

# Technical report

Lasse Letager Hansen  
Kira Kutscher

April 29, 2019

## Contents

<b>1</b>	<b>Introduction</b>	<b>2</b>
<b>2</b>	<b>Theory and existing frameworks</b>	<b>2</b>
2.1	$\mathcal{R}_{ml}$ . . . . .	2
2.2	<code>pwhile</code> . . . . .	3
<b>3</b>	<b>Our approach</b>	<b>3</b>
3.1	Translating <code>while</code> to a functional language . . . . .	3
3.2	Translation from <code>Rml</code> to typed $\lambda$ -calculus . . . . .	3
3.2.1	Example: <code>Fib</code> . . . . .	4
3.3	All translations (forward) . . . . .	5
<b>4</b>	<b>Our contribution</b>	<b>5</b>
<b>5</b>	<b>Comparisons and future work</b>	<b>5</b>
<b>6</b>	<b>Conclusion</b>	<b>5</b>
<b>7</b>	<b>Appendix</b>	<b>6</b>

# 1 Introduction

## 2 Theory and existing frameworks

### 2.1 $\mathcal{R}_{ml}$

We have two representations of  $\mathcal{R}_{ml}$ , continuations and distributions. Both build on a monad, for ease of use.

The data structure used to represent  $\mathcal{R}_{ml}$  terms is as follows:

```
Inductive  $\mathcal{R}_{ml}$  : Type :=  
  | Var : ( $\mathbb{N} * \text{Type}$ )  $\rightarrow$   $\mathcal{R}_{ml}$   
  | Const :  $\forall (A : \text{Type}), A \rightarrow \mathcal{R}_{ml}$   
  | Let_stm : ( $\mathbb{N} * \text{Type}$ )  $\rightarrow \mathcal{R}_{ml} \rightarrow \mathcal{R}_{ml} \rightarrow \mathcal{R}_{ml}$   
  | If_stm :  $\mathcal{R}_{ml} \rightarrow \mathcal{R}_{ml} \rightarrow \mathcal{R}_{ml} \rightarrow \mathcal{R}_{ml}$   
  | App_stm :  $\text{Type} \rightarrow \mathcal{R}_{ml} \rightarrow \mathcal{R}_{ml} \rightarrow \mathcal{R}_{ml}$   
  | Let_rec : ( $\mathbb{N} * \text{Type}$ )  $\rightarrow (\mathbb{N} * \text{Type}) \rightarrow \mathcal{R}_{ml} \rightarrow \mathcal{R}_{ml} \rightarrow \mathcal{R}_{ml}$ .
```

We use all cog types, as possible types of  $\mathcal{R}_{ml}$  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  $\mathcal{R}_{ml}$  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  $\mathcal{R}_{ml}$  expression can be typed under a given type. We have shown that if a  $\mathcal{R}_{ml}$  program is valid then it is well formed. We have then constructed a simplified form of  $\mathcal{R}_{ml}$  called `sRml` (for simple  $\mathcal{R}_{ml}$ ), to make it easier to reason about and evaluate expressions, with the following data structure:

```
Inductive  $s\mathcal{R}_{ml}$  : Type :=  
  | sVar : ( $\mathbb{N} * \text{Type}$ )  $\rightarrow s\mathcal{R}_{ml}$   
  | sConst :  $\forall (A : \text{Type}), A \rightarrow s\mathcal{R}_{ml}$   
  | sIf :  $s\mathcal{R}_{ml} \rightarrow s\mathcal{R}_{ml} \rightarrow s\mathcal{R}_{ml} \rightarrow s\mathcal{R}_{ml}$   
  | sApp :  $\text{Type} \rightarrow s\mathcal{R}_{ml} \rightarrow s\mathcal{R}_{ml} \rightarrow s\mathcal{R}_{ml}$   
  | sFix :  $\forall (p \ p0 : (\mathbb{N} * \text{Type})), @s\mathcal{R}_{ml} \ p.2 \rightarrow @s\mathcal{R}_{ml} \ (p.2 \rightarrow A) \rightarrow s\mathcal{R}_{ml}$ .
```

That is  $\mathcal{R}_{ml}$  where we remove expressions with variables, from `let_stm` statements (not `let_rec` statements). We then show that given a valid typing of an  $\mathcal{R}_{ml}$  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 possibility 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  $\mathcal{Rml}$ . The thought behind this is that once this translation is in place, all we have to do to translate **pwhile** to  $\mathcal{Rml}$  is to add nondeterminism.

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

The syntax of our functional language is the same as  $\mathcal{Rml}$  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.

- $$\begin{aligned} (3) \quad & \mathbf{skip} ; s \mapsto \llbracket s \rrbracket \\ (4) \quad & \mathbf{skip} \mapsto x_r \\ (5) \quad & x := e ; s \mapsto \mathbf{let}\ x := e\ \mathbf{in}\ \llbracket s \rrbracket \end{aligned}$$

### 3.2 Translation from Rml to typed $\lambda$ -calculus

Rml	typed $\lambda$ -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 $(f, A \rightarrow B)\ (x, A)\ e_1\ e_2$	$(\lambda f : A \rightarrow B. e_2)\ (Y\ (\lambda f : A \rightarrow B. \lambda x : A. e_1))$

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:

```
Let_rec (f,  $\mathbb{N} \rightarrow \mathbb{N}$ ) (x,  $\mathbb{N}$ )
  (if  $x \leq 0$ 
   then 0
   else  $f (x - 1) + f (x - 2)$ )
(f 3)
```

Typing:

```
Let_rec (f,  $\mathbb{N} \rightarrow \mathbb{N}$ ) (x,  $\mathbb{N}$ )
  ((if ( $x \leq 0 : \mathbb{B}$ )
    then ( $0 : \mathbb{N}$ )
    else ( $f : \mathbb{N} \rightarrow \mathbb{N}$ ) ( $x - 1 : \mathbb{N}$ ) + ( $f : \mathbb{N} \rightarrow \mathbb{N}$ ) ( $x - 2 : \mathbb{N}$ ) :  $\mathbb{N}$ )
  (( $f : \mathbb{N} \rightarrow \mathbb{N}$ ) ( $3 : \mathbb{N}$ ) :  $\mathbb{N}$ )
```

Semi-simple

```
Let_stm f
  sFix
    sFun (f,  $\mathbb{N} \rightarrow \mathbb{N}$ )
      sFun (x,  $\mathbb{N}$ )
        ((if ( $x \leq 0$ )
          then 0
          else  $f (x - 1) + f (x - 2)$ )))
(f 3)
```

Simple form:

```

sApp sFix
  sFun (f,  $\mathbb{N} \rightarrow \mathbb{N}$ )
    sFun (x,  $\mathbb{N}$ )
      ((if (x ≤ 0)
        then 0
        else f (x - 1) + f (x - 2)))
3

```

### 3.3 All translations (forward)

Rml	@sRml A	typed $\lambda$ -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, $A \rightarrow B$ ) (x, A) $e_1$ $e_2$		$(\lambda f : A \rightarrow B, e_2) (Y (\lambda f : A \rightarrow B, \lambda x : A, e_1))$

## 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)} : @sRml A
Proof.
  (** Structure **)
  {
    induction x ; intros ; refine (sVar (0,A)).
  }

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

```