Syntax

We will adopt the following meta-variable conventions:

x	$\in$	Var	variables
b	$\in$	$\{$`true, false`$\}$	booleans
n	$\in$	$\mathbb{Z}$	integers
s	$\in$	Σ^*	ASCII strings
ℓ	$\in$	Loc	memory locations
h	$\in$	Hand	process handles
ρ	$\in$	Prom	promises

The abstract syntax of **expressions** can be defined as follows, using auxiliary definitions for patterns p, unary operations $\odot$, binary operations $\oplus$, and types τ given below:

$e \in \mathit{Exp}$	$::=$	$()$	*Unit*
	$\mid$	b	*Booleans*
	$\mid$	n	*Integers*
	$\mid$	s	*Strings*
	$\mid$	x	*Variables*
	$\mid$	(e_1, e_2)	*Pairs*
	$\mid$	$[\,]$	*Empty list*
	$\mid$	$e_1 :: e_2$	*Non-empty lists*
	$\mid$	$\texttt{fun}\ (p : \tau) \to e$	*Functions*
	$\mid$	$\texttt{let}\ (p : \tau) = e_1\ \texttt{in}\ e_2$	*Let definitions*
	$\mid$	$\texttt{let rec}\ (f : \tau_1) = \texttt{fun}\ (p : \tau_2) \to e_1\ \texttt{in}\ e_2$	*Recursive function definitions*
	$\mid$	$e_1\ e_2$	*Function application*
	$\mid$	$\odot\ e$	*Unary operators*
	$\mid$	$e_1 \oplus e_2$	*Binary operators*
	$\mid$	$e_1;\ e_2$	*Sequence*
	$\mid$	$\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3$	*Conditionals*
	$\mid$	$\texttt{match}\ e_0\ \texttt{with} \mid p_1 \to e_1\ \ldots \mid p_n \to e_n\ \texttt{end}$	*Pattern matching*
	$\mid$	$\texttt{ref}\ e$	*Reference creation*
	$\mid$	$!e$	*Dereference*
	$\mid$	$e_1 := e_2$	*Assignment*
	$\mid$	$\texttt{return}\ e$	*Asynchronous return*
	$\mid$	$\texttt{await}\ (p : \tau) = e_1\ \texttt{in}\ e_2$	*Asynchronous bind*
	$\mid$	$e_1\ \texttt{>>=}\ e_2$	*Asynchronous bind - infix*
	$\mid$	$\texttt{send}\ e_1\ \texttt{to}\ e_2$	*Asynchronous message send*
	$\mid$	$\texttt{recv}\ e$	*Asynchronous message receive*
	$\mid$	$\texttt{spawn}\ e_1\ \texttt{with}\ e_2$	*Asynchronous spawn*
	$\mid$	$\texttt{self}$	*Self handle literal*

The syntax for patterns, unary operations, and binary operations is defined as follows:

$p \in \mathit{Pat}$	$::=$	$\_\_ \mid x \mid () \mid b \mid n \mid s \mid (p_1, p_2) \mid [\,] \mid p_1 :: p_2$
$\odot \in \mathit{UOp}$	$::=$	`-` $\mid$ `not`
$\oplus \in \mathit{BOp}$	$::=$	`+` $\mid$ `-` $\mid$ `*` $\mid$ `/` $\mid$ `%` $\mid$ `&&` $\mid$ `||` $\mid$ `<` $\mid$ `<=` $\mid$ `>` $\mid$ `>=` $\mid$ `=` $\mid$ `<>` $\mid$ `^` $\mid$ `|>`

The set of **types** is defined as follows:

$$
\begin{array}{rcll}
\tau \in \mathit{Type} & ::= & \texttt{unit} & \textit{Unit type} \\
& | & \texttt{bool} & \textit{Boolean Type} \\
& | & \texttt{int} & \textit{Integer Type} \\
& | & \texttt{string} & \textit{String Type} \\
& | & \tau_1 * \tau_2 & \textit{Pair Types} \\
& | & \tau\ \texttt{list} & \textit{List Types} \\
& | & \tau_1 \to \tau_2 & \textit{Function Types} \\
& | & \tau\ \texttt{ref} & \textit{Reference Types} \\
& | & \tau\ \texttt{promise} & \textit{Promise Types} \\
& | & \tau\ \texttt{handle} & \textit{Handle Types}
\end{array}
$$

The set of **values** is defined as follows, using the auxiliary definition for environments $\mathcal{E}$, which is given below:

$$
\begin{array}{rcll}
v \in \mathit{Val} & ::= & () & \textit{Unit} \\
& | & b & \textit{Boolean} \\
& | & n & \textit{Integers} \\
& | & s & \textit{Strings} \\
& | & (v_1, v_2) & \textit{Pairs} \\
& | & [\,] & \textit{Empty list} \\
& | & v_1 :: v_2 & \textit{Non-empty lists} \\
& | & (\!|\mathcal{E}, p, e|\!) & \textit{Closures} \\
& | & \ell & \textit{Locations} \\
& | & \rho & \textit{Promises} \\
& | & h & \textit{Handles}
\end{array}
$$

Environments and **stores** are defined as partial functions from variables to values and from locations to values respectively:

$$
\begin{array}{rcl}
\mathcal{E} & \in & \mathrm{Var} \rightharpoonup \mathit{Val} \\
\sigma & \in & \mathrm{Loc} \rightharpoonup \mathit{Val}
\end{array}
$$

We use the following notation for describing environments:

- $\mathit{dom}\ \mathcal{E}$ denotes the domain of $\mathcal{E}$, that is the set of variables that it is defined on,
- $\{\}$ denotes the environment that is undefined on all variables,
- $\{x \mapsto v\}$ denotes the environment that maps x to v and is otherwise undefined,
- $\mathcal{E}_1 \circ \mathcal{E}_2$ denotes the environment that maps x in $\mathit{dom}\ \mathcal{E}_2$ to $\mathcal{E}_2(x)$, x in $\mathit{dom}\ \mathcal{E}_1$ but not in $\mathit{dom}\ \mathcal{E}_2$ to $\mathcal{E}_1(x)$, and is otherwise undefined.

