Syntax

We will adopt the following meta-variable conventions:

- $x \in \text{Var}$ variables
- $b \in \{\text{true, false}\}$ booleans
- $n \in \mathbb{Z}$ integers
- $s \in \Sigma^*$ ASCII strings
- $\ell \in \text{Loc}$ memory locations
- $h \in \text{Hand}$ process handles
- $\rho \in \text{Prom}$ promises

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

\[
e \in \text{Exp} ::= (\quad) \quad \text{Unit} \quad \left\{ 
\begin{array}{l}
\mid b \\
\mid n \\
\mid s \\
\mid x \\
\mid (e_1, e_2) \\
\mid [] \\
\mid e_1 :: e_2 \\
\mid \text{fun } p \to e \\
\mid \text{let } p = e_1 \text{ in } e_2 \\
\mid \text{let rec } f = \text{fun } p \to e_1 \text{ in } e_2 \\
\mid e_1 \cdot e_2 \\
\mid e_1 \oplus e_2 \\
\mid e_1; e_2 \\
\mid \text{if } e_1 \text{ then } e_2 \text{ else } e_3 \\
\mid \text{match } e_0 \text{ with } | p_1 \to e_1 \ldots | p_n \to e_n \text{ end} \\
\mid !e \\
\mid e_1 := e_2 \\
\mid \text{return } e \\
\mid \text{await } p = e_1 \text{ in } e_2 \\
\mid \text{join } e \\
\mid \text{pick } e \\
\mid \text{send } e_1 \text{ to } e_2 \\
\mid \text{recv } e \\
\mid \text{spawn } e_1 \text{ with } e_2
\end{array} \right.
\]

\[
\begin{array}{l}
\text{BOp} ::= + | - | * | / | \% | & & | || | < | <= | > | >= | = | <> | ^
\end{array}
\]

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

\[
p \in \text{Pat} ::= \left\{ 
\begin{array}{l}
\mid x | () \\
\mid b | n | s | (p_1, p_2) | [] | p_1 :: p_2
\end{array} \right.
\]

\[
\left\{ 
\begin{array}{l}
\circ \in \text{UOp} ::= \left\{ 
\text{not} \\
- \right. \\
\oplus \in \text{BOp} ::= + | - | * | / | \% | & & | || | < | <= | > | >= | = | <> | ^
\end{array} \right.
\]
The set of values is defined as follows, using the auxiliary definition for environments $E$, which is given below:

\[
v \in \text{Val} \quad ::= \quad () \quad \text{Unit} \\
| \quad b \quad \text{Boolean} \\
| \quad n \quad \text{Integers} \\
| \quad s \quad \text{Strings} \\
| \quad (v_1, v_2) \quad \text{Pairs} \\
| \quad [] \quad \text{Empty list} \\
| \quad v_1 :: v_2 \quad \text{Non-empty lists} \\
| \quad [\mathcal{E}, p, e] \quad \text{Closures} \\
| \quad \ell \quad \text{Locations} \\
| \quad \rho \quad \text{Promises} \\
| \quad h \quad \text{Handles}
\]

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

\[
E \in \text{Var} \rightarrow \text{Val} \\
\sigma \in \text{Loc} \rightarrow \text{Val}
\]

We use the following notation for describing environments:

- $\text{dom}(E)$ denotes the domain of $E$, 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,
- $E_1 \circ E_2$ denotes the environment that maps $x$ in $\text{dom}(E_2)$ to $E_2(x)$, $x$ in $\text{dom}(E_1)$ but not in $\text{dom}(E_2)$ to $E_1(x)$, and is otherwise undefined.

We use the same notation for the analogous operations on stores $\sigma$.

**Concurrency Library**

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 $\Delta \in \text{State}$ range over the run-time state of this library and assume the following operations:

\[
\begin{align*}
\text{return} & \in \text{State} \rightarrow \text{Val} \rightarrow \text{State} \times \text{Prom} \\
\text{join} & \in \text{State} \rightarrow \text{Prom List} \rightarrow \text{State} \times \text{Prom} \\
\text{pick} & \in \text{State} \rightarrow \text{Prom List} \rightarrow \text{State} \times \text{Prom} \\
\text{bind} & \in \text{State} \rightarrow \text{Prom} \rightarrow (\text{State} \times \text{Store} \times \text{Val} \rightarrow \text{State} \times \text{Store} \times \text{Prom}) \rightarrow \text{State} \times \text{Prom} \\
\text{send} & \in \text{State} \rightarrow \text{Hand} \rightarrow \text{Val} \rightarrow \text{State} \times \{()\} \\
\text{recv} & \in \text{State} \rightarrow \text{Hand} \rightarrow \text{State} \times \text{Prom}
\end{align*}
\]

