# Exploring interpretations of probabilistic algorithms using measure theory in Coq

Kira Kutscher - 201509720
Lasse Letager Hansen - 201508114

17th of June 2019

# Contents


**1 Introduction (Finished 12.06.) 2**

**2 Theory and existing frameworks (READY(?)) 2**
- 2.1 Complete partial orders . . . 3
  - 2.1.1 Recursive definitions as fixed point iterations . . . 3
- 2.2 Interpreting probabilistic definitions . . . 5
  - 2.2.1 The concept of measures . . . 5
  - 2.2.2 A monadic interpretation . . . 6
- 2.3 Putting together recursion and probability . . . 8
- 2.4 The functional approach: $\mathcal{R}$ml . . . 9
- 2.5 EasyCrypt and `pwhile` (or 'The imperative approach') . . . 9

**3 Equivalence of $\mathcal{R}$ml and `pwhile` (READY) 12**
- 3.1 Translating `while` to a functional language . . . 13
- 3.2 Towards the $\lambda$-calculus . . . 15
- 3.3 Interpreting it both ways . . . 15
- 3.4 Adding probabilistic definitions . . . 19
- 3.5 Implementing the translations in Coq . . . 19

**4 The Coq development (READY) 19**
- 4.1 The definition of $\mathcal{R}$ml* . . . 20
- 4.2 Baby steps: The advent of s$\mathcal{R}$ml . . . 22
- 4.3 Typing $\mathcal{R}$ml* . . . 24
- 4.4 But what does it mean? Interpreting s$\mathcal{R}$ml . . . 25

**5 Comparisons and future work (Draft 10.06.) 27**

**6 Conclusion (Finished 12.06.) 27**

**References 28**

**A Translating $\mathcal{R}$ml* to s$\mathcal{R}$ml 29**

**B $\mathcal{R}$ml* typing rules 33**

**C Example - Error: Stack Overflow. 34**

# 1 Introduction (Finished 12.06.)

Probabilistic algorithms are widely used, but far less widely proved correct. We explore some probabilistic languages and how to embed them in the widely known and trusted proof assistant Coq. Having an interpretation of such a language in Coq makes it possible to prove facts about the algorithms in Coq's proof logic.
We explore both a functional design, as presented in [1], and an imperative one, as in [2].

Requirements to the reader:

- A rudimentary understanding of probability theory, but no knowledge of measure theory
- Section 2.2 requires an understanding of monads and how they are used in functional programming
- Section 4 requires a basic understanding of Coq

# 2 Theory and existing frameworks (READY(?))

As mentioned before we focused our work on developing a framework for proofs of randomised algorithms in the proof assistant Coq. Coq is known to be a reliable tool and proofs formalised with it are widely trusted. Coq is, however, also known to be notoriously difficult to code things in due to a strict type system that requires determinism and certain termination of all programs written in it. Obviously these two are not the optimal conditions for the encoding of randomised algorithms that may not terminate.

This means that we need a way to encode an interpretation general recursion (or iteration) as well as randomness in such a way that we can still reason about our programs in the proof system of Coq without having to run them.

We will in this section present a monadic interpretation into probability distribution over the outcome of a randomised program as well as a way of interpreting general recursion.

In this section we will have a look at how to use complete partial orders to interpret general recursion (Section 2.1), and how we can represent the result of a probabilistic computation using a monadic interpretation of probability measures (Section 2.2. Afterwards we will move on to presenting

two different developments that have worked with probabilistic languages in Coq: the functional $\mathcal{R}$`ml` (Section 2.4) and the imperative `pwhile` (Section 2.5).

## 2.1 Complete partial orders

A partially ordered set (poset) is a set with an associated binary ordering relation $\leq$ which is both reflexive and transitive. The order is partial when the ordering relation is not defined on every pair of elements in the set.

There exist a number of different completeness properties that a poset can have. We will here have a look at $\omega$-complete partial orders, which we will use in order to interpret general recursion and probabilistic programs.

**Definition 1.** *$\omega$-complete partial order ($\omega$-cpo)*
An $\omega$-cpo is a partially ordered set that, additionally, has a distinct least element and where there exist least upper bounds on all monotonic sequences.

### 2.1.1 Recursive definitions as fixed point iterations

Before using $\omega$-cpos to interpret recursion, let us first have a look at some interesting things that our definition entails.

A monotonic sequence on an $\omega$-cpo $X$ can be viewed as a monotonic function $f : \mathbb{N} \xrightarrow{m} X$ where $f(n)$ is the $n$th element of the sequence (or the least upper bound of the sequence, if $n$ is larger than the length of the sequence).

There is a standard way of defining fixed point iterations on an $\omega$-cpo: [1]

Consider an operator $F : X \xrightarrow{m} X$ on some $\omega$-cpo $X$; with this we define the monotonic sequence $F_i \mapsto \underbrace{F(F(\ldots F}_{i \text{ times}}(0_X)\ldots))$ of repeated application of $F$ to the least element of $X$. By our choice of $F$ and the definition of $\omega$-cpos, it is clear that there has to exist a least upper bound on $F_i$. This least upper bound is the fixed point of $F$ and it will hold that `fix` $F = F($`fix` $F)$ if $F$ is continuous.

For an $\omega$-cpo with underlying set $B$ we can also define an $\omega$-cpo on functions from any set $A$ whose co-domain is $B$.

To reiterate the definition, let us think of what we need for an $\omega$-cpo. We need an ordering relation, a least element, and a least upper bound

operation. Those can be defined as follows:

$$\begin{aligned} f \leq_{A\to B} g \Leftrightarrow \forall x : f(x) \leq_B g(x) &\quad \textit{(pointwise order)} \\ 0_{A\to B} := f(x) = 0_B &\quad \textit{(least element)} \\ \texttt{lub}_{A\to B} f_n := g(x) = \texttt{lub}_B(f_n(x)) &\quad \textit{(least upper bound operation)} \end{aligned}$$

The result of an interpretation of programs in the language of discourse will be in an $\omega$-cpo, so according to the above discussion functions will have an $\omega$-cpo structure as well. Together with the above definition of fixed points we can use this structure to interpret general recursive definitions.

We define a functional, $F$, taking as input a function and "adding a step to it". Let us look at the example of the factorial function $f(n) = n!$ [3]. The recursive definition is well known:

$$fac(n) := \texttt{ if } n = 0 \texttt{ then } 1 \texttt{ else } n \cdot fac(n-1)$$

For the interpretation of this definition, we want to define $F : (\mathbb{N} \to \mathbb{N}) \to (\mathbb{N} \to \mathbb{N})$ in such a way that its fixed point is the same as the above recursive definition. We choose

$$F(F_i(n)) := \texttt{ if } n = 0 \texttt{ then } 1 \texttt{ else } n \cdot F_i(n-1)$$

Where $F_0(n)$ is $0_{\mathbb{N}\to\mathbb{N}}$ (the function that takes a natural number and returns 0), by the above definition of the least element in the $\omega$-cpo defined on a function space. By repeated application of $F$ the function will slowly approach the real factorial function, which is the fixed point of $F$. The beginning of the iteration will be

$$F_1(n) = F(F_0(n)) = \begin{cases} 1 & \text{if } n \text{ is } 0 \\ 0 & \text{otherwise} \end{cases}$$

$$F_2(n) = F(F(F_0(n))) = \begin{cases} 1 & \text{if } n \text{ is } 0 \\ 1 & \text{if } n \text{ is } 1 \\ 0 & \text{otherwise} \end{cases}$$

$$F_3(n) = F(F(F(F_0(n)))) = \begin{cases} 1 & \text{if } n \text{ is } 0 \\ 1 & \text{if } n \text{ is } 1 \\ 2 & \text{if } n \text{ is } 2 \\ 0 & \text{otherwise} \end{cases}$$

In this case it is easy to see that $F_0 \leq F_1 \leq F_2 \leq F_3 \leq \ldots$. In the general case this follows from the fact that $F$ has to be monotone and $F_0$ is always the least element of the function space $F$ operates on.

## 2.2 Interpreting probabilistic definitions

In order to interpret probabilistic definitions, we need a way of expressing the distribution of possible results. For this we will use probability measures as described in [1].

### 2.2.1 The concept of measures

In layman's terms, we can describe a measure on a set $A$ as exactly that: a way of measuring subsets of $A$. More precisely, a measure on $A$ assigns a non-negative real number to every "suitable" subset of $A$, where "suitable" means fulfilling certain arbitrary conditions. We will henceforth write $\mu(X)$ to signify the value of $X \subseteq A$ under the measure $\mu$.

