\ (f : \alpha \to [0,1]) \Rightarrow f\ x \\
(\gg\!=) &: \mathrm{M}\alpha \to (\alpha \to \mathrm{M}\beta) \to \mathrm{M}\beta \\
\mu \gg\!= M &= \mathbf{fun}\ (f : \beta \to [0,1]) \Rightarrow \mu\ (\mathbf{fun}\ (x : \alpha) \Rightarrow M\ x\ f)
\end{aligned}$$

Thus in the example from the previous section we had

$$f = \mathbf{fun}\ x \Rightarrow f_1\ x \gg\!= f_2$$

. We have the following identity connecting the operators $\gg\!=$ and $\cdot$:

$$(\mathbf{fun}\ x \Rightarrow f_1\ x \gg\!= f_2) \cdot p = f_1 \cdot (f_2 \cdot p)$$

2.3 A simple randomized language

We use the simple functional language $\mathcal{R}$ml from [1]. In section 3, we will see how to reason on randomized algorithms. We will not reason directly on the syntax of the algorithm but rather on its denotation. In this section, we will see how to give a denotational semantics to $\mathcal{R}$ml expressions, using the framework introduced in sections 2.1 and 2.2.

2.3.1 Syntax

Expressions:

$$\mathit{Expr} ::= x \mid c \mid \texttt{if}\ b\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \mid f\ e_1\ \ldots\ e_n \mid \texttt{let}\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2$$

Top-level declarations:

$$\mathit{TopLevDecls} ::= \texttt{let}\ f\ x_1\ \ldots\ x_n\ \texttt{=}\ e \mid \texttt{let rec}\ f\ x_1\ \ldots\ x_n\ \texttt{=}\ e$$

2.3.2 Type system

$\mathcal{R}$ml uses a simple monomorphic type system which has some base types (unit type, booleans, natural numbers, the set $[0,1]$) and also has the constructor $\to$ for function types.

The denotation of an expression of a non-functional type α is of type $\mathrm{M}\alpha$ because an $\mathcal{R}$ml expression can use randomness. Similarly the denotation of an expression of a functional type $\alpha_1 \to \ldots \to \alpha_n \to \beta$ (where β is a non-functional type) is of type $\alpha_1 \to \ldots \to \alpha_n \to \mathrm{M}\beta$.

2.3.3 Denotational semantics

For each expression e, we define its denotation $[e]$:

$$
\begin{aligned}
[x] &= \mathrm{return}\, x \\
[c] &= \mathrm{return}\, c \\
[\texttt{if}\ e_0\ \texttt{then}\ e_1\ \texttt{else}\ e_2] &= [e_0] \gg\!= \mathbf{fun}\ (x : \mathbb{B}) \Rightarrow \mathbf{if}\ x\ \mathbf{then}\ [e_1]\ \mathbf{else}\ [e_2] \\
[f\ e_1\ \ldots\ e_n] &= [e_1] \gg\!= \mathbf{fun}\ x_1 \Rightarrow \ldots [e_n] \gg\!= \mathbf{fun}\ x_n \Rightarrow f\ x_1\ \ldots\ x_n \\
[\texttt{let}\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2] &= [e_1] \gg\!= \mathbf{fun}\ x \Rightarrow [e_2]
\end{aligned}
$$

The denotations of top-level declarations are top-level declarations in the meta-language:

$$
\begin{aligned}
[\texttt{let}\ f\ x_1\ \ldots\ x_n\ \texttt{=}\ e] &= (\mathbf{let}\ f = \mathbf{fun}\ x_1 \Rightarrow \ldots \mathbf{fun}\ x_n \Rightarrow [e]) \\
[\texttt{let rec}\ f\ x_1\ \ldots\ x_n\ \texttt{=}\ e] &= (\mathbf{let}\ f = \mathbf{fix}\ (\mathbf{fun}\ f \Rightarrow \mathbf{fun}\ x_1 \Rightarrow \ldots \mathbf{fun}\ x_n \Rightarrow [e]))
\end{aligned}
$$