**Semantics**

The semantics of an RML program can be obtained by modeling the behavior of the concurrency library. Intuitively, the run-time state $\Delta$ can be thought of as encoding multiple threads of execution, each with its own thread-local environment, store, and expression, and a single step $\Delta \rightarrow \Delta'$ models the sequential execution of a single thread until it relinquishes control back to the library. The overall behavior emerges by non-deterministically interleaving the steps for individual threads.

In this assignment, to keep things simple, we will not actually model this top-level operational semantics. Instead, we will formalize the evaluation of individual threads using a big-step semantics. Formally, we will define a relation,

\[
\langle \Delta, \sigma, E, e \rangle \Downarrow \langle \Delta', \sigma', v \rangle
\]
which can intuitively be read as follows: “under run-time state $\Delta$, with store $\sigma$, and environment $E$, the expression $e$ evaluates to run-time state $\Delta'$, store $\sigma'$, and value $v$.” Note that each big step does not necessarily fully evaluate $e$, but merely models its execution up until the next program point where it relinquishes control back to the concurrency library.

To define the big-step evaluation relation, we will use inference rules:

\[
\text{E-Sequence} \quad \frac{\langle \Delta, \sigma, E, e_1 \rangle \Downarrow \langle \Delta_1, \sigma_1, () \rangle \quad \langle \Delta, \sigma, E, e_2 \rangle \Downarrow \langle \Delta', \sigma', v \rangle}{\langle \Delta, \sigma, E, e_1; e_2 \rangle \Downarrow \langle \Delta, \sigma, v \rangle}
\]

Such rules are similar to the definitions we have seen in lecture, and can be read from bottom to top. The conclusion below the line, such as $\langle \Delta, \sigma, E, e \rangle \Downarrow \langle \Delta, \sigma, v \rangle$, holds if all of the premises above the line, such as $\langle \Delta, \sigma, E, e_1 \rangle \Downarrow \langle \Delta_1, \sigma_1, () \rangle$ and $\langle \Delta_1, \sigma_1, E, e_2 \rangle \Downarrow \langle \Delta', \sigma', v \rangle$, also hold. Note that any variables in the terms above the line, such as $\Delta_1$ and $\sigma_1$ may be filled in with arbitrary values, provided all of the constraints encoded in the premises are satisfied.

**Simple Expressions**

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

\[
\text{E-Value} \quad \frac{\langle \Delta, \sigma, E, v \rangle \Downarrow \langle \Delta, \sigma, v \rangle}{\langle \Delta, \sigma, E, v \rangle}
\]

\[
\text{E-Pair} \quad \frac{\langle \Delta, \sigma, E, (e_1, e_2) \rangle \Downarrow \langle \Delta', \sigma', (v_1, v_2) \rangle}{\langle \Delta, \sigma, E, e_1; e_2 \rangle \Downarrow \langle \Delta', \sigma', v_1 : v_2 \rangle}
\]

\[
\text{E-Cons} \quad \frac{\langle \Delta, \sigma, E, (e_1) \rangle \Downarrow \langle \Delta_1, \sigma_1, v_1 \rangle \quad \langle \Delta, \sigma, E, E, e_2 \rangle \Downarrow \langle \Delta', \sigma', e_2 \rangle}{\langle \Delta, \sigma, E, (e_1 :: e_2) \rangle \Downarrow \langle \Delta', \sigma', v_1 :: v_2 \rangle}
\]

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 $\Delta$ and store $\sigma$ 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 $\Delta$ and store $\sigma$ 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 use a three-place relation of the form $v : p \leadsto E$, read as “value $v$ matches pattern $p$ and produces the bindings in $E$.”

