Science of Computer Programming 74 (2009) 568–589

Contents lists available at ScienceDirect

Science of Computer Programming

journal homepage: www.elsevier.com/locate/scico

Proofs of randomized algorithms in Coq

Philippe Audebaud [a], Christine Paulin-Mohring [b,c,*]

[a] LIP, École normale supérieure de Lyon, 46 allée d'Italie, F-69437 Lyon, France
[b] ProVal, INRIA Saclay - Île-de-France, Parc Orsay Université, Zac des Vignes, 4 rue Jacques Monod, F-91893 Orsay, France
[c] LRI, Université Paris-Sud & CNRS, F-91405 Orsay, France

ARTICLE INFO

Article history:
Received 16 February 2007
Received in revised form 30 May 2007
Accepted 21 September 2007
Available online 10 February 2009

Keywords:
Randomized algorithms
Proof of partial and total correctness
Functional language
Axiomatic semantics
Probability framing
Call-by-value
Monadic interpretation

ABSTRACT

Randomized algorithms are widely used for finding efficiently approximated solutions to complex problems, for instance primality testing and for obtaining good average behavior. Proving properties of such algorithms requires subtle reasoning both on algorithmic and probabilistic aspects of programs. Thus, providing tools for the mechanization of reasoning is an important issue. This 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 distributions (Giry, Ramsey and Pfeffer). It does not require the definition of an operational semantics for the language nor the development of a complex formalization of measure theory. Instead it uses functional and algebraic properties of unit interval. Using this model, we show the validity of general rules for estimating the probability for a randomized algorithm to satisfy specified properties. This approach addresses only discrete distributions and gives rules for analyzing general recursive functions.

We apply this theory to the formal proof of a program implementing a Bernoulli distribution from a coin flip and to the (partial) termination of several programs. All the theories and results presented in this paper have been fully formalized and proved in the Coq proof assistant.

1. Introduction

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

1.1. 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 Kozen [15,16] which proposes to interpret randomized imperative programs as measure transformers. This approach has been studied further by Morgan and McIver [20] who 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 lower the probability for the result of the program to satisfy a given property.

* Corresponding author at: ProVal, INRIA Saclay - Île-de-France, Parc Orsay Université, Zac des Vignes, 4 rue Jacques Monod, F-91893 Orsay, France. Tel.: +33 1 72 92 59 05; fax: +33 1 74 85 42 29.
E-mail addresses: Philippe.Audebaud@ens-lyon.fr (P. Audebaud), Christine.Paulin@lri.fr (C. Paulin-Mohring).

0167-6423/$ – see front matter © 2009 Elsevier B.V. All rights reserved.
doi:10.1016/j.scico.2007.09.002

Studying the semantic foundations of probabilistic languages has been the concern of much research. 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 [10,11] or a uniform distribution [21]. 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 uses an interpretation of randomized programs as probability distributions. It is also possible to use a syntactic monadic transformation. In the discrete case, a probability distribution can be represented as a functional mapping from a subset of some σ into the interval $[0, 1]$, or, using expectation, mapping a real-valued function on σ into an element of $\mathbb{R}$. The monadic structure of probability theory was studied by [7] developing unpublished ideas of Lawvere [17]. This approach is used for instance by Ramsey and Pfeffer [24] who interpret a randomized functional term as a Haskell program using the so-called expectation monad, i.e., functions of type $(\sigma \to \mathbb{R}) \to \mathbb{R}$.

1.2. Proofs

Enabling the mechanized reasoning of probabilistic programs requires also tools for analyzing the behavior of these programs. This point is a research topic.

Hurd, McIver and Morgan [12] designed a mechanization of the quantitative logic for probabilistic guarded commands using the proof assistant HOL. Their goal is very similar to ours, except that they analyse a different source language, handling both probabilistic and non-deterministic choice in an imperative settings, while we are only considering probabilistic choice but in a functional language, including recursive functions. Their work also contains the formalization in HOL of meta-reasoning on the source language, while we have for the moment only considered a shallow embedding of our programs in Coq.

With regard to algorithms, Hurd [10,11] shows how to model and prove properties of randomized programs in the HOL proof assistant using a monadic transformation of programs, where Hurd assumes access to an infinite sequence of independent coin flips.

1.3. Our choices

In this article, we intend to prove specifications for probabilistic programs inside the Coq proof assistant. We start by turning a (probabilistic) functional program p on some type A into a pure functional term, denoted as $[p]$, with type $\mathsf{M}A \equiv (A \to [0, 1]) \to [0, 1]$, where $\mathsf{M}A$ is provided with a monadic structure. In this setting, $[p]$ will represent a (mathematical) discrete measure: a sub-probability. Although this monad appears more restrictive than the one proposed by Ramsey and Pfeffer [24], it turns out to be sufficient for the goal of providing approximations for probabilities. To keep the monadic transformation simple, we design a tiny probabilistic language $\mathcal{R}\mathsf{ml}$, equipped with a rather restricted type system, yet expressive enough for coding interesting algorithms. Program specifications are then proved along a specific inference system for axiomatic semantics.

For the proof assistant, tools are required for interactive reasoning about probabilistic programs (actually through the above transformation). We thus share Hurd's approach, while our design choices do not require full development of measure theory inside Coq. Our tools are based upon a specific library which *axiomatizes* the properties required on some abstract type U representing the real interval $[0, 1]$. This library is developed as an independent contribution [22], and designed to provide the back-end tools needed by the user.

Our axiomatic semantics enhances previous work by Morgan and McIver [20], where rules allowed only weakening for probabilities. We prove their validity with respect to our semantics. We also propose schemes to reason on general recursive functions which generalize the usual schemes for loops.

Our framework does not rely on a particular choice of a primitive randomized function. In this paper, we use a boolean flip and a finite random function and we show how to interpret directly a randomized choice operator. We only build discrete distributions: dealing with continuous distributions would require modification of the interpretation to restrict the functional to measurable functions, an extension we plan to investigate later.

1.4. Paper 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 framing the probability for a randomized program to satisfy a given property, in particular for the case of recursive programs. In Section 5, we apply our method to proofs of simple probabilistic properties of programs.

The current paper extends [1] by suggesting an interpretation of higher-order functional programs in Section 2.5.6 and introducing rules for intervals in Section 4.1 and also more general rules for reasoning on recursive functions in Section 4.4. We also develop an example of partial termination in Section 5.2.

1.5. 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, see [7], or more concretely for representing random programs in Haskell in Ramsey and Pfeffer's work [24]. To our knowledge, however, the approach of mechanizing reasoning about random functional expressions is new. In [24], the interpretation does not cover general recursive programs and its inefficiency is criticized, the authors propose instead an alternative method which only covers discrete distributions. The possibility to cover recursion was however studied in Jones's thesis [13], on which the approach of this paper is based. That the interpretation can lead to inefficient or even unfeasible computations in practice will be illustrated in Sections 2.6.1 and 2.6.2. Our work advocates that operational behavior is not relevant, since 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's second semantics in [15], and to the framework proposed in [16].

2. Monadic interpretation of randomized algorithms

Sections 2.1 and 2.2 provide background results on probabilities which underlie our framework. We present in Section 2.3 a reasonably simple probabilistic language $\mathcal{R}$ml. The monadic interpretation is the subject of Section 2.4, where we discuss also the consequences of relaxing the typing rules. We conclude this section by putting our interpretation at work on concrete examples in Section 2.6.

The approach in this part is very similar to the one proposed in Ramsey and Pfeffer [24]. The main differences are that we measure functions with values in the interval $[0, 1]$ instead of real numbers, we concentrate on a first-order language, which is sufficient for the applications we want to address, but we also show how to extend the approach for the general functional case. Unlike what is done in [24], we shall address the question of general recursive functions in Section 4.

2.1. Randomized programs as measure transformers

Usually, an imperative or functional program returns *at most* one state (or value in the functional case), from any given initial state. Moreover, the returned state is entirely determined by the program and the initial state. When dealing with probabilistic programs, this is no longer the case, even when running the program several times, starting with the same initial state. Rather, the distribution of returned states can be represented as some *random variable*, hence a measure over the states set. This change of view has been investigated in works by Kozen [15,16], Jones and Plotkin [14,13], McIver and Morgan [18] among others. Whilst the observation of the actual returned states is non-deterministic, the measure which can be built from the initial state by applying the denotation of a probabilistic program provides a deterministic value. This approach is then easily extended into randomized programs viewed as measure transformers.

The distribution of these output states is interesting. If this distribution is known, given a property P on the output 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 $\frac{1}{1+n}$, or a more basic `flip` function which produces boolean values `true` or `false` with equal probability $\frac{1}{2}$. Another classical operator is probabilistic choice $\mathtt{P}\ {}_p\!+\ \mathtt{Q}$ 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 given random primitive in the program is independent of the others.

Since we are concerned with a functional language, we do not have to take global states into consideration. Programs are interpreted as functions which compute values, and our aim is to estimate the distribution of these return 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 x such that $0 \leq x \leq 1$.

2.2.1. The measure perspective

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 $\int f \mathrm{d}\mu$. A required condition on μ to be a measure, besides linearity, is that μ preserves least upper bounds: $\int \bigvee_n f_n \mathrm{d}\mu = \bigvee_n \int f_n \mathrm{d}\mu$.

In the following, we shall use the notation $\mu(f)$ instead of $\int f \mathrm{d}\mu$.

2.2.2. Notations for 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 $\{k\}$.

2.2.3. Our abstract notion of measure

From now on in this article, probability distributions are represented as positive measures, which norm is bounded by 1.

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 on A can be interpreted as a function of type $(A \to [0,1]) \to [0,1]$ satisfying some extra algebraic properties, to be precised in Section 3.2.

2.3. Basic language for randomized programs

For sake of simplicity, we shall use in the following a simple first-order functional language. We will explain in Section 2.5.6 how it could be extended to full functional constructions.

2.3.1. Expressions

Our language (called $\mathcal{R}$ml) contains the following constructions:

- Variables: x
- Primitive constants: c
- Conditional: **if** b **then** e_1 **else** e_2
- Local binding: **let** $x = e_1$ **in** e_2
- Application: $f\ e_1 \ldots e_n$ with f a primitive or user-defined function.

We shall introduce parentheses in concrete notations when needed.

Functions can be declared the following way:

let $f\ x_1 \ldots x_n = e$

Remark. Recursive definitions can be defined as well:

let rec $f\ x_1 \ldots x_n = e$

However, their introduction raise some technical issues with respect to the material developed in this section. Therefore, we postpone any further detail to Section 3.3.

In order to deal with probabilistic programs, $\mathcal{R}$ml includes also a few random primitive functions, such as the `random` function which given a positive integer n, computes with uniform distribution an integer k such that $0 \leq k \leq n$ and the `flip` function which computes a boolean which is `true` with probability $\frac{1}{2}$.

2.3.2. Types

Our assumption on $\mathcal{R}$ml, is that all expressions will be well-formed using a restricted simple types system. This system is built over base types such as `bool` for boolean values and `nat` for natural numbers (non-negative integers), and allows arrow types in the restricted case where arguments have a base type. In the following we write $\beta, \beta_i \ldots$ in order to denote a base type. We shall write $e : \beta$ when e is a well-formed expression of type β and $f : \beta_1 \to \cdots \to \beta_n \to \beta$ when $f\ e_1 \ldots e_n : \beta$ whenever $e_i : \beta_i$ for $i = 1 \ldots n$.

2.3.3. Meta-language

Randomized expressions are interpreted in an higher-order functional language. The target type system is richer: a type τ will be either a base type β (including the base type $[0,1]$ for reals between 0 and 1) or some functional type $\tau_1 \to \tau_2$.