We use the same notation for the analogous operations on stores σ.

Simplified Semantics

As we have seen in lecture, we can model the semantics of our language in terms of a bigstep relation $\Longrightarrow$. For RML, we will start with the relation

$$\langle \mathcal{E}, e \rangle \Longrightarrow \langle v \rangle,$$

which can be read intuitively as "under environment $\mathcal{E}$, the expression e evaluates to the value v." We can use this relation to define rules for evaluation, again similar to how we have done in lecture. For example, the rule for sequencing can be defined as follows:

$\boxed{\text{E-Sequence}}$

$\langle \mathcal{E}, e_1; e_2 \rangle \Longrightarrow \langle v \rangle$
$\quad$ if $\quad \langle \mathcal{E}, e_1 \rangle \Longrightarrow \langle () \rangle$
$\quad$ and $\quad \langle \mathcal{E}, e_2 \rangle \Longrightarrow \langle v \rangle$

Simple Expressions

To warm up, let us consider the semantics of several simple expressions: values, pairs, lists, and variables.

$\boxed{\text{E-Value}}$

$\langle \mathcal{E}, v \rangle \Longrightarrow \langle v \rangle$

$\boxed{\text{E-Var}}$

$\langle \mathcal{E}, x \rangle \Longrightarrow \langle v \rangle$
$\quad$ if $\quad \mathcal{E}(x) = v$

$\boxed{\text{E-Pair}}$

$\langle \mathcal{E}, (e_1, e_2) \rangle \Longrightarrow \langle (v_1, v_2) \rangle$
$\quad$ if $\quad \langle \mathcal{E}, e_1 \rangle \Longrightarrow \langle v_1 \rangle$
$\quad$ and $\quad \langle \mathcal{E}, e_2 \rangle \Longrightarrow \langle v_2 \rangle$

$\boxed{\text{E-Cons}}$

$\langle \mathcal{E}, e_1 :: e_2) \rangle \Longrightarrow \langle v_1 :: v_2 \rangle$
$\quad$ if $\quad \langle \mathcal{E}, e_1 \rangle \Longrightarrow \langle v_1 \rangle$
$\quad$ and $\quad \langle \mathcal{E}, e_2 \rangle \Longrightarrow \langle v_2 \rangle$

Intuitively, these inference rules can be understood as follows:

E-Value: a value v evaluates to itself, as in most big-step semantics. The run-time state Δ and store σ are unchanged.

E-Var: a variable x evaluates to the value obtained by looking up x in the environment $\mathcal{E}$. Again, the run-time state Δ and store σ are unchanged.

E-Pair: a pair expression (e_1, e_2) evaluates to a pair value (v_1, v_2) in the obvious way. Note that the effects on the run-time state Δ and store σ are accumulated from left to right.

E-Cons: a cons expression $e_1 :: e_2$ evaluates to a non-empty list value $v_1 :: v_2$ in the obvious way.

Pattern Matching Expressions

To model pattern matching, we will introduce a new relation that we haven't seen before. It will be used to describe not only pattern matching but also variable bindings in general. The three-place relation of the form $v : p \rightsquigarrow \mathcal{E}$ will be read as "value v matches pattern p and produces the bindings in $\mathcal{E}$." Make sure you take the time to understand what is being said by this relation, and why it is useful. The rules for this relation are as follows:

$\boxed{\text{M-Wild}}$

$v : \_ \rightsquigarrow \{\}$

$\boxed{\text{M-Var}}$

$v : x \rightsquigarrow \{x \mapsto v\}$

$\boxed{\text{M-Unit}}$

$() : () \rightsquigarrow \{\}$

$\boxed{\text{M-Bool}}$

$b : b \rightsquigarrow \{\}$

$\boxed{\text{M-Int}}$

$n : n \rightsquigarrow \{\}$

$\boxed{\text{M-String}}$

$s : s \rightsquigarrow \{\}$

$\boxed{\text{M-EmptyList}}$

$[] : [] \rightsquigarrow \{\}$

$\boxed{\text{M-Pair}}$

$(v_1, v_2) : (p_1, p_2) \rightsquigarrow \mathcal{E}_1 \circ \mathcal{E}_2$
$\quad$ if $\quad v_1 : p_1 \rightsquigarrow \mathcal{E}_1$
$\quad$ and $\quad v_2 : p_2 \rightsquigarrow \mathcal{E}_2$

$\boxed{\text{M-Cons}}$

$v_1 :: v_2 : p_1 :: p_2 \rightsquigarrow \mathcal{E}_1 \circ \mathcal{E}_2$
$\quad$ if $\quad v_1 : p_1 \rightsquigarrow \mathcal{E}_1$
$\quad$ and $\quad v_2 : p_2 \rightsquigarrow \mathcal{E}_2$

Each of these rules are straightforward, recursing on the value and pattern in lock-step, and collecting up bindings in an environment.

The inference rule for pattern matching is as follows:

$\boxed{\text{E-Match}}$

$\langle \mathcal{E}, \texttt{match}\ e\ \texttt{with}\ \mid p_1 \to e_1\ \ldots\ \mid p_n \to e_n\ \texttt{end} \rangle \Longrightarrow \langle v \rangle$
$\quad$ if $\quad \langle \mathcal{E}, e \rangle \Longrightarrow \langle v \rangle$
$\quad$ and $\quad v : p_j \rightsquigarrow \mathcal{E}_j$ for some $0 < j < n + 1$
$\quad$ and $\quad \langle \mathcal{E} \circ \mathcal{E}_j, e_j \rangle \Longrightarrow \langle v \rangle$
$\quad$ and $\quad v : p_i \not\rightsquigarrow \mathcal{E}_i$ for all $i < j$

This inference rule evaluates e to a value v, finds the first pattern p_j that matches v, and then evaluates the corresponding expression e_j in an environment extended with the bindings from v obtained using p_j.