In order for a function to be a measure, there are three properties it has to have: It must take only non-negative values, $\mu(\varnothing) = 0$, and it has to be countably additive. Being countably additive means that for every set of pairwise disjoint objects, the value of this set is equal to the sum of the values of each object:

$$\mu(X) = \mu(\bigcup_{x \in X} x) = \Sigma_{x \in X} \mu(x)$$

An example might be to choose $A$ to be a set of 3-dimensional objects and a possible measure would be the total volume of objects in a subset.

We can understand a measure on $A$ as integral over functions from $A$ to $\mathbb{R}^+$. From this perspective, the above example would consist of the function that given an object in $A$ returns its volume. The integral over this function would be the volume of all objects in $A$. Now the challenge is to measure only a subset of $A$. We can do this by introducing the characteristic function of a subset $X$[4]:

$$\mathbb{I}_X(x) = \begin{cases} 1 & \text{if } x \in X \\ 0 & \text{otherwise} \end{cases}$$

By multiplying the characteristic function for $X$ with the volume function on objects in $A$, we get a new function, $f$, such that $\int f \; d\mu = \mu(X)$. With this in place, we will allow ourselves to be sloppy in our notation and write $\mu(f)$ instead of $\int f \; d\mu$.

It is easy to see that the integral perspective still satisfies all the requirements that a function has to fulfil in order to be a measure, and the reader is invited to check this for herself.

Now why all this talk about measures? Wasn't it probabilistic programs we were talking about?

Well, yes. The cool thing is, that being able to measure function whose co-domain is the real numbers in the unit interval, $\tau \to [0,1]$, gives us a way of representing probability distributions. A measure on type $\tau$ can be expressed with the type $(\tau \to [0,1]) \to [0,1]$. An interpretation of a probabilistic term whose type is $\tau$ can now be understood as a measure on type $\tau$, or equivalently as a transformation of a probability distribution.

Something of type $\tau \to [0,1]$ can be understood as the function that returns the probability of its input being part of a specified set, or in other words, we can understand it as a probability density function. We can view our programs as a transformation of a distribution: If we input an initial distribution (this might also just be the characteristic function of a single value), we will as output receive the integral over the transformed distribution; with other words, we will receive the probability distribution specifying the result of an actual pseudorandomised run of our program producing a result within the input distribution.

As an example we might consider the value `flip`, which returns a uniform distribution of `false` and `true`. If now we supply the measure representation of this value with the characteristic function of `true`, the result will be 0.5, since this is the probability of `true` being the outcome.

We could even go further and say: If we have a set that contains `true` with probability $p$ and `false` with probability $q$, what would the probability be of the result of evaluating `flip` being in this set. The answer to this is simply applying the measure representation of `flip` with the function that returns $p$ on `true` and $q$ on `false`. With other words, the outcome would be the joint probability distribution between the input distribution and the one represented by the measure, assuming that both are independent.

### 2.2.2 A monadic interpretation

We can represent measures with a monadic structure. This is sensible, since probabilistic programs are inherently not functional and once probability is involved in a computation we can not get rid of it again; this fits with a monad where we can not get a value back out of the monad once it is in the monad, unless we work with the representation of the monad in our language.

Since we are using a monad over the type $\tau$ to represent a measure on $\tau$, we will for simplicity of notation introduce the notation $\mathtt{M}\tau$ to mean $(\tau \to [0,1]) \to [0,1]$.

The monadic operators presented here have been formerly introduced by [1] and satisfy the usual monadic properties[1]. They are also used by the Mathematical Components compliant Analysis Library[2] in their representation of distributions.

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

These definitions look big and scary, so let's break them down.

The `unit` function could be described as the "wrapper" that takes a value and wraps the monad around it. The outermost lambda-abstraction takes the value that we want to produce the measure of and remembers it. The second function binding is to match the type of measures; this is where the probability distribution is received as input, and where we then give the output of said distribution function applied to our initial value.

A simple example could be `unit 5` applied to the uniform distributions of numbers between 1 and 5. The result would then be 0.2, since that is the probability of a (pseudo-)random sampling from our measure (we recall that in this case it is just the characteristic function $\mathbb{I}_{\{5\}}$) and a (psuedo-)random sampling from the initial distribution coinciding.

The `bind` function can be viewed as something transforming the value inside the monad. Let us take it from the inside and out. Clearly the last line of the definition is the result, the $\texttt{M}\sigma$ part; this is obvious from the types. $\mu$ and $M$ are the usual arguments to a `bind` operation, $\mu$ being the initial value and $M$ the transformation to be applied to it.

We can view the final construction like continuation passing style programming. We still have a value of type $\tau$ inside the monad before we bind

[1] There are many introduction to the theory of monads. One example is our previous development in Coq, which can be found at `https://bitbucket.org/Ninijura/functionalprogramming/raw/47de0f3f259370f3214789bbc90511313be451f8/project/report_final.pdf`

[2] `https://github.com/math-comp/analysis/tree/master/`

it into a function, and we want to retain that information in the result of the `bind`. But we can not access this information unless we supply the value of type $\texttt{M}\tau$ with a function of type $\tau \to [0,1]$; this function is the one that $\mu$ is applied to in the final value. Since the result is of type $\texttt{M}\sigma$, it would make sense to use its input on something of type $\sigma$ to get something in [0,1], which would satisfy the output type for the function we apply $\mu$ to.

The final function takes the input of type $\tau$, that $\mu$ will give it, applies $M$ to it to transform it to something of type $\texttt{M}\sigma$, and then supplies it with the last piece of the computation, $f$.

This way we retain all of the information that is contained in $\mu$, transform it with $M$, and lastly transform it with $f$, once that is supplied.

An example of the use of `bind` could be the following: We have a distribution of natural numbers, $\mu$, and we want to figure out what the probability is that a number drawn from this distribution is even. To this end, we use the function $M$, that takes a natural number and returns the probability of it being even. If now we apply the result of `bind` $\mu$ $M$ to, let's say, the characteristic function of `false`, the following computation will be carried out:

`fun` $(x : \tau) \Rightarrow M\ x\ f$ is now the function that takes a natural number and returns the probability of it being odd. Applying $\mu$ to this function yields the probability of a pseudorandom interpretation of $\mu$ being an odd number; and this is exactly the meaning of

(`bind` $\mu$ $M$) (`fun (`$x$ `:  bool) ⇒ if` $x$ `then 0 else 1`).

### 2.3 Putting together recursion and probability

This section just needs some fact-checking.

What now if we want both probability and a way of interpreting general recursion? Then we need the $\omega$-cpo structure on our functions even though they are wrapped in an additional abstraction.

In order to show this, we use the fact that monotonic functions from an ordered set to an $\omega$-cpo also have an $\omega$-cpo structure [1, p. 584].

It is easy to see that, because a measure is an integral over strictly positive functions, it is a monotonic function. It is a function to [0,1], which clearly is an $\omega$-cpo with zero-element being zero, the ordering relation being $\leq$ as we know it for real numbers, and the least upper bound operation simply being the largest number of a monotonic sequence.

Now we have that a measure is a monotonic function to an $\omega$-cpo, which

is exactly what we needed to be sure of, before we go ahead and combine probability and recursion.

## 2.4 The functional approach: $\mathcal{R}$ml

The first language for probabilistic programs implemented in Coq is called $\mathcal{R}$ml('Randomised monadic language') and is due to is due toe Philippe Audebaud and Christine Paulin-Mohring ([1]).

$\mathcal{R}$ml could be described as a functional version of the `while` language with an interpretation that allows for probabilistic algorithms. The language itself does not contain probabilistic expressions, but rather makes use of the functions `random(`$n$`)` and flip `flip`, which give a uniform distribution of natural numbers between 0 and $n$, and `true` and `false`, respectively.

Since calling either function would be considered a valid $\mathcal{R}$ml term, the result has to be a measure over natural numbers (in the case of `random(`$n$`)`) or booleans (in the case of `flip`).

The expressions that our language consists of (we recall that it is a functional language, so there are only expressions, no statements) are as follows:

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

Here $x$ refers to a variable name previously bound in a let-binding, and $c$ refers to a primitive constant. Since Paulin-Mohring and Adebaud don't give a clear definition of what a constant can be, it makes sense to assume, that it can be any Coq term.

According to [1], the $f$ in function application can be a "primitive or a user-defined function", where primitive would be `random(`$n$`)` or `flip`, and a user-defined function should be specified by `let` $f \ x_1 \ \ldots \ x_n = e$, according to the language specification. Recursive functions can be defined with the keyword `let rec`.

