Technical report

Lasse Letager Hansen
Kira Kutscher

April 29, 2019

Contents

1 Introduction **2**

2 Theory and existing frameworks **2**
2.1 $\mathcal{R}$`ml` . . . 2
2.2 `pwhile` . . . 3

3 Our approach **3**
3.1 Translating `while` to a functional language . . . 3
3.2 Translation from Rml to typed λ-calculus . . . 4
3.2.1 Example: Fib . . . 4
3.3 All translations (forward) . . . 5

4 Our contribution **5**

5 Comparisons and future work **5**

6 Conclusion **5**

7 Appendix **6**

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:

$$
\begin{aligned}
&\texttt{Inductive Rml} : \texttt{Type} := \\
&\quad |\ \texttt{Var} : (\mathbb{N} * \texttt{Type}) \to \texttt{Rml} \\
&\quad |\ \texttt{Const} : \forall\ (A : \texttt{Type}),\ A \to \texttt{Rml} \\
&\quad |\ \texttt{Let\_stm} : (\mathbb{N} * \texttt{Type}) \to \texttt{Rml} \to \texttt{Rml} \to \texttt{Rml} \\
&\quad |\ \texttt{If\_stm} : \texttt{Rml} \to \texttt{Rml} \to \texttt{Rml} \to \texttt{Rml} \\
&\quad |\ \texttt{App\_stm} : \texttt{Type} \to \texttt{Rml} \to \texttt{Rml} \to \texttt{Rml} \\
&\quad |\ \texttt{Let\_rec} : (\mathbb{N} * \texttt{Type}) \to (\mathbb{N} * \texttt{Type}) \to \texttt{Rml} \to \texttt{Rml} \to \texttt{Rml}.
\end{aligned}
$$

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:

$$
\begin{aligned}
&\texttt{Inductive sRml} : \texttt{Type} := \\
&\quad |\ \texttt{sVar} : (\mathbb{N} * \texttt{Type}) \to \texttt{sRml} \\
&\quad |\ \texttt{sConst} : \forall\ (A : \texttt{Type}),\ A \to \texttt{sRml} \\
&\quad |\ \texttt{sIf} : \texttt{sRml} \to \texttt{sRml} \to \texttt{sRml} \to \texttt{sRml} \\
&\quad |\ \texttt{sApp} : \texttt{Type} \to \texttt{sRml} \to \texttt{sRml} \to \texttt{sRml} \\
&\quad |\ \texttt{sFix} : \forall\ (p\ p0 : (\mathbb{N} * \texttt{Type})),\ @\texttt{sRml}\ p.2 \to @\texttt{sRml}\ (p.2 \to A) \to \texttt{sRml}.
\end{aligned}
$$

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 `pwhile`

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 $\mathcal{R}\texttt{ml}$. The thought behind this is that once this translation is in place, all we have to do to translate `pwhile` to $\mathcal{R}\texttt{ml}$ is to add nondeterminism.

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

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

The syntax of our functional language is the same as $\mathcal{R}\texttt{ml}$ 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

Furthermore we need to handle the fact that while the imperative `while` has a return memory that one could extract the wished results from, a functional language has no such thing. We therefore need to choose the memory positions we are interested in and encapsulate those in a variable. We will, in the following, choose x_r to be the name of said variable.

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

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

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

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

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

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

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

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

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

3.2 Translation from Rml to typed λ-calculus

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:

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

Typing:

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

Semi-simple

$$\begin{array}{l}
\texttt{Let_stm}\ f \\
\quad \texttt{sFix} \\
\qquad \texttt{sFun}\ (f, \mathbb{N} \to \mathbb{N}) \\
\qquad\quad \texttt{sFun}\ (x, \mathbb{N}) \\
\qquad\qquad ((\texttt{if}\ (x \leq 0) \\
\qquad\qquad\ \ \texttt{then}\ 0 \\
\qquad\qquad\ \ \texttt{else}\ f\ (x-1) + f\ (x-2))) \\
\quad (f\ 3)
\end{array}$$

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 All translations (forward)

Rml	@sRml A	typed λ-calculus
Var (x, A)	sVar x	$x : A$
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		$(\lambda f : A \to B, e_2)$ $(Y$ $(\lambda f : A \to B, \lambda x : A, e_1))$

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 (nat * Type * Rml))
         (fl : seq (nat * 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 (nat * Type * Rml))
       (fl : seq (nat * 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 (nat * Type * Rml))
       (fl : seq (nat * 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 (nat * Type * Rml))
       (fl : seq (nat * 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 (nat * Type * Rml))
       (fl : seq (nat * 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 (nat * Type * Rml))
       (fl : seq (nat * 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 (nat * Type
     (fl : seq (nat * 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)} : @sRn
Proof.
  (** Structure **)
  {
    induction x ; intros ; refine (sVar (0,A)).
  }

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