The denotations of primitive (built-in) randomized functions:

$$
\begin{aligned}
[\texttt{random}] &: \mathbb{N} \to \mathrm{M}\mathbb{N} \\
[\texttt{random}]\ n &= \mathbf{fun}\ (f : \mathbb{N} \to [0,1]) \Rightarrow \sum_{i=0}^{n} \frac{1}{1+n} \times (f\ i) \\
[\texttt{flip}] &: () \to \mathrm{M}\mathbb{B} \\
[\texttt{flip}]\ () &= \mathbf{fun}\ (f : \mathbb{B} \to [0,1]) \Rightarrow \frac{1}{2} \times (f\ \texttt{true}) + \frac{1}{2} \times (f\ \texttt{false})
\end{aligned}
$$

The primitive `random` n is used to choose a random number from the set $\{0, \ldots, n\}$ using the uniform distribution. The primitive `flip` is used to simulate a (fair) coin flip.

3 Reasoning on randomized algorithms

We would like to estimate the probability $(f \cdot p)\ x$. If the definition of f does not use recursion then it is straightforward to calculate the probability in a finite number of steps. Each random operator creates another (finite) summation.

3.1 Algebraic properties of measures

Here we describe the algebraic properties that will be used in the following examples. First we define on functions the addition and scaling operations and a partial order:

$$\begin{aligned}(f_1 + f_2)\ x &= f_1\ x + f_2\ x \\ (f_1 - f_2)\ x &= f_1\ x - f_2\ x \\ (k \times f)\ x &= k \times f\ x \\ f_1 \leq f_2 &\equiv \forall x.\ f_1\ x \leq f_2\ x\end{aligned}$$

These hold for any real-valued functions where f_1 and f_2 are of the same type and k is a real number. For measure functions, in addition the linearity and monotonicity properties hold:

$$\begin{aligned}f\ (g_1 + g_2) &= f\ g_1 + f\ g_2 \\ f\ (k \times g) &= k \times f\ g \\ g_1 \leq g_2 &\Rightarrow f\ g_1 \leq f\ g_2\end{aligned}$$

Here $f : (\alpha \to [0,1]) \to [0,1]$ is a measure function and $g, g_1, g_2 : \alpha \to [0,1]$ for some α.

Our λ-calculus also satisfies extensionality, thus we can use η-conversion:

$$(\textbf{fun}\ x \Rightarrow f\ x) = f$$

3.2 A nonrecursive example

This is a modified example from [1]. Suppose we have randomized algorithms $f_1, f_2 : \alpha \to \mathrm{M}\beta$ that satisfy a property $h : \alpha \to \beta \to [0,1]$ with probabilities less than 1. For example, the algorithms may try to determine whether the argument is a prime number. In this case $h\ x\ y = 1$ iff x is prime and $y = \texttt{true}$ or x is not prime and $y = \texttt{false}$. In all other cases $h\ x\ y = 0$.

We would like to combine the algorithms f_1 and f_2 to get an algorithm that satisfies the property h with a higher probability. Suppose we have a

non-randomized function $\texttt{choice} : \beta \to \beta \to \beta$ that combines two results (e.g. from f_1 and f_2) to get a new result such that

$$\forall x, y_1, y_2.\ (\mathbb{I} - h\ x)\ (\texttt{choice}\ y_1\ y_2) \leq (\mathbb{I} - h\ x)\ y_1 \times (\mathbb{I} - h\ x)\ y_2 \quad (1)$$

In the case where $h\ x = \mathbb{I}_P$ for some predicate P, this corresponds to

$$\forall x, y_1, y_2.\ P\ y_1 \vee P\ y_2 \Rightarrow P\ (\texttt{choice}\ y_1\ y_2)$$

For the primality testing example, $\texttt{choice}\ x\ y = x\ \texttt{and}\ y$.

