

BATTERY – Basic and Algebraic Data Type-Supported Typed Expressions with Recursion and Polymorphism

1 INTRODUCTION

BATTERY is a toy language for the [COSE212](#) course at Korea University. BATTERY stands for **B**asic and **A**lgebraic **D**ata **T**ype-Supported **T**yped **E**xpressions with **R**ecursion and **P**olymorphism, and it supports the following features:

- **unit value** `()`
- **number values** `(0, 1, -1, 2, -2, 3, -3, ...)`
- **boolean values** `(true and false)`
- **string values** `("", "abc", "def", ...)` and **string concatenation** `(++)`
- **arithmetic operators**: negation `(-)`, addition `(+)`, subtraction `(-)`, multiplication `(*)`, division `(/)`, and modulo `(%)`
- **equality operators** `(==, !=)` and **arithmetic relational operators** `(<, >, <=, >=)`
- **logical operators**: conjunction `(&&)`, disjunction `(||)`, and negation `(!)`
- **sequences** `(;)` and **conditionals** `(if-else)`
- **(immutable) variable definitions** `(val)`
- **first-class functions** `(=>)` and **function applications**
- **mutually recursive definitions**:
 - **lazy variable definitions** `(lazy val)`
 - **polymorphic recursive functions** `(def)`
 - **polymorphic algebraic data types** `(enum)`
- **pattern matching** `(match)`
- **exit** `(exit)`
- **static type checking**

This document is the specification of BATTERY. First, Section 2 describes the concrete syntax, and Section 3 describes the abstract syntax with the desugaring rules. Then, Section 4 describes the type system. Finally, Section 5 describes the big-step operational (natural) semantics of BATTERY.

2 CONCRETE SYNTAX

The concrete syntax of BATTERY is written in a variant of the extended Backus–Naur form (EBNF). The notation `<nt>` denotes a nonterminal, and `"t"` denotes a terminal. We use `?` to denote an optional element and `+` (or `*`) to denote one or more (or zero or more) repetitions of the preceding element. The notation `+{A}` or `*{A}` denotes the same as `+` or `*`, respectively, but the elements are separated by the element `A`. We use **butnot** to denote a set difference to exclude some strings from a producible set of strings. We omit some obvious terminals using the ellipsis `(...)` notation.

```
// basic elements
<digit>    ::= "0" | "1" | "2" | ... | "9"
<number>   ::= "-"? <digit>+
<alphabet> ::= "A" | "B" | "C" | ... | "Z" | "a" | "b" | "c" | ... | "z"
<idstart>  ::= <alphabet> | "_"
<idcont>   ::= <alphabet> | "_" | <digit>
<char>     ::= /* any character except "'" */
<string>   ::= "\" <char>* "\"
```

```

<keyword> ::= "Boolean" | "Number" | "String" | "Unit"
           | "case" | "def" | "else" | "exit" | "enum" | "false"
           | "if" | "lazy" | "match" | "true" | "val"
<id>      ::= <idstart> <idcont>* butnot <keyword>

// expressions
<expr> ::= "(" | <number> | "true" | "false" | <string> | <id>
         | <uop> <expr> | <expr> <bop> <expr>
         | "(" <expr> ")" | "{" <expr> "}" | <expr> ";"? <expr>
         | "if" "(" <expr> ")" <expr> "else" <expr>
         | "val" <id> [ ":" <type> ]? "=" <expr> ";"? <expr>
         | <params> ">" <expr>
         | <expr> [ "[" <type>*{"", "}" "]" ]? "(" <expr>*{"", "}" ")"
         | [ <recdef> ";"? ]+ <expr>
         | <expr> "match" "{" [ <mcase> ";"? ]+ "}"
         | "exit" "[" <type> "]" "(" <expr> ")"

// operators
<uop> ::= "-" | "!"
<bop> ::= "+" | "-" | "*" | "/" | "%" | "++" | "&&" | "||"
         | "==" | "!=" | "<" | "<=" | ">" | ">="

// function parameters
<params> ::= "(" <param>*{"", "}" ")"
<param>  ::= <id> ":" <type>

// recursive definitions
<recdef> ::= "lazy" "val" <id> "=" <expr>
         | "def" <id> <tvars>? <params> ":" <type> "=" <expr>
         | "enum" <id> <tvars>? "{" [ <variant> ";"? ]+ "}"

// type variables
<tvars> ::= "[" <id>*{"", "}" "]"

// variants
<variant> ::= "case" <id> "(" [ <id> ":" <type> ]*{"", "}" ")"

// match cases
<mcase>  ::= "case" <id> "(" <id>*{"", "}" ")" ">" <expr>

// types
<type> ::= "(" <type> ")" | "Unit" | "Number" | "Boolean" | "String"
         | <id> [ "[" <type>*{"", "}" "]" ]?
         | <tvars>? <type> ">" <type>
         | <tvars>? "(" <type>*{"", "}" ")" ">" <type>

