Technical report

Lasse Letager Hansen
Kira Kutscher

May 13, 2019

Contents

1 Introduction

2 Theory and existing frameworks

2.1 $\mathcal{R}$`ml`

We have two representations of Rml, continuations and distributions. Both build on a monad, for ease of use.

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 coq 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.

2.2 EasyCrypt and `pwhile`

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 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 [1]. 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:

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

Explain what prp is all about

The result of interpreting a program in `pwhile` is a distribution over memories.

Is it enough to describe what the semantics is, or should I extract the formal semantics from the xhl development?

2.3 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 randomised 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.

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

2.3.1 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 $F_0 \leq F_1 \leq F_2 \leq F_3 \leq \dots$. 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.

2.3.2 Interpreting random definitions

A measure is a linear function μ from a set A to non-negative real numbers. It also preserves least upper bounds.
Mß = (ß→[0,1])→[0,1] The (ß→[0,1]) part describes a probability distribution. We can view our programs as transformations of probability distributions.
"a term e of type ß is translated to a purely functional one [e] which is understood as a measure on the same type."

The w-cpo structure on [0,1]; why is it an w-cpo?
@Bas: What do we need the structure for?
I am confused by what it says on page 574: First we talk about μ(f) for f:A→[0,1] and then we say that μ is a measure on A. Shouldn't it be on probability distributions over A (so A→[0,1])?

Monadic transformation. Should we add a subsection on monads or should we just mention them?

3 Our approach

3.1 Translating `while` to a functional language

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

$$\begin{aligned}
&(1) \qquad exp &::= \;& x | n | \texttt{true} | \texttt{false} | f\ x \\
&(2) \qquad stm &::= \;& \texttt{skip} | x := e | \texttt{if}\ e\ \texttt{then}\ s_1\ \texttt{else}\ s_2 | \texttt{while}\ e\ \texttt{do}\ s | s_1 ; s_2
\end{aligned}$$

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

@Bas: Is this description of our return and x_r sufficiently clear?

$$\begin{aligned}
&(3) & \texttt{skip}\ ;\ s &\mapsto & [\![s]\!] \\
&(4) & \texttt{skip} &\mapsto & x_r \\
&(5) & x := e\ ;\ s &\mapsto & \texttt{let}\ x := e\ \texttt{in}\ [\![s]\!] \\
&(6) & x_r := e &\mapsto & e \\
&(7) & x := e &\mapsto & x_r \\
&(8) & & & \\
& & (\texttt{if}\ e\ \texttt{then}\ s_1\ \texttt{else}\ s_2)\ ;\ s_3 &\mapsto & \texttt{if}\ e\ \texttt{then}\ [\![s_1\ ;\ s_3]\!]\ \texttt{else}\ [\![s_2\ ;\ s_3]\!] \\
&(9) & \texttt{if}\ e\ \texttt{then}\ s_1\ \texttt{else}\ s_2 &\mapsto & \texttt{if}\ e\ \texttt{then}\ [\![s_1]\!]\ \texttt{else}\ [\![s_2]\!] \\
&(10) & (\texttt{while}\ e\ \texttt{do}\ s_1)\ ;\ s_2 &\mapsto & \texttt{let rec}\ f\ x := \texttt{if}\ e\ \texttt{then}\ [\![s_1\ ;\ f\ x]\!]\ \texttt{else}\ [\![s_2]\!]\ \texttt{in}\ f\ 0 \\
&(11) & \texttt{while}\ e\ \texttt{do}\ s_1 &\mapsto & \texttt{let rec}\ f\ x := \texttt{if}\ e\ \texttt{then}\ [\![s_1\ ;\ f\ x]\!]\ \texttt{else}\ x_r\ \texttt{in}\ f\ 0
\end{aligned}$$

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	typed λ-calculus
Var (x, A)	$x : A$
Const A c	$c : A$
Let (x, A) e_1 e_2	$(\lambda x : A.e_2)\ e_1$
Fun (x, A) e	$\lambda x : A.e$
App e_1 e_2	$e_1\ e_2$
Let rec $(f, A \to B)$ (x, A) e_1 e_2	$(\lambda f : A \to B.e_2)\ (Y\ (\lambda f : A \to B.\lambda x : A.e_1))$

The problem here is that we need to translate $e1$ and $e2$ to their simple forms, so we do an intermediate translation:

$$Let$$

3.2.1 Example: Fib

Expression:

$$\begin{aligned}
&\texttt{Let_rec}\ (f, \mathbb{N} \to \mathbb{N})\ (x, \mathbb{N})\\
&\qquad (\texttt{if}\ x \leq 0\\
&\qquad\ \texttt{then}\ 0\\
&\qquad\ \texttt{else}\ f\ (x-1) + f\ (x-2))\\
&\qquad (f\ 3)
\end{aligned}$$

Typing:

$$\begin{aligned}
&\texttt{Let_rec}\ (f, \mathbb{N} \to \mathbb{N})\ (x, \mathbb{N})\\
&\qquad ((\texttt{if}\ (x \leq 0 : \mathbb{B})\\
&\qquad\ \ \texttt{then}\ (0 : \mathbb{N})\\
&\qquad\ \ \texttt{else}\ (f : \mathbb{N} \to \mathbb{N})\ (x - 1 : \mathbb{N}) + (f : \mathbb{N} \to \mathbb{N})\ (x - 2 : \mathbb{N}) : \mathbb{N}) : \mathbb{N})\\
&\qquad ((f : \mathbb{N} \to \mathbb{N})\ (3 : \mathbb{N}) : \mathbb{N})
\end{aligned}$$

Semi-simple

```
Let_stm f
  sFix
    sFun (f, ℕ → ℕ)
      sFun (x, ℕ)
        ((if (x ≤ 0)
          then 0
          else f (x − 1) + f (x − 2)))
  (f 3)
```

Simple form:

```
sApp sFix
  sFun (f, ℕ → ℕ)
    sFun (x, ℕ)
      ((if (x ≤ 0)
        then 0
        else f (x − 1) + f (x − 2)))
  3
```

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

What do ω-cpos have to do with this?

3.4 Interpreting `while` directly

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

3.5 All translations (forward)

What is the point of this section?

Rml	@sRml A	typed λ-calculus
Var (x, A)	sVar x	$x : A$
Const A c	sConst c	$c : A$
Let (x, T) e_1 e_2	e'_2	$(\lambda x : T, e_2 : A)\ (e_1 : T) : A$
Fun (x, T) e	sFun S (x, T) e'	$(\lambda x : T, e : S) : T \to S$
App T e_1 e_2	sApp T e'_1 e'_2	$(e_1 : T \to A)\ (e_2 : T) : A$
Let rec T S f x e_1 e_2	sApp $(T \to S)$ (sFun A $(f, T \to S)$ e'_2) (sFun S (x, T) (sFix T f x e'_1 (sVar x)))	$(\lambda f : T \to S, e_2 : A)$ $(Y\ (\lambda f : T \to S, \lambda x : T, e_1 : S) : T \to S) : A$

4 Our contribution

5 Comparisons and future work

6 Conclusion

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.
```