Now consider the algorithm

$$\texttt{fun}\ x \Rightarrow \texttt{let}\ y_1 = f_1\ x\ \texttt{in let}\ y_2 = f_2\ x\ \texttt{in choice}\ y_1\ y_2$$

The denotation of this algorithm is

$$f\ x\ p = f_1\ x\ (\textbf{fun}\ y_1 \Rightarrow f_2\ x\ (\textbf{fun}\ y_2 \Rightarrow p\ (\texttt{choice}\ y_1\ y_2)))$$

The termination probability

$$\begin{aligned} f\ x\ \mathbb{I} &= f_1\ x\ (\textbf{fun}\ y_1 \Rightarrow f_2\ x\ (\textbf{fun}\ y_2 \Rightarrow \mathbb{I}\ (\texttt{choice}\ y_1\ y_2))) = \\ &= f_1\ x\ (\textbf{fun}\ y_1 \Rightarrow f_2\ x\ \mathbb{I}) = \\ &= f_1\ x\ (\mathbb{I} \times f_2\ x\ \mathbb{I}) = \\ &= f_1\ x\ \mathbb{I} \times f_2\ x\ \mathbb{I} \end{aligned}$$

We use (1) to estimate the probability

$$\begin{aligned} f\ x\ (\mathbb{I} - h\ x) &= f_1\ x\ (\textbf{fun}\ y_1 \Rightarrow f_2\ x\ (\textbf{fun}\ y_2 \Rightarrow (\mathbb{I} - h\ x)\ (\texttt{choice}\ y_1\ y_2))) \leq \\ &\leq f_1\ x\ (\textbf{fun}\ y_1 \Rightarrow f_2\ x\ (\textbf{fun}\ y_2 \Rightarrow (\mathbb{I} - h\ x)\ y_1 \times (\mathbb{I} - h\ x)\ y_2)) = \\ &= f_1\ x\ (\textbf{fun}\ y_1 \Rightarrow (\mathbb{I} - h\ x)\ y_1 \times f_2\ x\ (\textbf{fun}\ y_2 \Rightarrow (\mathbb{I} - h\ x)\ y_2)) = \\ &= f_1\ x\ (\textbf{fun}\ y_1 \Rightarrow (\mathbb{I} - h\ x)\ y_1 \times f_2\ x\ (\mathbb{I} - h\ x)) = \\ &= f_1\ x\ (\textbf{fun}\ y_1 \Rightarrow (\mathbb{I} - h\ x)\ y_1) \times f_2\ x\ (\mathbb{I} - h\ x) = \\ &= f_1\ x\ (\mathbb{I} - h\ x) \times f_2\ x\ (\mathbb{I} - h\ x) \end{aligned}$$

Thus, from the assumption (1), we have proved

$$f\ x\ (\mathbb{I} - h\ x) \leq f_1\ x\ (\mathbb{I} - h\ x) \times f_2\ x\ (\mathbb{I} - h\ x) \quad (2)$$

If we compare (1) and (2), we see that they are similar. They estimate the probability of the event corresponding to $\mathbb{I} - h\ x$ but (2) estimates it before the execution of algorithms f_1 and f_2 while (1) does it after executing the algorithms. Thus we can reduce estimating the former probability to the latter, which can be easier because there is less randomness left after executing the algorithms (or none at all as in the primality testing case).

Here we used only algebraic properties of measures, no probability theory was required. Thus this proof can easily be verified in Coq because the algebraic properties have been formalized.

3.3 A primitive-recursive example

Here we extend the nonrecursive example from section 3.2. Consider the algorithm

`let rec test` x n `= if n>0 then choice (`g x`) (test` x `(`n`-1)) else` c

The denotation of `test` is

$f\ x\ 0\ p = p\ c$
$f\ x\ n\ p = g\ x\ (\mathbf{fun}\ y_1 \Rightarrow f\ x\ (n-1)\ (\mathbf{fun}\ y_2 \Rightarrow p\ (\texttt{choice}\ y_1\ y_2)))$ if $n > 0$

We use the result from section 3.2 (taking $f_1\ x = g\ x$ and $f_2\ x = f\ x\ (n-1)$) and get

$$f\ x\ n\ (\mathbb{I} - h\ x) \leq g\ x\ (\mathbb{I} - h\ x) \times f\ x\ (n-1)\ (\mathbb{I} - h\ x)$$

