Ligo Formal Description

Syntax

The following describe the syntax of the simplify AST which is an internal of LIGO. The concrete syntax will be different depending of the choosen one but all the caterogies are present and the corresponding evaluation are the same

A LIGO program is a succession of declarations and expressions. Declarations add bindings to the environment while expressions are evaluated and yield values

variables (x, X)

Variable are the key to access elements in the environments. They are store with element they refers to at creation.

label (l)

Label identifies field in records, label always start with a letter

constructor (C)

Constructor creates custom type's literals for built-in types literals

declaration (d) $=$

$\mid$ *type* X *is* te	(Type variable declaration)
$\mid$ *const* x $(:\ te)?\ =\ e$	(Term Constant variable declaration)
$\mid$ *var* x $(:\ te)?\ =\ e$	(Term Mutable variable declaration)

declaration adds binding to the environment i.e. a pair of the variable and the expression it corresponds to

type expression (te) $=$

$\mid te\ (*\ te_i)+$	(type of tuple)
$\mid (\mid C_i\ of\ te_i)$	(type of sum)
$\mid \{\ l_i\ :\ te_i\ \}$	(type of record)
$\mid te1\ \rightarrow\ te2$	(type of function)
$\mid X$	(type variable)
$\mid Operator\ (te_i)$	(built in function on type)

The above describres all expression that are valid at the level of types

$$
\begin{array}{lr}
\textit{term expression } (e) \ = & \\
\quad | \ \textit{value} & \text{(values)} \\
\quad | \ \textit{built} - \textit{in } (e_i) & \text{(built-in function)} \\
\quad | \ x & \text{(variables)} \\
\quad | \ \lambda x \ . \ \textit{expr} & \text{(lambda absctraction)} \\
\quad | \ e_1 \ e_2 & \text{(application)} \\
\quad | \ \textit{let } x \ = \ e_1 \ \textit{in } e_2 & \text{(let in)} \\
\quad | \ (\ e_i\ ) & \text{(tuple)} \\
\quad | \ \{\ l_i \ = \ e_i\ \} & \text{(record)} \\
\quad | \ e(.a_i) & \text{(accessor)} \\
\quad | \ [\ e1_i \ = \ e2_i\ ] & \text{(map)} \\
\quad | \ [[\ e1_i \ = \ e2_i\ ]] & \text{(big map)} \\
\quad | \ e1[e2] & \text{(look up)} \\
\quad | \ \{\ e_i\ \} & \text{(set)} \\
\quad | \ [\ e_i\ ] & \text{(list)} \\
\quad | \ C \ e & \text{(constructor)} \\
\quad | \ \textit{match } e \textit{ with matching} & \text{(matching)} \\
\quad | \ e1; \ e2 & \text{(sequence)} \\
\quad | \ \textit{while } e1 \textit{ do } e2 & \text{(loop)} \\
\quad | \ x(.a_i) \ = \ e & \text{(assign)} \\
\quad | \ SKIP & \text{(skip)} \\
\quad | \ e \textit{ as } T & \text{(ascription)}
\end{array}
$$

The above describes all expression that are valid at the level of terms

$$
\begin{array}{lr}
\textit{value } (v) \ = & \\
\quad | \ \textit{literal} & \text{(values of built-in types)} \\
\quad | \ C \ v & \text{(values of construct types)} \\
\quad | \ \lambda x \ . \ \textit{expr} & \text{(lambda abstraction values)} \\
\quad | \ (\ v_i\ ) & \text{(tuple values)} \\
\quad | \ \{\ l_i \ = \ v_i\ \} & \text{(record values)} \\
\quad | \ [\ v1_i \ = \ v2_i\ ] & \text{(map values)} \\
\quad | \ [[\ v1_i \ = \ v2_i\ ]] & \text{(big map values)} \\
\quad | \ \{\ v_i\ \} & \text{(set values)} \\
\quad | \ [\ v_i\ ] & \text{(list values)}
\end{array}
$$

Values are valid termination of expression, it can be a built-in or construct literal or an abstraction

$$
\begin{array}{lr}
\textit{literal} \ = & \\
\quad | \ \textit{unit} & () \\
\quad | \ \textit{bool} & () \\
\quad | \ \textit{int} & () \\
\quad | \ \textit{nat} & () \\
\quad | \ \textit{mutez} & () \\
\quad | \ \textit{string} & () \\
\quad | \ \textit{bytes} & () \\
\quad | \ \textit{address} & () \\
\quad | \ \textit{timestamp} & () \\
\quad | \ \textit{operation} & ()
\end{array}
$$

the above lists all predefined literals supported by ligos

$$
\begin{aligned}
&\mathit{accessor}\ (a)\ = \\
&\qquad \mid i\ (\mathit{natural\ number}) && \text{(for tuples)} \\
&\qquad \mid l && \text{(for record)}
\end{aligned}
$$