\[
\begin{align*}
v : \_ & \leadsto \{\} & \text{M-Wild} \\
v : x & \leadsto \{x \mapsto v\} & \text{M-Var} \\
() : () & \leadsto \{\} & \text{M-Unit} \\
b : b & \leadsto \{\} & \text{M-Bool} \\
n : n & \leadsto \{\} & \text{M-Int} \\
s : s & \leadsto \{\} & \text{M-String} \\
(v_1, v_2) : (p_1, p_2) & \leadsto E_1 \circ E_2 & \text{M-Pair} \\
v_1 : p_1 & \leadsto E_1 & \text{M-Cons} \\
v_2 : p_2 & \leadsto E_2 \\
\end{align*}
\]

\[
\begin{align*}
\text{M-Cons} & \quad \frac{v_1 : p_1 \leadsto E_1 \quad v_2 : p_2 \leadsto E_2}{v_1 :: v_2 : p_1 :: p_2 \leadsto E_1 \circ E_2}
\end{align*}
\]
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.

$$\frac{\langle \Delta, \sigma, E, e \rangle \Downarrow \langle \Delta_e, \sigma_e, v \rangle \quad v : p_j \rightsquigarrow E_j \text{ for } 0 < j < n + 1 \quad \langle \Delta_e, \sigma_e \circ E_j, e_j \rangle \Downarrow \langle \Delta', \sigma', v \rangle \quad v : p_i \not\rightsquigarrow E_i \text{ for } i < j}{\langle \Delta, \sigma, E, \text{match } e \text{ with } p_1 \rightarrow e_1 \ldots \ldots \ p_n \rightarrow e_n \text{ end} \rangle \Downarrow \langle \Delta', \sigma', v \rangle}$$

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, Definitions, and Application Expressions**

The next few inference rules handle functions, let-definitions, and application expressions.

$$\frac{}{\langle \Delta, \sigma, E, \text{fun } p \rightarrow e \rangle \Downarrow \langle \Delta, \sigma, \{E, p, e\} \rangle}$$

$$\frac{}{\langle \Delta, \sigma, E, e_1 \rangle \Downarrow \langle \Delta_1, \sigma_1, \{E_{cl}, p_{cl}, e_{cl}\} \rangle \quad v_2 : p_{cl} \rightsquigarrow E_2 \quad \langle \Delta, \sigma, \{E_{cl} \circ E_2, e_{cl}\} \rangle \Downarrow \langle \Delta', \sigma', v \rangle}{\langle \Delta, \sigma, E, e_2 \rangle \Downarrow \langle \Delta', \sigma', v \rangle}$$

$$\frac{}{\langle \Delta, \sigma, E, e_1 \rangle \Downarrow \langle \Delta_1, \sigma_1, v_1 \rangle \quad v_1 : p \rightsquigarrow E_1 \quad \langle \Delta, \sigma, \{E, \{E, e_1\}, e_2\} \rangle \Downarrow \langle \Delta', \sigma', v \rangle}{\langle \Delta, \sigma, E, \text{let } p = e_1 \text{ in } e_2 \rangle \Downarrow \langle \Delta', \sigma', v \rangle}$$

$$\frac{}{\langle \Delta, \sigma, E, \text{fun } p \rightarrow e \rangle \Downarrow \langle \Delta, \sigma, \{E, p, e\} \rangle}$$

These inference rules can be understood as follows:

**E-Fun**: a function $\text{fun } p \rightarrow e$ evaluates to a closure

**E-App**: an application $e_1 e_2$ evaluates $e_1$ to a closure $\{E, p, e\}$, 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-Let**: a let-definition evaluates 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 $E_f$ 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.

$$\frac{}{\langle \Delta, \sigma, E, e_1 \rangle \Downarrow \langle \Delta_1, \sigma_1, v_1 \rangle \quad v = \{\circ\} v_1}{\langle \Delta, \sigma, E, \text{\circ } e_1 \rangle \Downarrow \langle \Delta', \sigma', v \rangle}$$

$$\frac{}{\langle \Delta_1, \sigma_1, v_1 \rangle \quad \langle \Delta_1, \sigma_1, E, e_2 \rangle \Downarrow \langle \Delta', \sigma', v \rangle}{\langle \Delta, \sigma, E, e_1 \oplus e_2 \rangle \Downarrow \langle \Delta', \sigma', v \rangle}$$
These inference rules can be understood as follows:

**E-UOr**: a unary operation $\odot e_1$ evaluates $e_1$ to a value $v_1$ and then uses the implementation of the operation, denoted $[\circ]$, 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-BOr**: 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.