We prove by induction that

$$f\ x\ n\ (\mathbb{I} - h\ x) \leq (g\ x\ (\mathbb{I} - h\ x))^n$$

The base case:

$$f\ x\ 0\ (\mathbb{I} - h\ x) = 1 - c \leq 1 = (g\ x\ (\mathbb{I} - h\ x))^0$$

The inductive step:

$$\begin{aligned} f\ x\ n\ (\mathbb{I} - h\ x) &\leq g\ x\ (\mathbb{I} - h\ x) \times f\ x\ (n-1)\ (\mathbb{I} - h\ x) \leq \\ &\leq g\ x\ (\mathbb{I} - h\ x) \times (g\ x\ (\mathbb{I} - h\ x))^{n-1} = \\ &= (g\ x\ (\mathbb{I} - h\ x))^n \end{aligned}$$

Because Coq supports primitive recursion, we can verify this proof in Coq.

Thus we can improve the precision of $g\ x$ by calling it many times. If we choose a large enough n, we can make the probability that $f\ x$ terminates but gives an incorrect answer smaller that any positive real number. Here we have not proved anything about the probability of termination but we can use a similar reasoning to prove that

$$f\ x\ n\ \mathbb{I} = (g\ x\ \mathbb{I})^n$$

If $g\ x\ (h\ x) > 0$, i.e. $g\ x\ (\mathbb{I} - h\ x) < g\ x\ \mathbb{I}$ then the probability of incorrect result decreases faster than the termination probability as n increases, thus the probability of correctness after termination increases and approaches 1 when $n \to \infty$.

3.4 General recursion

If the definition of f uses general recursion then we consider each recursive function separately. So we assume that $f : \alpha \to \mathrm{M}\beta$ is defined using the fixpoint operator: $f = \mathbf{fix}\, F$ where $F : (\alpha \to \mathrm{M}\beta) \to \alpha \to \mathrm{M}\beta$ is a monotonic function.

3.4.1 Partial correctness

We first consider deterministic functions. Let P be the predicate that defines partial correctness of f. Then the proposition $Q\ h = \forall x.\ \mathrm{I}_P(h, x)$ characterizes the partial correctness (with respect to the predicate P) of a function h. To prove $Q\ (\mathbf{fix}\, F)$ we may prove that if $Q\ h$ for some function h then also $Q\ (F\ h)$. In constructive logic, we would define a function that transforms a proof of $Q\ h$ to a proof of $Q\ (F\ h)$.

In the randomized case, we use a similar approach. Here we define (total) correctness (with respect to the functions g and φ) of a function h by functions $g : \beta \to [0, 1]$ and $\varphi : \alpha \to [0, 1]$. The function h is correct if $h \cdot g = \varphi$.

To prove the correctness of $\mathbf{fix}\, F$, we need to estimate $\mathbf{fix}\, F \cdot g$ for which we define a monotonic function $F_g : (\alpha \to [0, 1]) \to \alpha \to [0, 1]$ that transforms an estimate $h \cdot g$ to an estimate $F\ h \cdot g$, i.e. it satisfies

$$F_g\ (h \cdot g) = F\ h \cdot g$$

If this equality is satisfied, we say that the function F_g commutes with F for the expectation g. The value of F_g is easy to find from the definition of F if the value of $F\ h \cdot g$ is uniquely determined by the value $h \cdot g$ (even for different pairs of values of h and g). This is always the case when f passes its continuation g unchanged to all of its recursive calls, i.e. when all recursive calls in f are tail-recursive.

If f contains recursive calls that are not tail-recursive then it can be more difficult to find F_g that commutes with F. It may be easier to find a function F_g that satisfies

$$F_g\ (h \cdot g) \geq F\ h \cdot g$$

In this case we say that F_g weakly commutes with F (for the expectation g).

If we have a function F_g that commutes with F then $\mathbf{fix}\, F \cdot g = \mathbf{fix}\, F_g$. Now to prove $\mathbf{fix}\, F \cdot g \leq \varphi$ it suffices to prove that φ is a pre-fixpoint of F_g (i.e. $F_g\ \varphi \leq \varphi$) as this gives $\mathbf{fix}\, F_g \leq \varphi$ ($\mathbf{fix}\, F_g$ is the least pre-fixpoint). This also works when F_g weakly commutes with F.