Functions, Let-Expressions, and Application Expressions

The next few inference rules handle functions, `let`-expressions, and application expressions:

$\boxed{\text{E-Fun}}$

$\langle \mathcal{E}, \texttt{fun}\ (p : \tau) \to e \rangle \Longrightarrow \langle (\!| \mathcal{E}, p, e |\!) \rangle$

$\boxed{\text{E-App}}$

$\langle \mathcal{E}, e_1\ e_2 \rangle \Longrightarrow \langle v \rangle$
$\quad$ if $\quad \langle \mathcal{E}, e_1 \rangle \Longrightarrow \langle (\!| \mathcal{E}_{cl}, p_{cl}, e_{cl} |\!) \rangle$
$\quad$ and $\quad \langle \mathcal{E}, e_2 \rangle \Longrightarrow \langle v_2 \rangle \quad v_2 : p_{cl} \rightsquigarrow \mathcal{E}_2$
$\quad$ and $\quad \langle \mathcal{E}_{cl} \circ \mathcal{E}_2, e_{cl} \rangle \Longrightarrow \langle v \rangle$

$\boxed{\text{E-Pipe}}$

$\langle \mathcal{E}, e_1 \ \texttt{|>} \ e_2 \rangle \Longrightarrow \langle v \rangle$
$\quad$ if $\quad \langle \mathcal{E}, e_2 \ e_1 \rangle \Longrightarrow \langle v \rangle$

$\boxed{\text{E-Let}}$

$\langle \mathcal{E}, \texttt{let} \ (p : \tau) = e_1 \ \texttt{in} \ e_2 \rangle \Longrightarrow \langle v \rangle$
$\quad$ if $\quad \langle \mathcal{E}, e_1 \rangle \Longrightarrow \langle v_1 \rangle$
$\quad$ and $\quad v_1 : p \rightsquigarrow \mathcal{E}_1$
$\quad$ and $\quad \langle \mathcal{E} \circ \mathcal{E}_1, e_2 \rangle \Longrightarrow \langle v \rangle$

$\boxed{\text{E-LetRec}}$

$\langle \mathcal{E}, \texttt{let rec} \ (f : \tau_1) = \texttt{fun} \ (p : \tau_2) \rightarrow e_1 \ \texttt{in} \ e_2 \rangle \Longrightarrow \langle v \rangle$
$\quad$ if $\quad \mathcal{E}_f = \mathcal{E}[f \mapsto (\!| \mathcal{E}_f, p, e_1 |\!)]$
$\quad$ and $\quad \langle \mathcal{E}_f, e_2 \rangle \Longrightarrow \langle v \rangle$

These inference rules can be understood as follows:

E-Fun: a function $\texttt{fun} \ (p : \tau) \rightarrow e$ evaluates to a closure

E-App: an application $e_1 \ e_2$ evaluates e_1 to a closure $(\!| \mathcal{E}, p, e |\!)$, evaluates e_2 to a value v_2, matches v_2 against the pattern p, and finally evaluates the body of the closure e.

E-Pipe: an application of the binary operator $e_1 \ \texttt{|>} \ e_2$ de-sugars to the application $e_1 \ e_2$.

E-Let: a `let`-definition evalutes the first expression e_1 to a value v_1, and then evaluates the second expression e_2 in an environment in which variables bound in p are mapped to the corresponding values in v_1.

E-LetRec: is similar to the case for `let`-definitions. It builds a recursive environment $\mathcal{E}_f$ in which f is bound to the closure for the function with parameter p and body e_1, and then uses this environment to evaluate the second expression e_2.

Unary and Binary Operations

The next few rules model unary and binary operations:

$\boxed{\text{E-UOp}}$

$\langle \mathcal{E}, \odot \ e_1 \rangle \Longrightarrow \langle v \rangle$
$\quad$ if $\quad \langle \mathcal{E}, e_1 \rangle \Longrightarrow \langle v_1 \rangle$
$\quad$ and $\quad v = [\![\odot]\!] \ v_1$

$\boxed{\text{E-BOp}}$

$\langle \mathcal{E}, e_1 \oplus e_2 \rangle \Longrightarrow \langle v \rangle$
$\quad$ if $\quad \langle \mathcal{E}, e_1 \rangle \Longrightarrow \langle v_1 \rangle$
$\quad$ and $\quad \langle \mathcal{E}, e_2 \rangle \Longrightarrow \langle v_2 \rangle$
$\quad$ and $\quad v = [\![\oplus]\!]\ v_1\ v_2$

These inference rules can be understood as follows:

E-UOp**:** a unary operation $\odot\ e_1$ evaluates e_1 to a value v_1 and then uses the implementation of the operation, denoted $[\![\odot]\!]$, to produce the final value v. Note that implementations may require the value v_1 to have a specific type—e.g., unary negation is only defined on integers.

E-BOp**:** similar to the case for unary operations. Note that this inference rule is not quite correct in the case of boolean operators with short-circuit semantics, which may not necessarily evaluate e_2. We leave the task of formalizing the correct semantics as an exercise.