The monadic interpretation $[e] : \texttt{M}\tau$ of a term $e : \tau$ is given in Figure 2.1.

## 2.5 EasyCrypt and `pwhile` (or 'The imperative approach')

EasyCrypt is a framework that has been developed in order to help in the construction of machine-checkable proofs about cryptographic constructions and protocols [2]. A standard approach to this kind of proofs is based on so-called games; in EasyCrypt cryptographic algorithms as well as games

| $\mathcal{R}$`ml` term $e$ : $\tau$ | Coq value $[e]$ : `M`$\tau$ |
|---|---|
| $v$ | `unit` $v$ $v$ variable or constant |
| `let` $x = a$ `in` $b$ | `bind` $[a]$ (`fun` $x \Rightarrow [b]$) |
| $f\ a_1 \ldots a_n$ | `bind` $[a_1]$ (`fun` $x_1 \Rightarrow \ldots$ `bind` $[a_n]$ (`fun` $x_n \Rightarrow [f]\ x_1 \ldots x_n) \ldots$) |
| `if` $b$ `then` $a_1$ `else` $a_2$ | `bind` $[b]$ (`fun` $x$ : `bool`) $\Rightarrow$ (`if` $x$ `then` $[a_n]$ `else` $[a_2]$) |

Figure 2.1: Monadic interpretation of $\mathcal{R}$`ml` terms as presented in [1].

are modelled as *modules* consisting of procedures written in a simple imperative language called `pwhile`. The `p` in `pwhile` stands for "probabilistic", so in total the name refers to a probabilistic extension of the well-known minimalistic `while` language.

We will in this section give an overview of the language as well as its interpretation in Coq, which is due to a development by Pierre-Yves Strub [3]. We will not concern ourselves with the module system of EasyCrypt since the focus of the present development is on probabilistic languages and their interpretation rather than their use.

`pwhile` consists of the following expressions and commands:

$$\begin{aligned} exp &::= x \mid const \mid \texttt{prp (}p\ \texttt{:}\ \ \texttt{pred mem)} \mid e_1\ e_2 \\ cmd &::= \texttt{abort} \mid \texttt{skip} \mid x := e \mid x\ \$ = e \\ &\quad \mid \texttt{if}\ b\ \texttt{then}\ c_1\ \texttt{else}\ c_2 \mid \texttt{while}\ b\ \texttt{do}\ c \mid c_1; c_2 \end{aligned}$$

The embedding of `pwhile` in Coq is a so-called shallow embedding, which means that we use Coq terms and types as part of programs in `pwhile`. This is used in order to form expressions: Constants in `pwhile` are Coq constants, hence an expression in `pwhile` can have any Coq type. In the implementation, expressions are parameterised by their type.

Most of the above constructs are fairly standard and should be known to most readers. In Figure 2.2 we give a full formal semantics of commands, based on the monadic operations presented in Section 2.2.2.

[3] https://github.com/strub/xhl

$$
\begin{aligned}
[\![\texttt{abort}]\!]\ m &= \texttt{dnull} \\
[\![\texttt{skip}]\!]\ m &= \texttt{unit}\ \ m \\
[\![i;\ c]\!]\ m &= \texttt{bind}\ \ ([\![i]\!]\ m)\ [\![c]\!] \\
[\![x := e]\!]\ m &= \texttt{unit}\ \ m[[\![e]\!]\ m/x] \\
[\![x\ \$ = d]\!]\ m &= \texttt{bind}\ \ ([\![d]\!]\ m)\ (\lambda v.\texttt{unit}\ \ m[v/x]) \\
[\![\texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2]\!]\ m &= \begin{cases} [\![c_1]\!]\ m \text{ if} [\![e]\!]\ m = \texttt{true} \\ [\![c_2]\!]\ m \text{ if} [\![e]\!]\ m = \texttt{false} \end{cases} \\
[\![\texttt{while}\ e\ \texttt{do}\ c]\!]\ m &= \lambda f.\sup\ (\lambda n.[\![[\texttt{while}\ e\ \texttt{do}\ c]_n]\!])\ m\ f) \\
&\quad \text{where} \\
&\quad [\texttt{while}\ e\ \texttt{do}\ c]_0 \quad = \texttt{skip} \\
&\quad [\texttt{while}\ e\ \texttt{do}\ c]_{n+1} = \texttt{if}\ e\ \texttt{then}\ c; \\
&\qquad\qquad\qquad\qquad [\texttt{while}\ e\ \texttt{do}\ c]_n
\end{aligned}
$$

Figure 2.2: Denotational semantics of `pwhile` programs based on the monadic structure presented in Section 2.2.2. This interpretation is the same as presented in Chapter 2 of [5].

In addition to the standard expressions and commands, `pwhile` has the expression `prp` $p$, which we will have a look at here. `prp` takes a single argument of type `pred mem`. This is a Coq type specified with the type constructor `pred` $= \forall \tau\ :\ \tau \to \mathbb{B}$ applied to the type of memories defined in the `xhl` development. The predicate over memories that is `prp`s argument is mapped over the working memory at the time of evaluation.

For more clarity, we will look at an example. Let us take the situation where we only want to proceed with a computation, if a certain variable, $x$ is defined in the memory; we want to access $x$, but want to avoid our program crashing if $x$ has not been defined. We now write the function `x_defined` that, given a memory, returns true if $x$ is defined in said memory and false otherwise. By branching on `prp x_defined`, we can now make sure that we only take the branch accessing $x$ if it is present in the memory and do not end up with a program that may crash.

At this point a comment about "crashing" programs is in order. There are multiple ways in which a program can lead to an undefined result: encountering undefined behaviour, non-termination, and the `abort` command.

The interpretation of all of these is the same: We recall that the result of interpreting a `pwhile` program in Coq is a probability distribution over memories; now the result of interpreting a "crash" is by simply returning the null-distribution over memories.

# 3 Equivalence of $\mathcal{R}$ml and pwhile (READY)

The goal of the present project was to gain an actionable understanding of $\mathcal{R}$ml and `pwhile`, their respective interpretations, theoretic background, and similarities and differences between them. Our approach was to define a translation between `pwhile` and $\mathcal{R}$ml as well as an interpretation of $\mathcal{R}$ml, and subsequently show that translating from `pwhile` to $\mathcal{R}$ml and interpreting the resulting program would lead to the same interpretation as interpreting the `pwhile` program directly.

This approach allowed us to get acquainted with both languages as well as the difficulties in interpretation that both of them share.

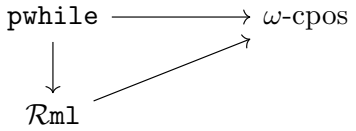


Figure 3.3: The languages we work with and the direction of the translations between them.

In this section we will concern ourselves with the translation between `pwhile` and $\mathcal{R}$ml, as well as their respective interpretations. Throughout this section we will disregard all typing rules. Adding types to the expressions should not influence the semantics of the translations we present and is straightforward.

For the translations, we will first have a look at deterministic versions of both in order to define the main part of the translations (Sections 3.1, 3.2). This means in order to show that the diagram of Figure 3.3 commutes, we will first show that the following diagram commutes (Section 3.3):

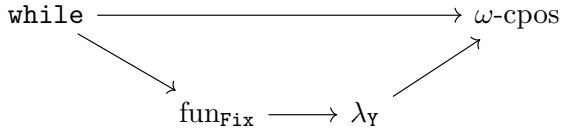


Figure 3.4: A deterministic version of Figure 3.3 with an extra step added.

To round off the theoretic part, we will have a look at how to add probabilistic constructs to all languages we use (Section 3.4).

We will briefly discuss how we would go about implementing the theory developed in the current section in Coq (Section 3.5), before we move on to discussion our own Coq development in the following Section (Section 4).

### 3.1 Translating `while` to a functional language

In order to do the translations properly, let us first have a look at a translation from the simple, widely known `while` language to a simple functional language resembling $\mathcal{R}$`ml`. The thought behind this is that once this translation is in place, all we have to do to translate `pwhile` to $\mathcal{R}$`ml` is to add probability.

$$\begin{aligned} exp &::= x|n|\texttt{true}|\texttt{false}|f\ x \\ stm &::= \texttt{skip}|x := e|\texttt{if}\ e\ \texttt{then}\ s_1\ \texttt{else}\ s_2|\texttt{while}\ e\ \texttt{do}\ s|s_1; s_2 \end{aligned}$$

The syntax of our functional language is the same as $\mathcal{R}$`ml` modulo the pre-defined probabilistic functions and our repeating it here is solely for the reader's convenience, not in order to introduce anything new.