```

For types, the arrow (\Rightarrow) operator is right-associative. For expressions, the precedence and associativity of operators are defined as follows:

Description	Operator	Precedence	Associativity
Unary	$-, !$	1	right
Multiplicative	$*, /, \%$	2	left
Additive	$++, +, -$	3	
Relational	$<, <=, >, >=$	4	
Equality	$==, !=$	5	
Logical Conjunction	$\&\&$	6	
Logical Disjunction	$ $	7	
Pattern Matching	<code>match</code>	8	

3 ABSTRACT SYNTAX

The abstract syntax of BATTERY is defined as follows:

Expressions	$\mathbb{E} \ni e ::= ()$	(EUnit)	$ e == e$	(EEq)
	$ n$	(ENum)	$ e < e$	(ELt)
	$ b$	(EBool)	$ e ; e$	(ESeq)
	$ s$	(EStr)	$ \text{if } (e) \ e \text{ else } e$	(EIf)
	$ x$	(EId)	$ \text{val } x [:\tau]^2 = e; e$	(EVal)
	$ e + e$	(EAdd)	$ \lambda([x:\tau]^*).e$	(EFun)
	$ e * e$	(EMul)	$ e[\tau^*](e^*)$	(EApp)
	$ e / e$	(EDiv)	$ d^+ e$	(ERecDefs)
	$ e \% e$	(EMod)	$ e \text{ match } \{ [\text{case } x(x^*) \Rightarrow e]^+ \}$	(EMatch)
	$ e ++ e$	(EConcat)	$ \text{exit}[\tau](e)$	(EExit)
Recursive Definitions	$\mathbb{D} \ni d ::= \text{lazy } x:\tau = e$			(LazyVal)
	$ \text{def } x[\alpha^*]([x:\tau]^*):\tau = e$			(RecFun)
	$ \text{enum } t[\alpha^*] \{ [\text{case } x([x:\tau]^*)]^+ \}$			(TypeDef)
Types	$\mathbb{T} \ni \tau ::= \text{unit}$	(UnitT)	Identifiers	$x \in \mathbb{X}$ (String)
	$ \text{num}$	(NumT)	Numbers	$n \in \mathbb{Z}$ (BigInt)
	$ \text{bool}$	(BoolT)	Booleans	$b \in \mathbb{B}$ (Boolean)
	$ \text{str}$	(StrT)	Strings	$s \in \mathbb{S}$ (String)
	$ t[\tau^*]$	(IdT)	Type Names	$t \in \mathbb{X}_t$ (String)
	$ \alpha$	(IdT)	Type Variables	$\alpha \in \mathbb{X}_\alpha$ (String)
	$ [\alpha^*](\tau^*) \rightarrow \tau$	(ArrowT)		

For type names $t[\tau^*]$ and arrow types $[\alpha^*](\tau^*) \rightarrow \tau$, we omit the square brackets (t and $(\tau^*) \rightarrow \tau$) when their type arguments (τ^*) or type variables (α^*) are empty, respectively. The types or semantics of the remaining cases are defined with the following desugaring rules:

$\mathcal{D}[-e]$	$= \mathcal{D}[e] * (-1)$	$\mathcal{D}[e_1 != e_2]$	$= \mathcal{D}[\neg (e_1 == e_2)]$
$\mathcal{D}[\neg e]$	$= \text{if } (\mathcal{D}[e]) \text{ false else true}$	$\mathcal{D}[e_1 <= e_2]$	$= \mathcal{D}[(e_1 < e_2) \vee (e_1 == e_2)]$
$\mathcal{D}[e_1 - e_2]$	$= \mathcal{D}[e_1] + \mathcal{D}[-e_2]$	$\mathcal{D}[e_1 > e_2]$	$= \mathcal{D}[\neg (e_1 <= e_2)]$
$\mathcal{D}[e_1 \&\& e_2]$	$= \text{if } (\mathcal{D}[e_1]) \ \mathcal{D}[e_2] \text{ else false}$	$\mathcal{D}[e_1 >= e_2]$	$= \mathcal{D}[\neg (e_1 < e_2)]$
$\mathcal{D}[e_1 \vee e_2]$	$= \text{if } (\mathcal{D}[e_1]) \text{ true else } \mathcal{D}[e_2]$		

The omitted cases recursively apply the desugaring rule to sub-expressions.

4 TYPE SYSTEM

This section explains type system of BATTERY, and we use the following notations:

Type Environments $\Gamma \in (\mathbb{X} \xrightarrow{\text{fin}} \mathbb{T}) \times (\mathbb{X}_t \xrightarrow{\text{fin}} (\mathbb{X}_\alpha^* \times (\mathbb{X} \xrightarrow{\text{fin}} \mathbb{T}^*))) \times \mathcal{P}(\mathbb{X}_\alpha)$ (TypeEnv)

A type environment Γ consists of three components: 1) a variable mapping $\mathbb{X} \xrightarrow{\text{fin}} \mathbb{T}$ that maps variables to their types, 2) a type name mapping $\mathbb{X}_t \xrightarrow{\text{fin}} (\mathbb{X}_\alpha^* \times (\mathbb{X} \xrightarrow{\text{fin}} \mathbb{T}^*))$ that maps type names to their type variables and commutative variants, and 3) a set of type variables that are currently in scope. In the type system, type checking is defined with the following typing rules:

$$\boxed{\Gamma \vdash e : \tau}$$

$$\begin{array}{c}
\tau\text{-EUnit} \frac{}{\Gamma \vdash () : \text{unit}} \quad \tau\text{-ENum} \frac{}{\Gamma \vdash n : \text{num}} \\
\\
\tau\text{-EBool} \frac{}{\Gamma \vdash b : \text{bool}} \quad \tau\text{-EStr} \frac{}{\Gamma \vdash s : \text{str}} \quad \tau\text{-EId} \frac{x \in \text{Domain}(\Gamma)}{\Gamma \vdash x : \Gamma(x)} \\
\\
\tau\text{-EAdd} \frac{\Gamma \vdash e_1 : \text{num} \quad \Gamma \vdash e_2 : \text{num}}{\Gamma \vdash e_1 + e_2 : \text{num}} \quad \tau\text{-EMul} \frac{\Gamma \vdash e_1 : \text{num} \quad \Gamma \vdash e_2 : \text{num}}{\Gamma \vdash e_1 * e_2 : \text{num}} \\
\\
\tau\text{-EDiv} \frac{\Gamma \vdash e_1 : \text{num} \quad \Gamma \vdash e_2 : \text{num}}{\Gamma \vdash e_1 / e_2 : \text{num}} \quad \tau\text{-EMod} \frac{\Gamma \vdash e_1 : \text{num} \quad \Gamma \vdash e_2 : \text{num}}{\Gamma \vdash e_1 \% e_2 : \text{num}} \\
\\
\tau\text{-EConcat} \frac{\Gamma \vdash e_1 : \text{str} \quad \Gamma \vdash e_2 : \text{str}}{\Gamma \vdash e_1 ++ e_2 : \text{str}} \\
\\
\tau\text{-EEq} \frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2 \quad \tau_1 \equiv \tau_2}{\Gamma \vdash e_1 == e_2 : \text{bool}} \quad \tau\text{-ELt} \frac{\Gamma \vdash e_1 : \text{num} \quad \Gamma \vdash e_2 : \text{num}}{\Gamma \vdash e_1 < e_2 : \text{bool}} \\
\\
\tau\text{-ESeq} \frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1 ; e_2 : \tau_2} \quad \tau\text{-EIf} \frac{\Gamma \vdash e_0 : \text{bool} \quad \Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2 \quad \tau_1 \equiv \tau_2}{\Gamma \vdash \text{if } (e_0) e_1 \text{ else } e_2 : \tau} \\
\\
\tau\text{-Eval} \frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma[x : \tau_1] \vdash e_2 : \tau_2}{\Gamma \vdash \text{val } x = e_1; e_2 : \tau_2} \quad \tau\text{-Eval}_\tau \frac{\Gamma \vdash e_1 : \tau_1 \quad \tau_0 \equiv \tau_1 \quad \Gamma[x : \tau_1] \vdash e_2 : \tau_2}{\Gamma \vdash \text{val } x : \tau_0 = e_1; e_2 : \tau_2} \\
\\
\tau\text{-EFun} \frac{\Gamma \vdash \tau_1 \quad \dots \quad \Gamma \vdash \tau_n \quad \Gamma[x_1 : \tau_1, \dots, x_n : \tau_n] \vdash e : \tau}{\Gamma \vdash \lambda(x_1 : \tau_1, \dots, x_n : \tau_n).e : (\tau_1, \dots, \tau_n) \rightarrow \tau} \\
\\
\tau\text{-EApp} \frac{\Gamma \vdash e_0 : [\alpha_1, \dots, \alpha_m](\tau'_1, \dots, \tau'_n) \rightarrow \tau' \quad \Gamma \vdash e_1 : \tau''_1 \quad \dots \quad \Gamma \vdash e_n : \tau''_n \quad \tau'_1 \equiv \tau'_1[\alpha_1 \leftarrow \tau_1, \dots, \alpha_m \leftarrow \tau_m] \quad \dots \quad \tau'_n \equiv \tau'_n[\alpha_1 \leftarrow \tau_1, \dots, \alpha_m \leftarrow \tau_m]}{\Gamma \vdash e_0[\tau_1, \dots, \tau_m](e_1, \dots, e_n) : \tau}
\end{array}$$

$$\begin{array}{c}
\tau\text{-ERecDefs} \frac{\Gamma \vdash d_1 \vdash \Gamma_1 \quad \dots \quad \Gamma_{n-1} \vdash d_n \vdash \Gamma_n \quad \Gamma_n \models d_1 \quad \dots \quad \Gamma_n \models d_n \quad \Gamma_n \vdash e : \tau}{\Gamma \vdash d_1; \dots; d_n; e : \tau} \\
\\
\tau\text{-EMatch} \frac{\begin{array}{c} \Gamma \vdash e : t[\tau_1, \dots, \tau_m] \quad \Gamma(t) = [\alpha_1, \dots, \alpha_m] \{x_1(\tau_{1,1}, \dots, \tau_{1,m_1}) + \dots + x_n(\tau_{n,1}, \dots, \tau_{n,m_n})\} \\ \forall 1 \leq i \leq n. \Gamma_i = \Gamma[x_{i,1} : \tau_{i,1}[\alpha_1 \leftarrow \tau_1, \dots, \alpha_m \leftarrow \tau_m], \dots, x_{i,m_i} : \tau_{i,m_i}[\alpha_1 \leftarrow \tau_1, \dots, \alpha_m \leftarrow \tau_m]] \\ \Gamma_1 \vdash e_1 : \tau_1 \quad \dots \quad \Gamma_n \vdash e_n : \tau_n \quad \tau_1 \equiv \dots \equiv \tau_n \end{array}}{\Gamma \vdash e \text{ match } \{ \text{case } x_1(x_{1,1}, \dots, x_{1,m_1}) \Rightarrow e_1; \dots; \text{case } x_n(x_{n,1}, \dots, x_{n,m_n}) \Rightarrow e_n \} : \tau} \\
\\
\tau\text{-EExit} \frac{\Gamma \vdash \tau \quad \Gamma \vdash e : \text{str}}{\Gamma \vdash \text{exit}[\tau](e) : \tau}
\end{array}$$