Standard Control-Flow Expressions

The next few rules model standard control-flow expressions:

$\boxed{\text{E-Sequence}}$

$\langle \mathcal{E}, e_1; e_2 \rangle \Longrightarrow \langle v \rangle$
$\quad$ if $\quad \langle \mathcal{E}, e_1 \rangle \Longrightarrow \langle () \rangle$
$\quad$ and $\quad \langle \mathcal{E}, e_2 \rangle \Longrightarrow \langle v \rangle$

$\boxed{\text{E-If-True}}$

$\langle \mathcal{E}, \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \rangle \Longrightarrow \langle v \rangle$
$\quad$ if $\quad \langle \mathcal{E}, e_1 \rangle \Longrightarrow \langle \texttt{true} \rangle$
$\quad$ and $\quad \langle \mathcal{E}, e_2 \rangle \Longrightarrow \langle v \rangle$

$\boxed{\text{E-If-False}}$

$\langle \mathcal{E}, \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \rangle \Longrightarrow \langle v \rangle$
$\quad$ if $\quad \langle \mathcal{E}, e_1 \rangle \Longrightarrow \langle \texttt{false} \rangle$
$\quad$ and $\quad \langle \mathcal{E}, e_3 \rangle \Longrightarrow \langle v \rangle$

These inference rules can be understood as follows:

E-Sequence**:** a sequential composition $e_1;\ e_2$ evaluates e_1 to unit () and then evaluates e_2 to a value v.

E-IF-TRUE **and** E-IF-FALSE a conditional `if` e_1 `then` e_2 `else` e_3 first evaluates e_1 to a boolean, and then either evaluates e_2 or e_3. Note however that it does *not* evaluate both e_2 and e_3.

Imperative Expressions

Next, we will model OCaml-style references. However, our current relation does not have enough information to describe the desired behavior. We need to add in the store, σ to keep track of stateful memory locations. The expanded bigstep-relation will take the following form:

$$\langle \sigma, \mathcal{E}, e \rangle \Longrightarrow \langle \sigma', v \rangle,$$

and it can be read as follows: "under store σ and environment $\mathcal{E}$, the expression e big-steps to the updated store σ' and the value v." For consistency, expanding the relation in this manner requires that we expand all of the rules already given. However, doing so is somewhat trivial and notationally heavy, so we have provided the expanded versions of the above rules in the appendix below. We also note that your implementation should not explicitly maintain the store, but instead make use of OCaml references. Here are the formal rules:

$\boxed{\text{E-REF}}$

$\langle \sigma, \mathcal{E}, \texttt{ref}\ e \rangle \Longrightarrow \langle \sigma', \ell \rangle$
$\quad$ if $\langle \sigma, \mathcal{E}, e \rangle \Longrightarrow \langle \sigma_e, v \rangle$
$\quad$ and $\ell \notin dom\ \sigma$
$\quad$ and $\sigma' = \sigma_e \circ \{\ell \mapsto v\}$

$\boxed{\text{E-DEREF}}$

$\langle \sigma, \mathcal{E}, !e \rangle \Longrightarrow \langle \sigma', v \rangle$
$\quad$ if $\langle \sigma, \mathcal{E}, e \rangle \Longrightarrow \langle \sigma', \ell \rangle$
$\quad$ and $v = \sigma(\ell)$

$\boxed{\text{E-ASSIGN}}$

$\langle \sigma, \mathcal{E}, e_1 := e_2 \rangle \Longrightarrow \langle \sigma', () \rangle$
$\quad$ if $\langle \sigma, \mathcal{E}, e_1 \rangle \Longrightarrow \langle \sigma_1, \ell \rangle$
$\quad$ and $\langle \sigma_1, \mathcal{E}, e_2 \rangle \Longrightarrow \langle \sigma_2, v \rangle$
$\quad$ and $\sigma' = \sigma_2 \circ \{\ell \mapsto v\}$

These inference rules can be understood as follows:

E-REF**:** a reference `ref` e evaluates e to a value v and then adds it to the store σ under a fresh location ℓ, which is returned as the result.

E-Deref: a dereference $!e_1$ evaluates e_1 to a location ℓ and the looks it up in the store σ.

E-Assign: an assignment $e_1 := e_2$ evaluates e_1 to a location ℓ and e_2 to a value, updates the store σ so that ℓ maps to v_2, and returns ().

Asynchronous Expressions

The final inference rules model asynchronous expressions. Once again, the relation we are working with does not have enough information to describe the behavior of these expressions. To simplify the task of specifying and implementing RML, we will assume the existence of an `Lwt`-like concurrency library that provides a set of basic primitives that can be used to implement the concurrent operations in RML. We let $\Delta \in \mathit{State}$ range over the run-time state of this library and assume the following operations:

$$
\begin{array}{rcl}
\mathit{return} & \in & \mathit{State} \to \mathit{Val} \to \mathit{State} \times \mathit{Prom} \\
\mathit{bind} & \in & \mathit{State} \to \mathit{Prom} \to (\mathit{State} \times \mathit{Store} \times \mathit{Val} \to \mathit{State} \times \mathit{Store} \times \mathit{Prom}) \to \mathit{Prom} \\
\mathit{send} & \in & \mathit{State} \to \mathit{Hand} \to \mathit{Val} \to \mathit{State} \times \{()\} \\
\mathit{recv} & \in & \mathit{State} \to \mathit{Hand} \to \mathit{State} \times \mathit{Prom} \\
\mathit{spawn} & \in & \mathit{State} \to \mathit{Val} \to \mathit{Val} \to \mathit{State} \times \mathit{Hand} \\
\mathit{self} & \in & \mathit{Env} \to \mathit{Hand}
\end{array}
$$

We then expand the big-step relation to the following, with modified versions of the above rules in the appendix:

$$\langle \Delta, \sigma, \mathcal{E}, e \rangle \Longrightarrow \langle \Delta', \sigma', v \rangle$$

Once again, you are not responsible to implement the runtime state on your own. We provide implementations of all of the functions above,, and your implementation need simply make use of these functions. Similarly, most of the rules simply call out to the corresponding functions from the concurrency library:

$\boxed{\text{E-Return}}$

$$
\begin{array}{l}
\langle \Delta, \sigma, \mathcal{E}, \texttt{return}\ e \rangle \Longrightarrow \langle \Delta', \sigma', \rho \rangle \\
\quad \text{if} \quad \langle \Delta, \sigma, \mathcal{E}, e \rangle \Longrightarrow \langle \Delta_e, \sigma', v \rangle \\
\quad \text{and} \quad \Delta', \rho = \mathit{return}\ \Delta_e\ v_1
\end{array}
$$

