CS 3110 – Data Structures and Functional Programming
Robot ML Formal Semantics

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 $\odot$, and binary operations $\oplus$, given below:

$e \in Exp$	$::=$	$()$	*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*
	$\mid$	$\texttt{fun}\ p \rightarrow e$	*Functions*
	$\mid$	$\texttt{let}\ p = e_1\ \texttt{in}\ e_2$	*Let definitions*
	$\mid$	$\texttt{let rec}\ f = \texttt{fun}\ p \rightarrow e_1\ \texttt{in}\ e_2$	*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*
	$\mid$	$\texttt{match}\ e_0\ \texttt{with}\ \mid p_1 \rightarrow e_1\ \ldots \mid p_n \rightarrow e_n\ \texttt{end}$	*Pattern matching*
	$\mid$	$\texttt{ref}\ e$	*Reference creation*
	$\mid$	$!e$	*Dereference*
	$\mid$	$e_1 := e_2$	*Assignment*
	$\mid$	$\texttt{return}\ e$	*Asynchronous return*
	$\mid$	$\texttt{await}\ p = e_1\ \texttt{in}\ e_2$	*Asynchronous bind*
	$\mid$	$\texttt{join}\ e$	*Asynchronous join*
	$\mid$	$\texttt{pick}\ e$	*Asynchronous choice*
	$\mid$	$\texttt{send}\ e_1\ \texttt{to}\ e_2$	*Asynchronous message send*
	$\mid$	$\texttt{recv}\ e$	*Asynchronous message receive*
	$\mid$	$\texttt{spawn}\ e_1\ \texttt{with}\ e_2$	*Asynchronous spawn*

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

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

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

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

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

$$
\begin{array}{lll}
\mathcal{E} & \in & \text{Var} \rightharpoonup \textit{Val} \\
\sigma & \in & \text{Loc} \rightharpoonup \textit{Val}
\end{array}
$$

We use the following notation for describing environments:

- $dom(\mathcal{E})$ 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 $dom(\mathcal{E}_2)$ to $\mathcal{E}_2(x)$, x in $dom(\mathcal{E}_1)$ but not in $dom(\mathcal{E}_2)$ to $\mathcal{E}_1(x)$, and is otherwise undefined.

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

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

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

Semantics

The semantics of an RML program can be obtained by modeling the behavior of the concurrency library. Intuitively, the run-time state Δ can be thought of as encoding multiple threads of execution, each with its own thread-local environment, store, and expression, and a single step $\Delta \to \Delta'$ 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,

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

which can intuitively be read as follows: "under run-time state Δ, with store σ, and environment $\mathcal{E}$, the expression e evaluates to run-time state Δ', store σ', 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**:

$$\textsc{E-Sequence}\ \frac{\langle \Delta, \sigma, \mathcal{E}, e_1 \rangle \Downarrow \langle \Delta_1, \sigma_1, () \rangle \qquad \langle \Delta, \sigma, \mathcal{E}, e_2 \rangle \Downarrow \langle \Delta', \sigma', v \rangle}{\langle \Delta, \sigma, \mathcal{E}, e_1; e_2 \rangle \Downarrow \langle \Delta, \sigma, v \rangle}$$

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 $\langle \Delta, \sigma, \mathcal{E}, e \rangle \Downarrow \langle \Delta, \sigma, v \rangle$, holds if all of the **premises** above the line, such as $\langle \Delta, \sigma, \mathcal{E}, e_1 \rangle \Downarrow \langle \Delta_1, \sigma_1, () \rangle$ and $\langle \Delta_1, \sigma_1, \mathcal{E}, e_2 \rangle \Downarrow \langle \Delta', \sigma', v \rangle$, also hold. Note that any variables in the terms above the line, such as Δ_1 and σ_1 may be filled in with arbitrary values, provided all of the constaints encoded in the premises are satisfied.

