Technical report

Lasse Letager Hansen
Kira Kutscher

May 26, 2019

Contents

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 [...], and an imperative one, as in [...].

2 Theory and existing frameworks (Finished: 27.05.)

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 2.3) and the imperative `pwhile` (Section 2.4).

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 ω-complete partial orders, which we will use in order to interpret general recursion and probabilistic programs.

Definition 1. *ω-complete partial order (ω-cpo)*
An ω-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 ω-cpos to interpret recursion, let us first have a look at some interesting things that our definition entails.

A monotonic sequence on an ω-cpo X can be viewed as a monotonic function $f : \mathbb{N} \xrightarrow{m} X$ where $f(n)$ is the nth 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 ω-cpo:

Consider an operator $F : X \xrightarrow{m} X$ on some ω-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 ω-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 $\texttt{fix}\ F = F(\texttt{fix}\ F)$ if F is continuous.

For an ω-cpo with underlying set B we can also define an ω-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 ω-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 ω-cpo, so according to the above discussion functions will have

an ω-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 $f(n) = n!$. 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 ω-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 [...].

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 μ.

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:

Is this understood correctly?

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

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 τ can

be expressed with the type $(\tau \to [0,1]) \to [0,1]$. An interpretation of a probabilistic term whose type is τ can now be understood as a measure on type τ, 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 happening, 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 of the result of an actual pseudorandomised run of our program producing a result within the input distribution.

Or something like that... I am very unsure about what exactly we get when we input an initial distribution... help?
It seems we get the integral over the joint distribution (given that both distributions are independent, which we will have to assume).

Geistesblitz! The input is a function that expresses the probability of something being memeber of some set. The characteristic function of a set is non-probabilistic and hence the result of inputting this will give the exact probability of our result being part of that set. If we, for example, have a set where both `true` and `false` are memebers with probability $\frac{1}{2}$, the result of inputting this into, e.g., `unit true` would be the probability of the value (`true`) being a member of said set.

2.2.2 A monadic interpretation

We can represent measures with a monadic structure. This is sensible since once we go measure we never go back; once probability is involved in a computation, we will not get rid of it again.

Add something about how the measure type now is called `M`.

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

Footnote too facetious?

[1] We refer the reader who is unfamiliar with monads (or requires proof of the authors' knowledge thereof) to an earlier work by the authors, 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/`

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

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 function binding 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.

Is this the right word for the `fun`-keyword)

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. μ and M are the usual arguments to a `bind` operation, μ 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 τ inside the monad before we bind 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 μ 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 σ to get something in [0,1], which would satisfy the output type for the function we apply μ to.

The final function takes the input of type τ, that μ will give it, uses M on 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 μ, transform it with M, and lastly transform it with f, once that is supplied.

Add an example.

2.3 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.

$\mathcal{R}$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 a `random(`n`)` or `flip` function giving a uniform distribution of natural numbers between 0 and n, and `true` and `false`, respectively.

check this sentence

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:

Is this parenthetical needed?

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

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.

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`.

quote

The monadic interpretation $[\![e]\!] : \texttt{M}\tau$ of a term $e : \tau$ is given in figure 2.3.

What the hell is wrong with figure numbering?

2.4 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. A standard approach to this kind of proofs is based on

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

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

so-called games; in EasyCrypt cryptographic algorithms as well as games 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:

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

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. We give a full formal semantics in Figure .

Insert reference to semantics-figure

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

Write the semantics for `pwhile`.

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.

is this the right word choice? It doesn't seem that that's what a null distribution is. But the keyword use is `dnull`.

3 Our approach (Ready draft: 03.06.)

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.

(1) $exp \ ::= \ x|n|\texttt{true}|\texttt{false}|f \ x$

(2) $stm \ ::= \ \texttt{skip}|x := e|\texttt{if} \ e \ \texttt{then} \ s_1 \ \texttt{else} \ s_2|\texttt{while} \ e \ \texttt{do} \ s|s_1; s_2$

The syntax of our functional language is the same as $\mathcal{R}$`ml` modulo the pre-defined probabilistic functions.

