Proofs of randomized algorithms in Coq

Philippe Audebaud[1] and Christine Paulin-Mohring[2]

[1] ENS Lyon and INRIA Sophia-Antipolis
[2] Université Paris Sud, LRI and INRIA Futurs*

Abstract. Randomized algorithms are widely used either for finding efficiently approximated solutions to complex problems, for instance primality testing, or for obtaining good average behavior, for instance in distributed computing. Proving properties of such algorithms requires subtle reasoning both on algorithmic and probabilistic aspects of the programs. Providing tools for the mechanization of reasoning is consequently an important issue. Our paper presents a new method for proving properties of randomized algorithms in a proof assistant based on higher-order logic. It is based on the monadic interpretation of randomized programs as probabilistic distribution [18]. It does not require the definition of an operational semantics for the language nor the development of a complex formalization of measure theory, but only use functionals and algebraic properties of the unit interval. Using this model, we show the validity of general rules for estimating the probability for a randomized algorithm to satisfy certain properties, in particular in the case of general recursive functions.
We apply this theory for formally proving a program implementing a Bernoulli distribution from a coin flip and the termination of a random walk. All the theories and results presented in this paper have been fully formalized and proved in the Coq proof assistant [19].

1 Introduction

Randomized algorithms are widely used either for finding efficiently approximated solutions to complex problems such as the primality test, or in order to obtain good average behavior, for instance in distributed computing. Proving properties of such algorithms requires subtle reasoning both on algorithmic and probabilistic aspects of the programs. Providing tools for the mechanization of reasoning is consequently an important issue.

Models. The first problem is to find an appropriate mathematical representation of a randomized algorithm. Methods for modeling randomized programs go back to the early work of D. Kozen [9, 10] which proposes to interpret randomized imperative programs as measure transformers. This approach has been

* Projet ProVal (http://proval.lri.fr), Pôle Commun de Recherche en Informatique du plateau de Saclay CNRS, Ecole Polytechnique, INRIA, Université Paris-Sud

studied further by A. McIver and C. Morgan [14] which extend the interpretation to non-deterministic as well as probabilistic choices and define a refinement relation. Using an extension of weakest-precondition computation to randomized programs, they propose a method to analyze the probability for the result of the program to satisfy a given properties by simple rules on the structure of the program and algebraic properties.

Studying the semantical foundations of probabilistic languages has been the concern of many works. There are at least two different approaches.

The first one is an operational view using access to an arbitrary number of independent random variables following a given distribution (which can be a coin flip or a uniform distribution as in [15]). This interpretation is a monadic transformation. If Ω denotes the type of infinite sequences of independent random values, then a computation of type A will be interpreted as a function of type $\Omega \to A \times \Omega$: it computes a value of type A and modifies the global state of type Ω after consuming a finite prefix of the sequence of random values. Reasoning on randomized programs using this approach requires to model the base probability distribution on Ω.

The second approach is to use directly the monadic structure of probability distributions in order to interpret directly a randomized program of type A as a distribution over the set of possible values in A. This is also a monadic interpretation but with a different space: a probability distribution can essentially be seen as a function from a set of subsets of A into the interval $[0, 1]$, an alternative [18] is to use the monad corresponding to expectations which is a functional which maps functions of type $A \to \mathbb{R}$ to $\mathbb{R}$.

Proofs. The second problem is to reason about probabilistic programs. There are few works on actually mechanizing the proofs in this area.

J. Hurd, A. McIver and C. Morgan designed a mechanization of the quantitative logic for probabilistic guarded commands using the proof assistant HOL [6].

In the domain of distributed protocols, the group of M. Kwiatkowska in Birmingham has designed a probabilistic model-checker PRISM [11], which uses Markov's chains as the underlying model and a probabilistic temporal logic for queries. Reasoning in this framework requires complex computations.

In the domain of algorithms, J. Hurd [4, 5] showed how to model and prove properties of randomized programs in the HOL proof assistant using a monadic transformation of programs, where he assumes access to an infinite sequence of independent coin flips.

Our work has the same goals as J. Hurd's development, to provide tools for interactive reasoning on probabilistic programs. We choose a different monadic transformation of probabilistic programs, interpreting directly programs as measures. One good thing about this method is that it does not require a complicated development within probability theory: the measure can be treated abstractly as a function with algebraic properties. Also the framework does not rely on a particular choice of a primitive randomized function, both discrete and uniform distributions can be manipulated. We propose an axiomatic semantics in the

spirit of the work of C. Morgan et al. and prove the validity of rules with respect to our semantics.

Outline. The paper is organized as follows. In section 2, we introduce the input language and its semantics: an interpretation of programs as measures using a monadic transformation. We analyze our monadic interpretation from the functional point of view. In section 3 we introduce the basic Coq theories for representing measures. In section 4, we show the derived rules for estimating the probability for a randomized program to satisfy a given property. In section 5, we apply our method to proofs of simple probabilistic properties of programs.

Remark. The possible interpretation of random functional programs as probabilistic distributions using a monadic interpretation is not new, it appears in many theoretical works on semantics, or more concretely for representing random programs in Haskell in [18]. To our knowledge, however, the approach of mechanizing reasoning on random functional expressions is new. In [18], the interpretation does not cover general recursive programs and its inefficiency is criticized, the authors propose instead an alternative method which only cover discrete distributions. The possibility to cover recursion was however studied in [7] and we shall take the same approach in this paper. That the interpretation can lead to inefficient or even unfeasible computations in practice will be illustrated in section 2.6. Our work advocates that operational behavior is not relevant, as our model allows anyway for abstract reasoning on programs, using the general rules presented in section 4 and illustrated on examples in section 5. This is to be related to Hoare rules for axiomatic semantics, which do not rely on computations per se, but to denotational semantics. From this point of view, we compare with Kozen second semantics in [9].

2 Monadic interpretation of randomized algorithms

2.1 Randomized programs as measure transformers

In works by D. Kozen [9, 10], G. Plotkin & C. Jones [8, 7], C. Morgan & A. McIver [12] and others, the basic idea is to interpret randomized programs as measure transformers instead of the usual interpretation of programs as state transformers.

The intuitive idea is that a randomized algorithm is non-deterministic and consequently, for a given input state, it may produce different output states. One is interested in the distribution of these output states. If this distribution is known, given a property P on the state, we can compute the probability for the result of the program to satisfy P. A randomized program uses basic randomized primitives such as a `random` function which, given a natural number n, produces a number between 0 and n with uniform probability $\frac{1}{n+1}$, or a more basic `flip` function which produces `true` (resp. `false`) with probability $\frac{1}{2}$. Another classical

operator is probabilistic choice $\texttt{P}\ {}_p\texttt{+}\ \texttt{Q}$ which behaves like the program P with probability p and as Q with probability $1-p$.

The implicit assumption is that any access to a random operator in the program is independent of the others.

In this work, we start from a functional language. We do not have to consider a global state: programs are functions which are computing values, and we want to estimate the distribution of these values.