We use the same notations as in $\mathcal{R}$ml for local bindings and conditionals, but we also introduce typed abstraction **fun** $(x : \tau) \Rightarrow e$ and binary application $(e_1\ e_2)$. Application is left associative and types can be omitted in lambda-abstraction, written **fun** $x \Rightarrow e$, when the type is clear from the context. As a matter of fact, $\mathcal{R}$ml (except for the randomized primitive functions) corresponds to a restricted subset of our meta-language where variables are always in base types and functions are in eta-long normal form.

An alternative could have been to use a monadic meta-language as in [19] or [23], but it would have introduced an extra level of syntax that we are able to avoid here, owing to the restrictions on $\mathcal{R}$ml syntax. Doing otherwise would result in introducing more complex notations, which would have obscure the key ideas. The Section 2.5.6 develops these points further.

2.4. Interpretation of random expressions

A (random) expression e in a base type β actually represents a set of values of type β, 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 $e : \beta$ as a measure on β, i.e. a function of type $(\beta \to [0,1]) \to [0,1]$. In the following, $\mathsf{M}\tau$ will represent the type $(\tau \to [0,1]) \to [0,1]$ of measures on values of type τ.

We write $[e]$ to represent the measure associated to the expression e. If we know $[e]$, given a property Q on β, 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 e to the characteristic function of the predicate Q, interpreted as a subset of β.

2.5. Monadic transformation

The interpretation of e of type β as a measure $[e]$ of type $\mathsf{M}\beta = (\beta \to [0,1]) \to [0,1]$ is defined by structural induction on e.

2.5.1. Definition of unit and bind

As usual in monadic transformations, we first introduce two operators:

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

As expected, theses definitions satisfy the usual monadic properties. The equality on $\mathsf{M}\beta$ is defined point-wise ($\mu_1 = \mu_2 \Leftrightarrow \forall f,\ \mu_1(f) = \mu_2(f)$).

- $\mathsf{bind}\ (\mathsf{unit}\ x)\ M = M\ x$
- $\mathsf{bind}\ (\mathsf{bind}\ \mu\ M_1)\ M_2 = \mathsf{bind}\ \mu\ \ (\textbf{fun}\ x \Rightarrow \mathsf{bind}\ (M_1\ x)\ M_2)$
- $\mathsf{bind}\ \mu\ \mathsf{unit} = \mu$

2.5.2. Interpretation of functions

A function with name f and type $\tau \equiv \beta_1 \to \cdots \to \beta_n \to \beta$ will lead to a new function name $[f]$ of type $[\tau] \equiv \beta_1 \to \cdots \to \beta_n \to \mathsf{M}\beta$.

Primitive randomized functions Each primitive randomized function is given a functional interpretation of the corresponding type. In this paper, we shall use the following constructions:

$$\begin{aligned}
&[\texttt{random}]\,n : \ \texttt{Mnat} \\
&\qquad = \textbf{fun}\ (f : \texttt{nat} \to [0,1]) \Rightarrow \Sigma_{i=0}^{n} \tfrac{1}{1+n} (f\ i) \\
&[\texttt{flip}]\,() \ : \ \texttt{Mbool} \\
&\qquad = \textbf{fun}\ (f : \texttt{bool} \to [0,1]) \Rightarrow \tfrac{1}{2}(f\ \texttt{true}) + \tfrac{1}{2}(f\ \texttt{false})
\end{aligned}$$

It is also possible to start from other primitive notions of randomness, like the random choice operator used in [24]:

$$\begin{aligned}
&e_1\ {}_p{+}\ e_2 : \ \mathsf{M}\beta \\
&\qquad = \textbf{fun}\ (f : \beta \to [0,1]) \Rightarrow p \times ([e_1]f) + (1-p) \times ([e_2]f)
\end{aligned}$$

User defined functions For a (non-recursive) user-defined function introduced by $\textbf{let}\ f\ x_1 \ldots x_n = e$, the interpretation $[f]$ will be introduced by $\textbf{let}\ [f]\,x_1 \ldots x_n = [e]$. This turn $[f]$ into a function with type $\beta_1 \to \cdots \beta_n \to \mathsf{M}\beta$, belonging to the target language.

Shortcut More generally, when f is any $\mathcal{R}\mathsf{ml}$ function of type $\beta_1 \to \cdots \beta_n \to \beta$, when $x_1, \ldots, x_n$ are terms of the meta-language such that x_i has type β_i (for each $1 \le i \le n$) and $g : \beta \to [0,1]$ is any function, we allow ourselves to write $[f\ x_1 \ldots x_n]g$ (instead of $([f]\,x_1 \ldots x_n)g$) for the expectation of g by the measure $[f]\,x_1 \ldots x_n$.
We also abusively use the same notation $[\phi\ x_1 \ldots x_n]g$ (instead of $(\phi\ x_1 \ldots x_n)g$) when ϕ is some function of type $\beta_1 \to \cdots \beta_n \to \mathsf{M}\beta$ defined in the meta-language, in order to emphasize the fact that we are computing the expectation of g with respect to the measure $(\phi\ x_1 \ldots x_n)$.

Recursively defined functions When dealing with $\textbf{let rec}\ f\ x_1 \ldots x_n = e$, we define as well $[f]$ as a new recursively defined function in the target language, introduced by $\textbf{let rec}\ [f]\,x_1 \ldots x_n = [e]$. However, this is not as simple, in spite of being quite the same from the sole syntactic point of view. We address this issue more deeply in Section 3.3.

2.5.3. Interpretation of expressions

Computation $a : \beta$	Functional value $[a] : \mathsf{M}\beta$
v	$\mathsf{unit}\ v$ $\quad v$ variable or constant
let $x = a$ **in** b	$\mathsf{bind}\ [a]\ (\mathbf{fun}\ x \Rightarrow [b])$
$f\ a_1 \ldots a_n$	$\mathsf{bind}\ [a_1]\ (\mathbf{fun}\ x_1 \Rightarrow \ \ldots \mathsf{bind}\ [a_n]\ (\mathbf{fun}\ x_n \Rightarrow [f]\, x_1 \ldots x_n) \ldots)$
if b **then** a_1 **else** a_2	$\mathsf{bind}\ [b]\ (\mathbf{fun}\ (x : \mathtt{bool}) \Rightarrow \mathbf{if}\ x\ \mathbf{then}\ [a_1]\ \mathbf{else}\ [a_2])$

2.5.4. Properties of the interpretation

It is easy to prove that our interpretation is well-typed:

Proposition 1. *Given an expression e in $\mathcal{R}$ml of (base) type β, $[e]$ is defined and has type $\mathsf{M}\beta$*

Proof. We prove a more precise result by a simple induction on the expression e: assume e has type β, contains the finite set of free variables $(x_i)_i$, where for each i, x_i has type β_i, and make calls to some finite set $(f_j)_j$ of functions. Then $[e]$ has type $\mathsf{M}\beta$ in an environment containing the same variables $(x_i)_i$ of same type $(\beta_i)_i$ and contains the corresponding functions symbols $([f_j])_j$, such that $[f_j]$ has now type $[\tau_j]$ when f_j has type τ_j. □

If random primitives are left aside in a term e then it is possible to simplify the translation $[e]$:

Proposition 2. *Let e be a pure expression of base type β in $\mathcal{R}$ml, i.e. an expression in which no randomized construction occurs, then e can be translated in our meta-language into a term of type β (still written e) and $[e] = \mathsf{unit}\, e$.*

Proof. The proof is by induction on the structure of terms not involving randomized constructions. The translation uses the meta-language abstraction, application and local definition for the interpretation of the corresponding $\mathcal{R}$ml constructions. The equality $[e] = \mathsf{unit}\, e$ is a consequence of the monadic properties of unit and bind. □

2.5.5. On the meaning of the interpretation

Let us have a look at this interpretation from the measure theory point of view,

- The monad operator $\mathsf{unit}\, x$ represents the Dirac measure δ_x at point x. If $x : \beta$ and $\theta : \beta \to [0, 1]$, then
$$[x]\theta \ = \mathsf{unit}\, x\, \theta = \theta(x) = \int \theta(y) \mathrm{d}\delta_x(y)$$
- Given μ a measure on $\mathsf{M}\alpha$ and $\mathbf{fun}\ x \Rightarrow e$ of type $\alpha \to \mathsf{M}\beta$ a family of measures on β parametrized with $x \in \alpha$, the measure $\mathsf{bind}\ \mu\ (\mathbf{fun}\ x \Rightarrow e)$ is defined as
$$\mathsf{bind}\ \mu\ (\mathbf{fun}\ x \Rightarrow e)\ \theta \ = \int \left(\int \theta(y) \mathrm{d}e(y) \right) \mathrm{d}\mu(x)$$
In particular,
$$[\mathbf{let}\ x = a\ \mathbf{in}\ b]\theta \ = \int \left(\int \theta(y) \mathrm{d}[b](y) \right) \mathrm{d}[a](x)$$
in such a way that both **let** and summation constructs *bind* the variable x.
- The measure associated to the conditional $e = \mathbf{if}\ e_0\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2$ behaves as expected:
$$\begin{aligned}[e]\theta \ &= \int \left(\int_{b=\mathtt{true}} \theta(y) \mathrm{d}[e_1](y) + \int_{b=\mathtt{false}} \theta(y) \mathrm{d}[e_2](y) \right) \mathrm{d}[e_0](b) \\ &= ([e_0]\mathbb{I}_{b=\mathtt{true}}) \int \theta(y) \mathrm{d}[e_1](y) + ([e_0]\mathbb{I}_{b=\mathtt{false}}) \int \theta(y) \mathrm{d}[e_2](y)\end{aligned}$$
as the variable b occurs neither in e_1 nor in e_2.
- Accordingly, the application $e = (f\ a_1 \ldots a_n)$ corresponds to a multiple summation
$$[e]\theta \ = \int \left(\cdots \int \left(\int \theta(y) \mathrm{d}[f\ x_1 \ldots x_n](y) \right) \mathrm{d}[a_n](x_n) \ldots \right) \mathrm{d}[a_1](x_1)$$
- The definitions for random primitives such as `flip` and `random`n involve actually finite summations, as already presented in Section 2.5.2. The general summation symbol includes obviously the particular case of finite and denumerable ones.

2.5.6. A general higher-order interpretation

The basic term language presented in Section 2.3 will turn out to be sufficient for dealing with interesting examples. Its main restriction however results from its proper design: we do not take into account programs which could generate randomized functions. For instance, the program Φ defined as

```
let Φ = let x = random 100 in fun (n:nat) ⇒ let y = random n in y+x
```

provides a random variable on type $\mathtt{nat} \to \mathtt{nat}$. As such, one would expect its monadic interpretation to be given over some type expression $[\![\mathtt{nat} \to \mathtt{nat}]\!] \equiv \mathsf{M}\rho \equiv (\rho \to [0,1]) \to [0,1]$, where ρ is some type, which is described more precisely below.

This leads to the general high-order interpretation, based upon the fact that $[\![\beta]\!] = \mathsf{M}\beta$ for any base type β, and the observation that variables (as well as abstractions) are expected to be considered *values* with respect to our interpretation. Given an arbitrary expression $e : \sigma$, we still want to turn e into a measure on some type expression $\bar{\sigma}$, i.e. a function of type $[\![\sigma]\!] = \mathsf{M}\bar{\sigma}$. The transformation turns out to be the well known Plotkin's *Call-by-Value* transformation, with $\bar{\beta} \equiv \beta$ and $\overline{\sigma \to \tau} \equiv \bar{\sigma} \to \mathsf{M}\bar{\tau}$.