**E-Sequence**

$$
\frac{\langle \Delta, \sigma, E, e_1 \rangle \Downarrow \langle \Delta_1, \sigma_1, () \rangle \quad \langle \Delta, \sigma, E, e_2 \rangle \Downarrow \langle \Delta', \sigma', v \rangle}{\langle \Delta, \sigma, E, e_1; e_2 \rangle \Downarrow \langle \Delta', \sigma', v \rangle}
$$

**E-If-True**

$$
\frac{\langle \Delta, \sigma, E, e_1 \rangle \Downarrow \langle \Delta_1, \sigma_1, \text{true} \rangle \quad \langle \Delta, \sigma, E, e_2 \rangle \Downarrow \langle \Delta', \sigma', v \rangle}{\langle \Delta, \sigma, E, \text{if } e_1 \text{ then } e_2 \text{ else } e_3 \rangle \Downarrow \langle \Delta', \sigma', v \rangle}
$$

**E-If-False**

$$
\frac{\langle \Delta, \sigma, E, e_1 \rangle \Downarrow \langle \Delta_1, \sigma_1, \text{false} \rangle \quad \langle \Delta, \sigma, E, e_3 \rangle \Downarrow \langle \Delta', \sigma', v \rangle}{\langle \Delta, \sigma, E, \text{if } e_1 \text{ then } e_2 \text{ else } e_3 \rangle \Downarrow \langle \Delta', \sigma', v \rangle}
$$

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

The next few inference rules model OCaml-style references:

**E-Ref**

$$
\frac{\langle \Delta, \sigma, E, e \rangle \Downarrow \langle \Delta', \sigma, v \rangle \quad \ell \not\in \text{dom}(\sigma) \quad \sigma' = \sigma_e \circ \{ \ell \mapsto v \}}{\langle \Delta, \sigma, E, \text{ref } e \rangle \Downarrow \langle \Delta', \sigma', \ell \rangle}
$$

**E-Deref**

$$
\frac{\langle \Delta, \sigma, E, e \rangle \Downarrow \langle \Delta', \sigma', \ell \rangle \quad v = \sigma(\ell)}{\langle \Delta, \sigma, E, !e \rangle \Downarrow \langle \Delta', \sigma', v \rangle}
$$

**E-Assign**

$$
\frac{\langle \Delta, \sigma, E, e_1 \rangle \Downarrow \langle \Delta_1, \sigma_1, \ell \rangle \quad \langle \Delta, \sigma, E, e_2 \rangle \Downarrow \langle \Delta', \sigma_2, v \rangle \quad \sigma' = \sigma_2 \circ \{ \ell \mapsto v \}}{\langle \Delta, \sigma, E, e_1 := e_2 \rangle \Downarrow \langle \Delta', \sigma', () \rangle}
$$

These inference rules can be understood as follows:

**E-Ref**: a reference $\text{ref } e$ evaluates $e$ to a value $v$ and then adds it to the store $\sigma$ under a fresh location $\ell$, which is returned as the result.

**E-Deref**: a dereference $!e_1$ evaluates $e_1$ to a location $\ell$ and the looks it up in the store $\sigma$.

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

5
Asynchronous Expressions

The final inference rules model asynchronous expressions. Most of these rules simply call out to the corresponding functions from the concurrency library.

\[
\text{E-Return} \quad \frac{\langle \Delta, \sigma, E, e \rangle \Downarrow \langle \Delta_e, \sigma', v \rangle}{\langle \Delta, \sigma, \text{return } e \rangle \Downarrow \langle \Delta_e, \sigma', \rho \rangle}
\]

\[
\text{E-Await} \quad \frac{\langle \Delta, \sigma, E, e \rangle \Downarrow \langle \Delta_1, \sigma_1, \rho_1 \rangle \quad \langle \lambda(\Delta''', \sigma''', v''') \quad \rho_2 \quad (v'' : p \rightarrow \mathcal{E}'' \quad \text{and } \langle \Delta''', \sigma''', E \circ E'', e_2 \rangle \Downarrow \langle \Delta_2, \sigma_2, \rho_2 \rangle)}{\langle \Delta, \sigma, \text{await } p = e_1 \text{ in } e_2 \rangle \Downarrow \langle \Delta', \sigma', \rho \rangle}
\]

\[
\text{E-Join} \quad \frac{\langle \Delta, \sigma, E, e \rangle \Downarrow \langle \Delta_e, \sigma'[p_1; \ldots; p_n] \rangle}{\langle \Delta, \sigma, \text{join } e \rangle \Downarrow \langle \Delta', \sigma', \rho \rangle}
\]