Simple Expressions

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

$$\textsc{E-Value}\ \frac{}{\langle \Delta, \sigma, \mathcal{E}, v \rangle \Downarrow \langle \Delta, \sigma, v \rangle} \qquad \textsc{E-Var}\ \frac{\mathcal{E}(x) = v}{\langle \Delta, \sigma, \mathcal{E}, x \rangle \Downarrow \langle \Delta, \sigma, v \rangle}$$

$$\textsc{E-Pair}\ \frac{\begin{array}{c}\langle \Delta, \sigma, \mathcal{E}, e_1 \rangle \Downarrow \langle \Delta_1, \sigma_1, v_1 \rangle \\ \langle \Delta_1, \sigma_1, \mathcal{E}, e_2 \rangle \Downarrow \langle \Delta', \sigma', v_2 \rangle\end{array}}{\langle \Delta, \sigma, \mathcal{E}, (e_1, e_2) \rangle \Downarrow \langle \Delta', \sigma', (v_1, v_2) \rangle} \qquad \textsc{E-Cons}\ \frac{\begin{array}{c}\langle \Delta, \sigma, \mathcal{E}, e_1 \rangle \Downarrow \langle \Delta_1, \sigma_1, v_1 \rangle \\ \langle \Delta_1, \sigma_1, \mathcal{E}, e_2 \rangle \Downarrow \langle \Delta', \sigma', v_2 \rangle\end{array}}{\langle \Delta, \sigma, \mathcal{E}, e_1 :: e_2 \rangle \Downarrow \langle \Delta', \sigma', v_1 :: v_2 \rangle}$$

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 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 use a three-place relation of the form $v : p \rightsquigarrow \mathcal{E}$, read as "value v matches pattern p and produces the bindings in $\mathcal{E}$."

$$\frac{}{v : \_ \rightsquigarrow \{\}}\ \textsc{M-Wild} \qquad \frac{}{v : x \rightsquigarrow \{x \mapsto v\}}\ \textsc{M-Var} \qquad \frac{}{() : () \rightsquigarrow \{\}}\ \textsc{M-Unit} \qquad \frac{}{b : b \rightsquigarrow \{\}}\ \textsc{M-Bool}$$

$$\frac{}{n : n \rightsquigarrow \{\}}\ \textsc{M-Int} \qquad \frac{}{s : s \rightsquigarrow \{\}}\ \textsc{M-String} \qquad \frac{}{[\,] : [\,] \rightsquigarrow \{\}}\ \textsc{M-EmptyList}$$

$$\frac{v_1 : p_1 \rightsquigarrow \mathcal{E}_1 \qquad v_2 : p_2 \rightsquigarrow \mathcal{E}_2}{(v_1, v_2) : (p_1, p_2) \rightsquigarrow \mathcal{E}_1 \circ \mathcal{E}_2}\ \textsc{M-Pair} \qquad \frac{v_1 : p_1 \rightsquigarrow \mathcal{E}_1 \qquad v_2 : p_2 \rightsquigarrow \mathcal{E}_2}{v_1 :: v_2 : p_1 :: p_2 \rightsquigarrow \mathcal{E}_1 \circ \mathcal{E}_2}\ \textsc{M-Cons}$$

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.

$$\textsc{E-Match}\ \frac{\langle \Delta, \sigma, \mathcal{E}, e\rangle \Downarrow \langle \Delta_e, \sigma_e, v\rangle \qquad v : p_j \rightsquigarrow \mathcal{E}_j \text{ for } 0 < j < n+1 \\ \langle \Delta_e, \sigma_e, \mathcal{E} \circ \mathcal{E}_j, e_j\rangle \Downarrow \langle \Delta', \sigma', v\rangle \qquad v : p_i \not\rightsquigarrow \mathcal{E}_i \text{ for } i < j}{\langle \Delta, \sigma, \mathcal{E}, \texttt{match}\ e\ \texttt{with} \mid p_1 \to e_1\ \ldots \mid p_n \to e_n\ \texttt{end}\rangle \Downarrow \langle \Delta', \sigma', v\rangle}$$