the following type environment update rules for recursive definitions:

$$\begin{array}{c}
\boxed{\Gamma \vdash d \vdash \Gamma} \\
\\
\text{LazyVal} \frac{}{\Gamma \vdash \text{lazy } x : \tau = e \vdash \Gamma[x : \tau]} \\
\\
\text{RecFun} \frac{}{\Gamma \vdash \text{def } x[\alpha_1, \dots, \alpha_m](x_1 : \tau_1, \dots, x_n : \tau_n) : \tau = e \vdash \Gamma[x : [\alpha_1, \dots, \alpha_m](\tau_1, \dots, \tau_n) \rightarrow \tau]} \\
\\
\text{TypeDef} \frac{\begin{array}{c} t \notin \text{Domain}(\Gamma) \quad \Gamma_0 = \Gamma[t = [\alpha_1, \dots, \alpha_m]\{x_1(\tau_{1,1}, \dots, \tau_{1,m_1}) + \dots + x_n(\tau_{n,1}, \dots, \tau_{n,m_n})\}] \\ \Gamma_1 = \Gamma_0[x_1 : [\alpha_1, \dots, \alpha_m](\tau_{1,1}, \dots, \tau_{1,m_1}) \rightarrow t[\tau_1, \dots, \tau_m]] \\ \dots \quad \Gamma_n = \Gamma_{n-1}[x_n : [\alpha_1, \dots, \alpha_m](\tau_{n,1}, \dots, \tau_{n,m_n}) \rightarrow t[\tau_1, \dots, \tau_m]] \end{array}}{\Gamma \vdash \text{enum } t[\alpha_1, \dots, \alpha_m] \left\{ \begin{array}{l} \text{case } x_1(x_{1,1} : \tau_{1,1}, \dots, x_{1,m_1} : \tau_{1,m_1}); \\ \dots \\ \text{case } x_n(x_{n,1} : \tau_{n,1}, \dots, x_{n,m_n} : \tau_{n,m_n}) \end{array} \right\} \vdash \Gamma_n}
\end{array}$$

the following typing rules for recursive definitions:

$$\begin{array}{c}
\boxed{\Gamma \models d} \\
\\
\text{LazyVal} \frac{\Gamma \vdash \tau_0 \quad \Gamma \vdash e_1 : \tau_1 \quad \tau_0 \equiv \tau_1}{\Gamma \models \text{lazy } x : \tau_0 = e_1} \\
\\
\text{RecFun} \frac{\begin{array}{c} \alpha_1 \notin \text{Domain}(\Gamma) \quad \dots \quad \alpha_m \notin \text{Domain}(\Gamma) \quad \Gamma' = \Gamma[\alpha_1, \dots, \alpha_m] \\ \Gamma' \vdash \tau_1 \quad \dots \quad \Gamma' \vdash \tau_n \quad \Gamma' \vdash \tau \quad \Gamma'[x_1 : \tau_1, \dots, x_n : \tau_n] \vdash e : \tau' \quad \tau \equiv \tau' \end{array}}{\Gamma \models \text{def } x[\alpha_1, \dots, \alpha_m](x_1 : \tau_1, \dots, x_n : \tau_n) : \tau = e} \\
\\
\text{TypeDef} \frac{\begin{array}{c} \alpha_1 \notin \text{Domain}(\Gamma) \quad \dots \quad \alpha_m \notin \text{Domain}(\Gamma) \\ \Gamma' = \Gamma[\alpha_1, \dots, \alpha_m] \quad \Gamma' \vdash \tau_{1,1} \quad \dots \quad \Gamma' \vdash \tau_{n,m_n} \end{array}}{\Gamma \models \text{enum } t[\alpha_1, \dots, \alpha_m] \left\{ \begin{array}{l} \text{case } x_1(x_{1,1} : \tau_{1,1}, \dots, x_{1,m_1} : \tau_{1,m_1}); \\ \dots \\ \text{case } x_n(x_{n,1} : \tau_{n,1}, \dots, x_{n,m_n} : \tau_{n,m_n}) \end{array} \right\}}
\end{array}$$