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.

$$\texttt{skip}\ ;\ s \mapsto [\![s]\!] \tag{3}$$

$$\texttt{skip} \mapsto x_r \tag{4}$$

$$x := e\ ;\ s \mapsto \texttt{let}\ x := e\ \texttt{in}\ [\![s]\!] \tag{5}$$

$$x_r := e \mapsto e \tag{6}$$

$$x := e \mapsto x_r \tag{7}$$

$$(\texttt{if}\ e\ \texttt{then}\ s_1\ \texttt{else}\ s_2)\ ;\ s_3 \mapsto \texttt{if}\ e\ \texttt{then}\ [\![s_1\ ;\ s_3]\!]\ \texttt{else}\ [\![s_2\ ;\ s_3]\!] \tag{8}$$

$$\texttt{if}\ e\ \texttt{then}\ s_1\ \texttt{else}\ s_2 \mapsto \texttt{if}\ e\ \texttt{then}\ [\![s_1]\!]\ \texttt{else}\ [\![s_2]\!] \tag{9}$$

$$(10) \quad (\texttt{while}\ e\ \texttt{do}\ s_1)\ ;\ s_2 \mapsto \texttt{let rec}\ f\ x := \texttt{if}\ e \\ \texttt{then}\ [\![s_1\ ;\ f\ x]\!] \\ \texttt{else}\ [\![s_2]\!] \\ \texttt{in}\ f\ 0$$

$$(11) \quad \texttt{while}\ e\ \texttt{do}\ s_1 \mapsto \texttt{let rec}\ f\ x := \texttt{if}\ e \\ \texttt{then}\ [\![s_1\ ;\ f\ x]\!] \\ \texttt{else}\ x_r \\ \texttt{in}\ f\ 0$$

Note that in 10 and 11 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 Translation from Rml to typed λ-calculus

This section is preliminary and needs either huge changes or deletion before the report is finalised.

Rml	@sRml A
Var (x, A)	sVar x
Const A c	sConst c
Let (x, T) e_1 e_2	e_2'
If b e_1 e_2	sIf b' e_1' e_2'
Fun (x, T) e	sFun S (x, T) e'
App T e_1 e_2	sApp T e_1' e_2'
Let rec T S f x e_1 e_2	sFix T S f x e_1' e_2'

@sRml A	typed λ-calculus	distr
sVar x	$x : A$	dunit (lookup
sConst c	$c : A$	dunit c
sFun S (x, T) e'	$(\lambda x : T, e : S) : T \to S$	$\backslash x \to$ dunit
sApp T e'_1 e'_2	$(e_1 : T \to A)$ $(e_2 : T) : A$	$e'_1 \gg\!= \backslash e''_1 \to e'_2 \gg\!= \backslash e''_2$
sFix T S f x e'_1 e'_2	$(\lambda f : T \to S, e_2)$ $(Y$ $(\lambda f : T \to S, (\lambda x : T, e_1\ x))) : A$	$(\backslash f \to e_2)(\backslash x \to$ dlim

3.3 Example: Fact

Expression:

$$\begin{aligned}&\texttt{Let\_rec}\ (f, \mathbb{N} \to \mathbb{N})\ (x, \mathbb{N})\\ &\qquad (\texttt{If}\ (x = 0)\ (1)\ (x * f\ (x - 1)))\\ &\qquad (f\ 3)\end{aligned}$$

Simple form:

$$\begin{aligned}&\texttt{sFix}\ f\ x\\ &\quad (\texttt{sIf}\ (x = 0)\ 1\ (x * f\ (x - 1)))\\ &\quad (f\ 3)\end{aligned}$$

Typed lambda expressions:

$$(\lambda f : \mathbb{N} \to \mathbb{N}, f\ 3)\ (Y(\lambda f : \mathbb{N} \to \mathbb{N}, \lambda x : \mathbb{N}, (x = 0)\ (1)\ (x * f\ (x - 1))))$$

Distribution:

$$\backslash\texttt{dlet\_}(f \leftarrow \texttt{dunit}\ (\lambda x, \backslash\texttt{dlim}(\backslash\texttt{ubn}\ (\lambda f, (x = 0)\ (1)\ (x * f\ (x - 1)))))\ (f\ 3)$$

3.4 Interpreting λ-calculus in the space of ω-cpos

What do ω-cpos have to do with this?