$\boxed{\text{E-Await}}$

$$
\begin{array}{l}
\langle \Delta, \sigma, \mathcal{E}, \texttt{await}\ (p : \tau) = e_1\ \texttt{in}\ e_2 \rangle \Longrightarrow \langle \Delta', \sigma', \rho \rangle \\
\quad \text{if} \quad \langle \Delta, \sigma, \mathcal{E}, e_1 \rangle \Longrightarrow \langle \Delta_1, \sigma', \rho_1 \rangle \\
\quad \text{and} \quad \rho = \mathit{bind}\ \Delta_1\ \rho_1\ (\lambda(\Delta'', \sigma'', v'').\ \rho_2\ \mathit{where}\ v'' : p \rightsquigarrow \mathcal{E}'' \\
\qquad\quad \mathit{and}\ \langle \Delta'', \sigma'', \mathcal{E} \circ \mathcal{E}'', e_2 \rangle \Longrightarrow \langle \Delta_2, \sigma_2, \rho_2 \rangle)
\end{array}
$$

$\boxed{\text{E-Bind}}$

$$\langle \Delta, \sigma, \mathcal{E}, e_1 \texttt{>>=} e_2 \rangle \Longrightarrow \langle \Delta', \sigma_2, \rho \rangle$$

$$
\begin{array}{l}
\quad \text{if} \quad \langle \Delta, \sigma, \mathcal{E}, e_1 \rangle \Longrightarrow \langle \Delta_1, \sigma_1, \rho_1 \rangle \\
\quad \text{and} \quad \langle \Delta_1, \sigma_1, \mathcal{E}, e_2 \rangle \Longrightarrow \langle \Delta_2, \sigma_2, (|\mathcal{E}_{cl}, p, e_{cl}|) \rangle \\
\quad \text{and} \quad \Delta', \rho = \mathit{bind}\ \Delta_2\ \rho_1\ (\lambda(\Delta'', \sigma'', v'').\ \rho_2\ \mathit{where}\ v'' : p \rightsquigarrow \mathcal{E}'' \\
\qquad \mathit{and}\ \langle \Delta'', \sigma'', \mathcal{E}_{cl} \circ \mathcal{E}'', e_{cl} \rangle \Longrightarrow \langle \Delta', \sigma', \rho_2 \rangle)
\end{array}
$$

$\boxed{\text{E-Send}}$

$$
\begin{array}{l}
\langle \Delta, \sigma, \mathcal{E}, \texttt{send}\ e_1\ \texttt{to}\ e_2 \rangle \Longrightarrow \langle \Delta', \sigma', () \rangle \\
\quad \text{if} \quad \langle \Delta, \sigma, \mathcal{E}, e_1 \rangle \Longrightarrow \langle \Delta_1, \sigma_1, v_1 \rangle \\
\quad \text{and} \quad \langle \Delta_1, \sigma_1, \mathcal{E}, e_2 \rangle \Longrightarrow \langle \Delta_2, \sigma', h_2 \rangle \\
\quad \text{and} \quad \Delta', () = \mathit{send}\ \Delta_2\ v_2\ h_1
\end{array}
$$

$\boxed{\text{E-Recv}}$

$$
\begin{array}{l}
\langle \Delta, \sigma, \mathcal{E}, \texttt{recv}\ e \rangle \Longrightarrow \langle \Delta', \sigma', \rho \rangle \\
\quad \text{if} \quad \langle \Delta, \sigma, \mathcal{E}, e \rangle \Longrightarrow \langle \Delta_e, \sigma_e, h \rangle \\
\quad \text{and} \quad \Delta', \rho = \mathit{recv}\ \Delta_e\ h
\end{array}
$$

$\boxed{\text{E-Spawn}}$

$$
\begin{array}{l}
\langle \Delta, \sigma, \mathcal{E}, \texttt{spawn}\ e_1\ \text{with}\ e_2 \rangle \Longrightarrow \langle \Delta', \sigma', h \rangle \\
\quad \text{if} \quad \langle \Delta, \sigma, \mathcal{E}, e_1 \rangle \Longrightarrow \langle \Delta_1, \sigma_1, v_1 \rangle \\
\quad \text{and} \quad \langle \Delta_1, \sigma_1, \mathcal{E}, e_2 \rangle \Longrightarrow \langle \Delta_2, \sigma', v_2 \rangle \\
\quad \text{and} \quad \Delta', h = \mathit{spawn}\ \Delta_2\ v_1\ v_2
\end{array}
$$

$\boxed{\text{E-Self}}$

$$
\begin{array}{l}
\langle \Delta, \sigma, \mathcal{E}, \texttt{self} \rangle \Longrightarrow \langle \Delta, \sigma, h \rangle \\
\quad \text{if} \quad h = \mathit{self}\mathcal{E}
\end{array}
$$

These inference rules can be understood as follows:

E-Return: a return expression `return` e evaluates e to a value v and uses the library function *return* to obtain a promise ρ.

E-Await: an await expression `await` $e_1 = p$ `in` e_2 evaluates e_1 to a promise ρ and then schedules a call-back with parameters p and body e_2 that is executed when ρ is resolved. The notation $\lambda v.\, v'$ used to describe the call-back function denotes the function that takes a parameter v and yields a result v'.

E-Bind: an application of the binary operator e_1 `>>=` e_2 evaluates e_1 to a promise ρ and e_2 to a function closure $(|\mathcal{E}, p, e|)$ and then schedule a call-back with parameters p and body e that is executed when ρ is resolved.

E-SEND: an send expression `send` e_1 `to` e_2 evaluates e_1 to a value v_1 and e_2 to a handle h_2 and then uses the library function *send* to send v to h.