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

The translation of expressions is completely straightforward: variables are mapped to variables, constants to constants, and function applications to function applications.

In order to translate statements we choose a set of SML-style matching rules; this choice is due to the translation of sequences being dependent on what the first statement is. We will in the following write the translation of a `while` statement $s$ to an expression in our functional language as $[\![s]\!]$.

The result of a computation in `while` is the state of a memory, while the result of a functional computation is a value. A simple way to make up for this difference is by choosing a variable name that is designated the return variable and encapsulates the information we are interested in after the computation. This is the result of a program translated from `while` to our functional language; in the following we choose $x_r$ as the symbol for the chosen return variable.

(1) $$\texttt{skip}\ ;\ s\ \mapsto [\![s]\!]$$

(2) $$\texttt{skip}\ \mapsto x_r$$

(3) $$x := e \ ; \ s \ \mapsto \texttt{let} \ x := e \ \texttt{in} \ [\![s]\!]$$

(4) $$x_r := e \ \mapsto e$$

(5) $$x := e \ \mapsto x_r$$

(6) $$(\texttt{if} \ e \ \texttt{then} \ s_1 \ \texttt{else} \ s_2) \ ; \ s_3 \ \mapsto \texttt{if} \ e \ \texttt{then} \ [\![s_1 \ ; \ s_3]\!] \ \texttt{else} \ [\![s_2 \ ; \ s_3]\!]$$

(7) $$\texttt{if} \ e \ \texttt{then} \ s_1 \ \texttt{else} \ s_2 \ \mapsto \texttt{if} \ e \ \texttt{then} \ [\![s_1]\!] \ \texttt{else} \ [\![s_2]\!]$$

(8) $$(\texttt{while} \ e \ \texttt{do} \ s_1) \ ; \ s_2 \ \mapsto \texttt{let rec} \ f \ x := \texttt{if} \ e \ \texttt{then} \ [\![s_1 \ ; \ f \ x]\!] \ \texttt{else} \ [\![s_2]\!] \ \texttt{in} \ f \ 0$$

(9) $$\texttt{while} \ e \ \texttt{do} \ s_1 \ \mapsto \texttt{let rec} \ f \ x := \texttt{if} \ e \ \texttt{then} \ [\![s_1 \ ; \ f \ x]\!] \ \texttt{else} \ x_r \ \texttt{in} \ f \ 0$$

Note that in 8 and 9 we create recursive functions with a name and an argument, both of which are not present in the while construct we translate from. This means that we have to be careful about the translation: Both $f$ and $x$ have to be chosen fresh; and even fresher than that, they can not occur in the body of the while loop we are translating either, because that would break the recursive call.

Further notice that the recursive functions are always called with a dummy argument. This is because they act as procedures, but since our syntax requires an argument for recursive definitions, we give a dummy argument.

### 3.2 Towards the $\lambda$-calculus

In order to make the interpretation of programs easier for ourselves, we decided to take an extra step in between the functional language and its interpretation: We use the lambda calculus with the `Y`-combinator.

The translation is straightforward in most cases, the only thing that is a bit tricky is the translation of recursive let-bindings. This is because the `Y`-combinator may look confusing at first, but when we take a closer look at it, we realise that it looks a lot like the fixed point iteration we presented in Section 2.1.1.

$$[\![x]\!] \mapsto x \tag{10}$$

$$[\![c]\!] \mapsto c \tag{11}$$

$$[\![\texttt{let } x := e_1 \texttt{ in } e_2]\!] \mapsto (\lambda x.[\![e_2]\!])\ [\![e_1]\!] \tag{12}$$

$$[\![\texttt{if } b \texttt{ then } e_1 \texttt{ else } e_2]\!] \mapsto [\![b]\!]\ [\![e_1]\!]\ [\![e_2]\!] \tag{13}$$

$$[\![f\ e_1 \dots e_n]\!] \mapsto [\![f]\!]\ [\![e_1]\!] \dots [\![e_n]\!] \tag{14}$$

$$[\![\texttt{let rec } f\ x := e_1 \texttt{ in } e_2]\!] \mapsto (\lambda f.[\![e_2]\!])\ (\texttt{Y}\ (\lambda f.\lambda x.[\![e_1]\!])) \tag{15}$$

In 13 we encode an if-expression as the application of a boolean value to the two branches. This just means that we assume Church-encoding of booleans.

Now for the `Y`-combinator: It is a so-called *fixed-point combinator*, this means it is a lambda-expression without any free variables which we can use to find the fixed point of the application of a recursive definition. The expansion of using the `Y`-combinator is, as mentioned before, substantially similar to the fixed-point iteration we presented, so we will not discuss it in detail here but refer any interested reader to [6].

$\beta$-reduction of function application in the $\lambda$-calculus is to insert the bound value instead of its identifier everywhere in the expression. In Sections 4.1 and 4.2, we will use this fact in order to rewrite $\mathcal{R}$`ml` expressions into something that is easier for us to interpret.

### 3.3 Interpreting it both ways

Finally, after having defined the translations, we would like to look at the interpretations of both the $\lambda$-calculus and `while` and see that the triangle shown in Figure 3.4 actually commutes.

To this end we will look at all the statements we have looked at for the translation from `while` to a functional language, and compare their direct interpretation with the interpretation of their translations to the $\lambda$-calculus.

We prove that the triangle commutes by induction. The base cases are the expressions; since these are the same in all languages and evaluated in the same way, they are trivially true.

For the proof we will allow ourselves to be sloppy with environments and instead of looking up variables simply replace occurrences of variable $x$ in statement $s$ by the value of $x$ before interpreting $s$.

Our induction hypothesis is that the interpretation of any sub-term is equivalent for direct interpretation (which we will write as $\mapsto^{\mathtt{W}}$) and translation to $\lambda$-calculus (which we will write as $\mapsto^{\lambda}$) with subsequent interpretation.

$$\texttt{skip} \ ; \ s \ \mapsto^{\mathtt{W}} [\![s]\!] \tag{16}$$
$$\texttt{skip} \ ; \ s \ \mapsto^{\lambda} [\![s]\!]$$

$$\texttt{skip} \ \mapsto^{\mathtt{W}} x_r \tag{17}$$
$$\texttt{skip} \ \mapsto^{\lambda} x_r$$

$$x := e \ ; \ s \ \mapsto^{\mathtt{W}} [\![s]\!][e/x]$$
$$x := e \ ; \ s \ \mapsto^{\lambda} (\lambda x.[\![s]\!]) \ e \tag{18}$$

$$x_r := e \ \mapsto^{\mathtt{W}} e \tag{19}$$
$$x_r := e \ \mapsto^{\lambda} e$$

$$x := e \ \mapsto^{\mathtt{W}} x_r \tag{20}$$
$$x := e \ \mapsto^{\lambda} x_r$$

$$(\texttt{if} \ e \ \texttt{then} \ s_1 \ \texttt{else} \ s_2) \ ; \ s_3 \ \mapsto^{\mathtt{W}} \texttt{if} \ e \ \texttt{then} \ [\![s_1 \ ; \ s_3]\!] \ \texttt{else} \ [\![s_2 \ ; \ s_3]\!] \tag{21}$$
$$(\texttt{if} \ e \ \texttt{then} \ s_1 \ \texttt{else} \ s_2) \ ; \ s_3 \ \mapsto^{\lambda} e \ [\![s_1 \ ; \ s_3]\!] \ [\![s_2 \ ; \ s_3]\!]$$

$$\texttt{if} \ e \ \texttt{then} \ s_1 \ \texttt{else} \ s_2 \ \mapsto^{\mathtt{W}} \texttt{if} \ e \ \texttt{then} \ [\![s_1]\!] \ \texttt{else} \ [\![s_2]\!] \tag{22}$$
$$\texttt{if} \ e \ \texttt{then} \ s_1 \ \texttt{else} \ s_2 \ \mapsto^{\lambda} e \ [\![s_1]\!] \ [\![s_2]\!]$$

(23)

$$(\texttt{while}\ e\ \texttt{do}\ s_1)\ ;\ s_2 \mapsto^{\mathtt{W}} sup_n(\llbracket\texttt{unfold-while-n-times}\ e\ s_1\rrbracket)\ ;\ \llbracket s_2\rrbracket$$

$$(\texttt{while}\ e\ \texttt{do}\ s_1)\ ;\ s_2 \mapsto^{\lambda} (\lambda f.f\ 0)\ (\mathtt{Y}\ (\lambda f.\lambda x.(e\ (\llbracket s_1\ ;\ f\ x\rrbracket)\ (\llbracket s_2\rrbracket))))$$

(24)

$$\texttt{while}\ e\ \texttt{do}\ s_1 \mapsto^{\mathtt{W}} sup_n(\llbracket\texttt{unfold-while-n-times}\ e\ s_1\rrbracket)$$

$$\texttt{while}\ e\ \texttt{do}\ s_1 \mapsto^{\lambda} (\lambda f.f\ 0)\ (\mathtt{Y}\ (\lambda f.\lambda x.(e\ (\llbracket s_1\ ;\ f\ x\rrbracket)\ (x_r))))$$

Most of the cases follow directly from the induction hypothesis, the base cases, or $\beta$-reduction of the $\lambda$-term in combination with the base cases and the induction hypothesis. We will leave it to the reader to convince himself of this.

