Exploring interpretations of probabilistic programs using measure theory in Coq

En undersøgelse af probabilistiske programmer med brug af målteori i Coq

Kira Kutscher - 201509720
Lasse Letager Hansen - 201508114

A thesis (15 ECTS) presented for the degree of
Bachelor of Computer Science
for Kira Kutscher

and

A term project (10 ECTS)
for Lasse Letager Hansen

Under the supervision of
Bastiaan Alexander Wilhelmus Spitters

Department of Computer Science
Aarhus University

17th of June 2019

Abstract

Probabilistic algorithms are widely used in computer science, especiallly in the fields of cryptography and security. Because of their many applications, reliable proofs of the correctness and security of such algorithms are desirable.

In this work we explore some existing designs and implementations of probabilistic languages and frameworks for proofs of probabilistic algorithms in Coq. We present a functional and an imperative language ($\mathcal{R}$`ml` and `pwhile`), discuss the theory behind their respective interpretations, and propose a translation from `pwhile` to $\mathcal{R}$`ml`, which we prove to preserve the semantics of the original program.

Finally we implement an interpretation of $\mathcal{R}$`ml` in Coq as a first step to a verified implementation of our translation from `pwhile` to $\mathcal{R}$`ml`.

Personal note

The present work is the result of a joint project between two people in different places of their educational journey. Furthermore the time to be spent on the project differed between the two of us. We will here give a quick overview of what each of us had to learn in order to produce the present result.

Kira Kutscher worked on this as a bachelor project of 15 ECTS (∼420 hours). I had only rudimentary knowledge of probability theory and none of measure theory; both I had to study in order to understand the semantics as well as the interpretation of probabilistic definitions. Also new to me were domain theory and fixed point iterations as well as denotational semantics. Lastly I did not know anything about category theory, which was part of our work although it is not explicitly mentioned in the report.

Lasse Letager Hansen worked on this as a masters level research project of 10 ECTS (∼280 hours). I have an entry level understanding of category theory, and deep knowledge of probability theory, but no prior understanding of measure theory, domain theory or the interpretation of probabilistic definition. I had a good understanding of the Coq proof assistant before this project, and now I have gained knowledge of how to do a large project in Coq, and experienced some of the challenges with doing deep embeddings and the limitations of Gallina.

Contents

1 Introduction

Probabilistic algorithms are widely used in computer science, especially in the fields of cryptography and security. Because of their many applications, reliable proofs of the correctness and security of such algorithms are desirable.

A number of developments exists that attempt to make this kind of proofs easier and more secure, but with such developments the next question is if the implementation is correct and the logic consistent. In the present work we will focus on developments in the Coq proof assistant, since it is one of the most widely used and most probably correct proof assistants currently in existence.

1.1 The Coq proof assistant

The Coq proof assistant was developed in 1984 by Coquand and Huet [1] and its logic, originally based on the Calculus of Constructions, was extended to the Calculus of Inductive Constructions by Christine Paulin in 1991. It has been under constant development ever since it was first published and many bugs have been found and fixed over time, making Coq one of the most reliable and trusted proof environments.

The reader is not expected to have an understanding of Coq, though it may prove helpful in Section 4, which deals with our Coq development.

1.2 Previous work

For our development we oriented ourselves by the landmarks of previous frameworks that have been built for proofs of probabilistic algorithms in Coq. We looked at three different probabilistic languages and the corresponding proof-frameworks; two of them are functional, $\mathcal{R}$`ml` [2] and FCF [3], and the last one, `pwhile`, is imperative [4].

Both FCF and $\mathcal{R}$`ml` are developments that have been implemented in Coq, but are not available to use. The FCF development is implemented in an older version of Coq, so it does not run on the current version; and $\mathcal{R}$`ml` has been implemented, but the development is not available.

`pwhile` originated in the Coq development CertiCrypt, then took wings as the language used for probabilistic algorithms in EasyCrypt, and is currently being brought back to Coq with the `xhl` development (cf. Section 2.5).