Proving a lower bound of $\mathbf{fix}\, F \cdot g$ is more difficult because F_g is not the greatest post-fixpoint. Thus we prove instead an upper bound of $\mathbf{fix}\, F \cdot \mathbb{I} -$

$\mathbf{fix}\, F \cdot g = \mathbf{fix}\, F \cdot (\mathbb{I} - g) = \mathbf{fix}\, F_{\mathbb{I}-g}$. If we can find a function ψ such that $\mathbb{I} - \psi$ is a pre-fixpoint of $F_{\mathbb{I}-g}$ then we have proved that

$$\begin{aligned} \mathbf{fix}\, F \cdot \mathbb{I} - \mathbf{fix}\, F \cdot g &\leq \mathbb{I} - \psi \\ \mathbf{fix}\, F \cdot g &\geq \psi - (\mathbb{I} - \mathbf{fix}\, F \cdot \mathbb{I}) \end{aligned} \tag{3}$$

This also works when $F_{\mathbb{I}-g}$ weakly commutes with F (for the expectation $\mathbb{I} - g$).

Thus, if we can find suitable functions ψ and φ, we can prove

$$\psi - (\mathbb{I} - \mathbf{fix}\, F \cdot \mathbb{I}) \leq \mathbf{fix}\, F \cdot g \leq \varphi \tag{4}$$

For correctness, the bounds should be tight:

$$\begin{aligned} &\psi - (\mathbb{I} - \mathbf{fix}\, F \cdot \mathbb{I}) = \mathbf{fix}\, F \cdot g = \varphi \\ &\varphi = \mathbf{fix}\, F \cdot g \\ &\psi = \mathbf{fix}\, F \cdot g + (\mathbb{I} - \mathbf{fix}\, F \cdot \mathbb{I}) \end{aligned}$$

For partial correctness, we only need to prove (4). For total correctness, we need in addition

$$\mathbf{fix}\, F \cdot \mathbb{I} = \varphi + (\mathbb{I} - \psi) \tag{5}$$

If (4) is already proved then it suffices to prove

$$\mathbf{fix}\, F \cdot \mathbb{I} \geq \varphi + (\mathbb{I} - \psi)$$

as the upper bound is implied by (4).

3.4.2 Termination

Analogously to the deterministic case, estimating the probability of termination is often more difficult than proving partial correctness. For the upper bound, we can use the techniques from section 3.4.1 but for the lower bound this does not work. Here we need to use another approach.

We define the monotonic sequence of functions (s_n):

$$\begin{aligned} s_0\ x &= 0 \\ s_{n+1}\ x &= F_{\mathbb{I}}\ s_n\ x \end{aligned}$$

where $F_{\mathbb{I}}$ commutes (not weakly) with F for the expectation $\mathbb{I}$. Then $\mathbf{fix}\, F_{\mathbb{I}}$ is the limit of the sequence (s_n), i.e. $\mathbf{fix}\, F \cdot \mathbb{I} = \mathbf{lub}(s_n)$.

To find the limit, we may, for example, first try to find a closed expression for s_n, prove it by induction, and then take its limit.

3.5 A general-recursive example

We consider the example (appearing in [1,2]) of simulating a biased coin flip (the Bernoulli distribution) using only fair coin flips. We use the following algorithm:

$$\begin{aligned}&\texttt{let rec bernoulli } p = \\ &\qquad \texttt{if flip() then if } p < \tfrac{1}{2} \texttt{ then false else bernoulli } (p \,\&\, p) \\ &\qquad\qquad \texttt{else if } p < \tfrac{1}{2} \texttt{ then bernoulli } (p + p) \texttt{ else true}\end{aligned}$$

