Technical report

Lasse Letager Hansen 201508114

April 27, 2019

1 Rml

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 : Type :=
  | Var : (ℕ * Type) → 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 : (ℕ * Type) → (ℕ * Type) → Rml → Rml → Rml.
```

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

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

with the following data structure:

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

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 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:

$$Let$$

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:

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

Semi-simple

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

Simple form:

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

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

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