\[
\text{E-Pick} \quad \frac{\langle \Delta, \sigma, E, e \rangle \Downarrow \langle \Delta_e, \sigma'[p_1; \ldots; p_n] \rangle}{\langle \Delta, \sigma, \text{pick } e \rangle \Downarrow \langle \Delta', \sigma', \rho \rangle}
\]

\[
\text{E-Send} \quad \frac{\langle \Delta, \sigma, E, e_1 \rangle \Downarrow \langle \Delta_1, \sigma_1, v_1 \rangle \quad \langle \Delta_1, \sigma_1, E, e_2 \rangle \Downarrow \langle \Delta_2, \sigma', h_2 \rangle}{\langle \Delta, \sigma, \text{send } e_1 \text{ to } e_2 \rangle \Downarrow \langle \Delta', \sigma', \langle \rangle \rangle}
\]

\[
\text{E-Recv} \quad \frac{\langle \Delta, \sigma, E, e \rangle \Downarrow \langle \Delta_e, \sigma_e, h \rangle}{\langle \Delta, \sigma, \text{recv } e \rangle \Downarrow \langle \Delta', \sigma', \rho \rangle}
\]

\[
\text{E-Spawn} \quad \frac{\langle \Delta, \sigma, E, e_1 \rangle \Downarrow \langle \Delta_1, \sigma_1, v_1 \rangle \quad \langle \Delta_1, \sigma_1, E, e_2 \rangle \Downarrow \langle \Delta_2, \sigma', v_2 \rangle}{\langle \Delta, \sigma, \text{spawn } e_1 \text{ with } e_2 \rangle \Downarrow \langle \Delta', \sigma', h \rangle}
\]

These inference rules can be understood as follows:

**E-Return:** a return expression \( \text{return } e \) evaluates \( e \) to a value \( v \) and uses the library function \( \text{return} \) to obtain a promise \( \rho \).

**E-Await:** an await expression \( \text{await } e_1 = p \) in \( e_2 \) evaluates \( e_1 \) to a promise \( \rho \) and then schedules a call-back with parameters \( p \) and body \( e_2 \) that is executed with \( \rho \) 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-Join:** a join expression \( \text{join } e \) evaluates \( e \) to a list of promises \( [\rho_1; \ldots; \rho_n] \) and then uses the library function \( \text{join} \) to obtain a promise \( \rho \) that resolves to a list containing the values that the promises \( \rho_1 \) to \( \rho_n \) resolve to.

**E-Pick:** a pick expression \( \text{pick } e \) evaluates \( e \) to a list of promises \( [\rho_1; \ldots; \rho_n] \) and then uses the library function \( \text{pick} \) to select a promise \( \rho \) from the list.

**E-Send:** an send expression \( \text{send } e_1 \text{ to } e_2 \) evaluates \( e_1 \) to a value \( v_1 \) and \( e_2 \) to a handle \( h_2 \) and then uses the library function \( \text{send} \) to send \( v \) to \( h \).

**E-Recv:** an receive expression \( \text{recv } e \) evaluates \( e \) to a handle \( h \) and then uses the library function \( \text{recv} \) to obtain a promise \( \rho \) that resolves to a message sent to \( h \).

**Simplified Semantics**

As you build your implementation of the semantics in OCaml, there are a few practical considerations worth keeping in mind. First, the \( \text{Dwt} \) module serves as the currency library and provides all of the operations needed.
to implement the asynchronous expressions. Second, it is not necessary to thread the run-time state $\Delta$ and store $\sigma$ 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 $\rightarrow$ expr $\rightarrow$ value. Generally speaking, you can simply elide the run-time state $\Delta$ and store $\sigma$ when mapping the big-step semantics rules to OCaml code.

It may be helpful to work with a simpler relation, $(\mathcal{E}, e) \Downarrow v$, which can intuitively be read as follows, “under environment $\mathcal{E}$, the expression $e$ evaluates to value $v$,” eliding the side effects on $\Delta$ and $\sigma$. Following is a formal definition of this relation, using streamlined versions of the concurrency library functions that elide $\Delta$.