1.3 Roadmap

This work builds upon the `xhl` implementation of `pwhile` as well as the description of $\mathcal{R}$`ml` that can be found in [2]. We aim at analysing differences and similarities between both languages and proving them equivalent.

We will begin our journey with introducing the theory necessary to understand both languages (Section 2); this includes the domain-theoretic interpretation of general recursion (Section 2.1.1) as well as a monadic interpretation of probabilistic programs based on concepts from measure theory (Section 2.2). Once the theory is in place we will have a closer look at $\mathcal{R}$`ml` (Section 2.4) and `pwhile` (Section 2.5).

We will then develop an approach to translating programs from `pwhile` to $\mathcal{R}$`ml` and proving their interpretations equivalent (Section 3).

Lastly we will present our own implementation of $\mathcal{R}$`ml` and its interpretation (Section 4); we will compare it to the previously discussed approaches and reflect on how to meaningfully expand the development in Section 5. Afterwards Section 6 wraps up the journey, presenting our conclusion.

2 Theory and existing frameworks

Our work is concerned with interpretations of possibly non-terminating probabilistic computations encoded in Coq. As mentioned earlier, Coq is widely trsuted; it is, however, also known to be notoriously difficult to program in due to a strict type system that requires determinism and certain termination of all programs written in it. Obviously these are not the optimal conditions for running probabilistic algorithms that may not terminate.

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

In this section we present a way of representing probability distributions using a monad and show how we can use this in order to interpret probabilistic definitions. Furthermore we will see the domain theoretic approach of interpreting general recursion by performing a fixed-point iteration.

More specifically we will have a look at how to use complete partial orders to interpret general recursion in Section 2.1, and then concern ourselves with representing the result of a probabilistic computation using a monadic interpretation of probability measures in Section 2.2. Afterwards we will move on to presenting two different developments concerning 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 reflexive, antisymmetric and transitive. The order is partial when the ordering relation is not defined on every pair of elements in the set.

There is a whole range of different completeness properties that a poset can have. For our purposes we will need ω-complete partial orders for the interpretation of general recursion.

Definition 1. *ω-complete partial order (ω-cpo)*
An ω-cpo is a partially ordered set, Σ, that has a distinct least element, 0_Σ, and where there exist least upper bounds, $\texttt{lub}_\Sigma$, 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 for n larger than the length of the sequence, if the sequence is finite).

There is a standard way of defining fixed point iterations on an ω-cpo[2]: 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 with co-domain B from any domain A. To define this we recall what requirements there are for a set to be an ω-cpo: We need an ordering relation, a least element, and a least upper bound operation. Those can be

defined as follows:

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

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 $f(n) = n!$ [5]. The recursive definition of this is:

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

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

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

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 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 on which F operates.

2.2 Interpreting probabilistic definitions

The most straightforward interpretation of a probabilistic definition is a probability distribution of the definition's results. In order to define this interpretation we need a way to represent probability distributions; for this we will use probability measures as described in [2].

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 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[6]:

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

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

The integral perspective still satisfies all the requirements that a function has to fulfil in order to be a measure. This is fairly easy to see when

we bear in mind that the integral is over strictly non-negative functions; 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?

The cool thing is that being able to measure function whose co-domain is the real numbers in the unit interval (in other words functions of type $\tau \to [0,1]$ for some type τ) 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 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 transforming distributions: 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 pseudo-randomised 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 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.

But the function we measure is not constrained to being the characteristic function of a finite set. We can use functions specifying infinite sets, or sets that contain values with some probability. We could, for example, have a set that contains `true` with probability p and `false` with probability q. The characteristic function of this set is

$$\mathbb{I}_X = \begin{cases} \texttt{true} \mapsto p \\ \texttt{false} \mapsto q \end{cases}$$

In order to find out what the probability is of a uniformly random choice between `true` and `false` being in this set, we apply the measure representation of `flip` to $\mathbb{I}_X$. The result is the integral over the joint probability distribution of `flip` and $\mathbb{I}_X$.