E-RECV: an receive expression `recv` e evaluates e to a handle h and then uses the library function *recv* to obtain a promise ρ that resolves to a message sent to h.

E-SPAWN: a spawn expression `spawn` e_1 `with` e_2 evaluates e_1 to a function closure v_1 and e_2 to v_1's argument v_2 and then uses the library function *spawn* to update the runtime state with a new thread scheduled to run the function call $v_1\ v_2$ and return a handle h to that new thread.

E-SELF: a self expression `self` uses the library function *self* to obtain the handle of the calling (current) thread.

Definitions and Programs

A complete RML program is defined as a non-empty list of definitions:

$$\begin{array}{lcll} \mathit{prog} \in \mathit{Prog} & ::= & d & \textit{Single definition} \\ & | & d\ \mathit{prog} & \textit{Definition followed by more definitions} \end{array}$$

A definition is either a let-definition or a let-rec-definition:

$$\begin{array}{lcll} d \in \mathit{Defn} & ::= & \texttt{let}\ (p : \tau) = e & \textit{Let definition} \\ & | & \texttt{let rec}\ (f : \tau) = \texttt{fun}\ (p : \tau') \to e & \textit{Let rec definition} \end{array}$$

The relation $\langle \Delta, \sigma, \mathcal{E}, d \rangle \Longrightarrow \langle \Delta', \sigma', \mathcal{E}' \rangle$, which can be read "under runtime state Δ, store σ, and environment $\mathcal{E}$, the definition d emits the updated runtime state Δ', updated store σ', and updated environment $\mathcal{E}'$", is given as follows:

$\boxed{\text{D-LET}}$

$$\begin{array}{l} \langle \Delta, \sigma, \mathcal{E}, \texttt{let}\ (p : \tau) = e \rangle \Longrightarrow \langle \Delta', \sigma', \mathcal{E} \circ \mathcal{E}' \rangle \\ \quad \text{if} \quad \langle \Delta, \sigma, \mathcal{E}, e \rangle \Longrightarrow \langle \Delta', \sigma', v \rangle \\ \quad \text{and} \quad v : p \rightsquigarrow \mathcal{E}' \end{array}$$

$\boxed{\text{D-LETREC}}$

$$\begin{array}{l} \langle \Delta, \sigma, \mathcal{E}, \texttt{let rec}\ (f : \tau) = e \rangle \Longrightarrow \langle \Delta', \sigma', \mathcal{E} \circ \mathcal{E}' \rangle \\ \quad \text{if} \quad \mathcal{E}_f = \mathcal{E}[f \mapsto (|\mathcal{E}_f, p, e|)] \\ \quad \text{and} \quad \langle \Delta, \sigma, \mathcal{E}_f, e \rangle \Longrightarrow \langle \Delta', \sigma', v \rangle \\ \quad \text{and} \quad v : f \rightsquigarrow \mathcal{E}' \end{array}$$

Program evaluation simply threads the empty environment through the definitions, accumulating more new mappings each time:

$\boxed{\text{PROG}}$

$$\begin{array}{l} \langle \Delta, \sigma, \mathcal{E}, d_1\ \ldots\ d_n \rangle \Longrightarrow \langle \Delta', \sigma', \mathcal{E}' \rangle \\ \quad \text{if} \quad \langle \Delta, \sigma, \mathcal{E}, d_1 \rangle \Longrightarrow \langle \Delta_1, \sigma_1, \mathcal{E}_1 \rangle \end{array}$$

...
and $\langle \Delta_{n-1}, \sigma_{n-1}, \mathcal{E}_{n-1}, d_n \rangle \Longrightarrow \langle \Delta', \sigma', \mathcal{E} \rangle$

Appendix Semantics

As you build your implementation of the semantics in OCaml, there are a few practical considerations worth keeping in mind. First, the `Promise` module serves as the concurrency library and provides all of the operations needed to implement the asynchronous expressions. Second, it is not necessary to thread the run-time state Δ and store σ through the entire evaluation. Instead, you can rely on OCaml's built-in imperative features to do that for you. Hence, the `eval` function has type: `env -> expr -> value`. Generally speaking, you can simply elide the run-time state Δ and store σ when mapping the big-step semantics rules to OCaml code.

It may be helpful to work with the simpler relation, $\langle \Delta, \sigma, \mathcal{E}, e \rangle \Longrightarrow \langle \Delta', \sigma', v \rangle$ we started with, eliding the side effects on Δ and σ. However, we provide the following fully-expanded formal definition complete with the runtime state Δ and the store σ:

$\boxed{\text{E-Value}}$

$\langle \Delta, \sigma, \mathcal{E}, v \rangle \Longrightarrow \langle \Delta, \sigma, v \rangle$

$\boxed{\text{E-Var}}$

$\langle \Delta, \sigma, \mathcal{E}, x \rangle \Longrightarrow \langle \Delta, \sigma, v \rangle$
  if $\mathcal{E}(x) = v$

$\boxed{\text{E-Pair}}$

$\langle \Delta, \sigma, \mathcal{E}, (e_1, e_2) \rangle \Longrightarrow \langle \Delta', \sigma', (v_1, v_2) \rangle$
  if $\langle \Delta, \sigma, \mathcal{E}, e_1 \rangle \Longrightarrow \langle \Delta_1, \sigma_1, v_1 \rangle$
  and $\langle \Delta_1, \sigma_1, \mathcal{E}, e_2 \rangle \Longrightarrow \langle \Delta', \sigma', v_2 \rangle$

$\boxed{\text{E-Cons}}$

$\langle \Delta, \sigma, \mathcal{E}, e_1 :: e_2) \rangle \Longrightarrow \langle \Delta', \sigma', v_1 :: v_2 \rangle$
  if $\langle \Delta, \sigma, \mathcal{E}, e_1 \rangle \Longrightarrow \langle \Delta_1, \sigma_1, v_1 \rangle$
  and $\langle \Delta_1, \sigma_1, \mathcal{E}, e_2 \rangle \Longrightarrow \langle \Delta', \sigma', v_2 \rangle$