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.

$$\textsc{E-Fun}\ \frac{}{\langle \Delta, \sigma, \mathcal{E}, \texttt{fun}\ p \to e\rangle \Downarrow \langle \Delta, \sigma, (\!| \mathcal{E}, p, e |\!)\rangle}$$

$$\textsc{E-App}\ \frac{\langle \Delta, \sigma, \mathcal{E}, e_1\rangle \Downarrow \langle \Delta_1, \sigma_1, (\!| \mathcal{E}_{cl}, p_{cl}, e_{cl} |\!)\rangle \\ \langle \Delta_1, \sigma_1, \mathcal{E}, e_2\rangle \Downarrow \langle \Delta_2, \sigma_2, v_2\rangle \quad v_2 : p_{cl} \rightsquigarrow \mathcal{E}_2 \qquad \langle \Delta, \sigma_2, \mathcal{E}_{cl} \circ \mathcal{E}_2, e_{cl}\rangle \Downarrow \langle \Delta', \sigma', v\rangle}{\langle \Delta, \sigma, \mathcal{E}, e_1\ e_2\rangle \Downarrow \langle \Delta', \sigma', v\rangle}$$

$$\textsc{E-Let}\ \frac{\langle \Delta, \sigma, \mathcal{E}, e_1\rangle \Downarrow \langle \Delta_1, \sigma_1, v_1\rangle \qquad v_1 : p \rightsquigarrow \mathcal{E}_1 \qquad \langle \Delta_1, \sigma_1, \mathcal{E} \circ \mathcal{E}_1, e_2\rangle \Downarrow \langle \Delta', \sigma', v\rangle}{\langle \Delta, \sigma, \mathcal{E}, \texttt{let}\ p = e_1\ \texttt{in}\ e_2\rangle \Downarrow \langle \Delta', \sigma', v\rangle}$$

$$\textsc{E-LetRec}\ \frac{\mathcal{E}_f = \mathcal{E}[f \mapsto (\!| \mathcal{E}_f, p, e_1 |\!)] \qquad \langle \Delta, \sigma, \mathcal{E}_f, e_2\rangle \Downarrow \langle \Delta', \sigma', v\rangle}{\langle \Delta, \sigma, \mathcal{E}, \texttt{let rec}\ f = \texttt{fun}\ p\ \to e_1\ \texttt{in}\ e_2\rangle \Downarrow \langle \Delta', \sigma', v\rangle}$$

These inference rules can be understood as follows:

E-Fun**:** a function `fun` $p \to e$ 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-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.

$$\textsc{E-UOp}\ \frac{\langle \Delta, \sigma, \mathcal{E}, e_1\rangle \Downarrow \langle \Delta', \sigma', v_1\rangle \qquad v = [\![\odot]\!]\ v_1}{\langle \Delta, \sigma, \mathcal{E}, \odot\ e_1\rangle \Downarrow \langle \Delta', \sigma', v\rangle}$$

$$\textsc{E-BOp}\ \frac{\langle \Delta, \sigma, \mathcal{E}, e_1\rangle \Downarrow \langle \Delta_1, \sigma_1, v_1\rangle \qquad \langle \Delta_1, \sigma_1, \mathcal{E}, e_2\rangle \Downarrow \langle \Delta', \sigma', v_2\rangle \qquad v = [\![\oplus]\!]\ v_1\ v_2}{\langle \Delta, \sigma, \mathcal{E}, e_1\ \oplus\ e_2\rangle \Downarrow \langle \Delta', \sigma', v\rangle}$$

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.