As for $e : \sigma$, we define $[\![e]\!] : [\![\sigma]\!]$ such that:

Term $e : \tau$	Interpretation $[\![e]\!] : [\![\tau]\!]$
x	unit x
fun $(x : \sigma) \Rightarrow t$	unit (**fun** $(x : \bar{\sigma}) \Rightarrow [\![t]\!]$)
let $x = a$ **in** b	bind $[\![a]\!]$ (**fun** $(x : \bar{\sigma}) \Rightarrow [\![b]\!]$)
$t\ u$	bind $[\![t]\!]$ (bind $[\![u]\!]$)
if b **then** e_1 **else** e_2	bind $[\![b]\!]$ (**fun** $(x : \mathsf{bool}) \Rightarrow$ **if** x **then** $[\![e_1]\!]$ **else** $[\![e_2]\!]$)

In other words, turning our former monadic transformation (Section 2.3) over $\mathcal{R}$ml into a more general transformation amounts at applying CPS transformations to our programs, where measurable functions $f : \beta \to [0,1]$ are now seen as particular cases of *continuations*. This interpretation extends this situation by interpreting random primitives such as `flip()` (resp. `random n`) as a *genuine* inhabitant in `Mbool` (resp. `Mnat`) as shown in Section 2.5.2.

It can be shown that this interpretation is a conservative extension from the former monadic one. They compare, when we restrict ourselves to the $\mathcal{R}$ml case:

Proposition 3. *Assume p is a well formed term from $\mathcal{R}$ml. If $p : \beta$, where β is a base type, then $[p] = [\![p]\!]$ as elements of $M\beta$.*

In this paper, randomized functions such as Φ cannot be considered since the type system chosen in this work does not allow for building measures on functional types.

2.6. Examples of functional interpretation

Now that the monadic translation is defined, we can transform an expression e which computes a value randomly into an deterministic expression $[e]$ which returns the measure associated with the expression e. Before looking at this interpretation in the prospect of proving facts over some program e, notice that $[e]$ is an ordinary functional term, and can be evaluated as such in the interactive main loop of, say, O'Caml.

2.6.1. Primality test

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 (`test`) which applies to $1 \leq k < p$ and p such that:

- If p is prime then (`test` $k\ p$) evaluates to `true` for all k
- If p is not prime then (`test` $k\ p$) evaluates to `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, the function which iterates n times the primality test for p can be written:

```
let rec prime_test p n =
   if n = 0 then true
   else let k = random (p-2) in
        if test (k+1) p then prime_test p (n-1) else false
```

Using the monadic transformation, and monad 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_fun (p-2))
         fun k ⇒ if test (k+1) p 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 `prime_correct`, the characteristic function of the correctness predicate, which says that the result is true exactly when p is prime:

```
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 O'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 (except for the interpretation of `random`). On the other hand, if we analyze the program, we remark that it is very inefficient:

- 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.

2.6.2. Random walk

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 x else walk (x+1)
```