2.2 Representation of distributions

In this section, we explain our choice for a mathematical representation of probability distributions. We introduce the notation $[0,1]$ for the set of real numbers between 0 and 1.

The probability point of view. From the mathematical point of view, a probability distribution on a set A is defined by a set of *events* $\mathcal{E}$ which is a set of subsets of A with good closure properties, and a function Pr from $\mathcal{E}$ to $[0,1]$ such that the following properties hold:

$\mathsf{Pr}(A) \quad = 1$
$\mathsf{Pr}(\bigcup_i E_i) = \Sigma_i \mathsf{Pr}(E_i)$ when $(E_i)_i$ is a denumerable set of disjoint sets

The measure point of view A (positive) measure on a set A, is a linear functional μ which given a (measurable) function f from A to $\mathbb{R}^+$, computes a non-negative real number, its integral $\int f d\mu$. In the following, we shall use the notation $\mu(f)$ instead of $\int f d\mu$.

Characteristic functions. If X is a subset of A, $\mathbb{I}_X \in A \to [0,1]$ will denote the characteristic function of X such that $\forall x \in A, \mathbb{I}_X(x) = 0 \Leftrightarrow x \notin X \ \wedge\ \mathbb{I}_X(x) = 1 \Leftrightarrow x \in X$. We write simply $\mathbb{I}$ for the function which is 1 everywhere. If $P(x)$ is a formula with a free variable x, we write $\mathbb{I}_{P(.)}$ for the characteristic function of the set X such that $x \in X \Leftrightarrow P(x)$. For instance, $\mathbb{I}_{.=k}$ is the characteristic function of the singleton $\{k\}$.

Measure and probability. There is a well-known correspondence between measures and probability.

Given a probability Pr on a set A, the functional which, given a function $f : A \to \mathbb{R}^+$, computes its expectation defines a measure.

For instance, if A is a finite set, the set of events can be generated by the singletons $\{x\}$ for $x \in A$. The expectation of a function f is defined by:

$$\mu(f) \ = \ \Sigma_{x \in A} f(x) \times \mathsf{Pr}(\{x\})$$

In the other direction, given a measure μ on a set A such that $\mu(\mathbb{I}) = 1$, one can define an associated probability Pr. The events are subsets X of A, such that $\mathbb{I}_X$ is measurable and $\mathsf{Pr}(X) = \mu(\mathbb{I}_X)$.

Our abstract notion of measure In this development, probability distributions are represented as positive bounded measures.

In order to define a probability distribution, it is sufficient to be able to measure functions which take values in the unit interval $[0, 1]$. We can remark that if $\forall x. f(x) \in [0, 1]$ then $\mu(f) \in [0, 1]$ because a probability distribution is bounded by one. Hence, a measure μ can be interpreted as a function of type $(X \to [0, 1]) \to [0, 1]$ satisfying some extra algebraic properties, to be precised in section 3.2.

2.3 Basic language for randomized programs

In the following, we shall be interested in a simple functional language with the following constructions:

- Primitive constants and functions: c
- Conditional: **if** b **then** e_1 **else** e_2
- Local binding: **let** $x = a$ **in** b
- Abstraction: **fun** $(x : \tau) \Rightarrow e$
- Application: $(e_1\ e_2)$

The term τ in the abstraction denotes a type. We assume given a simple (non-polymorphic) type system on this language, containing (at least) the base types `bool` for boolean values and `int` for integer as well as function types $\tau_1 \to \tau_2$.

In order to deal with probabilistic programs, we add primitive functions to this language, such as the `random` function which given a positive integer n, computes with uniform distribution an integer k such that $0 \leq k \leq n$. We shall also use the `flip` function which computes a boolean which is `true` with probability $\frac{1}{2}$.

In the following, we use the same language for expressions representing randomized computations and terms representing their functional interpretation instead to introduce a monadic meta-language as in [13] or [17]. There will be in general no possible confusion.

For the sake of simplification, this paper assumes that abstraction and application in programs are only done on objects in base (non-functional) types; in a local binding as well, the introduced variable has a base type. In the meta-theory and in the interpretation, however, we shall use the same notations for higher-order functions, in particular when writing fixpoints.

2.4 Interpretation of random expressions

A (random) expression e in a base type τ actually represents a set of values of type τ, as different evaluations of the expression will lead to different values in general.

As pointed out above, for analyzing the distribution of these values, we interpret $e : \tau$ as a measure on τ, i.e. a function of type $(\tau \to [0, 1]) \to [0, 1]$.

We write $[e]$ to represent the measure associated to the expression e. If we know $[e]$, given a property Q on τ, it is possible to compute the probability

for the evaluation of e to satisfy Q, it is just $[e](\mathbb{I}_Q)$, namely the application of the measure associated to the expression p to the characteristic function of the predicate Q, interpreted as a subset of τ.

2.5 Monadic transformation

The computation of the measure $[e]$ is defined by case analysis on the structure of the expression e, following a monadic transformation.

We extend the interpretation to expressions denoting functions and not just expressions in base types. Each random expression representing a computation of type τ is interpreted as a purely functional expression of type $[\tau]$.

For a base type τ, $[\tau]$ is defined as $(\tau \to [0,1]) \to [0,1]$ the type of measures on τ. For a functional type $\tau = \tau_1 \to \tau_2$, we define $[\tau] = \tau_1 \to [\tau_2]$, as our study is restricted to the first-order case where τ_1 is a basic type.

In the monadic approach, it is sufficient to define two operators

$$\mathsf{unit} : \tau \to [\tau] \quad \mathsf{bind} : [\tau] \to (\tau \to [\sigma]) \to [\sigma],$$

and for each non-functional construction of type τ (for instance `random`), its functional interpretation of type $[\tau]$.

Then the interpretation of expressions follows naturally:

Computation $p : \tau$	Functional value $[p] : [\tau]$
let $x = a$ **in** b	(bind $[a]$ **fun** $(x : \sigma) \Rightarrow [b]$)
fun $(x : \sigma) \Rightarrow t$	**fun** $(x : \sigma) \Rightarrow [t]$
$(t\ u)$	(bind $[u]$ $[t]$)
if b **then** e_1 **else** e_2	(bind $[b]$ **fun** $(x : \texttt{bool}) \Rightarrow$ **if** x **then** $[e_1]$ **else** $[e_2]$)

Definition of unit and bind. Given an expression e of base type τ, we want $[e]$ to be a measure, that is a functional object of type $(\tau \to [0,1]) \to [0,1]$. $[e]$ is analogous to the monadic interpretation of continuations. Assume τ and σ are base types, one defines:

$$\begin{aligned}
\mathsf{unit}_\tau \ &: \ \tau \to [\tau] \\
&= \textbf{fun}\ (x : \tau) \Rightarrow\ \textbf{fun}\ (f : \tau \to [0,1]) \Rightarrow\ (f\ x) \\
\mathsf{bind}_\sigma \ &: \ [\tau] \to (\tau \to [\sigma]) \to [\sigma] \\
&= \textbf{fun}\ (\mu : [\tau]) \Rightarrow \textbf{fun}\ (M : \tau \to [\sigma]) \Rightarrow \\
&\quad\ \textbf{fun}\ (f : \sigma \to [0,1]) \Rightarrow\ (\mu\ \textbf{fun}\ (x : \tau) \Rightarrow\ (M\ x\ f))
\end{aligned}$$