$$\text{E-SEQUENCE}\ \frac{\langle \Delta, \sigma, \mathcal{E}, e_1 \rangle \Downarrow \langle \Delta_1, \sigma_1, () \rangle \qquad \langle \Delta, \sigma, \mathcal{E}, e_2 \rangle \Downarrow \langle \Delta', \sigma', v \rangle}{\langle \Delta, \sigma, \mathcal{E}, e_1; e_2 \rangle \Downarrow \langle \Delta', \sigma', v \rangle}$$

$$\text{E-IF-TRUE}\ \frac{\langle \Delta, \sigma, \mathcal{E}, e_1 \rangle \Downarrow \langle \Delta_1, \sigma_1, \texttt{true} \rangle \qquad \langle \Delta_1, \sigma_1, \mathcal{E}, e_2 \rangle \Downarrow \langle \Delta', \sigma', v \rangle}{\langle \Delta, \sigma, \mathcal{E}, \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \rangle \Downarrow \langle \Delta', \sigma', v \rangle}$$

$$\text{E-IF-FALSE}\ \frac{\langle \Delta, \sigma, \mathcal{E}, e_1 \rangle \Downarrow \langle \Delta_1, \sigma_1, \texttt{false} \rangle \qquad \langle \Delta_1, \sigma_1, \mathcal{E}, e_3 \rangle \Downarrow \langle \Delta', \sigma', v \rangle}{\langle \Delta, \sigma, \mathcal{E}, \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \rangle \Downarrow \langle \Delta', \sigma', v \rangle}$$

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:

$$\text{E-REF}\ \frac{\langle \Delta, \sigma, \mathcal{E}, e \rangle \Downarrow \langle \Delta', \sigma_e, v \rangle \qquad \ell \notin dom(\sigma) \qquad \sigma' = \sigma_e \circ \{\ell \mapsto v\}}{\langle \Delta, \sigma, \mathcal{E}, \texttt{ref}\ e \rangle \Downarrow \langle \Delta', \sigma', \ell \rangle}$$

$$\text{E-DEREF}\ \frac{\langle \Delta, \sigma, \mathcal{E}, e \rangle \Downarrow \langle \Delta', \sigma', \ell \rangle \qquad v = \sigma(\ell)}{\langle \Delta, \sigma, \mathcal{E}, !e \rangle \Downarrow \langle \Delta', \sigma', v \rangle}$$

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

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. Most of these rules simply call out to the corresponding functions from the concurrency library.

$$\textsc{E-Return}\ \frac{\langle \Delta, \sigma, \mathcal{E}, e\rangle \Downarrow \langle \Delta_e, \sigma', v\rangle \qquad \Delta', \rho = \mathit{return}\ \Delta_e\ v_1}{\langle \Delta, \sigma, \mathcal{E}, \texttt{return}\ e\rangle \Downarrow \langle \Delta', \sigma', \rho\rangle}$$

$$\textsc{E-Await}\ \frac{\begin{array}{c}\langle \Delta, \sigma, \mathcal{E}, e_1\rangle \Downarrow \langle \Delta_1, \sigma_1, \rho_1\rangle \\ \Delta', \rho = \mathit{bind}\ \Delta_1\ \rho_1\ (\lambda(\Delta'', \sigma'', v'').\ \rho_2\ \mathit{where}\ v'' : p \rightsquigarrow \mathcal{E}''\ \mathit{and}\ \langle \Delta'', \sigma'', \mathcal{E} \circ \mathcal{E}'', e_2\rangle \Downarrow \langle \Delta_2, \sigma_2, \rho_2\rangle)\end{array}}{\langle \Delta, \sigma, \mathcal{E}, \texttt{await}\ p = e_1\ \texttt{in}\ e_2\rangle \Downarrow \langle \Delta', \sigma', \rho\rangle}$$

$$\textsc{E-Join}\ \frac{\langle \Delta, \sigma, \mathcal{E}, e\rangle \Downarrow \langle \Delta_e, \sigma', [\rho_1; \ldots; \rho_n]\rangle \qquad \Delta', \rho = \mathit{join}\ \Delta_e\ [\rho_1; \ldots; \rho_n]}{\langle \Delta, \sigma, \mathcal{E}, \texttt{join}\ e\rangle \Downarrow \langle \Delta', \sigma', \rho\rangle}$$