If we test this function in O'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
     (fun (b:bool) ⇒ if b then unit x else walk_fun (x+1))
```

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 false.

This example shows that, when general fix-points are involved, we cannot anymore use computation of the monadic interpretation 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

The monadic interpretation transforms a probabilistic term e of type β into a purely functional one, $[e]$ which is understood as a measure on this same type. Our next step towards reasoning on these randomized terms consists in providing tools on proof assistant Coq side to reason on *e through* its interpretation $[e]$. As a matter of consequence, we develop tools to reason on measures instead. The Section 3.1 presents an axiomatization U of the unit interval $[0, 1]$, sufficient for the purpose, and representation for types and terms from $\mathcal{R}$ml is explained in Sections 3.2 and 3.3.

3.1. U: An axiomatization of 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]. The complete development is available as a Coq contribution (see http://coq.inria.fr) [1]

3.1.1. Notations for complete partial orders

Our development extensively uses the notion of complete partial order. Our Coq library consequently starts with the definition of a structure for ordered sets, and one for complete partial orders.

An ordered set is given by a type O, a relation $\leq$ which is reflexive and transitive. An equality on O is defined by $x == y$ iff $x \leq y \wedge y \leq x$. Given two ordered sets O_1 and O_2, we introduce the type of monotonic functions $O_1 \xrightarrow{m} O_2$.

A ω-complete partial order (ω-cpo) is given by an ordered set D, a minimal element 0 and a least-upper bound operation $\mathtt{lub}\, f$ on monotonic sequences $f : \mathtt{nat} \xrightarrow{m} D$. Given two ω-cpos D_1 and D_2, a monotonic function $F : D_1 \xrightarrow{m} D_2$ is defined to be continuous whenever $F\ (\mathtt{lub}\, f) \leq \mathtt{lub}(F \circ f)$. Because the opposite inequality is always provable, a continuous function also satisfies $F\ (\mathtt{lub}\, f) == \mathtt{lub}(F \circ f)$.

There is a standard way to introduce fix-points in an ω-cpo D. Let F be a monotonic operator on D (i.e. $F : D \xrightarrow{m} D$), we introduce the sequence F_n defined by $F_n \equiv F^n 0$ (with $F^{n+1} = F \circ F^n$) and define **fix** $F = \mathtt{lub}\, F_n$.

It is easy to show that **fix** $F \leq F$ (**fix** F), the equality **fix** $F == F$ (**fix** F) requires that F is continuous.

The ω-cpo structure can be extended to functions spaces. If we have an ω-cpo structure on a set D, then we can define the same structure on the set $A \to D$ of functions with values in D, just taking:

$$f \leq_{A\to D} g \Leftrightarrow \forall x, f\, x \leq_D g\, x$$
$$0_{A\to D} = \mathbf{fun}\, x \Rightarrow 0_D \qquad \mathtt{lub}_{A\to D} f_n = \mathbf{fun}\, x \Rightarrow \mathtt{lub}_D(f_n\, x)$$

Given an ordered set O and an ω-cpo D, the set of monotonic functions from O to D is also an ω-cpo.

3.1.2. Definitions

Our axiomatization of [0, 1], starts by introducing an ω-cpo U. Consequently, we can use the following symbols:

- Constant: 0
- Predicates: $x \leq y, x == y$ with $x, y \in U$
- Least-upper bounds for monotonic sequences: $\mathtt{lub}\, f$ with $f \in \mathtt{nat} \xrightarrow{m} U$.
  If f is an expression with a free variable n, we write $\mathtt{lub}(f)_n$ instead of $\mathtt{lub}\ (\mathbf{fun}\ n \Rightarrow f)$.

We also introduce the following constructions building new elements in U:

- bounded addition: $x + y$ with $x, y \in U$
- multiplication: $x \times y$ with $x, y \in U$
- inverse: $1 - x$ with $x \in U$
- values: $\frac{1}{1+n}$ with $n : \mathtt{nat}$

The addition in U is bounded: it gives the minimum of addition on reals and 1.

3.1.3. Axioms

In addition to the ω-cpo properties, we introduce a set of axioms for the operations on U.

3.1.3.1. Order. We assume that 1 is different from 0 and not less than any element in U and that the order is total:

- Non-confusion: $\neg 0 == 1$
- Bounds: $\forall x, x \leq 1$
- Totality: $\forall xy, x \leq y \vee_c y \leq x$

Coq implements an intuitionistic logic, we did not want to commit ourselves to a classical axiomatization of real numbers. Consequently, we choose a classical version of disjunction for expressing the totality: the property $A \vee_c B$ is defined as $\forall C, (\neg\neg C \to C) \to (A \to C) \to (B \to C) \to C$ and we added an axiom stating that the order relation is classical:

- Classical: $\neg\neg(x \leq y) \to x \leq y$

[1] Our development currently runs with Coq V8.1.

3.1.3.2. Addition, multiplication and inverse. As expected, we include the usual axioms stating that addition and multiplication are symmetric and associative, with 0 and 1 as their respective neutral elements.

Some properties of addition are only valid when there is no overflow during addition. The non-overflow condition is expressed in our formalism as $x \leq 1-y$.

We express the relationship between least upper bounds (lubs) and addition and multiplication by the assumption of continuity of addition and multiplication with respect to their second argument.

The complete set of axioms is:

- Addition
  - . Symmetry: $\forall x\, y, x + y == y + x$
  - . Associativity: $\forall x\, y\, z, x + (y + z) == (x + y) + z$
  - . Neutral element: $\forall x, 0 + x == x$
  - . Compatibility: $\forall x\, y\, z, y \leq z \Rightarrow x + y \leq x + z$
  - . Simplification: $\forall x\, y\, z, z \leq 1 - x \Rightarrow x + z \leq y + z \Rightarrow x \leq y$
  - . $\mathtt{lub}$ and addition: $\forall (f : \mathtt{nat} \xrightarrow{m} U)\, k, k + \mathtt{lub} f \leq \mathtt{lub}(k + f\, n)_n$
- Multiplication
  - . Symmetry: $\forall x\, y, x \times y == y \times x$
  - . Associativity: $\forall x\, y\, z, x \times (y \times z) == (x \times y) \times z$
  - . Neutral element: $\forall x, 1 \times x == x$
  - . Distributivity on addition: $\forall x\, y\, z, x \leq 1 - y \Rightarrow (x + y) \times z == x \times z + y \times z$
  - . Compatibility: $\forall x\, y\, z, y \leq z \Rightarrow x \times y \leq x \times z$
  - . Simplification: $\forall x\, y\, z, \neg 0 == z \Rightarrow z \times x \leq z \times y \Rightarrow x \leq y$
  - . $\mathtt{lub}$ and multiplication: $\forall (f : \mathtt{nat} \xrightarrow{m} U)\, k, k \times \mathtt{lub} f \leq \mathtt{lub}(k \times f\, n)_n$
- Inverse
  - . Inverse maps 1 to 0 : $1 - 1 == 0$
  - . Inverse property: $\forall x, (1 - x) + x == 1$
  - . Compatibility: $\forall x\, y, x \leq y \Rightarrow 1 - y \leq 1 - x$
  - . Inverse and addition: $\forall x\, y, y \leq 1 - x \Rightarrow (1 - (x + y)) + x == 1 - y$
  - . Inverse and multiplication: $\forall x\, y, 1 - (x \times y) == (1 - x) \times y + 1 - y$

3.1.3.3. Constant $\frac{1}{1+n}$. The constant $\frac{1}{1+n}$ satisfies the axiom:

- $\frac{1}{1+n} == 1-(n \times \frac{1}{1+n})$
  where $n \times \frac{1}{1+n}$ is a generalized sum defined by induction on n.

Finally, the fact that U is Archimedean is axiomatized by the property

- $\forall x, \neg x == 0 \Rightarrow \exists_c n, \frac{1}{1+n} \leq x$

As for the total order property, we use a classical version of existential.

3.1.4. 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 few 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, and we plan to investigate a possible encoding using the reals defined by Geuvers and Niqui [5] or the axioms proposed for interval objects as described by Escardó and Simpson [2]. We use the functor mechanism of Coq in order to keep the axiomatization of [0, 1] as a parameter of the theory.

3.1.5. Derived operations

The usual minus operation $x - y$ (which is zero when $x \leq y$) can be defined using our special inverse by: $x - y \equiv 1-((1-x) + y)$ The operation $\mathtt{max}$ can be defined as $(x - y) + y$. Using the $\mathtt{max}$ operation, we can define the least-upper bound of an arbitrary sequence. The greatest lower bound can be defined by $\mathtt{glb} f \equiv 1-\mathtt{lub}(1-f)$. It is also easy to define $n \times x$ and x^n for an integer n by induction on n. Morgan and McIver [20] use an operation $x \,\&\, y$ defined on non-negative real numbers as the maximum of 0 and $x+y-1$. The same operation can be defined in our theory using the inverse operation 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 operation captures intersection of properties because $\mathbb{I}_{P \cap Q} == \mathbb{I}_P \,\&\, \mathbb{I}_Q$ and will be used in fix-point rules in Section 4.4.2.

Altogether, the Coq theory for [0, 1] contains approximately 1100 lines of definitions and lemmas (and almost twice as many lines of proofs).

3.2. Dealing with $\mathcal{R}$ml in Coq

Given $e : \beta$, we get $[e] : \mathsf{M}\beta = (\beta \to [0,1]) \to [0,1]$. The type $\mathsf{M}\beta$ is first represented in Coq as some record type $(\texttt{distr}\ \beta)$ which captures functionals in $\mathsf{M}\beta$ with good measure properties.

3.2.1. Representation of types

In the following, we extend in a standard way the operations on U, to operations and relations on functions of type $\beta \to U$ using the same notations: $f + g$ is the function **fun** $x \Rightarrow f\,x + g\,x$ and $k \times f$ is the function **fun** $x \Rightarrow k \times f\,x$.

Given a type β, we define a distribution on β to be a monotonic function μ of type $(\beta \to U) \xrightarrow{m} U$ which furthermore satisfies stability properties, namely:

- linearity :
  - $\forall f\ g : \beta \to U, f \le 1 - g \Rightarrow \mu(f+g) == \mu(f) + \mu(g)$
  - $\forall (k : U)(f : \beta \to U), \mu(k \times f) == k \times \mu(f)$
- compatibility with inverse : $\forall f : \beta \to U, \mu(1-f) \le 1 - \mu(f)$
- continuity : $\forall f : \texttt{nat} \xrightarrow{m} (\beta \to U), \mu(\texttt{lub}\,f) \le \texttt{lub}\,(\mu \circ f)$

In Coq, we introduce a type $(\texttt{distr}\ \beta)$ as a dependent record which contains the measure μ plus the proofs of compatibility properties for μ.
There is a natural order on that type inherited from the functional order on $(\beta \to U) \to U$.

Formally in the Coq development, there is a difference between the type $\mathsf{M}\beta$ of functionals and the type $(\texttt{distr}\ \beta)$ which contains the functional of type $\mathsf{M}\beta$ plus the proofs of stability properties. However, for the sake of readability we shall not emphasize this distinction in this paper and use simply the type $\mathsf{M}\beta$ in place of $(\texttt{distr}\ \beta)$ assuming all the objects in that type satisfy the requested stability properties.

3.2.2. Remarks

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.

The definition and properties in Coq of a measure on a type β is done for an arbitrary Coq type and not just base types coming from the $\mathcal{R}$ml interpretation.

3.2.3. Derived properties

From this definition, we can deduce further properties, such as

- $\mu(\textbf{fun}\ x \Rightarrow 0) == 0$,
- $\mu(1-f) == \mu(\mathbb{I}) - \mu(f)$,
- $\forall fg, \mu(f+g) \le \mu(f) + \mu(g)$ (even when there is an overflow),
- $\forall fg, \mu(f)\ \&\ \mu(g) \le \mu(f\ \&\ g)$.

3.2.4. Representation for $\mathcal{R}$ml terms

We easily check that the monadic operators unit and bind introduced in 2.5 satisfy the stability properties of measures given in Section 3.2.1. This is also the case for the primitive random constructions introduced in Section 2.5.2: [random] and [flip] or the choice operator P $_p$+ Q.

With the help of these operators, we can represent our $\mathcal{R}$ml terms. For example, following our general monadic translation scheme, one can also define a conditional operation Mif of type $\mathsf{M}\texttt{bool} \to \mathsf{M}\beta \to \mathsf{M}\beta \to \mathsf{M}\beta$:

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

We use this operator for interpreting conditional programs:

$$[\texttt{if}\ b\ \texttt{then}\ e_1\ \texttt{else}\ e_2] \equiv \texttt{Mif}\ [b]\ [e_1]\ [e_2]$$

3.2.5. Properties

We prove the monotonicity of the bind operation. Assuming $\mu_1, \mu_2 : \mathsf{M}\alpha, M_1, M_2 : \alpha \to \mathsf{M}\beta$:

$$\frac{\mu_1 \le \mu_2 \quad M_1 \le M_2}{\mathsf{bind}\ \mu_1\ M_1 \le \mathsf{bind}\ \mu_2\ M_2}$$

3.3. Managing recursive definitions

As expected, the difficult part is the interpretation of general fix-points. We distinguish two cases, one where termination is total, like in the case of primality testing, in which case we can use the fix-point constructions of Coq in order to interpret the recursively defined distribution and the general case, like in the example of the Random walk, where we use a limit construction.

3.3.1. Total recursive functions

We assume the function f is recursively defined in $\mathcal{R}$ml and has type $\beta_1 \to \cdots \to \beta_n \to \beta$.

$$\textbf{let rec}\, f\ x_1 \ldots x_n = e$$

A natural idea in order to interpret f in Coq as a function $[f]$ defining a measure of type $\beta_1 \to \cdots \to \beta_n \to \mathsf{M}\beta$, would be to use the same recursive definition in Coq:

$$\textbf{let rec}\, [f]\ x_1 \ldots x_n = [e]$$

However, this is not always possible in Coq. The prover accepts a recursive definition for f when there is an argument x_i of type β_i with β_i an inductive type and all recursive calls $(f\ a_1 \ldots a_n)$ in the body e are such that a_i is a value structurally smaller than x_i.

If the definition of f in $\mathcal{R}$ml satisfies this criteria (for one of its arguments) and if the structurally smaller elements a_i do not contain randomized constructions, then this is also the case of recursive calls to $[f]$ in $[e]$ and the recursive definition of $[f]$ in Coq will be valid. The function `prime_test` in Section 2.6 gives an example of this case: it is a structural recursion on the variable n.

Another important case of recursive definitions in Coq is the case of well-founded recursive definitions. We assume given a relation $\prec$ on one of the arguments x_i of type β_i which is proved to be well-founded and such that all recursive calls $(f\ a_1 \ldots a_n)$ in the body e are such that a_i is a non-randomized construction and $a_i \prec x_i$ is provable. Such that the Coq definition of $[f]$ using well-founded recursion is also valid.

3.3.2. Limit of distributions

In order to interpret recursive functions in which recursive calls are not obviously terminating as in the previous cases, we need to take limits of sequences of distributions.

As mentioned in Section 3.1.1, there is a ω-cpo structure on the functional type $\mathsf{M}\beta = (\beta \to [0,1]) \xrightarrow{m} [0,1]$, it is not difficult to show that the least-upper bound operation preserves the measure stability properties, such that the set $\mathtt{distr}\ \beta$ is also an ω-cpo.

3.3.3. Fix-points

For the sake of clarity, this explanation is restricted to unary recursive definitions; the n-ary case is handled similarly. Let us consider we want to define a function which satisfies the equation

$$\textbf{let rec}\, f\ x = e$$

where f is assumed to take an argument in type α, and returns a random value of type β, such that it has type $\alpha \to \beta$ and $[f]$ will have type $\alpha \to \mathsf{M}\beta$. We introduce F of type $(\alpha \to \mathsf{M}\beta) \to \alpha \to \mathsf{M}\beta$ defined by $(\textbf{fun}\ [f] \Rightarrow \textbf{fun}\ x \Rightarrow [e])$. We assume F to be monotonic: $h \leq g \Rightarrow F\,h \leq F\,g$. Using the ω-cpo structure on $\alpha \to \mathsf{M}\beta$, we construct the fix-point $\textbf{fix}\,F$ of type $\alpha \to \mathsf{M}\beta$, this function will be our interpretation of f.

As mentioned in Section 3.1.1, the inequality $\textbf{fix}\,F\,x \leq F\ (\textbf{fix}\,F)\,x$ holds. The equality is only provable when F is continuous.

We have proven lemmas stating that the bind operation seen as a monotonic function of type $\mathtt{distr}\ A \xrightarrow{m} (A \to \mathtt{distr}\ B) \xrightarrow{m} \mathtt{distr}\ B$ is continuous. We have also that the fixpoint operation seen as a monotonic function from $D \xrightarrow{c} D$ to D is continuous with $D \xrightarrow{c} D$ the set of continuous functions from D to D. We can deduce (as a meta-theorem that we did not formalize) that functionals generated from $\mathcal{R}$ml expressions will satisfy the continuity hypothesis.

To summarize this section, when a recursive function is introduced in $\mathcal{R}$ml using the declaration:

$$\textbf{let rec}\, f\ x = e$$

we interpret it as a function $[f]$ defined in our meta-language by

$$\textbf{let rec}\, [f]\ x = \textbf{fix}\ (\textbf{fun}\ [f] \xrightarrow{m} \textbf{fun}\ x \Rightarrow [e])\ x$$

We will explain in the next section how to prove properties of such programs.

4. Derived rules for reasoning on programs

As far as fix-points are concerned, well founded recursive definitions are dealt with as usual in Coq, and need no further development in this article. In this section, we develop an extended axiomatic semantics for $\mathcal{R}$ml programs (Section 4.1), with some particular attention to general recursive definitions. Actually, the very novelty when considering some probabilistic program e is the fact that e may not terminate on every initial state, but rather *terminates almost surely*, which is is a weaker property. From the operational point of view, this property expresses that e will terminate *eventually*. This is developed further in Section 4.4.

4.1. Extending Kozen's minoring derivation rules

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 [16]) in deriving some *information* on the probability for a certain property to hold. Given $e : \beta$, its monadic interpretation $[e] : \mathsf{M}\beta$ is meant to represent a measure on β, which computes for a function $f : \beta \to [0, 1]$, its expectation $[e]f \in [0, 1]$. (Usually f will be the characteristic function $\mathbb{I}_P$ of some predicate P of type $\beta \to$ bool, in which case $[e]\mathbb{I}_P$ computes the probability for the property P.)

The expression $[e]f$ computes the exact expectation, while in general it would be easier to reason on approximation of this value that will be given by a possible interval of values.

Obviously, $0 \leqslant [e]f \leqslant 1$ is the worst surrounding we can get for this expectation. Whenever $[e]\mathbb{I} = 1$, we understand that $[e]$ is a probability, which also means that *e terminates almost surely*. On the contrary, the obvious meaning of $[e]\mathbb{I} = 0$ is that *e diverges almost surely*. Besides these particular cases, we expect to derive $a \leqslant [e]f \leqslant b$ framings, where $a \leqslant b \in [0, 1]$, that is to say $[e]f \in [a, b]$. Therefore, our precondition is going to be some interval $I \subseteq [0, 1]$.

Post-conditions should be similar, but expected to depend on the value returned from the computation of e, since we are dealing with functional programs. Thus, post-conditions are taken to be *interval-valued* functions F, such that $\forall x : A, F\,x \subseteq [0, 1]$.

As a matter of consequence, we provide rules for deriving judgments of the form $[e]F \subseteq I$, which extends Kozen's $k \leq [e]f$ rules (where $k \in [0, 1]$, e is an expression of type β and f is a function of type $\beta \to [0, 1]$) in a consistent way:

The minoration $k \leq [e]f$ is rewritten as $k \leqslant [e]f \wedge [e]\mathbb{I} \leqslant 1$, owing to the fact the interpretation $[e]$ is *monotonic*.

Before going through the details, let us notice that this presentation could have been settled in the usual Scott's domains framework [25], where the set $\mathcal{I}$ of intervals included in $[0, 1]$ is turned into an ω-cpo, with ordering the converse of inclusion, $[0, 1]$ as bottom element and intersection as the least upper-bound operation. As a matter of fact, if we do not restrict ourselves to the unit interval, this is Scott's Interval Domain, which is the interpretation for abstract data type $\mathbb{R}$ in his model for functional programming. We do not need to deal with the full presentation for our purpose, but for two important points. First of, maximal elements of the Interval Domain are singleton sets $\{r\} \equiv [r, r]$, where $r \in \mathbb{R}$. In our framework, maximal elements are the same, restricted to $r \in [0, 1]$, and are associated (obviously) to equality proofs. In other words, maximal interval matches the best information we can derive for some probability, while $[0, 1]$ matches the worst, useless information. Secondly, we have to cope with recursive definitions, in which case we shall need monotonic interval sequences $(I_n)_n$ such that for all n, $I_{n+1} \subseteq I_n$. Then, the least upper bound $\cap_n I_n$ is well defined. This is going to be sufficient in this setting.

4.2. Definition on intervals

An interval I is given by its lower bound $\mathtt{low}\,I$ and its upper bound $\mathtt{up}\,I$ such that $0 \leq \mathtt{low}\,I \leq \mathtt{up}\,I \leq 1$, and we write it $[\mathtt{low}\,I, \mathtt{up}\,I]$, we use the notation $\{r\}$ for the singleton interval $[r, r]$. We write $\mathcal{I}$ the set of intervals.

We have the expected definition on membership and inclusion:

- $x \in [a, b]$ is defined as $a \leq x \leq b$
- $[a, b] \subseteq [c, d]$ is defined as $c \leq a \wedge b \leq d$.

Operations on intervals can be lifted to interval functions. For an interval function F, we write $\mathtt{low}\,F$ for the function **fun** $x \Rightarrow \mathtt{low}\,(F\,x)$ and similarly $\mathtt{up}\,F$ for the function **fun** $x \Rightarrow \mathtt{up}\,(F\,x)$.

The operation of a distribution e on A on an interval function F on A is written $[e]$F, it is an interval defined by $[[e](\mathtt{low}\,F), [e](\mathtt{up}\,F)]$. Given two functions f and g of type $\beta \to [0, 1]$, we shall write $[f, g]$ for the interval function **fun** $x \Rightarrow [f\,x, g\,x]$ and $\{f\}$ for the singleton function $[f, f]$.

Because of the monotonicity of distributions, it is easy to show that for a function f in $\beta \to [0, 1]$, if for all x, $f\,x$ belongs to the interval $F\,x$, then $[e]f \in [e]F$. We have also that $[e]\{f\} = \{[e]f\}$ such that nothing is lost when considering intervals.

We also extend operations of addition and multiplication to intervals:

- $[a_1, b_1] + [a_2, b_2] = [a_1 + a_2, b_1 + b_2]$
- $k \times [a, b] = [k \times a, k \times b]$

4.3. Basic (non-recursive) rules

From now on, $I, J, K \subseteq [0, 1]$ stand for intervals and F, G, H for interval-valued functions. We derive proofs for $[e]F \subseteq I$ along the following cases.

Representation of intervals in Coq is done with no additional effort. The interpretation of $\mathcal{R}$ml terms however need now being reconsidered as acting on interval-valued functions instead of simple functions. This is straightforward along the following points:

- $[v]G = G\,v$ when v is a variable, a constant or a non-randomized term
- $[\textbf{let}\ x = a\ \textbf{in}\ e]G = [a]\ (\textbf{fun}\ x \Rightarrow [e]G)$
- $[\textbf{if}\ e_0\ \textbf{then}\ e_1\ \textbf{else}\ e_2]G = [e_0]\mathbb{I}_{.=\texttt{true}} \times [e_1]G + [e_0]\mathbb{I}_{.=\texttt{false}} \times [e_2]G$

The functions [random] and [flip] associated to the primitive randomized constructions also operate on intervals functions like on real functions.

- $[\texttt{random}\,n]G = \Sigma_{i=0}^{n} \frac{1}{1+n}(G\,i)$
- $[\texttt{flip}\,()]G = \frac{1}{2}(G\,\texttt{true}) + \frac{1}{2}(G\,\texttt{false})$

From these equalities, we can derive the following rules:

$$\frac{G_2 \subseteq G_1 \quad [e]G_1 \subseteq I_1 \quad I_1 \subseteq I_2}{[e]G_2 \subseteq I_2}$$

$$\frac{[a]F \subseteq I \qquad \forall x, [e]G \subseteq F\,x}{[\textbf{let}\ x = a\ \textbf{in}\ e]G \subseteq I}$$

$$\frac{[e_1]G \subseteq I_1 \quad [e_2]G \subseteq I_2}{[\textbf{if}\ e_0\ \textbf{then}\ e_1\ \textbf{else}\ e_2]G \subseteq [e_0]\mathbb{I}_{.=\texttt{true}} \times I_1 + [e_0]\mathbb{I}_{.=\texttt{false}} \times I_2}$$

We can derive in our formalism useful schemes which generalize reasoning on deterministic programs. For instance, if we have established that 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:

$$\frac{[a]\mathbb{I}_P = 1 \qquad \forall x, P\,x \Rightarrow [e]F \subseteq I}{[\textbf{let}\ x = a\ \textbf{in}\ e]F \subseteq I}$$

4.4. Rules for fix-points

In that part, we use the same notations as in Section 3.3.3. We want to prove properties of a recursive definition in $\mathcal{R}$ml: $\textbf{let rec}\ f\ x = e$ with x of type α, and e of type β. We introduce F a monotonic operator of type $(\alpha \to \mathsf{M}\beta) \xrightarrow{m} \alpha \to \mathsf{M}\beta$ as in 3.3.3 such that $[f] = \textbf{fix}\,F$.

We also introduce the notation $f \cdot G$ when f has type $\alpha \to \mathsf{M}\beta$ and G has type $\alpha \to \beta \to \mathbb{1}$. The expression $f \cdot G$ will denote a function of type $\alpha \to \mathbb{1}$ defined by $(f \cdot G)\,x$ is the value $[f\,x](G\,x)$ of the measure $(f\,x)$ on the function $(G\,x)$.

We allow ourselves to use the same notation when g is a real-valued function of type $\alpha \to \beta \to [0, 1]$, in which case $f \cdot g$ will be a function of type $\alpha \to [0, 1]$.

The function g plays the role of an input–output relation: given a binary relation R on α and β then we can take g of type $\alpha \to \beta \to [0, 1]$ to be the characteristic function of R, in that case $f \cdot g$ corresponds to the function which associates to x the probability of $R(x, f\,x)$.

4.4.1. Basic estimation

We now justify the rule for estimating fix-points which agrees and extends the ideas presented by Jones [13]. Let us give the general idea in the first place. The $\mathcal{R}$ml definition $\textbf{let rec}\ f\ x = e$ for f can also be considered as the fix-point of some functional F such that $[f]\,x = \textbf{fix}\,F\,x$.

Given the interval-valued function G, we want to estimate $[f\,x]G$, so to find I such that $[f\,x]G \subseteq I$. The maximal interval $I = [0, 1]$ is a trivial solution. Now the fix-point is the result of the iteration of the functional F, so if it is possible to decrease the interval at each step, we can deduce an approximation for f.

This leads to the following provable rule, assuming a given monotonic sequence $(I_n)_n$ of interval-valued functions on type α such that: $\forall x, 0 \in I_0\,x$, and for $n \geq 0$, $I_{n+1} \subseteq I_n$.

$$\frac{\forall n, \forall h : \alpha \to \mathsf{M}\beta, (h \cdot G \subseteq I_n) \Rightarrow (F\,h) \cdot G \subseteq I_{n+1}}{\textbf{fix}\,F \cdot G \subseteq \bigcap_n I_n}$$

The proof is a direct consequence of the following equalities with $G = [g_1, g_2]$ and $I_n = [p_n, q_n]$, where $(p_n)_n$ is an increasing sequence starting from 0 and $(q_n)_n$ is a decreasing sequence:

$$\mathbf{fix}\, F \cdot [g_1, g_2] = [\mathtt{lub}(F^n 0) \cdot g_1, \mathtt{lub}(F^n 0) \cdot g_2] \subseteq [\mathtt{lub}\,(p_n), \mathtt{glb}\,(q_n)]$$

The rule above estimates an upper-bound of the fix-point using a decreasing sequence, it is sometimes more convenient to use increasing sequences both for lower and upper bounds of the intervals. In this case, assuming $(p_n)_n$ and $(q_n)_n$ are both increasing sequences of functions of type $\alpha \to [0, 1]$ with the proviso that for all x, $p_0\, x = 0$, we can prove the following result:

$$\frac{\forall n, \forall h : \alpha \to \mathsf{M}\beta, (h \cdot G \subseteq [p_n, q_n]) \Rightarrow (F\, h) \cdot G \subseteq [p_{n+1}, q_{n+1}]}{\mathbf{fix}\, F \cdot G \subseteq [\mathtt{lub}\,(p_n), \mathtt{lub}\,(q_n)]}$$

No continuity condition on F is required to validate the above rules. As mentioned in Section 3.3.3, continuity is only necessary to ensure that $\mathbf{fix}\, F$ is indeed a fixpoint of F.

4.4.2. Advanced schemes

The previous scheme gives the general idea. However, reasoning with fix-points is always tricky, and it would be handy to involve some more advanced schema in the process. While one is required to find an appropriate invariant, there are some systematic ways to find it depending on the form of F. In this section, we make intensive use of notations introduced at the beginning of the section.

In this part, we took inspiration from the loop rules in pGCL introduced by Morgan (as described in [18]) and propose a systematic generalization to the case of recursive functions.

Let us make some preliminary observations. We start from a recursive definition $\mathbf{let\ rec}\, f\, x = e$ on type $\alpha \to \beta$. Assuming f is deterministic and we want to prove that $\forall x, P\,(f\, x)$, a natural approach is to try to find an inductive argument which shows that the body e of the function f satisfies P assuming the recursive calls in e do. More formally, if the definition f corresponds to the functional F, we can try to prove for an arbitrary function h that, $\forall x, P(h\, x)$ implies $\forall x, P(F\, h\, x)$.

We use a similar approach for randomized programs. Instead of the property P, we start from a function $g : \alpha \to \beta \to [0, 1]$ to be estimated and we try to relate the estimation of the body of the recursive function $(F[f] \cdot g)$ to the estimation of the recursive calls by using properties of F. If we succeed, it means that we found a functional F_g (of type $(\alpha \to U) \xrightarrow{m} (\alpha \to U)$) such that the following diagram commutes for an arbitrary h of type $\alpha \to \mathsf{M}\beta$.

$$\begin{array}{ccc} h & \xmapsto{\omega_g} & h \cdot g \\ \Big\downarrow{\scriptstyle F} & & \Big\downarrow{\scriptstyle F_g} \\ F\, h & \xmapsto{\omega_g} & (F\, h) \cdot g \end{array}$$

Whenever F_g exists, we get for all $n > 0$, the relation: $\omega_g \circ F^n = F_g^n \circ \omega_g$ which expresses a simulation relation between the fix-point issued from the source program through iterations of the functional F when applied to g, and the fix-point which can be computed by applying the functional F_g.

Therefore, we understand that the value $[f] \cdot g$ can be reached as well from the sequence of iterations F_g^n. In fact:

$$[f] \cdot g = \mathbf{fix}\, F \cdot g = \mathtt{lub}\,(F^n 0) \cdot g = \mathtt{lub}\,(F^n 0 \cdot g) = \mathtt{lub}\,(F_g^n (0 \cdot g)) = \mathbf{fix}\, F_g$$

We now give the general definition.

Definition 4. Given a functional F of type $(\alpha \to \mathsf{M}\beta) \xrightarrow{m} (\alpha \to \mathsf{M}\beta)$ a function g of type $\alpha \to \beta \to [0, 1]$, we say that a functional F_g of type $(\alpha \to [0, 1]) \xrightarrow{m} (\alpha \to [0, 1])$ *commutes* with F for the expectation g when the following property holds:

$$\forall h, (F\, h) \cdot g = F_g\, (h \cdot g) \tag{1}$$

We will say that F_g *weakly commutes* with F when

$$\forall h, (F\, h) \cdot g \leq F_g\, (h \cdot g) \tag{2}$$

An important consequence of the existence of F_g is that the estimation of expectation for the fix-point can be related to the fix-point of F_g as stated in the following lemma.

Proposition 5. *Given a real-valued function g of type $\alpha \to \beta \to [0, 1]$ and a monotonic operator F_g of type $(\alpha \to [0, 1]) \xrightarrow{m} (\alpha \to [0, 1])$:*

- *if F_g weakly commutes with F for g then $\mathbf{fix}\, F \cdot g \leq \mathbf{fix}\, F_g$.*
- *if F_g commutes with F for g then $\mathbf{fix}\, F \cdot g = \mathbf{fix}\, F_g$.*

Now we can use the fact that $\mathbf{fix}\, F_g$ is an initial fix-point, such that if we can find a real-valued function ϕ of type $\alpha \to [0,1]$ such that $F_g\, \phi \leq \phi$ then we deduce $\mathbf{fix}\, F_g \leq \phi$ and combining this result with the last property, we obtain the following result:

Proposition 6. *Given a real function g of type $\alpha \to \beta \to [0,1]$ such that there exists a monotonic operator F_g which weakly commutes with F for g, if $F_g\, \phi \leq \phi$ then $\mathbf{fix}\, F \cdot g \leq \phi$.*

In most cases, we also want a minoration for $\mathbf{fix}\, F \cdot g$. For that, we have to reverse this result and consider how the distribution $\mathbf{fix}\, F$ operates on $1-g$.

Proposition 7. *Given a real function g of type $\alpha \to \beta \to [0,1]$ such that there exists a monotonic operator F_{1-g} which weakly commutes with F for $1-g$, if $F_{1-g}\, (1-\phi) \leq (1-\phi)$ then $\phi \ \&\ (\mathbf{fix}\, F \cdot \mathbb{I}) \leq \mathbf{fix}\, F \cdot g$.*

The function $\mathbf{fix}\, F \cdot \mathbb{I}$ associates to each x the probability that the recursive function terminates on x.

Proof. The value $x \,\&\, y$ is defined in our formalism as $1-((1-x)+(1-y))$ using our bounded addition and corresponds to the real $\max(0, x+y-1)$. In particular $x \,\&\, 1 = x$ so for any function f, $f \,\&\, \mathbb{I} = f$.

The proof uses the fact that for any distribution μ of type $\mathsf{M}\beta$, we have $(1-\mu(1-h_1)) \,\&\, \mu(h_2) \leq \mu(h_1 \,\&\, h_2)$. From the previous proposition applied to $1-\phi$ we have $\mathbf{fix}\, F \cdot (1-g) \leq 1-\phi$. so $\phi \leq 1-\mathbf{fix}\, F \cdot (1-g)$ then:

$$\begin{aligned} \phi \ \&\ (\mathbf{fix}\, F \cdot \mathbb{I}) &\leq (1-\mathbf{fix}\, F \cdot (1-g)) \ \&\ (\mathbf{fix}\, F \cdot \mathbb{I}) \\ &\leq \mathbf{fix}\, F \cdot (g \,\&\, \mathbb{I}) = \mathbf{fix}\, F \cdot g \quad \square \end{aligned}$$

There is a special case where we can get a minoration by ϕ, this is when $\phi \leq \mathbf{fix}\, F \cdot \mathbb{I}$ which can be seen as a generalisation of the fact that our invariant estimation ϕ implies termination of the fix-point. In order to obtain this result, we need $(\mathbf{fix}\, F \cdot \mathbb{I}) - \phi$ to be a pre-fixpoint of F_{1-g}.

Proposition 8. *Let g be a real function of type $\alpha \to \beta \to [0,1]$ such that there exists a monotonic operator F_{1-g} which weakly commutes with F for $1-g$. If the properties $F_{1-g}\, ((\mathbf{fix}\, F \cdot \mathbb{I}) - \phi) \leq (\mathbf{fix}\, F \cdot \mathbb{I}) - \phi$ and $\phi \leq \mathbf{fix}\, F \cdot \mathbb{I}$ hold, then $\phi \leq \mathbf{fix}\, F \cdot g$.*

Proof. This results is obtained using the previous proposition with the invariant $\phi' = \phi + 1 - (\mathbf{fix}\, F \cdot \mathbb{I})$. We have $1-\phi' = (\mathbf{fix}\, F \cdot \mathbb{I}) - \phi$ such that $F_{1-g}\, (1-\phi') \leq 1-\phi'$ by hypothesis, consequently $\phi' \ \&\ \mathbf{fix}\, F \cdot \mathbb{I} \leq \mathbf{fix}\, F \cdot g$.

The final result comes from properties of $+$ and $\&$ on $[0,1]$:

$$\phi' \ \&\ \mathbf{fix}\, F \cdot \mathbb{I} = (\phi + (1-\mathbf{fix}\, F \cdot \mathbb{I}) \ \&\ \mathbf{fix}\, F \cdot \mathbb{I} = \phi \quad \square$$

4.4.3. Application to loops

We can define recursively a loop function in $\mathcal{R}$ml. We assume given a type S for states, a boolean condition `cond` of type $S \to$ `bool` and a body named `body` of type $S \to S$.