This definition obviously satisfies the expected monadic properties, for instance $(\mathsf{bind}\ (\mathsf{unit}_\tau\ x)\ M) = (M\ x)$ and $(\mathsf{bind}\ (\mathsf{bind}\ \mu\ M_1)\ M_2) = (\mathsf{bind}\ \mu\ (\textbf{fun}\ x \Rightarrow (\mathsf{bind}\ (M_1 x)\ M_2))$. It is actually possible to extend these operators to functional types:

$$\begin{aligned}
\mathsf{unit}_{\tau_1 \to \tau_2} \ &: \ (\tau_1 \to \tau_2) \to [\tau_1 \to \tau_2] \\
&= \textbf{fun}\ (f : \tau_1 \to \tau_2) \Rightarrow \textbf{fun}\ (x : \tau_1) \Rightarrow\ \mathsf{unit}_{\tau_2}(f\ x) \\
\mathsf{bind}_{\sigma_1 \to \sigma_2} \ &: \ [\tau] \to (\tau \to [\sigma_1 \to \sigma_2]) \to [\sigma_1 \to \sigma_2] \\
&= \textbf{fun}\ (\mu : [\tau]) \Rightarrow \textbf{fun}\ (M : \tau \to [\sigma_1 \to \sigma_2]) \Rightarrow \\
&\quad\ \textbf{fun}\ (y : \sigma_1) \Rightarrow\ \mathsf{bind}_{\sigma_2}\ \mu\ (\textbf{fun}\ (x : \tau) \Rightarrow\ (M\ x\ y))
\end{aligned}$$

Notice that, since we are only manipulating first-order programs in this paper, these generalized operators will not be needed in the examples. Following the translation scheme, if f has type $\tau_1 \to \tau_2 \to \sigma$, the binary application $((f\ a)\ b)$ should be translated into $\mathsf{bind}_\sigma\ [b]\ (\mathsf{bind}_{\tau_2 \to \sigma}\ [a]\ [f])$ but it is always possible, and probably more readable, to use the equivalent expanded form: $\mathsf{bind}_\sigma\ [b]\ (\mathbf{fun}\ (y : \tau_2) \Rightarrow \mathsf{bind}_\sigma\ [a]\ \mathbf{fun}(x : \tau_1) \Rightarrow ([f]\ x\ y))$.

Interpretation. From the measure point of view, $(\mathsf{unit}_\tau\ x)$ is the Dirac measure at point x. If x is an expression of type τ with no randomized construction then it evaluates deterministically to a value v and the probability of the result to satisfy P is one when $P(v)$ is true and zero otherwise.

In the definition of bind_σ, μ is a measure on τ, and M is a family of measures on σ parameterized with $x \in \tau$. Given a function f on σ, $\mathsf{bind}\ \mu\ M$ measures with μ the function which associates with x the measure of f given by $(M\ x)$. For example, assume a is a randomized expression of type τ and e is a function which associates with $x : \tau$ a randomized expression of type σ. Given a property P on σ, we want to evaluate the probability for $(e\ a)$ to satisfy P. Interpreting e as a parameterized measure M we can compute, for a given value x, the probability for $(e\ x)$ to satisfy P. Then if we integrate this function with respect to x, using the measure associated with the expression a, we end up with the probability for $(e\ a)$ to satisfy P. That is exactly what bind is doing. This definition of bind captures the independence of random choices done in f and a.

Interpretation of randomized constructions For the additional primitives, we get

$$\begin{array}{lll} \texttt{random}(n) : [\texttt{int}] & = \mathbf{fun}\ (f : \texttt{int} \to [0,1]) \Rightarrow \Sigma_{i=0}^{n} \frac{1}{n+1}(f\ i) \\ \texttt{flip}() : [\texttt{bool}] & = \mathbf{fun}\ (f : \texttt{bool} \to [0,1]) \Rightarrow \frac{1}{2}(f\ \texttt{true}) + \frac{1}{2}(f\ \texttt{false}) \\ e_1\ {}_p\!+\ e_2 : [\tau] & = \mathbf{fun}\ (f : \tau \to [0,1]) \Rightarrow p \times ([e_1]\ f) + (1-p) \times ([e_2]\ f) \end{array}$$

2.6 Functional interpretation : an example

Now that the monadic translation is defined, we can transform an expression e which computes a value randomly into a expression $[e]$ which does a deterministic computation of the measure associated with the expression e. Before looking at this interpretation for proofs, we can use it simply for computation, in a functional language like Caml. A basic example of a randomized algorithm is the primality test. The principle of this algorithm is the following. We want to check whether a number p is prime. There is a deterministic test ($\texttt{test}$) which applies to $1 \leq k < p$ and p such that:

- If p is prime then $(\texttt{test}\ k\ p)$ evaluates to $\texttt{true}$ for all k
- If p is not prime then $(\texttt{test}\ k\ p)$ evaluates to $\texttt{true}$ for a limited number of k, say N less than $\frac{p-1}{2}$.

We choose k randomly and run the test: if the answer is false, then p is not prime; if the answer is true then p is not prime with a probability $\frac{N}{p-1}$ which is less than $\frac{1}{2}$. Iterating the test improves the level of confidence, provided the random choices of k are independent.

In our language (extended with a simple bounded fixpoint), the function which iterates n times the primality test for p can be written: [3]

```
let rec prime_test p n =
    if n = 0 then true
    else if test (random' (p-1)) p then prime_test p (n-1)
    else false
```

Using the monadic transformation, and monadic simplification laws, we get the functional computation of the associated measure:

```
let rec prime_test_fun p n =
    if n = 0 then (unit true)
    else bind (random' (p-1))
              fun a ⇒ if (test p a) then (prime_test_fun p (n-1))
                      else (unit false)
```

Now if we want to evaluate the probability for our program to give a correct answer, we define the characteristic function of the correctness predicate, which says that the result is true exactly when p is prime, and which is encoded as:

```
let prime_correct p b = if b = exact_prime p then 1. else 0.
```

One can now explicitly compute the probability that our program gives a correct answer after n iterations:

```
let evaluate p n = prime_test_fun p n (prime_correct p)
```

The function can be run in Caml and gives the following results.

```
# evaluate 23 1;;
- : float = 1
# [evaluate 9 0;evaluate 9 1;evaluate 9 2;evaluate 9 3];;
- : float list = [0.;0.75;0.9375;0.984375]
```

If the number is prime (example $p = 23$), then the result will be correct with probability one. On the other hand, if p is not prime (example $p = 9$) then the probability that the program gives a correct answer after 0 iteration is 0, after 1 iteration, we get the good answer 3 times out of 4 and it goes to more than 98% of good answers after 4 iterations.