$$\textsc{E-Pick}\ \frac{\langle \Delta, \sigma, \mathcal{E}, e\rangle \Downarrow \langle \Delta_e, \sigma', [\rho_1; \ldots; \rho_n]\rangle \qquad \Delta', \rho = \mathit{pick}\ \Delta_e\ [\rho_1; \ldots; \rho_n]}{\langle \Delta, \sigma, \mathcal{E}, \texttt{pick}\ e\rangle \Downarrow \langle \Delta', \sigma', \rho\rangle}$$

$$\textsc{E-Send}\ \frac{\langle \Delta, \sigma, \mathcal{E}, e_1\rangle \Downarrow \langle \Delta_1, \sigma_1, v_1\rangle \qquad \langle \Delta_1, \sigma_1, \mathcal{E}, e_2\rangle \Downarrow \langle \Delta_2, \sigma', h_2\rangle \qquad \Delta', () = \mathit{send}\ \Delta_2\ v_2\ h_1}{\langle \Delta, \sigma, \mathcal{E}, \texttt{send}\ e_1\ \texttt{to}\ e_2\rangle \Downarrow \langle \Delta', \sigma', ()\rangle}$$

$$\textsc{E-Recv}\ \frac{\langle \Delta, \sigma, \mathcal{E}, e\rangle \Downarrow \langle \Delta_e, \sigma_e, h\rangle \qquad \Delta', \rho = \mathit{recv}\ \Delta_e\ h}{\langle \Delta, \sigma, \mathcal{E}, \texttt{recv}\ e\rangle \Downarrow \langle \Delta', \sigma', \rho\rangle}$$

$$\textsc{E-Spawn}\ \frac{\langle \Delta, \sigma, \mathcal{E}, e_1\rangle \Downarrow \langle \Delta_1, \sigma_1, v_1\rangle \qquad \langle \Delta_1, \sigma_1, \mathcal{E}, e_2\rangle \Downarrow \langle \Delta_2, \sigma', v_2\rangle \qquad \Delta', h = \mathit{spawn}\ \Delta_2\ v_1\ v_2}{\langle \Delta, \sigma, \mathcal{E}, \texttt{spawn}\ e_1\ \text{with}\ e_2\rangle \Downarrow \langle \Delta', \sigma', h\rangle}$$

These inference rules can be understood as follows:

E-Return: a return expression `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 with 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-Join: a join expression `join` e evaluates e to a list of promises $[\rho_1; \ldots; \rho_n]$ and then uses the library function join to obtain a promise ρ that resolves to a list containing the values that the promises ρ_1 to ρ_n resolve to.

E-Pick: a pick expression `pick` e evaluates e to a list of promises $[\rho_1; \ldots; \rho_n]$ and then uses the library function pick to select a promise ρ from the list.

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.

Simplified Semantics

As you build your implementation of the semantics in OCaml, there are a few practical considerations worth keeping in mind. First, the `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 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 a simpler relation, $\langle \mathcal{E}, e \rangle \Downarrow v$, which can intuitively be read as follows, "under environment $\mathcal{E}$, the expression e evaluates to value v," eliding the side effects on Δ and σ. Following is a formal definition of this relation, using streamlined versions of the concurrency library functions that elide Δ.