$$
\begin{align*}
&\text{E-Value} & & (\mathcal{E}, v) \Downarrow v & & (\mathcal{E}(x) = v) \\
&\text{E-Var} & & (\mathcal{E}(x) \Downarrow v) & & \mathcal{E}(x) = v \\
&\text{E-Cons} & & (\mathcal{E}, e_1) \Downarrow v_1 & & \mathcal{E}, e_2) \Downarrow v_2 \quad (\mathcal{E}, e_1 :: e_2) \Downarrow v_1 :: v_2 \\
&\text{E-App} & & (\mathcal{E}, e_1) \Downarrow (\mathcal{E}, e_2) \Downarrow v_2 & & (\mathcal{E}, e_1 \circ e_2) \Downarrow (v_1, v_2) \\
&\text{E-Let} & & (\mathcal{E}, e_1) \Downarrow v_1 & & v_1 : p \rightarrow \mathcal{E}_1 & & \mathcal{E} \circ \mathcal{E}_1, e_2) \Downarrow v \\
&\text{E-LetRec} & & (\mathcal{E}, e_1) \Downarrow (\mathcal{E}, e_2) \Downarrow v_2 & & \mathcal{E}_1, e_2) \Downarrow v_2 & & (\mathcal{E}, e_1 + e_2) \Downarrow v \\
&\text{E-If-True} & & (\mathcal{E}, e_1) \Downarrow \text{true} & & (\mathcal{E}, e_2) \Downarrow v \\
&\text{E-If-False} & & (\mathcal{E}, e_1) \Downarrow \text{false} & & (\mathcal{E}, e_3) \Downarrow v \\
&\text{E-UOp} & & (\mathcal{E}, e_1) \Downarrow v_1 & & v_1 = [\bigcirc] v_1 \\
&\text{E-BOp} & & (\mathcal{E}, e_1) \Downarrow v_1 & & (\mathcal{E}, e_2) \Downarrow v_2 & & v = [\bigcirc] v_1 v_2 \\
&\text{E-Seq} & & (\mathcal{E}, e_1) \Downarrow () & & (\mathcal{E}, e_2) \Downarrow v \\
&\text{E-Par} & & (\mathcal{E}, e_1) \Downarrow () & & (\mathcal{E}, e_2) \Downarrow v \\
&\text{E-If} & & (\mathcal{E}, e_1) \Downarrow v_1 & & \text{if } e_1 \text{ then } v_1 \text{ else } v_2 \Downarrow v \\
&\text{E-Match} & & (\mathcal{E}, e_1) \Downarrow v_1 & & \text{match } e_1 \text{ with } v_1 & & \text{match } e_1 \text{ with } v_2 & & \text{match } e_1 \text{ with } v_3 & & \text{match } e_1 \text{ with } v_4 \\
&\text{E-Ref} & & (\mathcal{E}, e_1) \Downarrow \ell & & \ell \not\in \text{dom}(\sigma) & & \sigma' = \sigma \circ (\ell \mapsto v) \\
&\text{E-Deref} & & (\mathcal{E}, e_1) \Downarrow \ell & & v = \sigma(\ell) \\
&\text{E-Assign} & & (\mathcal{E}, e_1) \Downarrow \ell & & \ell \not\in \text{dom}(\sigma) & & \sigma' = \sigma \circ (\ell \mapsto v) \\
&\text{E-Return} & & (\mathcal{E}, e_1) \Downarrow \ell & & v = \sigma(\ell) \\
&\text{E-Await} & & (\mathcal{E}, e_1) \Downarrow p_1 & & p_1 = \text{bind } p_1 (\lambda v'. \text{ where } v' : p \rightarrow \mathcal{E}' \text{ and } \mathcal{E} \circ \mathcal{E}'', e_2) \Downarrow p_2 \\
&\text{E-Join} & & (\mathcal{E}, e_1) \Downarrow [p_1; \ldots ; p_n] & & p = \text{join } [p_1; \ldots ; p_n] \\
&\text{E-Pick} & & (\mathcal{E}, e_1) \Downarrow [p_1; \ldots ; p_n] & & p = \text{pick } [p_1; \ldots ; p_n] \\
&\text{E-Send} & & (\mathcal{E}, e_1) \Downarrow v_1 & & (\mathcal{E}, e_2) \Downarrow h_2 & & (\mathcal{E}, \text{send } e_1 \text{ to } e_2) \Downarrow () \\
&\text{E-Recv} & & (\mathcal{E}, e_1) \Downarrow v_1 & & (\mathcal{E}, e_2) \Downarrow v_2 & & v_1 = \text{spawn } e_1 v_2 \\
&\text{E-Spawn} & & (\mathcal{E}, e_1) \Downarrow v_1 & & (\mathcal{E}, e_2) \Downarrow v_2 & & v_1 = \text{spawn } e_1 v_2 \\
\end{align*}
$$