2.2.2 A monadic interpretation

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

Since we are using a monad over the type τ to represent a measure on τ, 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 [2] and satisfy the usual monadic properties[1]. They are also used by the Mathematical Components compliant Analysis Library[2] for Coq in order to represent distributions.

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

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 abstraction 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 distribution 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 (pseudo-)random

[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/`

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 g are the usual arguments to a `bind` operation, μ being the initial value and g 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 cannot 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, which μ will give it, applies g 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 μ, transform it with g, 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, μ, 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 g, that takes a natural number and returns the probability of it being even. If now we apply the result of $\texttt{bind}\ \mu\ g$ to, let's say, the characteristic function of `false`, the following computation will be carried out:

$\texttt{fun}\ (x\ :\ \tau) \Rightarrow g\ x\ f$ is now the function that takes a natural number and returns the probability of it being odd. Applying μ to this function yields the probability of a pseudo-random interpretation of μ being an odd number; and this is exactly the meaning of

$$(\texttt{bind}\ \mu\ g)\ (\texttt{fun}\ (x\ :\ \ \texttt{bool}) \Rightarrow \texttt{if}\ x\ \texttt{then 0 else 1})$$

2.3 Putting together recursion and probability

What now if we want both probability and a way of interpreting general recursion? Then we need the ω-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 ω-cpo also have an ω-cpo structure [2, p. 584]. Because a probability measure is an integral over strictly non-negative functions, it is a monotonic function. It is a function to [0,1], which is an ω-cpo with the 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 ω-cpo, which is 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 Philippe Audebaud and Christine Paulin-Mohring[2].

$\mathcal{R}$ml is a simple functional language whose interpretation allows for probabilistic algorithms. The language itself does not contain probabilistic expressions, but rather makes use of the functions `random(`n`)` and `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, their semantics is a distribution. Their typing is therefore:

$$\texttt{random} \ : \ \mathbb{N} \to \texttt{M} \ \mathbb{N}$$
$$\texttt{flip} \ : \ \_ \to \texttt{M} \ \mathbb{B}$$

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 Audebaud and Paulin-Mohring don't give a clear definition of what a constant can be, it makes sense to assume that it can be any Coq term. This is because their proposed approach to interpreting $\mathcal{R}$ml in Coq is a shallow embedding, which means that it uses Coq's type system and values instead of defining its own.

According to [2], 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` $f\ x_1\ \ldots\ x_n = e$. Recursive functions can be defined with the keyword `let rec`. Here, again, we could make use of the fact that the paper proposes a shallow embedding and use Coq top-level functions.

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

$\mathcal{R}$`ml` term $e\ :\ \tau$	Interpretation $[e]\ :\ \mathtt{M}\tau$
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 [2].

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 regarding cryptographic algorithms and protocols [4]. For the implementation of algorithms, it makes use of 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.

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

`pwhile` consists of the following expressions and commands:

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

The embedding of `pwhile` in Coq is a 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.

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

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 [7].

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 the null-distribution over memories.

3 Equivalence of $\mathcal{R}$`ml` and `pwhile`

The goal of the present project was to explore probabilistic languages, focusing on $\mathcal{R}$`ml` and `pwhile`, their respective interpretations, theoretic background, and similarities and differences between them. Our approach was to define a translation from `pwhile` to $\mathcal{R}$`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.

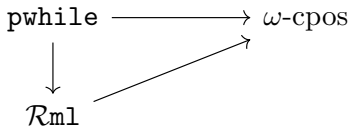

Figure 3.3: The languages we work with and the direction of the translations between them.

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 be implicit about typing of expressions, since the addition of types would not have any influence on the translations we define.

For the translations, we will first have a look at deterministic versions of both languages in order to define the main part of the translations (Sections 3.1, 3.2). For a deterministic version of `pwhile` we will look at `while`, and for $\mathcal{R}$`ml`, we will look at a simple functional language with recursive definitions, $\text{fun}_{\texttt{Fix}}$. Additionally we will take an extra step in the interpretation of the functional language that we will use, translating it to the λ-calculus with `Y`-combinator first. 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):

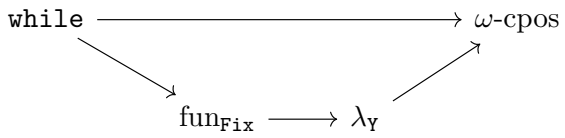

Figure 3.4: A deterministic version of Figure 3.3 with an extra step added.