One nice point is that we have been able to compute these probabilities with a simple ML program without any specific knowledge on probability theory nor number theory. On the other hand, if we analyze the program, we remark that it is very inefficient:

[3] We use a function `random'` defined as `random'` $n =$ `random` $(n-1)+1$ in order to get a number between 1 and n

– in order to build the characteristic function to be tested we need to know (or to test) exactly if p is prime or not;
– because of the interpretation of `random`, the program is executed for all the values of k between 1 and $p-1$ before computing the average number of good answers.

Furthermore, this computational approach does not work in all cases. Our previous program uses a structural recursion which always terminates. Many interesting probabilistic programs only terminate with probability one, which is a weaker requirement. For instance the following function flips a coin and returns how many flips it took to get `false`, this is a typical example of a random walk:

```
let rec walk x = if flip () then walk (x+1) else x
```

If we test this function in Caml several times, we get small number answers such as $1, 2, 3$. We may apply our translation scheme:

```
let rec walk_fun x =
     bind flip (fun (b:bool) ⇒ if b then walk_fun (x+1)
                               else (unit x))
```

and measure the function which is 1 everywhere:

```
# walk_fun 1 (fun n -> 1.);;
Stack overflow during evaluation (looping recursion?).
```

it loops because our interpretation tests all the cases, in particular the one where the result of `flip` is always true...

This example shows that, when general fixpoints are involved, we cannot anymore use computation for analyzing the probability of events. We shall need to reason about these programs instead. For that, we first define a Coq theory for representing distributions, then we prove several theorem for analyzing programs.

3 Coq representation of randomized programs

We present now our model of randomized programs in the proof assistant Coq. We follow the ideas presented in the previous section in order to associate with each program a measure and to reason directly on these measures.

3.1 The set $[0, 1]$

Our model is based on measures seen as functionals of type $(A \to [0,1]) \to [0,1]$. For constructing this model in Coq, we have chosen to axiomatize a type U which corresponds to the interval $[0, 1]$.

Definitions Let two special constants 0 and 1 in U. The basic operations are multiplication, addition and a special inversion function. The addition is bounded: it gives the minimum of addition on reals and 1. The inversion function associates $1-x$ with x. We have also two predicates on U, $x \leq y$ and $x = y$, with the standard meaning.

For each natural number n, we introduce a special element $\frac{1}{n+1}$ in U

To deal with unbounded computations, we also need the least-upper bound (*lub*) of any denumerable set of elements of U, represented as a function from `nat` to U: we consequently adjoin a parameter `lub` with type $(\texttt{nat} \to U) \to U$. If f is an expression with a free variable n, we write $\texttt{lub}(f)_n$ instead of `lub` (**fun** $n \Rightarrow f$).

Axioms We have axioms which say that $\forall x : U, 0 \leq x \leq 1$ and that $0 \neq 1$.

As expected, the previous operators come with the usual axioms stating that addition and multiplication are symmetric and associative, with 0 and 1 as their respective neutral elements, and so on.

Our inversion function enjoys good properties such as $1-(1-x) = x$. Some properties of addition are only valid when there is no overflow during addition. The non-overflow condition is expressed in our formalism as $x \leq 1-y$. For instance, assuming $x \leq 1-y$, we have:

$$(1-(x+y))+x = 1-y \qquad (x+y)\times z = x\times z + y\times z \qquad x+y \leq x+z \Rightarrow y \leq z$$

The axioms for least upper bounds include the two basic properties of *lubs* and the fact that *lubs* are compatible with addition and multiplication

$$\texttt{lub}\ ((f\ n)+k)_n = \texttt{lub}\ f + k \quad \texttt{lub}\ ((f\ n)\times k)_n = \texttt{lub}\ f \times k$$

We also need two extra properties:

$$\neg\neg(x \leq y) \to x \leq y \quad x \leq y \vee y \leq x$$

The first property is required because Coq implements an intuitionistic logic in which $\neg\neg A \Rightarrow A$ is not satisfied for all propositions. The second property states that the order is total.

The operation $\frac{1}{n+1}$ satisfies the axiom $\frac{1}{n+1} = 1 - n \times \frac{1}{n+1}$ where $n \times \frac{1}{n+1}$ is a generalized sum defined by induction on n.

Finally the fact that U is archimedian is axiomatized by the property

$$\forall x, x \neq 0 \Rightarrow \exists n, \frac{1}{n+1} \leq x$$

Remarks. Our modeling of randomized programs does not depend on our particular axiomatization of $[0,1]$. Our choices are somehow arbitrary, we tried to find an axiomatization with a minimum number of operations and axioms such that the theory could be easily instantiated by different representations of real numbers (we are interested in particular by constructive reals). We use the functor mechanism of Coq in order to keep the axiomatization of $[0,1]$ as a parameter of the theory.

Derived operators The usual minus operation $x - y$ (which is zero when $x \leq y$) can be defined using our special inverse by: $x - y = 1 - ((1 - x) + y)$ The operator `max` can be defined as $(x - y) + y$. It is also easy to define $n \times x$ and x^n for an integer n by induction on n. In [14], C. Morgan and A. McIver use an operator $x\&y$ defined on non-negative real numbers as the maximum of 0 and $x + y - 1$. The same operator can be defined in our theory using the inverse operator and addition by $x\&y \equiv 1 - ((1 - x) + (1 - y))$. It is the dual operation of addition because we have $(1-(x\&y)) = (1-x)+(1-y)$ and $(1-(x+y)) = (1-x)\&(1-y)$. This operator captures intersection of properties because $\mathbb{I}_{P \cap Q} = \mathbb{I}_P \& \mathbb{I}_Q$.

3.2 Definition of a distribution

In the following, we extend in a standard way the operations and relations on U, to operations and relations on functions of type $A \to U$ using the same notations: $f \leq g$ will stand for $\forall x, f\ x \leq g\ x$ and $f + g$ is the function **fun** $x \Rightarrow f\ x + g\ x$.

Given a type A, we define a distribution on A to be a measure μ of type $(A \to U) \to U$ which furthermore satisfies stability properties, namely:

- monotonicity : $\forall f\ g : A \to U, (f \leq g) \Rightarrow \mu(f) \leq \mu(g)$
- compatibility with addition :
  $\forall f\ g : A \to U, (f \leq 1 - g) \Rightarrow \mu(f + g) = \mu(f) + \mu(g)$
- compatibility with inverse : $\forall (f : A \to U), \mu(1 - f) \leq 1 - \mu(f)$
- compatibility with multiplication :
  $\forall (k : U)(f : A \to U), \mu(k \times f) = k \times \mu(f)$

In Coq, we use a dependent record type in order to introduce a type (`distr` A) which contains the measure μ plus the proofs of compatibility properties for μ.

Remarks. Because the addition is bounded, the compatibility with respect to addition is only assumed when there is no overflow in the addition of f and g. We also need the extra condition of compatibility with respect to inversion which is usually derived from linearity.

We allow a distribution to be a sub-probability with possibly $\mu(1 - f) < 1 - \mu(f)$ (i.e. $\mu(\mathbb{I}) < 1$). This is useful for interpreting non terminating programs.