$$\textsc{E-Value}\ \frac{}{\langle \mathcal{E}, v \rangle \Downarrow v} \qquad \textsc{E-Var}\ \frac{\mathcal{E}(x) = v}{\langle \mathcal{E}, x \rangle \Downarrow v} \qquad \textsc{E-Pair}\ \frac{\langle \mathcal{E}, e_1 \rangle \Downarrow v_1 \qquad \langle \mathcal{E}, e_2 \rangle \Downarrow v_2}{\langle \mathcal{E}, (e_1, e_2) \rangle \Downarrow (v_1, v_2)}$$

$$\textsc{E-Cons}\ \frac{\langle \mathcal{E}, e_1 \rangle \Downarrow v_1 \qquad \langle \mathcal{E}, e_2 \rangle \Downarrow v_2}{\langle \mathcal{E}, e_1 :: e_2) \rangle \Downarrow v_1 :: v_2} \qquad \textsc{E-Fun}\ \frac{}{\langle \mathcal{E}, \texttt{fun}\ p \to e \rangle \Downarrow (\!| \mathcal{E}, p, e |\!)}$$

$$\textsc{E-App}\ \frac{\langle \mathcal{E}, e_1 \rangle \Downarrow (\!| \mathcal{E}_{cl}, p_{cl}, e_{cl} |\!) \qquad \langle \mathcal{E}, e_2 \rangle \Downarrow v_2 \quad v_2 : p_{cl} \rightsquigarrow \mathcal{E}_2 \qquad \langle \mathcal{E}_{cl} \circ \mathcal{E}_2, e_{cl} \rangle \Downarrow v}{\langle \mathcal{E}, e_1\ e_2 \rangle \Downarrow v}$$

$$\textsc{E-Let}\ \frac{\langle \mathcal{E}, e_1 \rangle \Downarrow v_1 \qquad v_1 : p \rightsquigarrow \mathcal{E}_1 \qquad \langle \mathcal{E} \circ \mathcal{E}_1, e_2 \rangle \Downarrow v}{\langle \mathcal{E}, \texttt{let}\ p = e_1\ \texttt{in}\ e_2 \rangle \Downarrow v} \qquad \textsc{E-LetRec}\ \frac{\mathcal{E}_f = \mathcal{E}[f \mapsto (\!| \mathcal{E}_f, p, e_1 |\!)] \qquad \langle \mathcal{E}_f, e_2 \rangle \Downarrow v}{\langle \mathcal{E}, \texttt{let rec}\ f\ p = e_1\ \texttt{in}\ e_2 \rangle \Downarrow v}$$

$$\textsc{E-UOp}\ \frac{\langle \mathcal{E}, e_1 \rangle \Downarrow v_1 \qquad v = [\![\odot]\!]\ v_1}{\langle \mathcal{E}, \odot\ e_1 \rangle \Downarrow v} \qquad \textsc{E-BOp}\ \frac{\langle \mathcal{E}, e_1 \rangle \Downarrow v_1 \qquad \langle \mathcal{E}, e_2 \rangle \Downarrow v_2 \qquad v = [\![\oplus]\!]\ v_1\ v_2}{\langle \mathcal{E}, e_1\ \oplus\ e_2 \rangle \Downarrow v}$$

$$\textsc{E-Sequence}\ \frac{\langle \mathcal{E}, e_1 \rangle \Downarrow () \qquad \langle \mathcal{E}, e_2 \rangle \Downarrow v}{\langle \mathcal{E}, e_1; e_2 \rangle \Downarrow v} \qquad \textsc{E-If-True}\ \frac{\langle \mathcal{E}, e_1 \rangle \Downarrow \texttt{true} \qquad \langle \mathcal{E}, e_2 \rangle \Downarrow v}{\langle \mathcal{E}, \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \rangle \Downarrow v}$$

$$\textsc{E-If-False}\ \frac{\langle \mathcal{E}, e_1 \rangle \Downarrow \texttt{false} \qquad \langle \mathcal{E}, e_3 \rangle \Downarrow v}{\langle \mathcal{E}, \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \rangle \Downarrow v} \qquad \textsc{E-Match}\ \frac{\begin{array}{c}\langle \mathcal{E}, e \rangle \Downarrow v \qquad v : p_j \rightsquigarrow \mathcal{E}_j \text{ for } 0 < j < n+1 \\ \langle \mathcal{E} \circ \mathcal{E}_j, e_j \rangle \Downarrow v \qquad v : p_i \not\rightsquigarrow \mathcal{E}_i \text{ for } i < j\end{array}}{\langle \mathcal{E}, \texttt{match}\ e\ \texttt{with}\ |\ p_1 \to e_1\ \ldots\ |\ p_n \to e_n\ \texttt{end} \rangle \Downarrow v}$$