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 `while` to a simple functional language resembling $\mathcal{R}$`ml`, $\text{fun}_{\texttt{Fix}}$. 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 probabilistic computations.

First let us have a look at the grammar for `while`:

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

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; nothing new is introduced.

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

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 depending on its first statement. We will in the following write the translation of a statement s in the `while` language 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) $$(\texttt{while}\ e\ \texttt{do}\ s_1)\ ;\ s_2 \ \mapsto \texttt{let rec}\ f\ x := \texttt{if}\ e\ \texttt{then}\ [\![s_1\ ;\ f\ x]\!]\ \texttt{else}\ [\![s_2]\!]\ \texttt{in}\ f\ 0$$

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

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 cannot occur in the body of the while loop we are translating either, because that would break the recursive call.

Further, note that the recursive functions are always called with a dummy argument. This is because they act as procedures and do not use their argument; the argument is, however, required by our syntax, so we give a dummy argument.

3.2 Towards the λ-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.

$$
\begin{aligned}
(10) \qquad [\![x]\!] &\mapsto x \\
(11) \qquad [\![c]\!] &\mapsto c \\
(12) \qquad [\![\texttt{let}\ x := e_1\ \texttt{in}\ e_2]\!] &\mapsto (\lambda x.[\![e_2]\!])\ [\![e_1]\!] \\
(13) \qquad [\![\texttt{if}\ b\ \texttt{then}\ e_1\ \texttt{else}\ e_2]\!] &\mapsto \texttt{if}\ [\![b]\!]\ \texttt{then}\ [\![e_1]\!]\ \texttt{else}\ [\![e_2]\!] \\
(14) \qquad [\![f\ e_1 \dots e_n]\!] &\mapsto [\![f]\!]\ [\![e_1]\!] \dots [\![e_n]\!] \\
(15) \qquad [\![\texttt{let rec}\ f\ x := e_1\ \texttt{in}\ e_2]\!] &\mapsto (\lambda f.[\![e_2]\!])\ (\texttt{Y}\ (\lambda f.\lambda x.[\![e_1]\!]))
\end{aligned}
$$

For simplicity, we decided to use a typed λ-calculus with the type of booleans and an if-primitive, which we use in 13.

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 [8].

β-reduction of function application in the λ-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 λ-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 λ-calculus. The proof that the triangle commutes is 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 λ-calculus (which we will write as $\mapsto^{\lambda}$) with subsequent interpretation.

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

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

$$x := e\ ;\ s\ \mapsto^{\mathtt{W}} [\![s]\!][e/x]$$

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

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

(20)
$$x := e \;\mapsto^{\mathtt{w}}\; x_r$$
$$x := e \;\mapsto^{\lambda}\; x_r$$

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

(22)
$$(\texttt{if } e \texttt{ then } s_1 \texttt{ else } s_2)\ ;\ s_3 \;\mapsto^{\lambda}\; \texttt{if } e \texttt{ then } [\![s_1\ ;\ s_3]\!] \texttt{ else } [\![s_2\ ;\ s_3]\!]$$

(23)
$$\texttt{if } e \texttt{ then } s_1 \texttt{ else } s_2 \;\mapsto^{\mathtt{w}}\; \texttt{if } e \texttt{ then } [\![s_1]\!] \texttt{ else } [\![s_2]\!]$$
$$\texttt{if } e \texttt{ then } s_1 \texttt{ else } s_2 \;\mapsto^{\lambda}\; \texttt{if } e \texttt{ then } [\![s_1]\!] \texttt{ else } [\![s_2]\!]$$