```
let rec loop s =
    if cond s then let s' = body s in loop s' else s
```

The interpretation [cond] will have type $S \to$ `Mbool` and [body] will have type $S \to \mathsf{M}S$.

We introduce the terms $\mathtt{ctrue}\, s = [\mathtt{cond}\, s]\mathbb{I}_{.=\mathtt{true}}$ and $\mathtt{cfalse}\, s = [\mathtt{cond}\, s]\mathbb{I}_{.=\mathtt{false}}$

We want to measure a function g of type $S \to [0,1]$ on the output state of `loop`, which does not depend on the input state. We still use the notation $f \cdot g$ in place of the more verbose $f \cdot \mathbf{fun}\, s \Rightarrow g$.

We write F for the functional associated to `loop`. We have:

$$(F\, f) \cdot g = \mathbf{fun}\, s \Rightarrow (\mathtt{ctrue}\, s) \times [\mathtt{body}\, s](f \cdot g) + (\mathtt{cfalse}\, s) \times (g\, s)$$

Such that the functional F_g which commutes with F for g can be defined the following way:

$$F_g\, h = \mathbf{fun}\, s \Rightarrow (\mathtt{ctrue}\, s) \times [\mathtt{body}\, s]h + (\mathtt{cfalse}\, s) \times (g\, s)$$

It is easy to check the following property: $F_{1-g}\, (1-h) \leq 1-(F_g\, h)$ such that the condition $\phi \leq F_g\, \phi$ is sufficient to ensure $F_{1-g}\, (1-\phi) \leq 1-\phi$. And we can derive the following theorem:

Proposition 9. *Given g, ϕ and ψ of type $S \to [0,1]$, assuming $\forall s, \phi\, s \leq (\mathtt{ctrue}\, s) \times [\mathtt{body}\, s]\phi + (\mathtt{cfalse}\, s) \times (g\, s)$ and $\forall s, (\mathtt{ctrue}\, s) \times [\mathtt{body}\, s]\psi + (\mathtt{cfalse}\, s) \times (g\, s) \leq \psi\, s$ we can deduce $\phi \ \&\ [\mathtt{loop}] \cdot \mathbb{I} \leq [\mathtt{loop}] \cdot g \leq \psi$.*

In case `cond` is a non-randomized construction, let $C\, s$ be the property $\mathtt{cond}\, s = \mathtt{true}$. The condition:
$\phi\, s \leq (\mathtt{ctrue}\, s) \times [\mathtt{body}\, s]\phi + (\mathtt{cfalse}\, s) \times (g\, s)$ becomes:
$C\, s \Rightarrow \phi\, s \leq [\mathtt{body}\, s]\phi$ and $\neg C\, s \Rightarrow \phi\, s \leq g\, s$
which is a generalization of the loop rule in axiomatic semantics, ϕ being the invariant which should be preserved in the body (when the condition is true) and should establish the post-condition at the end (when the condition is false).

We consequently have the following rule which corresponds to the total loop correctness rule in [18]:

$$\frac{\forall s,\, C\, s \Rightarrow \phi\, s \leq [\mathtt{body}\, s]\phi}{\forall s,\, \phi\, s \;\&\; [\mathtt{loop}\, s]\mathbb{I} \leq [\mathtt{loop}\, s](\phi \;\&\; \mathbb{I}_{\neg C})}$$

5. Applications

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

5.1. Probabilistic termination

We return to our example of Section 2.6.2, a random walk which illustrates probabilistic termination.

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

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

$$\forall x, [\mathtt{walk}\, x]\mathbb{I} = 1.$$

The functional F to be considered is:

$$\mathbf{fun}\, [\mathtt{walk}] \stackrel{m}{\Rightarrow} \mathbf{fun}\, x \Rightarrow [\mathbf{if}\, \mathtt{flip}() \,\mathbf{then}\, x \,\mathbf{else}\, \mathtt{walk}\, (x+1)]$$

when $w : \mathtt{nat} \to \mathsf{M}\mathtt{nat}$, $x : \mathtt{nat}$ and $g : \mathtt{nat} \to [0, 1]$ to be measured, we have:

$$(F\, w \cdot g)\, x = \frac{1}{2}(g\, x) + \frac{1}{2}(w \cdot g)\, (x+1)$$

We can introduce F_g of type $(\mathtt{nat} \to [0, 1]) \stackrel{m}{\to} (\mathtt{nat} \to [0, 1])$ such that

$$F_g\, h\, x = \frac{1}{2} g\, x + \frac{1}{2}(h\, (x+1))$$

and check the commutation property between F_g and F.

In case g is the function $\mathbb{I}$ we get the functional

$$F_{\mathbb{I}}\, h\, x = \frac{1}{2} + \frac{1}{2}(h\, (x+1))$$

we know by Proposition 5 that

$$[\mathtt{walk}\, x]\mathbb{I} = \mathbf{fix}\, F_{\mathbb{I}}\, x$$

what remains to be computed is $\mathbf{fix}\, F_{\mathbb{I}}\, x$.

The real $\mathbf{fix}\, F_{\mathbb{I}}\, x$ is the least-upper bound of a sequence $(p_i)_i$ such that $p_0 = 0$ and $p_{i+1} = \frac{1}{2} + \frac{1}{2} p_i$.

It is easy to show that $p_n = 1 - \frac{1}{2^n}$, that the least upper bound of the sequence $(p_i)_i$ is 1 such that $\mathbf{fix}\, F_{\mathbb{I}}\, x = \mathtt{lub}(p_n)_n = 1$.

5.2. Parametrized termination

This example is taken from Ycart [26], adapted here to fit with our restriction to discrete random distributions. It can be seen as a generalisation of `walk` where the probability to stop or continue is given in each point by an arbitrary function $K\, x$.

We assume given a non-randomized function K of type $\mathtt{nat} \to \mathtt{nat}$ and an integer N. We write also $Y\, x$ for the element of $[0, 1]$ defined as $(K\, x)/1 + N$. The function we want to study is defined by the following $\mathcal{R}\mathsf{ml}$ program:

```
let rec ω x = if random N < K x then x else ω (x+1)
```

We have $[\omega] = \mathbf{fix}\, F$, where $F\, f\, x \equiv [\mathbf{if}\, \mathrm{random}\, N < K\, x \,\mathbf{then}\, x \,\mathbf{else}\, f\, (x+1)]$.

Let us start with some informal observations. Given $\theta : \mathtt{nat} \to [0, 1]$, assume we want to approximate the value of $[\omega\, x]\theta \in [0, 1]$. From a mathematical point of view, this is a summation. Let us have a naive look at it:

$$\int \theta(y)\mathrm{d}[\omega\, x](y) \;=\; (Y\, x)\theta\, x + (1 - (Y\, x)) \int \theta(y)\mathrm{d}[\omega\, (x+1)](y)$$

From the Section 2.5.5, we know our monadic interpretation expresses the same idea, in a more formal setting.

5.2.1. Putting advanced schemes at work

$$\begin{aligned}[\omega\, x]\theta &= (Y\, x)\theta\, x + (1-(Y\, x))[\omega\,(x+1)]\theta \\ &= (Y\, x)\theta\, x + (1-(Y\, x))(Y\,(x+1))\theta\,(x+1) \\ &\quad +(1-(Y\, x))(1-(Y\,(x+1))[\omega\,(x+2)]\theta \\ &= \ldots \\ &= (Y\, x)\times\theta\, x + \cdots + \prod_{k=x}^{x+n}(1-Y\, k)[\omega\,(x+1+n)]\theta\end{aligned}$$

We observe that the potential source of divergence depends on the behavior of the infinite product $R_\infty(x)$, limit of the sequence $R_n(x) \equiv \prod_{k=x}^{x+n-1}(1-Y\, k)$. Let us make this observation more formal. Considering the functional F which defines the fix-point, we rather get:

$$[F\, f\, x]\theta = (Y\, x)\theta\, x + (1-(Y\, x))[f\,(x+1)]\theta \tag{3}$$

This turns out to be an application of the properties presented in Section 4.4.2.

From Eq. (3), we get that the commutation property holds with the functional

$$F_\theta\, h\, x = (Y\, x)\times(\theta\, x) + (1-Y\, x)\times(h\,(x+1))$$

When θ is the unit function $\mathbb{I}$, we obtain:

$$F_{\mathbb{I}}\, h\, x = (Y\, x) + (1-Y\, x)\times(h\,(x+1))$$

The Proposition 5 ensures that $[\omega\, x]\mathbb{I} = \mathbf{fix}\, F_{\mathbb{I}}\, x$ so it remains to compute this fixpoint, it is the limit of a sequence s_n such that $s_0\, x = 0$ and $s_{n+1}\, x = (Y\, x) + (1-Y\, x)\times(s_n\,(x+1))$.

One shows by induction on n that

$$s_n\, x = \sum_{k=0}^{n-1} Y(x+k)\times R_k(x)$$

with $R_n(x)$ as defined above. Then using the fact that $Y(x+k)\times R_k(x) = R_k(x) - R_{k+1}(x)$ we deduce $s_n\, x = R_0(x) - R_n(x) = 1-R_n(x)$ and consequently the expected limit of $s_n\, x$ is equal to $1-\prod_{i=x}^{\infty} 1-(Y\, i)$. We deduce the expected result:

$$[\omega\, x]\mathbb{I} = 1-\prod_{i=x}^{\infty} 1-(Y\, i)$$

We now illustrate the use of other rules for fix-points. We may be interested to show that the function ω applied on x never outputs value less than x. Because it is a property always true, one possibility would be to use the power of the Coq type system and have a semantic which associates to x a distribution on numbers greater or equal to x. However, if we stay in our $\mathcal{R}$ml framework, we may want to prove that the probability for $\omega\, x$ to output a value less than x is 0, which can be rephrased as $[\omega\, x]\mathbb{I}_{.<x} = 0$. This is a case where the function to be measured $\mathbb{I}_{.<x}$ depends on the input x. With g of type $\mathtt{nat} \to \mathtt{nat} \to [0, 1]$ we have

$$(F\, f\cdot g)\, x = [F\, f\, x](g\, x) = (Y\, x)\times(g\, x\, x) + (1-Y\, x)\times[f\,(x+1)](g\, x)$$

We consider $g\, x = \mathbb{I}_{.<x}$. This does not lead directly to a commutation property, because we need a sub-expression of the form $[f\,(x+1)](g\,(x+1))$ in order to abstract with respect to the function $f\cdot g$. We remark that $g\, x\, x = \mathbb{I}_{.<x}\, x = 0$ and also that $g\, x = \mathbb{I}_{.<x} \le \mathbb{I}_{.<x+1} = g\,(x+1)$ such that we have for this particular g:

$$(F\, f\cdot g)\, x \le (1-Y\, x)\times(f\cdot g)\,(x+1)$$

and we can introduce the function F_g which weakly-commutes with F

$$F_g\, h\, x = (1-Y\, x)\times(h\,(x+1))$$

Now we remark that $h = 0$ is an invariant of F_g such that using Proposition 6, we deduce $[\omega\, x]\mathbb{I}_{.<x} \le 0$ which is the expected result.

We can deduce using the same kind of reasoning that $[\omega\, x]\mathbb{I}_{.=x} = Y\, x$. The general method is again to rewrite $F\, f\cdot g$ for that particular case. We obtain because $g\, x\, x = 1$:

$$(F\, f\cdot g)\, x = (Y\, x) + (1-Y\, x)\times[f\,(x+1)](g\, x)$$

now we would like to reuse our previous result which ensures that $\omega\,(x+1)\cdot\mathbb{I}_{.=x} \le \omega\,(x+1)\cdot\mathbb{I}_{.<x+1} = 0$. This is possible using a stronger notion of commutation in Proposition 5 where we force the variable f to be less than the fixpoint we analyse, in our case, we may assume $f \le [\omega]$ and consequently use $[f\,(x+1)](g\, x) = 0$.

We obtain $(F\, f\cdot g)\, x = (Y\, x)$ so the (constant) functional $F_g\, h\, x = Y\, x$ commutes with F for g and $[\omega\, x]\mathbb{I}_{.=x} = \mathbf{fix}\ F_g = Y\, x$.

The lemmas in Coq involving commutation in Section 4.4.2 have been developed with this stronger notion of commutation i.e. $\forall h,\ h \le \mathbf{fix}\, F \Rightarrow F\, h\cdot g = F_g\,(h\cdot g)$.

5.2.2. Some practical consequences

Turning back to our program ω, taking $x = 0$ as an example, we have proved so far:

$$\int (\mathbb{I}\, y)\mathrm{d}[\omega\, 0](y) \;=\; 1-\prod_{k\geq 0}^{\infty}(1-Y\, k)$$

Therefore, the termination depends upon the asymptotic behavior of $Y\, x = (K\, x)/1 + N$, through the existence of the limit $R_\infty(x)$. For instance, whenever K is non-zero, $(\omega\, 0)$ terminates almost surely; while if K always returns the 0 value, then this program diverges almost surely. If $K\, n = 0$ as soon as $n \geq p$ for some integer $p > 0$, then $(\omega\, 0)$ terminates with probability $1-\prod_{k=0}^{p}(1-Y\, k)$.

5.3. 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 by Hurd [9] uses a simple idea: write p in binary form $\sum_{i=1}^{\infty} p_i \frac{1}{2^i}$, 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 3.1.5). 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
```