Here we use an operator &, which is the dual operator of addition on the interval $[0, 1]$ and is defined as

$$a \,\&\, b = 1 - ((1 - a) + (1 - b))$$

The denotation of this algorithm is $\mathbf{fix}\, F$ where

$$\begin{aligned}F\ f\ p\ g = &\frac{1}{2} \times (\mathbf{if}\ p < \frac{1}{2}\ \mathbf{then}\ g\ \texttt{false}\ \mathbf{else}\ f\ (p \,\&\, p)\ g) + \\ &+ \frac{1}{2} \times (\mathbf{if}\ p < \frac{1}{2}\ \mathbf{then}\ f\ (p + p)\ \mathbf{else}\ \texttt{true}) = \\ = &\mathbf{if}\ p < \frac{1}{2}\ \mathbf{then}\ \frac{1}{2} \times g\ \texttt{false} + \frac{1}{2} \times f\ (p + p)\ g \\ &\qquad \mathbf{else}\ \frac{1}{2} \times g\ \texttt{true} + \frac{1}{2} \times f\ (p \,\&\, p)\ g = \\ = &\mathbf{if}\ p < \frac{1}{2}\ \mathbf{then}\ \frac{1}{2} \times g\ \texttt{false} + \frac{1}{2} \times (f \cdot g)\ (p + p) \\ &\qquad \mathbf{else}\ \frac{1}{2} \times g\ \texttt{true} + \frac{1}{2} \times (f \cdot g)\ (p \,\&\, p)\end{aligned}$$

We define a function F_g that commutes with F for the expectation g (because $F_g\ (f \cdot g)\ p = F\ f\ p\ g$):

$$\begin{aligned}F_g\ h\ p = &\mathbf{if}\ p < \frac{1}{2}\ \mathbf{then}\ \frac{1}{2} \times g\ \texttt{false} + \frac{1}{2} \times h\ (p + p) \\ &\qquad \mathbf{else}\ \frac{1}{2} \times g\ \texttt{true} + \frac{1}{2} \times h\ (p \,\&\, p)\end{aligned}$$

Then $\mathbf{fix}\, F \cdot g = \mathbf{fix}\, F_g$.

3.5.1 Partial correctness

We now take $g = \mathbb{I}_{=\texttt{true}}$. Then

$$\begin{aligned}F_g\ h\ p &= \mathbf{if}\ p < \frac{1}{2}\ \mathbf{then}\ \frac{1}{2} \times h\ (p + p)\ \mathbf{else}\ \frac{1}{2} + \frac{1}{2} \times h\ (p \,\&\, p) \\ F_{\mathbb{I}_{-g}}\ h\ p &= \mathbf{if}\ p < \frac{1}{2}\ \mathbf{then}\ \frac{1}{2} + \frac{1}{2} \times h\ (p + p)\ \mathbf{else}\ \frac{1}{2} \times h\ (p \,\&\, p)\end{aligned}$$

Let $\phi\ p = p$. We would like to prove $\mathbf{fix}\, F \cdot g = \phi$.

We first prove $\mathbf{fix}\, F_g \leq \phi$, for this it suffices to prove that ϕ is a pre-fixpoint of F_g, i.e. $F_g\ \phi \leq \phi$ (here even the equality holds):

$$F_g\ \phi\ p = \mathbf{if}\ p < \frac{1}{2}\ \mathbf{then}\ \frac{1}{2} \times (p + p)\ \mathbf{else}\ \frac{1}{2} + \frac{1}{2} \times (p\ \&\ p) =$$
$$= \mathbf{if}\ p < \frac{1}{2}\ \mathbf{then}\ p\ \mathbf{else}\ p = p = \phi\ p$$

This gives us $\mathbf{fix}\, F \cdot g \leq \phi$.

Now we prove $\mathbf{fix}\, F_{\mathbb{I}-g} \leq \mathbb{I} - \phi$. For this it suffices to prove that $\mathbb{I} - \phi$ is a pre-fixpoint of $F_{\mathbb{I}-g}$ (here it is even a fixpoint):

$$F_{\mathbb{I}-g}\ (\mathbb{I} - \phi)\ p = \mathbf{if}\ p < \frac{1}{2}\ \mathbf{then}\ \frac{1}{2} + \frac{1}{2} \times (1 - (p + p))\ \mathbf{else}\ \frac{1}{2} \times (1 - (p\ \&\ p)) =$$
$$= \mathbf{if}\ p < \frac{1}{2}\ \mathbf{then}\ 1 - p\ \mathbf{else}\ 1 - p = 1 - p = (\mathbb{I} - \phi)\ p$$

Using (3), this gives us $\mathbf{fix}\, F \cdot g \geq \phi - (\mathbb{I} - \mathbf{fix}\, F \cdot \mathbb{I})$.