(24)
$$(\texttt{while } e \texttt{ do } s_1)\ ;\ s_2 \;\mapsto^{\mathtt{w}}\; sup_n([\![\texttt{unfold-while-n-times } e\ s_1]\!])\ ;\ [\![s_2]\!]$$
$$(\texttt{while } e \texttt{ do } s_1)\ ;\ s_2 \;\mapsto^{\lambda}\; (\lambda f.f\ 0)\ (\texttt{Y}\ (\lambda f.\lambda x.(e\ ([\![s_1\ ;\ f\ x]\!])\ ([\![s_2]\!]))))$$

(25)
$$\texttt{while } e \texttt{ do } s_1 \;\mapsto^{\mathtt{w}}\; sup_n([\![\texttt{unfold-while-n-times } e\ s_1]\!])$$
$$\texttt{while } e \texttt{ do } s_1 \;\mapsto^{\lambda}\; (\lambda f.f\ 0)\ (\texttt{Y}\ (\lambda f.\lambda x.(e\ ([\![s_1\ ;\ f\ x]\!])\ (x_r))))$$

Most of the cases follow directly from the induction hypothesis, the base cases, or β-reduction of the λ-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 due to the way we interpret sequences where the first statement is an assignment.

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 unfolding the `while`-loop.

So far so good, now what about the equivalence with the λ-term? For convenience let us first argue about the case of 25 and afterwards we will see that case 24 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.

$$(\lambda f.f\ 0)(\texttt{Y}\ \ (\lambda f.\lambda x.(e\ (\llbracket s_1\ ;\ f\ x \rrbracket)\ (x_r)))$$

$$\equiv_\beta$$

$$(\texttt{Y}\ \ (\lambda f.\lambda x.(e\ (\llbracket s_1\ ;\ f\ x \rrbracket)\ (x_r))))\ 0$$

$$\equiv_\beta$$

$$(\lambda f.\lambda x.(e\ (\llbracket s_1\ ;\ f\ x \rrbracket)\ (x_r)))\ (\texttt{Y}\ \ (\lambda f.\lambda x.(e\ (\llbracket s_1\ ;\ f\ x \rrbracket)\ (x_r))))\ 0$$

$$\equiv_\beta$$

$$e\ (\llbracket s_1\ ;\ (\texttt{Y}\ \ (\lambda f.\lambda x.(e\ (\llbracket s_1\ ;\ f\ x \rrbracket)\ (x_r))))\ 0 \rrbracket)\ x_r$$

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 `while` e `do` s_1 `;` `skip`, which is equivalent to the direct evaluation of 25.

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

$$e\ (\llbracket s_1\ ;\ (\texttt{Y}\ (\lambda f.\lambda x.(e\ (\llbracket s_1\ ;\ f\ x\rrbracket)\ (x_r))))\ 0\rrbracket)\ s_2$$

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 `if` e `then` s_1 `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. To this end we use the monadic structure presented in Section 2.2.2 as the space to interpret in.

We have already seen a monadic interpretation of $\mathcal{R}$`ml` in Figure 2.1 and one of `pwhile` in Figure 2.2. Adding an interpretation like this for the λ-calculus would be the first step from proving that the diagram in Figure 3.4 commutes to proving that the diagram for the non-deterministic languages in Figure 3.3 commutes.

We would then have to repeat the proof of Section 3.3 with the interpretation in said monadic structure instead of the proposed straightforward interpretation.

An important part of this proof is that, since we want to interpret into the space of ω-cpos, we have to make sure that the space of our interpretation is actually an ω-cpo. Here we recall Section 2.3, in which we argue that the monad we presented in Section 2.2.2 has an ω-cpo structure to it. We use this structure in order to interpret recursive definitions and looping constructs.

Lastly we 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 λ-calculus–and its interpretation using the ω-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.

3.5.1 About `prp` and probabilistic assignment

When translating from `while` to a simple functional language, we left out some of the syntax of `pwhile` other than the probabilistic functions. These parts are the expression `prp` and the statement x `$=` e. In order to implement a translation from `pwhile` to $\mathcal{R}$`ml`, these have to be translated as well, so here we give an idea of how to do that.