As before, given a function g of type `bool` $\to [0, 1]$ (not depending on the input p of the function), we compute the value of the functional F associated to `bernoulli`:

$$[F\, f\, p]g = \begin{array}{l}\textbf{if}\ p < \frac{1}{2}\ \textbf{then}\ \frac{1}{2}(g\ \texttt{false}) + \frac{1}{2}[f\ (p+p)]g \\ \textbf{else}\ \frac{1}{2}[f\ (p\ \&\ p)]g + \frac{1}{2}(g\ \texttt{true})\end{array}$$

So F_g commutes with F for g with F_g defined by:

$$F_g\, h\, p = \begin{array}{l}\textbf{if}\ p < \frac{1}{2}\ \textbf{then}\ \frac{1}{2}(g\ \texttt{false}) + \frac{1}{2}(h\ (p+p)) \\ \textbf{else}\ \frac{1}{2}(h\ (p\ \&\ p)) + \frac{1}{2}(g\ \texttt{true})\end{array}$$

In case g is the function $\mathbb{I}$, we have

$$F_{\mathbb{I}}\, h\, p = \textbf{if}\ p < \tfrac{1}{2}\ \textbf{then}\ \tfrac{1}{2} + \tfrac{1}{2}(h\ (p+p))\ \textbf{else}\ \tfrac{1}{2}(h\ (p\ \&\ p)) + \tfrac{1}{2}$$

In order to compute $\textbf{fix}\, F_{\mathbb{I}}$ we introduce the sequence $p_0 = 0\ p_{n+1} = \frac{1}{2} + \frac{1}{2}p_n$, which is the same sequence we used for the termination of `walk`, its limit is 1.

So we know that `bernoulli` terminates almost surely, i.e. $\textbf{fix}\, F \cdot \mathbb{I} = 1$ we consequently can use Propositions 6 and 7 in order to study the probability of the result to be `true`.

With $g = \mathbb{I}_{.=\texttt{true}}$ we have

$$F_g\, h\, p \;=\; \textbf{if}\ p < \tfrac{1}{2}\ \textbf{then}\ \tfrac{1}{2}h\ (p+p)\ \textbf{else}\ \tfrac{1}{2}h\ (p\ \&\ p) + \tfrac{1}{2}$$

$$F_{1-g}\, h\, p \;=\; \textbf{if}\ p < \tfrac{1}{2}\ \textbf{then}\ \tfrac{1}{2} + \tfrac{1}{2}h\ (p+p)\ \textbf{else}\ \tfrac{1}{2}h\ (p\ \&\ p)$$

We take for invariant $\phi\, p = p$, in order to deduce $\textbf{fix}\, F\, p \cdot \mathbb{I}_{.=\texttt{true}} = p$ it is enough to prove that ϕ is a pre fix-point of F_g (i.e. $F_g\, \phi \leq \phi$) and $1-\phi$ is a pre fix-point of F_{1-g} (i.e. $F_{1-g}\ (1-\phi) \leq 1-\phi$). So we simply have to prove the following properties which are consequences of properties of $[0, 1]$:

- $\textbf{if}\ p < \frac{1}{2}\ \textbf{then}\ \frac{1}{2}(p+p)\ \textbf{else}\ \frac{1}{2}(p\ \&\ p) + \frac{1}{2} \leq p$
- $\textbf{if}\ p < \frac{1}{2}\ \textbf{then}\ \frac{1}{2} + \frac{1}{2}(1-(p+p))\ \textbf{else}\ \frac{1}{2}(1-(p\ \&\ p)) \leq 1-p$

5.4. Improving precision

The previous examples show the proof of properties of particular programs. Our Coq development gives us the possibility to also derive more abstract properties involving program schemes.