Monotonicity could be replaced by compatibility with respect to equality $\forall f\ g : A \to U, (f = g) \Rightarrow \mu(f) = \mu(g)$. Assuming this property, monotonicity comes from the fact that $g = (g - f) + f$ and stability with respect to addition.

Derived properties. From this definition, we can deduce further properties, such as $\mu($**fun** $x \Rightarrow 0) = 0$, or $\mu(1 - f) = \mu(\mathbb{I}) - \mu(f)$

It is possible to prove that $\mu(f+g) \leq \mu(f)+\mu(g)$ is valid without extra non-overflow condition. In a dual manner, we have proved $\mu(f)\&\mu(g) \leq \mu(f\&g)$.

Monadic operators. We define the monadic operators on distributions: `Munit` of type $\forall A, A \to$ `distr` A and `Mlet` of type $\forall A\ B,$ `distr` $A \to (A \to$ `distr` $B) \to$ `distr` B. These operations are based on the transformations bind and unit for measures, while including extra proofs stating that these operations are stable with respect to the expected properties of distributions.

Properties We can define an order and an equality on the type (`distr` A) by a simple extensions of the relations on U. This leads to proofs of monadic equalities, as well as monotonicity of the bind operation.

In particular we prove [16]:

- $\forall(\mu : \texttt{distr}\ A), \texttt{Mlet}\ \mu\ (\textbf{fun}\ (x : A) \Rightarrow \texttt{Munit}\ x) = \mu$
- $\forall(\mu : \texttt{distr}\ A)\ (M : A \to \texttt{distr}\ B)\ (N : B \to \texttt{distr}\ C)$,
  $\texttt{Mlet}\ (\texttt{Mlet}\ \mu\ M)\ N = \texttt{Mlet}\ \mu\ (\textbf{fun}\ (x : A) \Rightarrow \texttt{Mlet}\ (M\ x)\ N)$
- $\forall(\mu_1\ \mu_2 : \texttt{distr}\ A)\ (M_1\ M_2 : A \to \texttt{distr}\ B)$,
  $\mu_1 \leq \mu_2 \Rightarrow (\forall x, (M_1\ x) \leq (M_2\ x)) \Rightarrow \texttt{Mlet}\ \mu_1\ M_1 \leq \texttt{Mlet}\ \mu_2\ M_2$

Random distributions. Following the interpretation of random primitives we gave in section 2.5, we can define in Coq the corresponding distributions, we have to formally prove the stability properties.

The primitive `flip` has type (`distr bool`), `random` has type `int` $\to$ (`distr int`) and the choice operator has type $U \to$ (`distr` A) $\to$ (`distr` A) $\to$ (`distr` A).

The framework is not limited to discrete distributions. While defining completely a measure on U could require the development of a non-trivial part of analysis, it is already possible, for example as found in [15], to introduce as a parameter a new distribution `uniform` of type (`distr` U) with the extra assumption that for all $a, b \in U$, the measure of the interval $[a, b]$ is equal to $b - a$, i.e. $(\texttt{uniform}\ \mathbb{I}_{a \leq . \leq b}) = b - a$.

Interpretation of simple programs. The constructors `Mlet`, `Munit`, `flip`, `random` are sufficient for interpreting simple random programs. Following our general monadic translation scheme, one can also define a conditional operation `Mif` of type (`distr bool`) $\to$ (`distr` A) $\to$ (`distr` A) $\to$ (`distr` A) by

$$\texttt{Mif}\ \mu_b\ \mu_1\ \mu_2 \equiv \texttt{Mlet}\ \mu_b\ (\textbf{fun}\ b \Rightarrow \texttt{if}\ b\ \texttt{then}\ \mu_1\ \texttt{else}\ \mu_2).$$

We use this operator for interpreting conditional programs:
$[\texttt{if}\ b\ \texttt{then}\ e_1\ \texttt{else}\ e_2] \equiv \texttt{Mif}\ [b]\ [e_1]\ [e_2]$

3.3 Interpretation of fixpoints

As expected, the difficult part is the interpretation of general fixpoints. This is achieved through the following steps.

Limit of distributions. In order to interpret recursive functions, we need to take limits of sequences of distributions.

We assume given a denumerable family of distributions $(\mu_n)_{n \in \mathbb{N}}$ of type `distr` A, such that $\forall n\ m, n \leq m \Rightarrow \mu_n \leq \mu_m$. Then we can define a new distribution as the least upper bound of $(\mu_n)_n$. The associated measure, $\mu_{\texttt{lub}}(\mu_n)_n$, is defined by $\mu_{\texttt{lub}}(\mu_n)_n\ (f) \equiv \texttt{lub}\ (\mu_n\ (f))_n$.

Fixpoints. Let us consider we want to define a function which satisfies the equation

let rec `f x = F f x`

where f is assumed to take an argument in type A, and returns a random value of type B, so that it is interpreted as a function of type $A \to \texttt{distr}\ B$. Thus, F will have type $(A \to \texttt{distr}\ B) \to A \to \texttt{distr}\ B$, and we assume this functional to be monotonic: $f \leq g \Rightarrow F\ f \leq F\ g$.

Let us define the sequence M_n of functions of type $A \to \texttt{distr}\ B$, by repeated iterations of F from the null distribution:

$$M_0\ x = \textbf{fun}\ f \Rightarrow 0 \qquad M_{n+1}\ x = F\ M_n\ x$$

The limit distribution `Mfix` is defined, for each given x, as the least upper bound of the sequence which associates with n the distribution $(M_n\ x)$:

$$\texttt{Mfix}\ F\ x \equiv \mu_{\texttt{lub}}(M_n\ x)_n$$

We can derive the inequalities

$$\texttt{Mfix}\ F\ x \leq F\ (\texttt{Mfix F})\ x \quad \text{and} \quad F\ (\texttt{Mfix}\ F)\ x \leq \texttt{Mfix}\ F\ x$$

The second inequality requires an extra hypothesis of continuity namely that for all monotonic sequences $(g_n)_{n\in\mathbb{N}}$ of type $A \to \texttt{distr}\ B$,

$$F\ (\textbf{fun}\ y \Rightarrow \mu_{\texttt{lub}}(g_n\ y)_n)\ x \leq \mu_{\texttt{lub}}(F\ g_n\ x)_n$$

However, as we will see in section 4.2, estimating programs built with fixpoints can be done without using this rule.

4 Derived rules for reasoning on programs

For reasoning about programs, it is convenient to use an axiomatic semantics that provides rules by induction on the structure of the program, stating as usual, how some post-condition is satisfied after execution, provided some precondition holds. In fact, in the context of probabilistic programs, we are interested (see also [10]) in deriving that the probability for a certain property to hold is greater than a certain value.

Thus we look forward deriving judgements of the form $k \leq [e](f)$ where $k \in [0,1]$, e is an expression of type A and f is a function of type $A \to [0,1]$.