Accessor are use to access fields of data structure. If S is a structure, S.a if field a of S.

$$
\begin{aligned}
&\mathit{matching}\ (m)\ = \\
&\qquad \mid \{\ \mathit{true}\ =>\ e;\ \mathit{false}\ =>\ e;\} && \text{(match bool)} \\
&\qquad \mid \{\ \mathit{nil}\ =>\ e;\ \mathit{cons}(\mathit{hd} :: \mathit{tl})\ =>\ e;\} && \text{(match list)} \\
&\qquad \mid \{\ \mathit{none}\ =>\ e;\ \mathit{some}(x)\ =>\ e;\} && \text{(match option)} \\
&\qquad \mid (x_i)\ =>\ e && \text{(match tuple)} \\
&\qquad \mid (\{\mathit{const}_i(x_i)\ =>\ e_i;\}\ ) && \text{(match variant)}
\end{aligned}
$$

Matchings represent the different branch of the control flow that are taken depending on what the value it is matched to i.e in "match a :bool with {true =¿ print 'toto'; false =¿ print 'tata'}" the program will display 'toto' if a is true and 'tata' if a is false. notice that if clause are just a boolean matching.

Evaluation of expression

The following describes how expression are evaluated to yield expressions $A \to A'$ reads as A evaluates to A', $\frac{P}{Q}$ reads as P implies Q

base

$$x\ \to\ v\ (\mathit{corresponding\ value\ in\ the\ environment}) \qquad \text{(E-VARIABLE)}$$

$$\mathit{built\ in}\ (e_i)\ \to\ \mathit{built\ in\ result}\ (*\ \mathit{evaluated\ depending\ on\ each\ case}\ *) \qquad \text{(E-BUILTIN)}$$

$$\mathit{SKIP}\ \to\ \mathit{unit} \qquad \text{(E-SKIP)}$$

$$(\lambda x.e)\ v\ \to\ [\ x\ \mapsto\ v\ ]\ e \qquad \text{(E-LAMBDA)}$$

Lambda expression are evaluated by replacing the bound variable in the inner expression with the value it is applied to

$$\frac{e1\ \to\ e1'}{e1\ e2\ \to\ e1'\ e2} \qquad \text{(E-APP1)}$$

$$\frac{e2\ \to\ e2'}{v1\ e2\ \to\ v1\ e2'} \qquad \text{(E-APP2)}$$

In application, expressions are evaluated from left to right

$$\frac{e1\ \to\ e1'}{\mathit{let}\ x = e1\ \mathit{in}\ e2\ \to\ \mathit{let}\ x = e1'\ \mathit{in}\ e2} \qquad \text{(E-LET)}$$

$$\mathit{let}\ x = v1\ \mathit{in}\ e2\ \to\ [x \to v1]\ e2 \qquad \text{(E-LETIN)}$$

In let in the first expression is evaluated to a value before using this value in place of the bound variable in the second expression

$$\frac{e1\ \to\ e1'}{e1;\ e2\ \to\ e1';\ e2} \qquad \text{(E-SEQ)}$$

$$\mathit{unit};\ e2\ \to\ e2 \qquad \text{(E-SEQNEXT)}$$

In sequence the left expression are evaluated first and shall yield a unit

$$\frac{\mathit{while}\ e1\ \mathit{then}\ e2}{\mathit{match}\ e1\ \mathit{with}\ \{\ \mathit{true}\ =>\ \text{`}e2;\ \mathit{while}\ e1\ \mathit{then}\ e2\text{`}\ ;\ \mathit{false}\ =>\ \mathit{skip}\}} \qquad \text{(E-LOOP)}$$

In loop, the condition expression is evaluted first, if it is true, the loop yield a sequence with the inner expression follow by a replication of the loop, otherwise the loop yield a unit

$$\frac{e\ \to\ e'}{x(.a_i)\ =\ e\ \to\ x(.a_i)\ =\ e'} \qquad \text{(E-ASSIGN1)}$$

$$x(.a_i) \;=\; v \;\rightarrow\; x' \; with \; x' \; as \; x \; with \; field \; (.a_i) \; replace \; by \; v \qquad \text{(E-ASSIGN2)}$$