$\boxed{\text{E-Match}}$

$\langle \Delta, \sigma, \mathcal{E}, \texttt{match}\ e\ \texttt{with}\ \mid p_1 \rightarrow e_1\ \ldots\ \mid p_n \rightarrow e_n\ \texttt{end} \rangle \Longrightarrow \langle \Delta', \sigma', v \rangle$
  if $\langle \Delta, \sigma, \mathcal{E}, e \rangle \Longrightarrow \langle \Delta_e, \sigma_e, v \rangle$

and $\quad v : p_j \rightsquigarrow \mathcal{E}_j$ for some $0 < j < n+1$
and $\quad \langle \Delta_e, \sigma_e, \mathcal{E} \circ \mathcal{E}_j, e_j \rangle \Longrightarrow \langle \Delta', \sigma', v \rangle$
and $\quad v : p_i \not\rightsquigarrow \mathcal{E}_i$ for all $i < j$

$\boxed{\text{E-Fun}}$

$\langle \Delta, \sigma, \mathcal{E}, \texttt{fun}\ (p : \tau) \to e \rangle \Longrightarrow \langle \Delta, \sigma, (\!| \mathcal{E}, p, e |\!) \rangle$

$\boxed{\text{E-App}}$

$\langle \Delta, \sigma, \mathcal{E}, e_1\ e_2 \rangle \Longrightarrow \langle \Delta', \sigma', v \rangle$
if $\quad \langle \Delta, \sigma, \mathcal{E}, e_1 \rangle \Longrightarrow \langle \Delta_1, \sigma_1, (\!| \mathcal{E}_{cl}, p_{cl}, e_{cl} |\!) \rangle$
and $\quad \langle \Delta_1, \sigma_1, \mathcal{E}, e_2 \rangle \Longrightarrow \langle \Delta_2, \sigma_2, v_2 \rangle \quad v_2 : p_{cl} \rightsquigarrow \mathcal{E}_2$
and $\quad \langle \Delta, \sigma_2, \mathcal{E}_{cl} \circ \mathcal{E}_2, e_{cl} \rangle \Longrightarrow \langle \Delta', \sigma', v \rangle$

$\boxed{\text{E-Pipe}}$

$\langle \Delta, \sigma, \mathcal{E}, e_1\ \texttt{|>}\ e_2 \rangle \Longrightarrow \langle \Delta', \sigma', v \rangle$
if $\quad \langle \Delta, \sigma, \mathcal{E}, e_2\ e_1 \rangle \Longrightarrow \langle \Delta', \sigma', v \rangle$

$\boxed{\text{E-Let}}$

$\langle \Delta, \sigma, \mathcal{E}, \texttt{let}\ (p : \tau) = e_1\ \texttt{in}\ e_2 \rangle \Longrightarrow \langle \Delta', \sigma', v \rangle$
if $\quad \langle \Delta, \sigma, \mathcal{E}, e_1 \rangle \Longrightarrow \langle \Delta_1, \sigma_1, v_1 \rangle$
and $\quad v_1 : p \rightsquigarrow \mathcal{E}_1$
and $\quad \langle \Delta_1, \sigma_1, \mathcal{E} \circ \mathcal{E}_1, e_2 \rangle \Longrightarrow \langle \Delta', \sigma', v \rangle$

$\boxed{\text{E-LetRec}}$

$\langle \Delta, \sigma, \mathcal{E}, \texttt{let rec}\ (f : \tau_1) = \texttt{fun}\ (p : \tau_2) \to e_1\ \texttt{in}\ e_2 \rangle \Longrightarrow \langle \Delta', \sigma', v \rangle$
if $\quad \mathcal{E}_f = \mathcal{E}[f \mapsto (\!| \mathcal{E}_f, p, e_1 |\!)]$
and $\quad \langle \Delta, \sigma, \mathcal{E}_f, e_2 \rangle \Longrightarrow \langle \Delta', \sigma', v \rangle$

$\boxed{\text{E-UOp}}$

$\langle \Delta, \sigma, \mathcal{E}, \odot\ e_1 \rangle \Longrightarrow \langle \Delta', \sigma', v \rangle$
if $\quad \langle \Delta, \sigma, \mathcal{E}, e_1 \rangle \Longrightarrow \langle \Delta', \sigma', v_1 \rangle$
and $\quad v = [\![\odot]\!]\ v_1$

$\boxed{\text{E-BOp}}$

$\langle \Delta, \sigma, \mathcal{E}, e_1 \oplus e_2 \rangle \Longrightarrow \langle \Delta', \sigma', v \rangle$
if $\quad \langle \Delta, \sigma, \mathcal{E}, e_1 \rangle \Longrightarrow \langle \Delta_1, \sigma_1, v_1 \rangle$
and $\quad \langle \Delta_1, \sigma_1, \mathcal{E}, e_2 \rangle \Longrightarrow \langle \Delta', \sigma', v_2 \rangle$

and $\quad v = [\![\oplus]\!]\ v_1\ v_2$

$\boxed{\text{E-Sequence}}$

$\langle \Delta, \sigma, \mathcal{E}, e_1; e_2 \rangle \Longrightarrow \langle \Delta', \sigma', v \rangle$
if $\quad \langle \Delta, \sigma, \mathcal{E}, e_1 \rangle \Longrightarrow \langle \Delta_1, \sigma_1, () \rangle$
and $\quad \langle \Delta, \sigma, \mathcal{E}, e_2 \rangle \Longrightarrow \langle \Delta', \sigma', v \rangle$