The interesting cases are those with a looping construct, so let us have a look at those.

For the interpretation of `while`, we introduced two new constructs in the meta language: $sup_n$ and `unfold-while-n-times`.

The first one is obvious after reading Section 2.1.1: $sup_n$ takes a monotonic sequence (so a monotonic function of the natural numbers) and finds its supremum.

`unfold-while-n-times` does exactly what its name says. It takes a loop condition, $e$, a loop body, $s_1$, and a numeric argument, $n$, and computes $\underbrace{\texttt{if}\ e\ \texttt{then}\ s_1\ \texttt{else skip}\ ;\ \ldots\ ;\ \texttt{if}\ e\ \texttt{then}\ s_1\ \texttt{else skip}}_{n\text{ times}}$. It is worth noting, that this way of unfolding the loop saves us the trouble of introducing an environment for the interpretation. Since we assumed the environment to be properly updated after each statement, and all we do is produce a number of subsequent statements, this lies implicitly in the calculation.

The supremum of this sequence is the result of the `while` loop after it has terminated, or undefined if it does not terminate. It may not seem to the reader as if this sequence of unfolding `while` a number of times is monotonic, since the result may change both up and down, but this is not what we consider when we talk of a monotonic sequence in this case. The sequence is monotonic in how defined it is. When it is fully defined, the result will stay the same, no matter how many more times we unfold, since at that point the looping condition will be false in the environment. Because of this, we can also talk about the supremum as the fixed point of the `while`-loop.

So far so good, now what about the equivalence with the $\lambda$-term? For convenience let us first argue about the case of 24 and afterwards we will see that case 23 follows with ease.

We know that $\texttt{Y}\ f$ evaluates to $f\ (\texttt{Y}\ f)$, for simplicity's sake, we will, without loss of generality, assume that the evaluation is lazy; this means that we can take steps evaluating the recursion.

Let us look at the evaluation of the full term.

$$(\lambda f.f\ 0)(\texttt{Y}\ \ (\lambda f.\lambda x.(e\ ([\![s_1\ ;\ f\ x]\!])\ (x_r)))$$

$$\equiv_\beta$$

$$(\texttt{Y}\ \ (\lambda f.\lambda x.(e\ ([\![s_1\ ;\ f\ x]\!])\ (x_r))))\ 0$$

$$\equiv_\beta$$

$$(\lambda f.\lambda x.(e\ ([\![s_1\ ;\ f\ x]\!])\ (x_r)))\ (\texttt{Y}\ \ (\lambda f.\lambda x.(e\ ([\![s_1\ ;\ f\ x]\!])\ (x_r))))\ 0$$

$$\equiv_\beta$$

$$e\ ([\![s_1\ ;\ (\texttt{Y}\ \ (\lambda f.\lambda x.(e\ ([\![s_1\ ;\ f\ x]\!])\ (x_r))))\ 0]\!])\ x_r$$

At this point we notice, that we now have the exact same construct as unfolding the `while`-loop once: If $e$ is evaluated to `true`, we will keep unfolding until we reach the fixed point of the function that the `Y`-combinator is applied to. If the function diverges, the fixed point is undefined, which is the same result that we get when we evaluate a diverging `while`-loop.

If the sequence converges, we end up evaluating $s_1$ as many times as is necessary for $e$ to be evaluated to `false` (just as we did in the direct interpretation), and afterwards we return $x_r$. We note that $x_r$ is the return variable for any `while` program in our setting, so this is the same as the direct interpretation of `skip`. So in principle we have found the supremum of unfolding $\texttt{while}\ e\ \texttt{do}\ s_1\ \texttt{;}\ \texttt{skip}$, which is equivalent to the direct evaluation of 24.

The last step is to argue that this still works if the `while`-loop is the first statement in a sequence. When translating this, we will end up with

$$e\ ([\![s_1\ ;\ (\texttt{Y}\ \ (\lambda f.\lambda x.(e\ ([\![s_1\ ;\ f\ x]\!])\ (x_r))))\ 0]\!])\ s_2$$

Since we know that if the construct does not diverge, we will arrive at this expression at a point where $e$ evaluates to `false`, so we will take the $s_2$ branch. This is the same as interpreting $\texttt{if}\ e\ \texttt{then}\ s_1\ \texttt{else skip ;}\ s_2$ where we know that $e$ evaluates to false. This is exactly the statement we will end up with after unfolding the `while`-loop often enough to arrive at its fixed point in the direct interpretation. The rest follows from the induction hypothesis.

### 3.4 Adding probabilistic definitions

After having looked at the skeleton of the translations, now we want to make sure, that it all still works when adding probabilistic functions. In order to do this, we encapsulate all of our computations in the monadic structure presented in Section 2.2.2. This means that the result of a computation is no longer a definite values anymore, but rather a monadic value representing a probability distribution over results of the internal computation.

This is where the argument from Section 2.3 comes into play. This tells us that we can add probability to our languages without changing the interpretation of recursive definitions.

We also note that the probabilistic computations for both $\mathcal{R}$`ml` and `pwhile` are fully contained in the probabilistic functions `flip` and `random` $n$. This makes it easy to add probabilistic computations to the languages without changing the translations between them, since using probability boils down to applying a predefined function that is the same for all languages.

### 3.5 Implementing the translations in Coq

We tried implementing the approach of this Section in Coq, but it turns out that we were not skilled enough in the use of Coq in order to implement all of the translations of this section in the time given. What we have implemented is a translation from $\mathcal{R}$`ml` to `s`$\mathcal{R}$`ml`–a language of our own making that bears resemblance to the $\lambda$-calculus–and its interpretation using the $\omega$-cpo structure on the probability monad.

In order to implement the rest of the translations, we would use the `xhl` development for a Coq implementation of `pwhile`. We would then define the translation to $\mathcal{R}$`ml` according to what we presented in Section 3.1, and then prove that for all well-formed `pwhile`-programs, $p$, evaluating $p$ using the semantics implemented in `xhl` leads to the same value as translating $p$ to $\mathcal{R}$`ml` and interpreting the result using our implementation.

## 4 The Coq development (READY)

Need to fix the colour scheme for Coq-blocks.

According to Boileau, "what one understands well is expressed clearly". Few forms of expressing an idea are as concise as code written for a computer.

Since the main goal of the present project was to gain an actionable understanding of a number of concepts within the field of language design,

we chose to implement some of the theory present in the previous section in Coq.

Our development contains an implementation of a language based on $\mathcal{R}$ml, a number of definitions concerning the correctness of programs in said language, as well as an interpreter for such correct programs. Our goal was to implement $\mathcal{R}$ml according to its specification, but due to a number of wrestling matches with Coq's typechecker, we ended up deviating from what was presented in [1].

We will in the following call the language we actually implemented for $\mathcal{R}$ml*.

## 4.1 The definition of $\mathcal{R}$ml*

In order to write an interpreter for a language, we need to have a way to represent the language's abstract syntax. To this end we define a datatype using the Coq keyword `Inductive`.

```
Inductive Rml :=
| Var : (ℕ * Type) → bool → Rml
| Const : ∀ {A : Type}, A → Rml
| Let_stm : (ℕ * Type) → Rml → Rml → Rml
| If_stm : Rml → Rml → Rml → Rml
| App_stm : Type → Rml → Rml → Rml
| Let_rec : Type → Type → ℕ→ ℕ→ Rml → Rml → Rml
| Random : Rml → Rml
| Flip : Rml.
```

Figure 4.5: The inductive definition or $\mathcal{R}$ml* abstract syntax.

The names of the constructors are straightforward and easily recognised from the discussion of $\mathcal{R}$ml in Section 2.4. The arguments that each constructor takes are specific to our implementation so let us have a look at them.