the following rules for well-formedness of types:

$$\begin{array}{c}
 \boxed{\Gamma \vdash \tau} \\
 \\
 \overline{\Gamma \vdash \text{unit}} \quad \overline{\Gamma \vdash \text{num}} \quad \overline{\Gamma \vdash \text{bool}} \quad \overline{\Gamma \vdash \text{str}} \\
 \\
 \frac{\Gamma(t) = [\alpha_1, \dots, \alpha_m] \{x_1(\tau_{1,1}, \dots, \tau_{1,m_1}) + \dots + x_n(\tau_{n,1}, \dots, \tau_{n,m_n})\} \quad \Gamma \vdash \tau_1 \quad \dots \quad \Gamma \vdash \tau_m}{\Gamma \vdash t[\tau_1, \dots, \tau_m]} \\
 \\
 \frac{\alpha \in \text{Domain}(\Gamma)}{\Gamma \vdash \alpha} \quad \frac{\Gamma' = \Gamma[\alpha_1, \dots, \alpha_m] \quad \Gamma' \vdash \tau_1 \quad \dots \quad \Gamma' \vdash \tau_n \quad \Gamma' \vdash \tau}{\Gamma \vdash [\alpha_1, \dots, \alpha_m](\tau_1, \dots, \tau_n) \rightarrow \tau}
 \end{array}$$

and the following rules for type equivalence:

$$\begin{array}{c}
 \boxed{\tau \equiv \tau} \\
 \\
 \overline{\text{unit} \equiv \text{unit}} \quad \overline{\text{num} \equiv \text{num}} \quad \overline{\text{bool} \equiv \text{bool}} \quad \overline{\text{str} \equiv \text{str}} \\
 \\
 \frac{\tau_1 \equiv \tau'_1 \quad \dots \quad \tau_m \equiv \tau'_m}{t[\tau_1, \dots, \tau_m] \equiv t[\tau'_1, \dots, \tau'_m]} \quad \overline{\alpha \equiv \alpha} \\
 \\
 \frac{\tau_1 \equiv \tau'_1[\alpha'_1 \leftarrow \alpha_1, \dots, \alpha'_m \leftarrow \alpha_m] \quad \dots \quad \tau_n \equiv \tau'_n[\alpha'_1 \leftarrow \alpha_1, \dots, \alpha'_m \leftarrow \alpha_m]}{[\alpha_1, \dots, \alpha_m](\tau_1, \dots, \tau_n \rightarrow \tau) \equiv [\alpha'_1, \dots, \alpha'_m](\tau'_1, \dots, \tau'_n \rightarrow \tau')}
 \end{array}$$

5 SEMANTICS

We use the following notations in the semantics:

$$\begin{array}{lcl}
 \text{Environments} & \sigma \in \mathbb{X} \xrightarrow{\text{fin}} \mathbb{V} & (\text{Env}) \\
 \\
 \text{Values} \quad \mathbb{V} \ni v ::= () & (\text{UnitV}) & | \langle \lambda x. (e, \dots, e), \sigma \rangle \quad (\text{CloV}) \\
 & | n & (\text{NumV}) \quad | \langle \langle e, \sigma \rangle \rangle \quad (\text{ExprV}) \\
 & | b & (\text{BoolV}) \quad | \langle x \rangle \quad (\text{ConstrV}) \\
 & | s & (\text{StrV}) \quad | x(v^*) \quad (\text{VariantV})
 \end{array}$$