`prp` is the expression that takes a predicate on memories and maps it over the working memory at the time of evaluation (cf. Section 2.5). The working memory throughout the interpretation of `pwhile` programs is a distribution over memories, so we need to create a distribution over memories in $\mathcal{R}$`ml` in order to be able to evaluate this expression.

This can be done by keeping an environment during the translation of `pwhile` programs and defining a function, `mem_from_env`, that, given such an environment, computes a memory distribution based on the variables currently defined in the environment. An environment should hold the names of all defined variables as well as an $\mathcal{R}$`ml` expression corresponding to their value. We can then translate `prp P` as

$$\texttt{prp P} \mapsto \texttt{P (mem\_from\_env } env\texttt{)}$$

where `mem_from_env` *env* is a term of the meta-language and the translated term should hold the value of its evaluation instead.

Probabilistic assignment is the statement that is used in `pwhile` for assigning a probabilistic value to a variable. This is used to make interpretation of `pwhile` easier for non-probabilistic assignments. All non-probabilistic assignments could be treated as probabilistic assignments with only one possible outcome that has probability 1. In $\mathcal{R}$`ml` all bindings are probabilistic, hence the translation of probabilistic assignments is the same as that for non-probabilistic assignments.

4 The Coq development

The proof in the previous section concerns itself with languages that are designed for an implementation in Coq. So what would make more sense than implementing the proof in Coq?

We did not implement the full proof of Section 3; 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 [2].

Since our implementation does not adhere to the exact syntax of $\mathcal{R}$`ml`, we will call it $\mathcal{R}$`ml`* to make it clear to the reader when we are talking about the specification and when we are talking about the implementation. This is presented in Section 4.1. Furthermore we went with an approach similar to that presented in Section 3 of not interpreting $\mathcal{R}$`ml`* directly but rather taking a step in between its abstract syntax and its interpretation. We call the type of the augmented abstract syntax `s`$\mathcal{R}$`ml` and it is presented in Section 4.2. Finally we will talk about the typing rules we implemented for $\mathcal{R}$`ml`* (Section 4.3) as well as how we interpret it (Section 4.4).

The entirety of our development can be found at `https://github.com/cmester0/probabilistic-while-to-random-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 interpretation 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 top-level 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.

`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 cannot 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 [2]) 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 s$\mathcal{R}$`ml` abstract syntax.

Having this definition in place, the next step is to translate $\mathcal{R}$`ml`* to s$\mathcal{R}$`ml`. 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 vari-

able that is bound. As we go, we construct the abstract syntax of an s$\mathcal{R}$ml expression. Most of the cases are straightforward and an $\mathcal{R}$ml* construct is just replaced with the corresponding s$\mathcal{R}$ml 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 cannot 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 [9] 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 $\mathtt{s}\mathcal{R}\mathtt{ml}$ 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 $\mathcal{R}\mathtt{ml}^*$ to $\mathtt{s}\mathcal{R}\mathtt{ml}$. This means that in order to translate, the $\mathcal{R}\mathtt{ml}^*$ expression we are trying to translate has to be well-typed; we assure this by 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 $\mathtt{s}\mathcal{R}\mathtt{ml}$ 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 $\mathcal{R}\mathtt{ml}^*$ to $\mathtt{s}\mathcal{R}\mathtt{ml}$, 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 $\mathcal{R}\mathtt{ml}^*$ as well as for the function environment for $\mathtt{s}\mathcal{R}\mathtt{ml}$.

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 $\mathtt{s}\mathcal{R}\mathtt{ml}$

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

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

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

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

`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`.

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

Our work is mainly based on two different approaches to developing a framework for proofs of probabilistic algorithms in Coq. On one hand there is $\mathcal{R}$`ml` and the ALEA library presented by Audebaud and Paulin-Mohring in [2]; on the other hand there is EasyCrypt's `pwhile` and the Coq implementation of `pwhile` in the `xhl` development.