3.5 Interpreting `while` directly

This should probably mainly refer back to the interpretation of `pwhile`.

3.6 All translations (forward)

What is the point of this section?

Rml	@sRml A
Var (x, A)	sVar x
Const A c	sConst c
Let (x, T) e_1 e_2	e_2'
If b e_1 e_2	sIf b' e_1' e_2'
Fun (x, T) e	sFun S (x, T) e'
App T e_1 e_2	sApp T e_1' e_2'
Let rec T S f x e_1 e_2	sFix T S f x e_1' e_2'

@sRml A	typed λ-calculus	distr
sVar x	$x : A$	dunit (lookup x)
sConst c	$c : A$	dunit c
sFun S (x, T) e'	$(\lambda x : T, e : S) : T \to S$	
sApp T e_1' e_2'	$(e_1 : T \to A)$ $(e_2 : T) : A$	$e_1' \gg\!= \backslash e_1'' \to e_2' \gg\!= \backslash e_2'' \to$ ret $(e_1'$ $e_2')$
sFix T S f x e_1' e_2'	$(e_2 : (T \to S) \to A)$ $(Y$ $(e_1 : (T \to S) \to T \to S) : T \to S) : A$	$(\backslash f \to e_2)(\backslash x \to$ `dlim` (`ubn` e_1 $x))$

Problem we get a function $f : A \to distr\ B$, instead of $f : distr\ (A \to B)$ (solution bind)

4 Our contribution (Draft: 03.06; Finished 10.06.)

4.1 Our implementation of $\mathcal{R}$`ml`

The data structure used to represent Rml terms is as follows:

```
Inductive Rml :=
| Var : (ℕ * Type) → Rml
| Const : ∀ (A : Type), A → Rml
| Let_stm : (ℕ * Type) → Rml → Rml → Rml
| Fun_stm : Type → (ℕ * Type) → Rml → Rml
| If_stm : Rml → Rml → Rml → Rml
| App_stm : Type → Rml → Rml → Rml
| Let_rec : Type → Type → ℕ→ ℕ→ Rml → Rml → Rml.
```

We use all cog types, as possible types of Rml expressions, since there are no real restrictions on the types. We encode variables, as a type and a natural number, so two variables are the same only if they have the same number and refer to the same type.

We have defined a relation `well_formed`, that checks that no variables are escaping the scope of an Rml program, that is there is always a binding for an expression of type `Var` p. We furthermore define a relation `rml_valid_type`, which checks that a given Rml expression can be typed under a given type. We have shown that if a Rml program is valid then it is well formed. We have then constructed a simplified form of Rml called sRml (for simple Rml), to make it easier to reason about and evaluate expressions, with the following data structure:

```
Inductive sRml {A : Type} :=
| sVar : ℕ → sRml
| sConst : A → sRml
| sFun : ∀ C (p : ℕ * Type), A = (p.2 → C) →
·sRml C → sRml
| sIf : · sRml bool → sRml → sRml → sRml
| sApp : ∀ T, · sRml (T → A) → ·sRml T → sRml
| sFix : ∀ B (nf nx : ℕ), · sRml (B → A) → ·sRml B →
sRml.
```

That is Rml where we remove expressions with variables, from `let_stm` statements (not `let_rec` statements). We then show that given a valid typing of an Rml expression, we can simplify that expression, and maintain the valid typing (under the same type). With this we can make an interpreter from an interpreter of sRml, which can be constructed as (for continuations). We have a similar function for Rml, using the posibility distributions as interpretations. We see similar patterns arising, since both interpretations are monadic.

5 Comparisons and future work (Finished 12.06.)

6 Conclusion (Finished 12.06.)

7 Appendix

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.
```