The big-step operational (natural) semantics of BATTERY is defined as follows:

$$\begin{array}{c}
 \boxed{\sigma \vdash e \Rightarrow v} \\
 \\
 \text{EUnit} \frac{}{\sigma \vdash () \Rightarrow ()} \quad \text{ENum} \frac{}{\sigma \vdash n \Rightarrow n} \quad \text{EBool} \frac{}{\sigma \vdash b \Rightarrow b} \quad \text{EStr} \frac{}{\sigma \vdash s \Rightarrow s} \\
 \\
 \text{EId} \frac{x \in \text{Domain}(\sigma)}{\sigma \vdash x \Rightarrow \sigma(x)} \quad \text{EAdd} \frac{\sigma \vdash e_1 \Rightarrow n_1 \quad \sigma \vdash e_2 \Rightarrow n_2}{\sigma \vdash e_1 + e_2 \Rightarrow n_1 + n_2} \quad \text{EMul} \frac{\sigma \vdash e_1 \Rightarrow n_1 \quad \sigma \vdash e_2 \Rightarrow n_2}{\sigma \vdash e_1 * e_2 \Rightarrow n_1 * n_2} \\
 \\
 \text{EDiv} \frac{\sigma \vdash e_1 \Rightarrow n_1 \quad \sigma \vdash e_2 \Rightarrow n_2 \quad n_2 \neq 0}{\sigma \vdash e_1 / e_2 \Rightarrow n_1 / n_2} \quad \text{EMod} \frac{\sigma \vdash e_1 \Rightarrow n_1 \quad \sigma \vdash e_2 \Rightarrow n_2 \quad n_2 \neq 0}{\sigma \vdash e_1 \% e_2 \Rightarrow n_1 \% n_2} \\
 \\
 \text{EConcat} \frac{\sigma \vdash e_1 \Rightarrow s_1 \quad \sigma \vdash e_2 \Rightarrow s_2 \quad s_2 \neq 0}{\sigma \vdash e_1 ++ e_2 \Rightarrow s_1 ++ s_2}
 \end{array}$$