The meaning of this judgement is that the measure associated with the program e computed on the function f is no less than k. Usually f will be the characteristic function $\mathbb{I}_P$ of some predicate P of type $A \to \texttt{bool}$. The judgement $k \leq [e](\mathbb{I}_P)$ therefore means that the probability for the result of e to satisfy P is at least k.

4.1 Basic rules

We can prove the following rule for application:

$$\frac{k \leq [a](f) \quad \forall x, f\ x \leq [e\ x](g)}{k \leq [e\ a](g)}$$

For the case of conditional, we can prove the rule:

$$\frac{k_1 \leq [e_1](f) \quad k_2 \leq [e_2](f)}{k_1 \times [b](\mathbb{I}_{\_=\texttt{true}}) + k_2 \times [b](\mathbb{I}_{\_=\texttt{false}}) \leq [\textsf{if}\ b\ \textsf{then}\ e_1\ \textsf{else}\ e_2](f)}$$

4.2 Rule for fixpoints

We now justify the rule for estimating fixpoints which follows the ideas presented in [7]. We assume F has type $(A \to \texttt{distr}\ B) \to A \to \texttt{distr}\ B$ and is monotonic. We take a monotonic sequence $(p_i)_i$ of functions of type $A \to U$ such that $\forall x, p_0\ x = 0$. The following rule is valid:

$$\frac{\forall f : A \to \texttt{distr}\ B, (\forall x, p_n\ x \leq [f\ x](q)) \Rightarrow (\forall x, p_{n+1}\ x \leq [F\ f\ x](q))}{\forall x, \texttt{lub}\ (p_n\ x)_n \leq [\textsf{fix}\ F\ x](q)}$$

No continuity condition on F is required to validate this rule. The sequence $(p_n)_n$ can be seen as a generalized invariant for randomized programs: assuming that the recursive goal establishes a post-condition Q with probability at least p_n, we prove that one further iteration establishes Q with probability at least p_{n+1}, and we finally get that the recursive program establishes Q with a probability which is at least the $\texttt{lub}$ of $(p_n)_n$.

4.3 Other rules

We can derive in our formalism useful schemes which generalize reasoning on deterministic programs. For instance, if we have established that the an expression a satisfies a predicate P with probability 1, then it is possible to reason subsequently exactly as if P was true for the result of the computation of a.

This is stated in the following derivable rule:

$$\frac{1 \leq [a](\mathbb{I}_P) \quad \forall x, (P\ x) \Rightarrow k \leq [b](f)}{k \leq [\textsf{let}\ x = a\ \textsf{in}\ b](f)}$$

5 Applications

We apply our approach for proving properties of simple randomized programs.

5.1 Probabilistic termination

We return to our example of section 2.6, a random walk which illustrates probabilistic termination.

```
let rec walk x = if flip() then walk (x+1) else x
```

We show that this program terminates with probability one. For that it is enough to prove that:

$$\forall x, 1 \leq [\texttt{walk}\ x](\mathbb{I}).$$

We shall apply the fixpoint rule with a functional F defined by

$$F\ f\ x \equiv \texttt{Mif}\ \mu_{\texttt{flip}}\ (f(x+1))\ (\texttt{Munit}\ x)$$

We introduce a sequence p_i defined by $p_0 = 0$ and $p_{i+1} = \frac{1}{2}p_i + \frac{1}{2}$. It is easy to show that $p_n = 1 - \frac{1}{2^n}$ and that the least upper bound of the sequence $(p_i)_i$ is 1. In order to prove $1 \leq [\texttt{Mfix}\ F\ x](\mathbb{I})$, we use the fixpoint rule and show:

$$\forall f, (\forall x, p_i \leq [f\ x](\mathbb{I})) \Rightarrow \forall x, p_{i+1} \leq [F\ f\ x](\mathbb{I})$$

We assume $\forall x, p_i \leq [f\ x](\mathbb{I})$ and we simplify as follows

$$\begin{array}{rcl} p_{i+1} \leq [F\ f\ x](\mathbb{I}) & \Leftrightarrow & \frac{1}{2}p_i + \frac{1}{2} \leq [\texttt{Mif}\ \mu_{\texttt{flip}}\ (f(x+1))\ (\texttt{Munit}\ x)](\mathbb{I}) \\ & \Leftrightarrow & \frac{1}{2}p_i + \frac{1}{2} \leq \frac{1}{2}f(x+1)(\mathbb{I}) + \frac{1}{2}\mathbb{I}(x) \end{array}$$

This is trivially true because $p_i \leq f(x+1)(\mathbb{I})$ by hypothesis and $\mathbb{I}(x) = 1$.

5.2 The Bernoulli distribution

We now apply our technique to the proof of an algorithm to simulate a Boolean function following Bernoulli's distribution (which is `true` with some probability p and false with probability $1 - p$) using only a coin flip. The algorithm which is also taken as an example in [3] uses a simple idea : write p in binary form $\Sigma_{i=1}^{\infty} p_i \frac{1}{2^i}$, if we flip a coin and get a sequence $(q_i)_{i \geq 1}$ then the first time we get $q_i \neq p_i$, we answer true when $q_i < p_i$ and false otherwise. Now this function can be expressed recursively. If $p < \frac{1}{2}$ then $p_1 = 0$ and the remainder of the sequence corresponds to $2 \times p = p + p$. If $\frac{1}{2} \leq p$ then $p_1 = 1$ and the remainder of the sequence corresponds to $2 \times p - 1 = p\&p$ (using the special operation $x\&y$ we introduced in section 3.1). Our Bernoulli program can be written as

```
let rec bernoulli p =
    if flip() then if p < 1/2 then false else bernoulli (p & p)
              else if p < 1/2 then bernoulli (p + p) else true
```

We directly translate this definition into a distribution, as was done in the case of the random walk. In order to analyze this program, we use the fixpoint rule and prove that

$$\forall p, \textbf{lub}_n\ (p - \frac{1}{2^n}) \leq [\texttt{bernouilli}\ p](\mathbb{I}_{\_=\texttt{true}}).$$

Assuming $\forall p, (p - \frac{1}{2^n}) \leq [\texttt{bernouilli}\ p](\mathbb{I}_{.=\texttt{true}})$, we just simplify the expression corresponding to the body of `bernoulli`. In case $p < \frac{1}{2}$, we have to show that

$$p - \frac{1}{2^{n+1}} \leq \frac{1}{2}\texttt{bernouilli}(p+p)$$

and in case $\frac{1}{2} \leq p$, we have to show that

$$p - \frac{1}{2^{i+1}} \leq \frac{1}{2}\texttt{bernouilli}(p\&p) + \frac{1}{2} \times 1$$

this follows easily using the fixpoint rule hypothesis and algebraic properties. The same reasoning allows to prove:

$$\forall p, \mathbf{lub}_n\ ((1-p) - \frac{1}{2^n}) \leq [\texttt{bernouilli}\ p](\mathbb{I}_{.=\texttt{false}}).$$

Using the fact that $\mathbb{I}_{.=\texttt{false}} = 1 - \mathbb{I}_{.=\texttt{true}}$ and the property of measures of inverse functions, we conclude that $[(\texttt{bernouilli}\ p)]\mathbb{I}_{.=\texttt{true}} = p$.