`Var` takes a pair of a natural number and a type as well as a boolean. The number and the type represent the information to identify a variable with. We use natural numbers as identifiers because they are easy to work with, and the only requirement for an identifier is that there is an equality operation on its type. The second part of the pair is the type of the variable. The type information of a variable is needed for type-checking and interpre-

tation of an expression, because it enables us to check that the value of said variable has the type we expect it to have.

The second argument for the construction of a variable is a boolean value. This is used to indicate if the variable refers to something bound by a previous let binding, otherwise it is something we need for type-checking the function body of a recursive definition. We will go into more detail about how this is used in Section 4.2.

`Const` constructs a constant and is straightforward. The only explicit argument it takes is a Coq value; the implicit argument `A` is the type of said value.

At this point it should be noted, that in Coq functions are values as well, so it is possible to have a constant whose type is a function type. In other words: When we want to access a Coq-toplevel function in our $\mathcal{R}$`ml`* program, we can do this by encapsulating it in an $\mathcal{R}$`ml`* constant.

`Let_stm` is rather straightforward. The pair of a natural number and a type is the identifier of the variable being defined. The first argument of type `Rml` is the value of the variable, and the other is the expression the value is being used in.

For example `Let_stm` (`x`,`T`) $e_1$ $e_2$ would, in $\mathcal{R}$`ml` syntax, be written as `Let (x:T) :=` $e_1$ `in` $e_2$.

`If_stm` is straightforward. All this constructor takes are the condition and the two branches, all in form of `Rml` values.

`App_stm` takes an argument more or one less than we would intuitively expect. It takes one type as argument instead of either inferring the types or asking for both the function type and the result type.

The trick here is that when type-checking, we know the type we expect the whole application expression to have, but we do not know what to expect from the function or its argument. We therefore explicitly supply the intermediate type.

An example would be `App_stm` $\mathbb{N}$ (`Const is_even`) (`Const` 50), where we would typecheck the whole expression to have type `bool`, while we need the explicit $\mathbb{N}$ to check that the argument has the right type for the function.

This is because we can't construct on types in Coq, right...?

`Let_rec` is the most complicated of the linguistic constructs of $\mathcal{R}$`ml`*. It takes two types, two identifiers (natural numbers) and two `Rml` expressions.

The two types are rather straightforwardly the domain and co-domain of the recursively defined function.

The identifiers are the name of the function and the name of its argument, respectively. Both can be referred to in the function body, which is the first argument of type `Rml`.

The last argument should be an `Rml` value of the same type as the newly defined function's argument. This is probably our largest deviation from the original specification of $\mathcal{R}$`ml`. We do not have any expressions of the form `Let rec ...in ...`, instead we require to be recursive definitions to be of the form `Let rec f x := ...in f(...)`. This way of writing and using recursive functions has the exact same expressiveness as the common form, but it is easier to interpret in Coq.

Note that this way of defining `Let rec`s removes the necessity of having a function environment. Each function is used right where it is defined and can not be called in the rest of the program, so effectively there is no binding of anything happening in our so-called `Let_rec`.

**`Flip` and `Random`** are the two probabilistic functions mentioned in Section 2.4. We added these to our abstract syntax, since this makes it easier for us to interpret them as `Rml` terms instead of just as distributions. Effectively the types are the same, but having them as part of our abstract syntax makes them more readily accessible in $\mathcal{R}$`ml`* programs.

It is easy to see that all of the additional information we store in our abstract syntax tree is something that a parser would be able to infer and add. It would even be possible to transform the shape of a recursive let-binding at parse-time so it would fit our abstract syntax. This might introduce more recursive let-bindings, if the function is used in multiple places in the original code, but apart from this, the result would be equivalent.

## 4.2 Baby steps: The advent of s$\mathcal{R}$`ml`

In our first try to interpret $\mathcal{R}$`ml` directly, we were not able to figure out how to use environments for the interpretation of let-bindings in a way that Coq would accept. So instead of going from $\mathcal{R}$`ml` directly to its interpretation (as was suggested in [1]) we took a step in between. We defined a language of simplified $\mathcal{R}$`ml`, s$\mathcal{R}$`ml`. The main difference between $\mathcal{R}$`ml` and s$\mathcal{R}$`ml` is that s$\mathcal{R}$`ml` does not have let-bindings; there is, however, still a construct that is the same as the previous `Let_rec` which we call `sFix`, but as mentioned

earlier, this construction does not actually bind anything to an identifier, so there is no problem with environments here.

```
Inductive sRml {A : Type} :=
| sVar : ℕ → sRml
| sConst : A → sRml
| sIf : @sRml bool → sRml → sRml → sRml
| sApp : ∀ T, @sRml (T → A) → @sRml T → sRml
| sFix : ∀ B (nf nx : ℕ), @sRml A → @sRml B → sRml
| sRandom : (A = ℕ) → @sRml ℕ→ sRml
| sFlip : (A = bool) → sRml.
```

Figure 4.6: The inductive definition or sℛml abstract syntax.

Having this definition in place, the next step is to translate ℛml* to sℛml. This is done by the function `replace_all_variables_type`, which can be found in Appendix A.

We replace the variables by traversing our abstract syntax tree and building an environment that contains the identifier, type, and value of each variable that is bound. As we go, we construct the abstract syntax of an sℛml expression. Most of the cases are straightforward and an ℛml* construct is just replaced with the corresponding sℛml construct. The interesting cases are `Var` and `Let`.

**Variables** refer to let-bindings, so we want to get rid of them. But we still want to be able to have recursive function definitions, so we can not get rid of the variables that are the function name and argument in a recursive definition. This means we need a way to distinguish them. Here the extra boolean argument that the constructor `Var` takes comes into play:

- If it is `false`, we look up the variable in the environment and replace it with its value.
- If it is `true`, we know that we are currently looking at the body of a recursive definition and have no value with which to replace the variable, so we construct an `sVar` for it.

**Let-bindings** are what we want to get rid of in our simplification. We just saw how we get rid of variables referring back to bindings, so now for the actual binding part.

When we encounter a let-binding in our translation, we have to make sure that we can replace all variables in its body with their value by just looking into the environment. This means that we have to extend the environment with the newly bound variable and then replace all the variables in its body. And this is exactly what we do. We call the function to replace variables recursively, on the body of the let-binding with the newly extended environment.

## 4.3 Typing $\mathcal{R}$ml*

But how can we be sure that the translation from $\mathcal{R}$ml* to s$\mathcal{R}$ml actually works and does not "go wrong"? Having learned from Robin Milner [7] we know that in order to ensure this we must implement some typing rules and a way to check that a program is well-typed.

Before we commence the discussion of what it means for $\mathcal{R}$ml* or s$\mathcal{R}$ml expressions to be well-typed, we should mention that we decided on using the axiom of decidable equality for types. We could alternatively have defined an even deeper embedding of $\mathcal{R}$ml in Coq by defining a datastructure for $\mathcal{R}$ml-types instead of using Coq-types. This, however, would not have changed our outcome, so we decided against the additional work.

The rules we implemented are straightforward and can be found in Appendix B.

The attentive reader will have noticed that s$\mathcal{R}$ml is parameterised by a type representing the type of the constructed expression. This is in order to help us (and Coq) during the interpretation. The required types are consistent with the specified typing rules and are added during the translation from $\mathcal{R}$ml* to s$\mathcal{R}$ml. This means that in order to translate, the $\mathcal{R}$ml* expression we are trying to translate has to be well-typed; we assure this by our taking as argument a proof of the well-typedness of the expression to be translated.

The output of our translation is of type $\forall$ (A : Type), verified_srml A nil, whose definition can be seen in Figure 4.7. This means that the output of our translation effectively is a pair of the resulting s$\mathcal{R}$ml expression and a proof of its well-typedness. This, again, is to help us through the actual interpretation.

```
Inductive verified_srml (A : Type) (fl : seq (ℕ * Type)) : Type :=
    verified : ∀ y : sRml, srml_valid_type A fl y → verified_srml A fl
```

Figure 4.7: The inductive definition of the type verified_srml.

When defining the translation from $\mathcal{R}$ml* to s$\mathcal{R}$ml, we had to define a few helping lemmas along the way, in order to be able to prove everything correct. The most important one of these is weakening, which we had to prove for both the variable environment and the function environment for $\mathcal{R}$ml* as well as for the function environment for s$\mathcal{R}$ml.