$$\textsc{E-Ref}\ \frac{\langle \mathcal{E}, e \rangle \Downarrow v \qquad \ell \notin dom(\sigma) \qquad \sigma' = \sigma_e \circ \{\ell \mapsto v\}}{\langle \mathcal{E}, \texttt{ref}\ e \rangle \Downarrow \ell} \qquad \textsc{E-Deref}\ \frac{\langle \mathcal{E}, e \rangle \Downarrow \ell \qquad v = \sigma(\ell)}{\langle \mathcal{E}, !e \rangle \Downarrow v}$$

$$\textsc{E-Assign}\ \frac{\langle \mathcal{E}, e_1 \rangle \Downarrow \ell \qquad \langle \mathcal{E}, e_2 \rangle \Downarrow v \qquad \sigma' = \sigma_2 \circ \{\ell \mapsto v\}}{\langle \mathcal{E}, e_1 := e_2 \rangle \Downarrow ()} \qquad \textsc{E-Return}\ \frac{\langle \mathcal{E}, e \rangle \Downarrow v \qquad \rho = \mathit{return}\ v_1}{\langle \mathcal{E}, \texttt{return}\ e \rangle \Downarrow \rho}$$

$$\textsc{E-Await}\ \frac{\langle \mathcal{E}, e_1 \rangle \Downarrow \rho_1 \qquad \rho = \mathit{bind}\ \rho_1\ (\lambda v''.\ \rho_2\ \mathit{where}\ v'' : p \rightsquigarrow \mathcal{E}''\ \mathit{and}\ \langle \mathcal{E} \circ \mathcal{E}'', e_2 \rangle \Downarrow \rho_2)}{\langle \mathcal{E}, \texttt{await}\ p = e_1\ \texttt{in}\ e_2 \rangle \Downarrow \rho}$$

$$\textsc{E-Join}\ \frac{\langle \mathcal{E}, e \rangle \Downarrow [\rho_1; \ldots; \rho_n] \qquad \rho = \mathit{join}\ [\rho_1; \ldots; \rho_n]}{\langle \mathcal{E}, \texttt{join}\ e \rangle \Downarrow \rho} \qquad \textsc{E-Pick}\ \frac{\langle \mathcal{E}, e \rangle \Downarrow [\rho_1; \ldots; \rho_n] \qquad \rho = \mathit{pick}\ [\rho_1; \ldots; \rho_n]}{\langle \mathcal{E}, \texttt{pick}\ e \rangle \Downarrow \rho}$$

$$\textsc{E-Send}\ \frac{\langle \mathcal{E}, e_1 \rangle \Downarrow v_1 \qquad \langle \mathcal{E}, e_2 \rangle \Downarrow h_2 \qquad () = \mathit{send}\ v_2\ h_1}{\langle \mathcal{E}, \texttt{send}\ e_1\ \texttt{to}\ e_2 \rangle \Downarrow ()} \qquad \textsc{E-Recv}\ \frac{\langle \mathcal{E}, e \rangle \Downarrow h \qquad \rho = \mathit{recv}\ h}{\langle \mathcal{E}, \texttt{recv}\ e \rangle \Downarrow \rho}$$

$$\textsc{E-Spawn}\ \frac{\langle \mathcal{E}, e_1 \rangle \Downarrow v_1 \qquad \langle \mathcal{E}, e_2 \rangle \Downarrow v_2 \qquad h = \mathit{spawn}\ v_1\ v_2}{\langle \mathcal{E}, \texttt{spawn}\ e_1\ \text{with}\ e_2 \rangle \Downarrow h}$$