We study 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 tests 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 β which satisfies a property Q with a certain probability. The hypotheses are that we have two programs p_1 and p_2 of type β, thus interpreted as objects of type $\mathsf{M}\beta$. 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 non-randomized function `choice` of type $\beta \to \beta \to \beta$ 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 $(b = \texttt{true} \Rightarrow p\ \texttt{is prime})$ and $(\texttt{choice}\, b_1\, b_2)$ defined as $(b_1\ \texttt{and}\ b_2)$. The opposite direction $p\ \texttt{is prime} \Rightarrow b = \texttt{true}$ is always satisfied for the output of the program so does not require further analysis.

Now we build a new program p:

$$\textbf{let}\ x = p_1\ \textbf{in let}\ y = p_2\ \textbf{in}\ \texttt{choice}\, x\, y$$

We assume that we have estimations for the probability of p_1 (resp. p_2) to satisfy Q, i.e. $k_1 \leq [p_1]\mathbb{I}_Q$ (resp. $k_2 \leq [p_2]\mathbb{I}_Q$) and we want to prove that the program p satisfies Q with a better probability.

Let k stand for the expression $k_1 + k_2 - k_1k_2$, and notice that $k = k_1(1-k_2) + k_2 = k_2(1-k_1) + k_1$ such that k is greater than both k_1 and k_2.

We are going to show that $k_1 \leq [p_1]\mathbb{I}_Q$ and $k_2 \leq [p_2]\mathbb{I}_Q$ implies $k \leq [p]\mathbb{I}_Q$.

Actually we establish a more general result, using an arbitrary function q of type $\beta \to [0, 1]$ 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

$$k \leq [p_1]\ (\textbf{fun}\ x \Rightarrow [p_2]\ (\textbf{fun}\ y \Rightarrow q\ (\texttt{choice}\, x\, y)))$$

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

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

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:

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

which is true because k is, by construction, monotonic with respect to both k_1 and k_2.

This example illustrates the possibility to do abstract modular reasoning in our framework. In Coq, the expressions $[p_1]$ and $[p_2]$ are just represented as variables of type $\mathsf{M}\beta$.

6. Related work

Park et al. [21] propose a probabilistic functional language, named $\lambda_\bigcirc$ which extends the ML functional kernel on the basis of the monadic metalanguage developed by Pfenning and Davies [23]. A key feature is the clear syntactical separation between deterministic terms and probabilistic expressions. The latter correspond to mathematical random variables. Any term can be seen as an expression: the Dirac mass distribution on this term. From any expression E, the operator **prob** E builds the associated image measure. As for random primitives, the language introduces the constant expression **S** which denotes a random variable following the uniform law on $[0, 1]$. In $\mathcal{R}$ml, one does not distinguish between these two syntactic categories; the monadic transformation forces any $\mathcal{R}$ml term into a measure of some kind. The monadic operators unit and bind get as close as possible from the corresponding **prob** and **sample** x **from** $\cdots$ **in** $\cdots$ from $\lambda_\bigcirc$.

The language $\lambda_\bigcirc$ 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 [15], 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 instead. Therefore, examples developed with $\lambda_\bigcirc$ are better analyzed through simulation techniques. These 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) O'Caml expressions.

In this paper, we have limited ourselves to discrete distributions, with the benefit of ensuring our monadic transformation to interpret properly programs as mathematical measures. We think the continuous case could be reached, starting from the formal development done so far with the U axiomatization, but this point requires further investigations. The current presentation does not take measurability property into account. This is not required in the discrete case, but cannot be

ignored in the general case anymore. We strongly consider Jones's [14] work as a possible direction to follow, providing that the interpretation of type β is given a cpo-structure. Also the recent work by Hasan and Tahar [8], which develops a formalization of continuous probability distributions based on Hurd's approach, deserves interest towards this goal.

McIver and Morgan [18] 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 according to 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 $\mu(f) + \mu(g) \leq \mu(f + g)$. Developing the corresponding theory still remains to be done. A mechanization of this calculus using the HOL theorem prover is presented in Hurd et al.'s paper [12]. 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. A so-called *deep-embedding* is proposed. The syntax of the language of guarded commands and the weakest-precondition generator are explicitly encoded in the proof assistant. Our approach uses instead a *shallow* embedding where we directly encode the semantics of the language. Also, their approach allows to measure an arbitrary (measurable) $\mathbb{R}^+$-valued function; We choose to restrict ourselves to $[0, 1]$-valued ones 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). We plan to investigate how to extend our development in that direction.

As already said in the introduction, our approach owes much to Kozen as well as to Hurd's thesis, where formal verification of probabilistic programs is handled with the HOL theorem prover. Hurd 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, inverse $(1-x)$, least upper-bounds of monotonic sequences.

We have derived axiomatic rules for estimating the probability that programs satisfy some given properties, following the structure of the program. When dealing with probabilistic termination of programs, we provide several fix-point rules, which could cover a wide class of situations. We provide few basic examples for showing how to take benefit of them. 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 [22].

Further research topics concern both theoretical issues and more practical concerns. On the former side, we want to deepen the relations between the $[0, 1]$-segment of $\mathbb{R}$ which is formalized by our axiomatization type U and other axiomatizations for the reals studied elsewhere. We are also interested in the approach taken by Escardó and Simpson [2] in the development of Real PCF. See also Geuvers et al. [6] for a quite comprehensive survey.

On the other side, we plan the development of an environment for analyzing randomized functional programs. Basically, the tool should automatically generate verification conditions from the specification of pre and post conditions plus a validation (the correctness proof in Coq obtained from the monadic translation of the program). This gets close to design infrastructure offered by the Why tool [3,4], with the consequence of allowing eventually other prover assistants as well for the logical back-end.

We are also looking forward to more advanced examples that certainly will require more challenging automation of their proofs.

Acknowledgements

We thank A. McIver and C. Morgan for useful comments on the first version of this paper and the referees for pointing out inaccuracies in an early version of this paper. We also thank R. Lassaigne for stimulating discussions on formal proofs for analyzing random programs and P. Lescanne for his feedback on the introduction part. This research was partly supported by Agence Nationale de la Recherche (references ANR-07-SESU-010-03 and 04).

References

[1] P. Audebaud, C. Paulin-Mohring, Proofs of randomized algorithms in Coq, in: T. Uustalu (Ed.), Proc. of 8th Int. Conf. on Mathematics of Program Construction, MPC 2006, Kuressaare, July 2006, in: Lect. Notes in Comput. Sci., vol. 4014, Springer, Berlin, 2006, pp. 49–68.

[2] M. Escardó, A. Simpson, A universal characterization of the closed euclidean interval (extended abstract), in: Proc. of 16th Ann. IEEE Symp. on Logic in Computer Science, LICS 2001 Boston, MA, June 2001, IEEE CS Press, Los Alamitos, CA, pp. 115–125.
[3] J.-C. Filliâtre, The Why Verification Tool, 2002. URL http://why.lri.fr/.
[4] J.-C. Filliâtre, Verification of non-functional programs using interpretations in type theory, J. Funct. Programming 13 (4) (2003) 709–745.
[5] H. Geuvers, M. Niqui, Constructive reals in Coq: Axioms and categoricity, in: P. Callaghan, Z. Luo, J. McKinna, R. Pollack (Eds.), Selected Papers from 1st Int. Wksh. on Types for Proofs and Programs, TYPES 2000, Durham, Dec. 2000, in: Lect. Notes in Comput. Sci., vol. 2277, Springer, Berlin, 2002, pp. 79–95.
[6] H. Geuvers, M. Niqui, B. Spitters, F. Wiedijk, Constructive analysis, types and exact real numbers, Math. Structures Comput. Sci. 17 (1) (2007) 3–36.
[7] M. Giry, A categorical approach to probability theory, in: B. Banaschewski (Ed.), Categorical Aspects of Topology and Analysis, in: Lect. Notes in Math., vol. 915, Springer, Berlin, 1982, pp. 69–85.
[8] O. Hasan, S. Tahar, Formalization of continuous probability distributions, in: F. Pfenning (Ed.), Proc. of 21st Conf. on Automated Deduction, CADE-21, Bremen, July 2007, in: Lect. Notes in Artif. Intell., vol. 4603, Springer, Berlin, 2007, pp. 3–18.
[9] J. Hurd, A formal approach to probabilistic termination, in: V.A. Carreño, C.A. Muñoz, S. Tahar (Eds.), Proc. of 15th Int. Conf. on Theorem Proving in Higher-Order Logics, TPHOLs 2002, Hampton, VA, Aug. 2002, in: Lect. Notes in Comput. Sci., vol. 2410, Springer, Berlin, 2002, pp. 230–245.
[10] J. Hurd, Formal verification of probabilistic algorithms, Ph.D. Thesis, Univ. of Cambridge, 2002.
[11] J. Hurd, Verification of the Miller–Rabin probabilistic primality test, J. Log. Algebr. Program. 50 (1–2) (2003) 3–21.
[12] J. Hurd, A. McIver, C. Morgan, Probabilistic guarded commands mechanized in HOL, in: A. Cerone, A. Di Pierro (Eds.), Proc. of 2nd Wksh. on Quantitative Aspects of Programming Languages, QAPL 2004, Barcelona, March 2004, in: Electron. Notes in Theor. Comput. Sci., vol. 112, Elsevier, Amsterdam, 2005, pp. 95–111.
[13] C. Jones, Probabilistic non-determinism, Ph.D. Thesis, Univ. of Edinburgh, 1989.
[14] C. Jones, G. Plotkin, A probabilistic powerdomain of evaluations, in: Proc. of 4th Ann. IEEE Symp. on Logic in Computer Science, LICS '89, Pacific Grove, CA, June 1989, IEEE CS Press, Los Alamitos, CA, 1989, pp. 186–195.
[15] D. Kozen, Semantics of probabilistic programs, J. Comput. System Sci. 22 (1981) 328–350.
[16] D. Kozen, A probabilistic PDL, in: Proc. of 15th Ann. ACM Symp. on Theory of Computing, STOC '83, Boston, MA, Apr. 1983, ACM Press, New York, 1983, pp. 291–297.
[17] F.W. Lawvere, The category of probabilistic mappings, 1962. Preprint.
[18] A. McIver, C. Morgan, Abstraction, refinement and proof for probabilistic systems, in: Technical Monographs in Computer Science, Springer, New York, 2005.
[19] E. Moggi, Notions of computation and monads, Inform. and Comput. 93 (1) (1991) 55–92.
[20] C. Morgan, A. McIver, pGCL: Formal reasoning for random algorithms, South African Comput. J. 22 (1999) 14–27.
[21] S. Park, F. Pfenning, S. Thrun, A probabilistic language based upon sampling functions, in: Proc. of 32nd ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages, POPL 2005, Long Beach, CA, Jan. 2005, ACM Press, New York, 2005, pp. 171–182.
[22] C. Paulin-Mohring, A library for reasoning on randomized algorithms in Coq, Description of a Coq Contribution, INRIA & Univ. Paris Sud, 2007. URL http://www.lri.fr/~paulin/ALEA/library.pdf.
[23] F. Pfenning, R. Davies, A judgmental reconstruction of modal logic, Math. Structures Comput. Sci. 11 (4) (2001) 511–540.
[24] N. Ramsey, A. Pfeffer, Stochastic lambda calculus and monads of probability distributions, in: Conf. Record of 29th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages, POPL 2002, Portland, OR, Jan. 2002, ACM Press, New York, 2002, pp. 154–165.
[25] D. Scott, Lattice theory, data types, and semantics, in: R. Rustin (Ed.), Formal Semantics of Programming Languages, in: Courant Computer Science Symposia, vol. 2, 1972, pp. 65–106.
[26] B. Ycart, Modèles et algorithmes Markoviens, in: Collection SMAI Mathématiques et Applications, vol. 39, Springer, Berlin, 2002.