Thus we have proved $\phi - (\mathbb{I} - \mathbf{fix}\, F \cdot \mathbb{I}) \leq \mathbf{fix}\, F \cdot g \leq \phi$, i.e. the partial correctness of `bernoulli`. To prove the total correctness, it remains to prove that $\mathbf{fix}\, F \cdot \mathbb{I} \geq 1$.

3.5.2 Termination

We now investigate the termination by taking $g = \mathbb{I}$:

$$F_{\mathbb{I}}\ h\ p = \mathbf{if}\ p < \frac{1}{2}\ \mathbf{then}\ \frac{1}{2} + \frac{1}{2} \times h\ (p + p)$$
$$\mathbf{else}\ \frac{1}{2} + \frac{1}{2} \times h\ (p\ \&\ p)$$

We define the sequence of functions (s_n):

$$s_0\ p = 0$$
$$s_{n+1}\ p = F_{\mathbb{I}}\ s_n\ p$$

We prove by induction that

$$s_n\ p = 1 - \frac{1}{2^n}$$

The base case is obvious. The inductive step:

$$s_{n+1}\ p = \mathbf{if}\ p < \frac{1}{2}\ \mathbf{then}\ \frac{1}{2} + \frac{1}{2} \times \left(1 - \frac{1}{2^n}\right)\ \mathbf{else}\ \frac{1}{2} + \frac{1}{2} \times \left(1 - \frac{1}{2^n}\right) =$$
$$= \frac{1}{2} + \frac{1}{2} \times \left(1 - \frac{1}{2^n}\right) = 1 - \frac{1}{2^{n+1}}$$

Now we can calculate the termination probability:

$$\mathbf{fix}\, F \cdot \mathbb{I} = \mathbf{lub}\, s_n = \mathbf{lub} \left(\left(1 - \frac{1}{2^n} \right) \times \mathbb{I} \right) = 1 \times \mathbb{I} = \mathbb{I}$$

Thus we have proved the total correctness of `bernoulli`:

$$\mathbf{fix}\, F \cdot g = \phi$$

4 Related work

This overview is mostly based on the paper by Audebaud and Paulin-Mohring [1]. They use the Coq proof assistant to formalize the techniques of reasoning on randomized algorithms. The necessary Coq contributions are described in [3].

An introduction to formal verification of probabilistic algorithms is given in Hurd's thesis [2]. Unlike the simpler functional interpretation of [1], Hurd's thesis gives a formalization (in the HOL theorem prover) of probability theory (and the more general measure theory) and randomized functions are modeled as transformers of infinite streams of random bits.

For deterministic algorithms, methods for formally proving correctness have been embedded in the ATS programming language [4].

5 Conclusion and further work

Here we saw how randomized algorithms can be formally modeled and reasoned about. This required to formally encode the semantics (denotation) of the algorithm. This encoding is done manually and is not formally verified, thus the original implementation of the algorithm can still be incorrect even if the encoded denotation is verified to be correct but the algorithm had been encoded incorrectly.

To remove this deficiency, it would be better to include the proofs in the same language where the algorithms are implemented, allowing to verify the implementation directly. For deterministic algorithms, this has been done in the ATS programming language [4]. It would be interesting to investigate whether ATS can be extended with random operations in a way that allows proofs to be embedded in the implemented algorithm.

References

[1] Philippe Audebaud and Christine Paulin-Mohring. Proofs of randomized algorithms in Coq. *Science of Computer Programming*, 2008.

[2] Joe Hurd. *Formal verification of probabilistic algorithms.* PhD thesis, University of Cambridge, 2002.

[3] Christine Paulin-Mohring. A library for reasoning on randomized algorithms in Coq, 2007. Description of a Coq contribution, Univ. Paris Sud. URL `http://www.lri.fr/paulin/ALEA/library.pdf`.

[4] Chiyan Chen and Hongwei Xi. Combining programming with theorem proving. In *ICFP '05: Proceedings of the tenth ACM SIGPLAN International Conference on Functional Programming*, pages 66–77. ACM, 2005.