$\boxed{\text{E-If-True}}$

$\langle \Delta, \sigma, \mathcal{E}, \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \rangle \Longrightarrow \langle \Delta', \sigma', v \rangle$
if $\quad \langle \Delta, \sigma, \mathcal{E}, e_1 \rangle \Longrightarrow \langle \Delta_1, \sigma_1, \texttt{true} \rangle$
and $\quad \langle \Delta_1, \sigma_1, \mathcal{E}, e_2 \rangle \Longrightarrow \langle \Delta', \sigma', v \rangle$

$\boxed{\text{E-If-False}}$

$\langle \Delta, \sigma, \mathcal{E}, \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \rangle \Longrightarrow \langle \Delta', \sigma', v \rangle$
if $\quad \langle \Delta, \sigma, \mathcal{E}, e_1 \rangle \Longrightarrow \langle \Delta_1, \sigma_1, \texttt{false} \rangle$
and $\quad \langle \Delta_1, \sigma_1, \mathcal{E}, e_3 \rangle \Longrightarrow \langle \Delta', \sigma', v \rangle$

$\boxed{\text{E-Return}}$

$\langle \Delta, \sigma, \mathcal{E}, \texttt{return}\ e \rangle \Longrightarrow \langle \Delta', \sigma', \rho \rangle$
if $\quad \langle \Delta, \sigma, \mathcal{E}, e \rangle \Longrightarrow \langle \Delta_e, \sigma', v \rangle$
and $\quad \Delta', \rho = \mathit{return}\ \Delta_e\ v_1$

$\boxed{\text{E-Await}}$

$\langle \Delta, \sigma, \mathcal{E}, \texttt{await}\ (p : \tau) = e_1\ \texttt{in}\ e_2 \rangle \Longrightarrow \langle \Delta', \sigma', \rho \rangle$
if $\quad \langle \Delta, \sigma, \mathcal{E}, e_1 \rangle \Longrightarrow \langle \Delta_1, \sigma', \rho_1 \rangle$
and $\quad \rho = \mathit{bind}\ \Delta_1\ \rho_1\ (\lambda(\Delta'', \sigma'', v'').\ \rho_2\ \mathit{where}\ v'' : p \rightsquigarrow \mathcal{E}''$
$\mathit{and}\ \langle \Delta'', \sigma'', \mathcal{E} \circ \mathcal{E}'', e_2 \rangle \Longrightarrow \langle \Delta_2, \sigma_2, \rho_2 \rangle)$

$\boxed{\text{E-Bind}}$

$\langle \Delta, \sigma, \mathcal{E}, e_1 \texttt{>>=} e_2 \rangle \Longrightarrow \langle \Delta', \sigma_2, \rho \rangle$
if $\quad \langle \Delta, \sigma, \mathcal{E}, e_1 \rangle \Longrightarrow \langle \Delta_1, \sigma_1, \rho_1 \rangle$
and $\quad \langle \Delta_1, \sigma_1, \mathcal{E}, e_2 \rangle \Longrightarrow \langle \Delta_2, \sigma_2, (|\mathcal{E}_{cl}, p, e_{cl}|) \rangle$
and $\quad \Delta', \rho = \mathit{bind}\ \Delta_2\ \rho_1\ (\lambda(\Delta'', \sigma'', v'').\ \rho_2\ \mathit{where}\ v'' : p \rightsquigarrow \mathcal{E}''$
$\mathit{and}\ \langle \Delta'', \sigma'', \mathcal{E}_{cl} \circ \mathcal{E}'', e_{cl} \rangle \Longrightarrow \langle \Delta', \sigma', \rho_2 \rangle)$

$\boxed{\text{E-Send}}$

$\langle \Delta, \sigma, \mathcal{E}, \texttt{send}\ e_1\ \texttt{to}\ e_2 \rangle \Longrightarrow \langle \Delta', \sigma', () \rangle$

$$
\begin{aligned}
&\text{if} \quad \langle \Delta, \sigma, \mathcal{E}, e_1 \rangle \Longrightarrow \langle \Delta_1, \sigma_1, v_1 \rangle \\
&\text{and} \quad \langle \Delta_1, \sigma_1, \mathcal{E}, e_2 \rangle \Longrightarrow \langle \Delta_2, \sigma', h_2 \rangle \\
&\text{and} \quad \Delta', () = \mathit{send}\ \Delta_2\ v_2\ h_1
\end{aligned}
$$

$\boxed{\text{E-Recv}}$

$$
\begin{aligned}
&\langle \Delta, \sigma, \mathcal{E}, \texttt{recv}\ \ e \rangle \Longrightarrow \langle \Delta', \sigma', \rho \rangle \\
&\quad \text{if} \quad \langle \Delta, \sigma, \mathcal{E}, e \rangle \Longrightarrow \langle \Delta_e, \sigma_e, h \rangle \\
&\quad \text{and} \quad \Delta', \rho = \mathit{recv}\ \Delta_e\ h
\end{aligned}
$$

$\boxed{\text{E-Spawn}}$

$$
\begin{aligned}
&\langle \Delta, \sigma, \mathcal{E}, \texttt{spawn}\ \ e_1 \text{ with } e_2 \rangle \Longrightarrow \langle \Delta', \sigma', h \rangle \\
&\quad \text{if} \quad \langle \Delta, \sigma, \mathcal{E}, e_1 \rangle \Longrightarrow \langle \Delta_1, \sigma_1, v_1 \rangle \\
&\quad \text{and} \quad \langle \Delta_1, \sigma_1, \mathcal{E}, e_2 \rangle \Longrightarrow \langle \Delta_2, \sigma', v_2 \rangle \\
&\quad \text{and} \quad \Delta', h = \mathit{spawn}\ \Delta_2\ v_1\ v_2
\end{aligned}
$$

$\boxed{\text{E-Self}}$

$$
\begin{aligned}
&\langle \Delta, \sigma, \mathcal{E}, \texttt{self} \rangle \Longrightarrow \langle \Delta, \sigma, h \rangle \\
&\quad \text{if} \quad h = \mathit{self}\mathcal{E}
\end{aligned}
$$