Weakening means that even though we extend our environment (with a let-binding), the interpretation of the expression is still well-typed. We had a tiny problem along the way here: What if we bind a new variable with the same name but of a different type? Then the expression inside the new let-binding might be ill-typed.

We fixed this by defining the lookup of a variable to not only refer to its name, but also its type (cf. Figure 4.5 and the subsequent discussion). This way we can be sure that the variable we retrieve is of the right type.

## 4.4 But what does it mean? Interpreting s$\mathcal{R}$ml

After all this talk about how we define well-typedness of $\mathcal{R}$ml* and s$\mathcal{R}$ml, we want to finally make use of it.

Our interpretation of s$\mathcal{R}$ml is based heavily on the previously presented interpretations for `pwhile` and $\mathcal{R}$ml, and hence its result type is ∀ (`A` : `Type`), `distr R` (`Choice A`); or with other words: We interpret s$\mathcal{R}$ml expressions of type $\tau$ as distributions over the type of $\tau$. This should remind the careful reader of how we presented the interpretations of $\mathcal{R}$ml and `pwhile` in Sections 2.4 and 2.5.

The additional elements of `R` and `Choice` are due to the implementation of distributions in the Mathematical Components compliant Analysis Library that we mentioned in Section 2.2.2.

Since most of the set-up for the interpretations has been presented throughout Section 2, we will here only present a formal version of the semantics whose implementation can be found in the Coq definitions `ssem_aux` and `ssem`.

`uniform` and `flip` are functions defined by the `mathcomp` library that return probabilistic distributions represented in the same way as we represent them in our development.

`uniform` takes as input a list and outputs a uniform distribution over the elements in this list. Since $[\![e]\!]_{\texttt{env}}$ is a number and not a list, we defined the function `range`, which takes as input a natural number $x$ and returns a list of all numbers from 0 to $x$.

`flip` simply returns a uniform distribution of `true` and `false`.

$$
\begin{aligned}
[\![\texttt{sVar}\ n]\!]_{\texttt{env}} &= \texttt{env}(n) \\
[\![\texttt{sConst}\ a]\!]_{\texttt{env}} &= \texttt{unit}\ a \\
[\![\texttt{sIf}\ b\ c_1\ c_2]\!]_{\texttt{env}} &= \texttt{bind}\ [\![b]\!]\texttt{env}\ (\lambda x.\texttt{if}\ x \\
&\qquad\qquad\qquad\qquad\qquad\qquad \texttt{then}\ [\![c_1]\!]_{\texttt{env}} \\
&\qquad\qquad\qquad\qquad\qquad\qquad \texttt{else}\ [\![c_2]\!]_{\texttt{env}}) \\
[\![\texttt{sApp}\ \tau\ f\ x]\!]_{\texttt{env}} &= \texttt{bind}\ (\lambda t.\texttt{bind}\ (\lambda u.\texttt{unit}\ (t\ u)) \\
&\qquad\qquad\qquad\qquad\qquad\qquad [\![x]\!]_{\texttt{env}}) \\
&\qquad\qquad [\![f]\!]_{\texttt{env}} \\
[\![\texttt{sFix}\ \tau\ f\ x\ e_1\ e_2]\!]_{\texttt{env}} &= \texttt{bind}\ (\texttt{dlim}\ F^n)\ [\![e_2]\!]_{\texttt{env}} \\
&\qquad \text{where} \\
&\qquad F^0 \ = \ \texttt{dnull} \\
&\qquad F^{n+1} \ = \ (\lambda f'.\lambda x'.[\![e_1]\!]_{\texttt{env}}[x'/x][f'/f])\ F^n \\
[\![\texttt{sRandom}\ e]\!]_{\texttt{env}} &= \texttt{bind}\ (\lambda x.\texttt{uniform}\ (\texttt{range}\ x))\ [\![e]\!]_{\texttt{env}} \\
[\![\texttt{sFlip}\ ]\!]_{\texttt{env}} &= \texttt{flip}
\end{aligned}
$$

Figure 4.8: Denotational semantics of $\texttt{s}\mathcal{R}\texttt{ml}$.

Our interpretation relies on an environment that is used in order to interpret recursive definitions. This environment is empty to begin with and the values we store in it are of distribution type, so a lookup (the interpretation of a variable) will always yield a distribution, just what we expect the interpretation of an expression to be.

For the supremum in the interpretation of `sFix`, we rely on the `mathcomp` function `dlim`. This takes the limit of a monotonic sequence of distributions. Since the result of a recursive definition is either `dnull` or the result, the sequence will look something like `dnull, dnull, dnull, ... result, result, result, ...`; this means that the supremum of the sequence will be the result of the computation. When a function diverges, the sequence will consist solely of `dnull`s and hence the all-zero distribution is going to be the supremum.

## 5 Comparisons and future work (Draft 10.06.)

We had a brief look at FCF, a Foundational Proof Framework for Cryptography [8], in the beginning of the project, but decided that it would be more interesting to work on the basis of Paulin-Mohring's and Audebaud's ALEA library [1]. This is because $\mathcal{R}$`ml` seemed like more of a complete language than FCF.
In the beginning of the project we were aware of EasyCrypt and the `xhl` implementation of `pwhile` and found this interesting and worth exploring, in the hope of making a contribution to the development.
Instead our work compares $\mathcal{R}$`ml` and `pwhile` and shows that their semantics are equivalent.

## 6 Conclusion (Finished 12.06.)

We did good.

## References


[1] P. Audebaud and C. Paulin-Mohring, "Proofs of randomized algorithms in coq," *Science of Computer Programming*, no. 74, pp. 568–589, 2009.

[2] G. Barthe, F. Dupressoir, B. Grégoire, B. Schmidt, and pierre Yves Strub, "Computer-aided cyprotgraphy: some tools and applications," *Proc. All about Proofs, Proofs for All*, 2014.

[3] "Haskell/denotational semantics." https://en.wikibooks.org/wiki/Haskell/Denotational_semantics#Recursive_Definitions_as_Fixed_Point_Iterations.

[4] "Measure (mathematics)." https://en.wikipedia.org/wiki/Measure_(mathematics).

[5] S. J. Z. Béguelin, *Formal Certification of Game-Based Cryptographic Proofs*. PhD thesis, l'École nationale supérieure des mines de Paris, 2010.

[6] "The y combinator (slight return) or: How to succeed at recursion without really recursing." https://mvanier.livejournal.com/2897.html.

[7] R. Milner, "A theory of type polymorphism in programming," *Journal of Computer and System Sciences*, no. 17, pp. 348–375, 1978.

[8] A. Petcher, *A Foundational Proof Framework for Cryptography*. PhD thesis, Harvard University, 2015.