Using $\mathbb{I}_{.=\texttt{true}} + \mathbb{I}_{.=\texttt{false}} = \mathbb{I}$, we also have $[(\texttt{bernouilli}\ p)]\mathbb{I} = 1$ which shows that the process terminates with probability one.

5.3 Improving precision

Another example is an abstract version of a program scheme where a randomized program is executed twice in order to improve the probability of getting a correct result. The implicit assumption is that given two runs on the program we can choose the better of the two answers. In case of primality for instance, if one of the test answers that p is not prime, we are sure that p is not prime; only when the two programs assert that p is prime, we can still pretend (but with higher confidence) that p is prime.

We want to compute a value in a type A which satisfies a property Q with a certain probability. The hypotheses are that we have two programs p_1 and p_2 of type A, thus interpreted as objects of type `distr` A. We want to combine p_1 and p_2 in order to get a better program i.e. we want to improve the probability that the result is correct.

We assume we have a function `choice` of type $A \to A \to A$ such that $(Q\ x) \Rightarrow Q\ (\texttt{choice}\ x\ y)$ and $(Q\ y) \Rightarrow Q\ (\texttt{choice}\ x\ y)$ are provable.

In case of a Boolean test for primality of p, we have $(Q\ b)$ defined as $(b = \texttt{true} \Leftrightarrow p\ \texttt{is prime})$ and $(\texttt{choice}\ b_1\ b_2)$ defined as $(b_1\ \texttt{and}\ b_2)$.

Now we build a new program p:

let `x =` p_1 **in let** `y =` p_2 **in** `choice x y`

We want to show that $k_1 \leq [p_1](\mathbb{I}_Q)$ and $k_2 \leq [p_2](\mathbb{I}_Q)$ implies $k_1(1-k_2)+k_2 \leq [p](\mathbb{I}_Q)$. The new estimation $k_1(1-k_2)+k_2$ (also equal to $k_2(1-k_1)+k_1$) is greater than both k_1 and k_2.

Actually we established a more general result, using an arbitrary function q of type $A \to U$ instead of the characteristic function $\mathbb{I}_Q$ of a predicate Q. We

assume that $\forall x\ y, (q\ x) + (q\ y) \leq q\ (\texttt{choice}\ x\ y)$ (with bounded addition). It is easy to see that when q is the characteristic function $\mathbb{I}_Q$, then the assumptions $(Q\ x) \Rightarrow Q\ (\texttt{choice}\ x\ y)$ and $(Q\ y) \Rightarrow Q\ (\texttt{choice}\ x\ y)$ are equivalent to $(\mathbb{I}_Q\ x) + (\mathbb{I}_Q\ y) \leq \mathbb{I}_Q\ (\texttt{choice}\ x\ y)$. We also need the fact that both programs p_1 and p_2 terminate with probability one, otherwise our choice function could give a result which is not as good as p_1 and p_2.

Now, the property to be shown amounts to

$$k_1(1-k_2) + k_2 \leq [p_1](\textbf{fun}\ x \Rightarrow [p_2](\textbf{fun}\ y \Rightarrow (q\ (\texttt{choice}\ x\ y))))$$

Using the fact that

$$(q\ x) \times (1 - (q\ y)) + (q\ y) \leq (q\ x) + (q\ y) \leq (q\ (\texttt{choice}\ x\ y))$$

the proof reduces to

$$k_1(1-k_2) + k_2 \leq [p_1](\textbf{fun}\ x \Rightarrow [p_2](\textbf{fun}\ y \Rightarrow (q\ x) \times (1 - (q\ y)) + (q\ y)))$$

Algebraic properties of measures lead to simplification of the right-hand side:

$$[p_1](q) \times [p_2](1-q) + [p_2](q)$$

Because p_2 terminates, we have $[p_2](1-q) = 1 - [p_2](q)$ (only the inequality is true in general) so we have to show:

$$k_1(1-k_2) + k_2 \leq [p_1](q)(1 - [p_2](q)) + [p_2](q)$$

which is true because $k_1(1-k_2)+k_2 = k_2(1-k_1)+k_1$ is monotonic with respect to both k_1 and k_2.

This example illustrates the possibility to do abstract modular reasoning in our framework.

6 Related work

In [15], Park and al. propose a functional language, named $\lambda_{\bigcirc}$ which extends the ML functional kernel on the basis of the monadic metalanguage developed by Pfenning and Davies [17]. It is a reformulation of Moggi's monadic metalanguage (the **let**...**in**... construction) which augments the λ-calculus, consisting of terms, with a separate syntactic category, consisting of expressions which denote probabilistic computations. A term can be cast to a (random) expression. From any expression E, the operator **prob** E builds the image measure. In our work, both terms and (random) expressions are not distinguished, unit providing the corresponding operator into measures. Besides, the current bind operation is represented by **sample** x **from** M **in** E in $\lambda_{\bigcirc}$. The language introduces a new constant $\mathcal{S}$ which denotes an expression, i.e. a random variable which follows the uniform law on the real interval $[0,1]$. The system is simply typed, where types are limited to arrows and pairs, enriched with the monadic construction $\bigcirc_A$ for each type A.

We do not have these two syntactic levels in our system where we chose to represent in Coq only the level of terms. The $\bigcirc_A$ type play the role of (`distr` A) in our formalism and the value `prob`(E) corresponds to our definition of $[E]$. Their formalism allows to build distributions on arbitrary types (possibly functional), an extension we did not investigate yet.

$\lambda_\bigcirc$ is mainly designed toward expressiveness as a programming language, for which the paper provides a small steps operational semantics. This corresponds to Kozen's first semantics in [9], where any computation involved in a reasoning step about a program requires the user to refer to the measurable space of random streams over $[0, 1]$. As far as reasoning on programs is concerned, this is not of great help, since axiomatic semantics relies on denotational semantics. Therefore, examples developed with $\lambda_\bigcirc$ are better analysed through simulation techniques. Both approaches are complementary: we are not able to simulate the programs as sampling functions but we can directly and easily reason on the probabilistic properties of (a subset of) Caml expressions.