Another notable development in this subject is the "Foundational Proof Framework for Cryptography", FCF, presented in [3]. This work presents a more low-level functional approach to the same problem, but we did not consider it in our own development.

In difference to the works mentioned above, our development focused on comparing and unifying the different approaches instead of developing a new framework.

The theory we developed in Section 3 was not fully implemented in our development, and it would be interesting to continue working on the implementation of a translation from `pwhile` to $\mathcal{R}$`ml` and compare their respective interpretations, as was our suggestion in Section 3.5.

Furthermore, our proofs only show `pwhile` to be a subset of $\mathcal{R}$`ml` and do not show the languages equivalent. Though it intuitively seems that they might be, proving the equivalence of `pwhile` and $\mathcal{R}$`ml` might be something upon which to base future work about developing frameworks for proofs of probabilistic algorithms in Coq.

Another thing that would be interesting to explore is how FCF fits into the picture. Intuitively we expect it to have the same expressivity as $\mathcal{R}$`ml` and `pwhile`, but it would be interesting to explore the differences in the interpretations of probabilistic definitions as well as the ease with which algorithms would be implemented in them.

Lastly it would make sense to explore the possibilities of the different approaches; which are most usable in practice, and if they could be combined in order to develop a framework that is more useful than either of the approaches on its own.

6 Conclusion

We have in this work looked at two probabilistic languages and their interpretation in Coq: The functional $\mathcal{R}$`ml`, and the imperative `pwhile`.

We started by going through the theoretical background of such interpretations, where we explored how we can interpret probabilistic definitions as probability measures and how we can represent these measures by use of a monad. We also looked at the domain theoretic approach to interpreting recursion and iteration without running the risk of trying to construct a non-terminating algorithm within Coq, since this would either be disallowed by Coq's termination-checker, or show that we added axioms that ended up introducing an inconsistency in the logic.

After looking at the interpretations for recursion/iteration, and for probabilistic definitions, we made sure that it was possible to interpret recursion/iteration in a probabilistic setting.

After having established the theoretic background, we had a look at the syntax and semantics of $\mathcal{R}$`ml` and `pwhile`. Seeing their similarity we went ahead proving that the interpretation of `pwhile` is a subset of the interpretation of $\mathcal{R}$`ml` by defining a translation from `pwhile` to $\mathcal{R}$`ml` and proving that interpreting the original `pwhile` program produced a result equivalent to interpreting the result of translating this `pwhile` program to $\mathcal{R}$`ml`.

This proof was carried out by hand. We attempted to prove the same

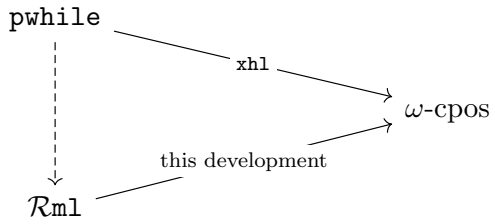

Figure 6.9: The translations of which implementations exist are represented by solid arrows labelled by the source they are implemented in, the translation that still lacks implementation is marked with a dashed arrow.

thing in Coq, but we approached this from the wrong direction and without first gaining a proper understanding of our goals and the way there, so what was actually implemented is an interpretation of a language equivalent to $\mathcal{R}$ml.

We tried to prove the diagram in Figure 3.3 commutative, but we did not implement a translation from pwhile to $\mathcal{R}$ml. The translations that are currently implemented can be seen in Figure 6.9.

Acknowledgements We thank our advisor Bas Spitters for his time, his expertise, and for continuously pushing us forwards.

A special thanks goes to Pierre-Yves Strub who took the time to introduces us to pwhile and detail the xhl development to us.

Special thanks is also due to Olivier Danvy for sparking an interest in research and for being generous with advice to those who seek it.

We also thank Christoffer Müller Madsen and Steffan Christ Sølvsten who have offered up some of their time in order to meliorate the result of our work.

References

[1] T. Coquand and G. Huet, "The coq proof assistant." http://coq.inria.fr, 1984.

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

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

[4] 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.

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

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

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

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

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

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.
```