# Appendices

## A Translating $\mathcal{R}$ml* to s$\mathcal{R}$ml

This appendix presents the function translating $\mathcal{R}$ml* abstract syntax to s$\mathcal{R}$ml abstract syntax. The auxiliary function, which is the main workhorse, is defined recursively and written in the form of a proof. This is possible because Coq-proofs are program terms that can be run.

```
Fixpoint replace_all_variables_aux_type
        A (x : Rml) (env : seq (ℕ * Type * Rml))
        (fl : seq (ℕ * Type)) `{env_valid : valid_env env fl}
        `{x_valid : @rml_valid_type A (map fst env) fl x} {struct x}
  : verified_srml A fl.
Proof.
  (** Structure **)
  generalize dependent fl.
  generalize dependent env.
  generalize dependent A.
  induction x ; intros.
  (** Var *)
  {
    assert (List.In p (map fst env) ∨ List.In p fl)
    by (inversion x_valid ; subst ; auto).
    destruct p.
    assert (A = T) by (inversion x_valid ; subst ; reflexivity) ; subst.
    apply (@lookup (n,T) env fl env_valid H).
  }

  (** Const **)
  {
    assert (A0 = A) by (inversion x_valid ; subst ; reflexivity) ; subst.
    exists (sConst a).
    constructor.
  }

  (** Let-stm **)
  {
    assert (x1_valid : rml_valid_type p.2 (map fst env) fl x1)
    by (inversion x_valid ; subst ; assumption).

    pose (x1' := replace_all_variables_aux_type p.2 x1 env fl env_valid x1_valid).
    destruct x1' as [x1'].
    pose (x1'' := @sRml_to_rml p.2 x1').
```

```
  assert (x1''_simple : @rml_is_simple fl x1'').
  apply sRml_simple.
  assumption.

  assert (x1''_valid : @rml_valid_type p.2 (map fst env) fl x1'').
  apply sRml_valid.
  assumption.

  pose (@rml_to_sRml_l p.2 x1'' (map fst env) fl).

  assert (x2_valid : rml_valid_type A (p :: [seq i.1 | i ← env]) fl x2)
  by (inversion x_valid ; subst ; assumption).

  assert (env_valid' : valid_env ((p,x1'') :: env) fl)
  by (constructor ; assumption).

  refine (replace_all_variables_aux_type A x2 ((p,x1'') :: env) fl env_valid' x2_valid).
}

(** If-stm **)
{
  assert (x1_valid : rml_valid_type bool (map fst env) fl x1)
  by (inversion x_valid ; subst ; assumption).
  assert (x2_valid : rml_valid_type A (map fst env) fl x2)
  by (inversion x_valid ; subst ; assumption).
  assert (x3_valid : rml_valid_type A (map fst env) fl x3)
  by (inversion x_valid ; subst ; assumption).

  pose (b' := replace_all_variables_aux_type bool x1 env fl env_valid x1_valid).
  pose (m1' := replace_all_variables_aux_type A x2 env fl env_valid x2_valid).
  pose (m2' := replace_all_variables_aux_type A x3 env fl env_valid x3_valid).

  destruct b' as [b'].
  destruct m1' as [m1'].
  destruct m2' as [m2'].

  pose (b'' := sRml_to_rml b').
  pose (m1'' := sRml_to_rml m1').
  pose (m2'' := sRml_to_rml m2').

  refine (rml_to_sRml_l (If_stm b'' m1'' m2'') [seq i.1 | i ← env] fl).
  constructor ; eauto using sRml_simple.
  constructor ; eauto using sRml_valid.
}
```

```
(** App-stm **)
{
  assert (x1_valid : rml_valid_type (T → A) (map fst env) fl x1)
  by (inversion x_valid ; subst ; assumption).

  assert (x2_valid : rml_valid_type T (map fst env) fl x2)
  by (inversion x_valid ; subst ; assumption).

  pose (e1' := replace_all_variables_aux_type (T → A) x1 env fl env_valid x1_valid).
  pose (e2' := replace_all_variables_aux_type T x2 env fl env_valid x2_valid).

  destruct e1' as [e1'].
  destruct e2' as [e2'].

  pose (e1'' := sRml_to_rml e1').
  pose (e2'' := sRml_to_rml e2').

  refine (rml_to_sRml_l (App_stm T e1'' e2'') [seq i.1 | i ← env] fl).
  constructor ; eauto 2 using sRml_simple.
  constructor ; eauto 2 using sRml_valid.
}

(** Let rec **)
{
  pose (fl_x1 := [:: (n0, T), (n, T → T0) & fl]).

  assert (x1_valid : rml_valid_type A [seq i.1 | i ← env] fl_x1 x1)
  by (inversion x_valid ; subst ; assumption).

  assert (x2_valid : rml_valid_type T [seq i.1 | i ← env] fl x2)
  by (inversion x_valid ; subst ; assumption).


  assert (env_valid_x1 : valid_env env fl_x1)
  by (repeat apply extend_fl_still_valid ; assumption).

  pose (x1' := replace_all_variables_aux_type A x1 env fl_x1 env_valid_x1 x1_valid).
  assert (env_valid_x2 : valid_env env fl) by (repeat apply extend_fl_still_valid ; assumption).

  pose (x2' := replace_all_variables_aux_type T x2 env fl env_valid_x2 x2_valid).

  destruct x1' as [x1'].
  destruct x2' as [x2'].
```

```
    assert (A = T0) by (inversion x_valid ; subst ; reflexivity) ; subst.

    exists (sFix T n n0 x1' x2').
    constructor ; assumption.
  }

  (** Random **)
  {
    assert (inner_x_valid : rml_valid_type ℕ(map fst env) fl x)
    by (inversion x_valid ; assumption).

    pose (x' := replace_all_variables_aux_type ℕx env fl env_valid inner_x_valid).

    assert (type_eq : A = ℕ) by (inversion x_valid ; reflexivity).

    destruct x' as [x' x'_valid].

    exists (sRandom type_eq x').
    constructor ; assumption.
  }

  (** Flip **)
  {
    assert (A = bool) by (inversion x_valid ; reflexivity).
    exists (sFlip H).
    constructor.
  }
Defined.

Definition replace_all_variables_type A (x : Rml)
           `{x_valid : rml_valid_type A nil nil x} :=
  @replace_all_variables_aux_type A x nil nil (env_nil nil) x_valid.
```

## B $\mathcal{R}$ml* typing rules

```
Inductive rml_valid_type : Type → seq (ℕ * Type) → seq (ℕ * Type) → Rml →
Prop :=
| valid_var : ∀vl fl p,
    List.In p vl →
    rml_valid_type p.2 vl fl (Var p false)

| valid_fun_var : ∀vl fl p,
    List.In p fl →
    rml_valid_type p.2 vl fl (Var p true)

| valid_const : ∀(A : Type) vl fl (c : A),
    rml_valid_type A vl fl (@Const A c)

| valid_let : ∀A vl fl p a b,
    @rml_valid_type p.2 vl fl a →
    @rml_valid_type A (p :: vl) fl b →
    rml_valid_type A vl fl (Let_stm p a b)

| valid_if : ∀A vl fl b m1 m2,
    rml_valid_type bool vl fl b →
    rml_valid_type A vl fl m1 →
    rml_valid_type A vl fl m2 →
    rml_valid_type A vl fl (If_stm b m1 m2)

| valid_app : ∀A vl fl (B : Type) e1 e2,
    rml_valid_type (B → A) vl fl e1 →
    rml_valid_type B vl fl e2 →
    rml_valid_type A vl fl (App_stm B e1 e2)

| valid_let_rec : ∀A vl fl B nf nx e1 e2,
    @rml_valid_type A vl ((nx,B) :: (nf,B → A) :: fl) e1 →
    @rml_valid_type B vl fl e2 →
    rml_valid_type A vl fl (Let_rec B A nf nx e1 e2)

| valid_random : ∀vl fl e,
    rml_valid_type ℕvl fl e →
    rml_valid_type ℕvl fl (Random e)

| valid_flip : ∀vl fl,
    rml_valid_type bool vl fl Flip.
```

## C Example - Error: Stack Overflow.

```
Fixpoint replace_all_variables_aux_type
        A (x : Rml) (env : seq (ℕ * Type * Rml))
        (fl : seq (ℕ * Type)) `{env_valid : valid_env env fl}
        `{x_valid : ·rml_valid_type A (map fst env) fl x} : ·sRml A

with replace_all_variables_aux_type_const
       A0 A a (env : seq (ℕ * Type * Rml))
       (fl : seq (ℕ * Type)) `{env_valid : valid_env env fl}
       `{x_valid : ·rml_valid_type A0 (map fst env) fl (Const A a)} : ·sRml A0
with replace_all_variables_aux_type_let
       A p x1 x2 (env : seq (ℕ * Type * Rml))
       (fl : seq (ℕ * Type)) `{env_valid : valid_env env fl}
       `{x_valid : ·rml_valid_type A (map fst env) fl (Let_stm p x1 x2)} : ·sRml A
with replace_all_variables_aux_type_fun
       A T p x (env : seq (ℕ * Type * Rml))
       (fl : seq (ℕ * Type)) `{env_valid : valid_env env fl}
       `{x_valid : ·rml_valid_type A (map fst env) fl (Fun_stm T p x)} : ·sRml A
with replace_all_variables_aux_type_if
       A x1 x2 x3 (env : seq (ℕ * Type * Rml))
       (fl : seq (ℕ * Type)) `{env_valid : valid_env env fl}
       `{x_valid : ·rml_valid_type A (map fst env) fl (If_stm x1 x2 x3)} : ·sRml A
with replace_all_variables_aux_type_app
       A T x1 x2 (env : seq (ℕ * Type * Rml))
       (fl : seq (ℕ * Type)) `{env_valid : valid_env env fl}
       `{x_valid : ·rml_valid_type A (map fst env) fl (App_stm T x1 x2)} : ·sRml A
with replace_all_variables_aux_type_let_rec A T T0 n n0 x1 x2 (env : seq (ℕ * Type * Rml))
     (fl : seq (ℕ * Type)) `{env_valid : valid_env env fl}
     `{x_valid : ·rml_valid_type A (map fst env) fl (Let_rec T T0 n n0 x1 x2)} : ·sRml A.
Proof.
  (** Structure **)
  {
    induction x ; intros ; refine (sVar (0,A)).
  }

  all: refine (sVar (0,A)).
Defined.
```