$$\begin{array}{c}
\text{EEq} \frac{\sigma \vdash e_1 \Rightarrow v_1 \quad \sigma \vdash e_2 \Rightarrow v_2}{\sigma \vdash e_1 == e_2 \Rightarrow \text{eq}(v_1, v_2)} \quad \text{ELt} \frac{\sigma \vdash e_1 \Rightarrow n_1 \quad \sigma \vdash e_2 \Rightarrow n_2}{\sigma \vdash e_1 < e_2 \Rightarrow n_1 < n_2} \\
\\
\text{ESeq} \frac{\sigma \vdash e_1 \Rightarrow v_1 \quad \sigma \vdash e_2 \Rightarrow v_2}{\sigma \vdash e_1 ; e_2 \Rightarrow v_2} \\
\\
\text{EIf}_T \frac{\sigma \vdash e_0 \Rightarrow \text{true} \quad \sigma \vdash e_1 \Rightarrow v_1}{\sigma \vdash \text{if } (e_0) e_1 \text{ else } e_2 \Rightarrow v_1} \quad \text{EIf}_F \frac{\sigma \vdash e_0 \Rightarrow \text{false} \quad \sigma \vdash e_2 \Rightarrow v_2}{\sigma \vdash \text{if } (e_0) e_1 \text{ else } e_2 \Rightarrow v_2} \\
\\
\text{Eval} \frac{\sigma \vdash e_1 \Rightarrow v_1 \quad \sigma[x \mapsto v_1] \vdash e_2 \Rightarrow v_2}{\sigma \vdash \text{val } x = e_1; e_2 \Rightarrow v_2} \quad \text{Eval}_\tau \frac{\sigma \vdash e_1 \Rightarrow v_1 \quad \sigma[x \mapsto v_1] \vdash e_2 \Rightarrow v_2}{\sigma \vdash \text{val } x : \tau_0 = e_1; e_2 \Rightarrow v_2} \\
\\
\text{EFun} \frac{}{\sigma \vdash \lambda(x_1 : \tau_1, \dots, x_n : \tau_n).e \Rightarrow \langle \lambda(x_1, \dots, x_n).e, \sigma \rangle} \\
\\
\text{EApp}_\lambda \frac{\sigma \vdash e_0 \Rightarrow \langle \lambda(x_1, \dots, x_n).e, \sigma' \rangle \quad \sigma \vdash e_1 \Rightarrow v_1 \quad \dots \quad \sigma \vdash e_n \Rightarrow v_n \quad \sigma'[x_1 \mapsto v_1, \dots, x_n \mapsto v_n] \vdash e \Rightarrow v}{\sigma \vdash e_0[\tau_1, \dots, \tau_m](e_1, \dots, e_n) \Rightarrow v} \\
\\
\text{EApp}_{\langle - \rangle} \frac{\sigma \vdash e_0 \Rightarrow \langle x \rangle \quad \sigma \vdash e_1 \Rightarrow v_1 \quad \dots \quad \sigma \vdash e_n \Rightarrow v_n}{\sigma \vdash e_0[\tau_1, \dots, \tau_m](e_1, \dots, e_n) \Rightarrow x(v_1, \dots, v_n)} \\
\\
\text{ERecDefs} \frac{(\sigma, \sigma_n) \vdash d_1 \vdash \sigma_1 \quad \dots \quad (\sigma_{n-1}, \sigma_n) \vdash d_n \vdash \sigma_n \quad \sigma_n \vdash e \Rightarrow v}{\sigma \vdash d_1; \dots; d_n; e \Rightarrow v} \\
\\
\text{EMatch} \frac{1 \leq i \leq n \quad \sigma \vdash e \Rightarrow x_i(v_1, \dots, v_{m_i}) \quad \sigma[x_{i,1} \mapsto v_1, \dots, x_{i,m_i} \mapsto v_{m_i}] \vdash e_i \Rightarrow v}{\sigma \vdash e \text{ match } \{ \text{case } x_1(x_{1,1}, \dots, x_{1,m_1}) \Rightarrow e_1; \dots; \text{case } x_n(x_{n,1}, \dots, x_{n,m_n}) \Rightarrow e_n \} \Rightarrow v}
\end{array}$$

with the following environment update rules for recursive definitions:

$$\boxed{(\sigma, \sigma) \vdash d \vdash \sigma}$$

$$\text{LazyVal} \frac{}{(\sigma, \sigma') \vdash \text{lazy } x : \tau = e \vdash \sigma[x \mapsto \langle \langle e, \sigma' \rangle \rangle]}$$

$$\text{RecFun} \frac{}{(\sigma, \sigma') \vdash \text{def } x[\alpha_1, \dots, \alpha_m](x_1 : \tau_1, \dots, x_n : \tau_n) : \tau = e \vdash \sigma[x \mapsto \langle \lambda x_1, \dots, x_n. e, \sigma' \rangle]}$$

$$\text{TypeDef} \frac{}{(\sigma, \sigma') \vdash \text{enum } t[\dots] \{ \text{case } x_1(\dots); \dots \text{case } x_n(\dots) \} \vdash \sigma[x_1 \mapsto \langle x_1 \rangle, \dots, x_n \mapsto \langle x_n \rangle]}$$

and the following auxiliary function:

$$\boxed{\text{eq} : \mathbb{V} \times \mathbb{V} \rightarrow \mathbb{B}}$$

$$\begin{aligned}
\text{eq}((), ()) &= \text{true} & \text{eq}(n, n') &= (n = n') & \text{eq}(b, b') &= (b = b') & \text{eq}(s, s') &= (s = s') \\
\text{eq}(x(v_1, \dots, v_n), x'(v'_1, \dots, v'_n)) &= (x = x') \wedge \text{eq}(v_1, v'_1) \wedge \dots \wedge \text{eq}(v_n, v'_n) \\
\text{eq}(_, _) &= \text{false}(\text{otherwise})
\end{aligned}$$