In an assign expression, the expression to be assign is evaluated first and then the expression yield the data structure with the corresponding field replace buy the value yield by right expresion

$$\frac{e \;\rightarrow\; e'}{e \; as \; T \;\rightarrow\; e' \; as \; T} \qquad \text{(E-ASCR1)}$$

$$v \; as \; T \;\rightarrow\; v \qquad \text{(E-ASCR2)}$$

Ascription are dropped while being evaluated

data structure

$$\frac{e_j \;\rightarrow\; e'_j}{(v_i,\; e_j,\; e_k) \;\rightarrow\; (v_i,\; e'_j,\; e_k)} \qquad \text{(E-TUPLES)}$$

$$\frac{e_j \;\rightarrow\; e'_j}{\{l_i = v_i,\; l_j = e_j,\; l_k = e_k\} \;\rightarrow\; \{l_i = v_i,\; l_j = e'_j,\; l_k = e_k\}} \qquad \text{(E-RECORDS)}$$

$$\frac{e \;\rightarrow\; e'}{e(.a_i) \;\rightarrow\; e'(.a_i)} \qquad \text{(E-ACCESS)}$$

$$\frac{e1_j \;\rightarrow\; e1'_j}{[v1_i = v_i,\; e1_j = e2_j,\; e1_k = e2_k] \;\rightarrow\; [v1_i = v_i,\; e1'_j = e2_j,\; e1_k = e2_k]} \qquad \text{(E-MAP1)}$$

$$\frac{e2_j \;\rightarrow\; e2'_j}{[v1_i = v_i,\; v1_j = e2_j,\; e1_k = e2_k] \;\rightarrow\; [v1_i = v_i,\; v1_j = e2'_j,\; e1_k = e2_k]} \qquad \text{(E-MAP2)}$$

$$\frac{e1_j \;\rightarrow\; e1'_j}{[[v1_i = v_i,\; e1_j = e2_j,\; e1_k = e2_k]] \;\rightarrow\; [[v1_i = v_i,\; e1'_j = e2_j,\; e1_k = e2_k]]} \qquad \text{(E-BIGMAP1)}$$

$$\frac{e2_j \;\rightarrow\; e2'_j}{[[v1_i = v_i,\; v1_j = e2_j,\; e1_k = e2_k]] \;\rightarrow\; [[v1_i = v_i,\; v1_j = e2'_j,\; e1_k = e2_k]]} \qquad \text{(E-BIGMAP2)}$$

$$\frac{e_j \;\rightarrow\; e'_j}{[v_i,\; e_j,\; e_k] \;\rightarrow\; [v_i,\; e'_j,\; e_k]} \qquad \text{(E-LIST)}$$

$$\frac{e_j \;\rightarrow\; e'_j}{\{v_i,\; e_j,\; e_k\} \;\rightarrow\; \{v_i,\; e'_j,\; e_k\}} \qquad \text{(E-SET)}$$

These rules Means in a data structure, expression are fully evaluated from left to right until it reach a value.

look up

$$[v1_i = v2_i][v1_j] \;\rightarrow\; v2_j \qquad \text{(E-LUPMAP)}$$

$$[[v2_i = v2_i]][v1_j] \;\rightarrow\; v2_j \qquad \text{(E-LUPBIGMAP)}$$

These rules indicated that when a loop up is evaluate to the element in the data-structure whose key/label/indice correspond to the argument of the lookup (no evaluation if there is no correspondence)

matching

$$\frac{e \;\rightarrow\; e'}{C \; e \;\rightarrow\; C \; e'} \qquad \text{(E-CONST)}$$

Constructor are avaluated by evaluating the inner expression

$$\frac{e \;\rightarrow\; e'}{match \; e \; with \; m \;\rightarrow\; match \; e' \; with \; m} \qquad \text{(E-MATCH1)}$$

$$match \; v_i \; with \; m \;\rightarrow\; e_i \; (if \; \{v_i \;=>\; e_i\} \; in \; m) \qquad \text{(E-MATCH2)}$$

To sum up, in matching expression, the element to be match is first evaluated, then the expression is evaluated by looking in the matching value, the element to be match against and extracting the expression given.

Derive form

The following describe equivalent notation. The AST could define only the right expression and the CST define the left one as syntactic sugar

$$e1;\ e2 \iff (\lambda x : Unit.e2)\ e1\ \textit{with } x \textit{ not a } free\ variable\ in\ e1$$

$$let\ x = e1\ in\ e2 \iff (\lambda x : T1.e2)\ e1$$