In [12], A. McIver and C. Morgan describe an axiomatic semantics for probabilistic programs written in imperative style. The state-predicates in Hoare logic are replaced by so-called *expectations* which are functions from states to $\mathbb{R}^+$, to be evaluated following the distribution defined by the program. An important aspect of this work is to introduce in the language a non-deterministic (demonic) choice $p \sqcap q$. The probability for a property P to hold after executing $p \sqcap q$ is the minimum of the probabilities that P holds after executing p and after executing q. This operator is used to represent specifications and for defining a refinement relation. In order to adapt our approach to the non-deterministic case, an idea could be to relax the compatibility condition for addition in the definition of a distribution into the weaker condition $\mu(f + g) \leq \mu(f) + \mu(g)$. Developing the corresponding theory still remains to be done. A mechanization of this calculus using the HOL theorem prover is presented in [6]. In this work programs are interpreted as functionals of type $(\alpha \to \mathbb{R}^+_\infty) \to (\alpha \to \mathbb{R}^+_\infty)$ where $\mathbb{R}^+_\infty \equiv \mathbb{R}^+ \cup \{\infty\}$ and α is the type of states. They propose a so-called *deep-embedding* where the syntax of the language of guarded commands and the weakest-precondition generator are explicitely encoded in the proof assistant, while we use a *shallow* embedding where we directly use the semantics of the language. Their approach allows to measure an arbitrary function with value in $\mathbb{R}^+$ and not only $[0, 1]$. We choose to restrict ourselves to $[0, 1]$ in order to simplify the formal development in Coq and because it is sufficient for correctness. Measuring arbitrary function can nevertheless be interesting in some cases. For instance, in the random walk example, one could measure the average of the result of the function (how many flips before we get false). It is possible to represent an element in $\mathbb{R}^+$ with a pair (n, x) with $n \in \mathbb{N}$ and $x \in [0, 1]$ and reuse a large part of our development in order to extend a measure of type $(A \to [0, 1]) \to [0, 1]$ into a measure of type $(A \to \mathbb{R}^+) \to \mathbb{R}^+$. We may introduce for each $n \in \mathbb{N}$ a function $f_n : A \to [0, 1]$ such that $f_n(x) = y$ when $f(x) = (n, y)$ and $f(x) = 0$ otherwise. We have $f = \Sigma_{n=0}^{\infty} f_n$ and we can define $\mu(f)$ as $\Sigma_{n=0}^{\infty} \mu(f_n)$ when it exists.

As already said in the introduction, our approach comes actually closer to J. Hurd's thesis, where formal verification of probabilistic programs is handled with the HOL theorem prover. He uses a monadic translation based on a global state with a stream of boolean values. Reasoning on programs required to define within HOL an adequate distribution over this infinite structure, while we only use simple mathematical constructions. It would be interesting to compare more carefully the complexity of proofs of high-level programs in both systems.

7 Conclusion

We have studied the interpretation of probabilistic programs in a functional framework using a monadic interpretation of programs as probability distributions represented by measures.

We have applied this technique for building an environment for reasoning about probabilistic programs in the Coq proof assistant. We have developed an axiomatization for the set $[0, 1]$ which uses a few primitive operations : bounded addition, multiplication and inverse $(1 - x)$.

We have derived axiomatic rules for estimating the probability that programs satisfy certain properties, following the structure of the program. The fixpoint rule is especially useful for dealing with probabilistic termination of programs. We use these rules for studying a few basic examples such as the computation of a function following a Bernoulli distribution. The development and results presented in this paper have been formally derived and checked in the Coq proof assistant and are available as a contribution [16].

Future works include automatic translation from functional randomized programs to Coq terms representing the corresponding distribution. One possibility could be to use a monadic meta-language in the spirit of [15] on top of the Coq proof assistant. Another possibility is to follow the approach of the Why tool [2, 1], a generic environment for analysing non-purely functional programs. It automatically generates verification conditions from the specification of pre and post conditions and invariants plus a validation (the correctness proof in Coq obtained from the monadic translation of the program).

We also plan to study advanced examples that certainly will require a more sophisticated automation of proofs.

Acknowledgments We thank A. McIver and C. Morgan for useful comments on an earlier version of this paper. We also thank R. Lassaigne for stimulating discussions on formal proofs for analyzing random programs.

References

1. Jean-Christophe Filliâtre. The why verification tool, 2002. http://why.lri.fr/.
2. Jean-Christophe Filliâtre. Verification of Non-Functional Programs using Interpretations in Type Theory. *Journal of Functional Programming*, 13(4):709–745, July 2003.

3. Joe Hurd. A formal approach to probabilistic termination. In Victor A. Carreño, César A. Muñoz, and Sofiène Tahar, editors, *Theorem Proving in Higher Order Logics: 15th International Conference, TPHOLs 2002*, volume 2410 of *Lecture Notes in Computer Science*, pages 230–245, Hampton, VA, USA, August 2002. Springer-Verlag.
4. Joe Hurd. *Formal Verification of Probabilistic Algorithms*. PhD thesis, University of Cambridge, 2002.
5. Joe Hurd. Verification of the Miller-Rabin probabilistic primality test. *Journal of Logic and Algebraic Programming*, 50(1–2):3–21, May–August 2003. Special issue on Probabilistic Techniques for the Design and Analysis of Systems.
6. Joe Hurd, Annabelle McIver, and Carroll Morgan. Probabilistic guarded commands mechanized in HOL. In A. Cerone and A. Di Pierro, editors, *Proceedings of the 2nd Workshop on Quantitative Aspects of Programming Languages (QAPL 2004)*, volume 112 of *Electronic Notes in Theoretical Computer Science (ENTCS)*, pages 95–111, Barcelona, Spain, January 2005. Elsevier.
7. Claire Jones. *Probabilistic Non-determinism*. PhD thesis, University of Edinburgh, 1989.
8. Claire Jones and Gordon Plotkin. A probabilistic powerdomain of evaluations. In *Proceedings of the Fourth Annual Symposium on Logic in Computer Science*, Pacific Grove, California, 1989. IEEE Comp. Soc. Press.
9. Dexter Kozen. Semantics of probabilistic programs. *Journal of Computer and System Sciences*, 1981.
10. Dexter Kozen. A probabilistic PDL. In *15th ACM Symposium on Theory of Computing*, 1983.
11. Marta Kwiatkowska, Gethin Norman, and David Parker. Probabilistic symbolic model checking with PRISM: A hybrid approach. *International Journal on Software Tools for Technology Transfer (STTT)*, 2004.
12. Annabelle McIver and Carroll Morgan. *Abstraction, Refinement and Proof for Probabilistic Systems*. Monographs in Computer Science. Springer-Verlag, 2005.
13. Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55–92, 1991.
14. Carroll Morgan and Annabelle McIver. pGCL: formal reasoning for random algorithms. *South African Computer Journal*, 1999.
15. Sungwoo Park, Frank Pfenning, and Sebastian Thrun. A probabilistic language based upon sampling functions. In Jens Palsberg and Martín Abadi, editors, *POPL*, pages 171–182. ACM Press, 2005.
16. Christine Paulin-Mohring. A library for reasoning on randomized algorithms in Coq. Description of a Coq contribution, Universit Paris Sud, January 2006.
17. Frank Pfenning and Rowan Davies. A judgmental reconstruction of modal logic. *Mathematical Structures in Computer Science*, 11(4):511–540, 2001.
18. Norman Ramsey and Avi Pfeffer. Stochastic lambda calculus and monads of probability distributions. In John Mitchell, editor, *Conference Record of the 29th Symposium on Principles of Programming Languages*, pages 154–165, Portland, OR, USA, January 2002. ACM Press.
19. The Coq Development Team. *The Coq Proof Assistant Reference Manual – Version V8.0*, April 2004. `http://coq.inria.fr`.