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 Data **T**ype-Supported **T**yped **E**xpressions with **R**ecursion and Pol**Y**morphism, and it supports the following features:

- **unit value** (`()`)
- **number (integer) 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 (`%`)
- **comparison operators**: equality (`==` and `!=`) and relational (`<`, `>`, `<=`, and `>=`)
- **logical operators**: conjunction (`&&`), disjunction (`||`), and negation (`!`)
- **sequences** (`;`)
- **conditionals** (`if-else`)
- **immutable variable definitions** (`val`)
- **first-class functions** (`=>`)
- **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.

```
<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>
```

Duplicate field names are not allowed in record expressions and record types. For types, the arrow (=>) 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	match	8	

3 ABSTRACT SYNTAX

The abstract syntax of BATTERY is defined as follows:

$$
\begin{array}{llll}
\text{Expressions} \quad \mathbb{E} \ni e ::= () & (\texttt{EUnit}) & \mid e == e & (\texttt{EEq}) \\
\mid n & (\texttt{ENum}) & \mid e < e & (\texttt{ELt}) \\
\mid b & (\texttt{EBool}) & \mid e \ ; \ e & (\texttt{ESeq}) \\
\mid s & (\texttt{EStr}) & \mid \texttt{if}\ (e)\ e\ \texttt{else}\ e & (\texttt{EIf}) \\
\mid x & (\texttt{EId}) & \mid \texttt{val}\ x\ [:\tau]^{?} = e;\ e & (\texttt{EVal}) \\
\mid e + e & (\texttt{EAdd}) & \mid \lambda([x:\tau]^{*}).e & (\texttt{EFun}) \\
\mid e * e & (\texttt{EMul}) & \mid e[\tau^{*}](e^{*}) & (\texttt{EApp}) \\
\mid e\ /\ e & (\texttt{EDiv}) & \mid d^{+}\ e & (\texttt{ERecDefs}) \\
\mid e\ \%\ e & (\texttt{EMod}) & \mid e\ \texttt{match}\ \{\ [\texttt{case}\ x(x^{*})\ \texttt{=>}\ e]^{+}\ \} & (\texttt{EMatch}) \\
\mid e\ \texttt{++}\ e & (\texttt{EConcat}) & \mid \texttt{exit}[\tau](e) & (\texttt{EExit})
\end{array}
$$

$$
\begin{array}{ll}
\text{Recursive Definitions} \quad \mathbb{D} \ni d ::= \texttt{lazy}\ x:\tau = e & (\texttt{LazyVal}) \\
\mid \texttt{def}\ x[\alpha^{*}]([x:\tau]^{*}):\tau = e & (\texttt{RecFun}) \\
\mid \texttt{enum}\ t[\alpha^{*}]\ \{\ [\texttt{case}\ x([x:\tau]^{*})]^{+}\ \} & (\texttt{TypeDef})
\end{array}
$$

$$
\begin{array}{llllll}
\text{Types} \quad \mathbb{T} \ni \tau ::= \texttt{unit} & (\texttt{UnitT}) & \text{Identifiers} & x & \in \mathbb{X} & (\texttt{String}) \\
\mid \texttt{num} & (\texttt{NumT}) & \text{Numbers} & n & \in \mathbb{Z} & (\texttt{BigInt}) \\
\mid \texttt{bool} & (\texttt{BoolT}) & \text{Booleans} & b & \in \mathbb{B} & (\texttt{Boolean}) \\
\mid \texttt{str} & (\texttt{StrT}) & \text{Strings} & s & \in \mathbb{S} & (\texttt{String}) \\
\mid t[\tau^{*}] & (\texttt{IdT}) & \text{Type Names} & t & \in \mathbb{X}_t & (\texttt{String}) \\
\mid \alpha & (\texttt{IdT}) & \text{Type Variables} & \alpha & \in \mathbb{X}_\alpha & (\texttt{String}) \\
\mid [\alpha^{*}](\tau^{*}) \to \tau & (\texttt{ArrowT}) & & & &
\end{array}
$$

For type names $t[\tau^*]$ and arrow types $[\alpha^*](\tau^*) \to \tau$, we omit the square brackets (t and $(\tau^*) \to \tau$) 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:

$$
\begin{array}{ll}
\mathcal{D}[\![-\ e]\!] = \mathcal{D}[\![e]\!] * (\texttt{-1}) & \mathcal{D}[\![e_1\ \texttt{!=}\ e_2]\!] = \mathcal{D}[\![\texttt{!}\ (e_1 == e_2)]\!] \\
\mathcal{D}[\![\texttt{!}\ e]\!] = \texttt{if}\ (\mathcal{D}[\![e]\!])\ \texttt{false else true} & \mathcal{D}[\![e_1 \texttt{<=} e_2]\!] = \mathcal{D}[\![(e_1 < e_2)\ \texttt{||}\ (e_1 == e_2)]\!] \\
\mathcal{D}[\![e_1 - e_2]\!] = \mathcal{D}[\![e_1]\!] + \mathcal{D}[\![-\ e_2]\!] & \mathcal{D}[\![e_1 > e_2]\!] = \mathcal{D}[\![\texttt{!}\ (e_1 \texttt{<=} e_2)]\!] \\
\mathcal{D}[\![e_1\ \texttt{\&\&}\ e_2]\!] = \texttt{if}\ (\mathcal{D}[\![e_1]\!])\ \mathcal{D}[\![e_2]\!]\ \texttt{else false} & \mathcal{D}[\![e_1 \texttt{>=} e_2]\!] = \mathcal{D}[\![\texttt{!}\ (e_1 < e_2)]\!] \\
\mathcal{D}[\![e_1\ \texttt{||}\ e_2]\!] = \texttt{if}\ (\mathcal{D}[\![e_1]\!])\ \texttt{true else}\ \mathcal{D}[\![e_2]\!] &
\end{array}
$$

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:

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

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:

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

$$\tau\text{–}\texttt{EUnit}\ \frac{}{\Gamma \vdash () : \texttt{unit}} \qquad \tau\text{–}\texttt{ENum}\ \frac{}{\Gamma \vdash n : \texttt{num}}$$

$$\tau\text{–}\texttt{EBool}\ \frac{}{\Gamma \vdash b : \texttt{bool}} \qquad \tau\text{–}\texttt{EStr}\ \frac{}{\Gamma \vdash s : \texttt{str}} \qquad \tau\text{–}\texttt{EId}\ \frac{x \in \text{Domain}(\Gamma)}{\Gamma \vdash x : \Gamma(x)}$$

$$\tau\text{–}\texttt{EAdd}\ \frac{\Gamma \vdash e_1 : \texttt{num} \qquad \Gamma \vdash e_2 : \texttt{num}}{\Gamma \vdash e_1 + e_2 : \texttt{num}} \qquad \tau\text{–}\texttt{EMul}\ \frac{\Gamma \vdash e_1 : \texttt{num} \qquad \Gamma \vdash e_2 : \texttt{num}}{\Gamma \vdash e_1 * e_2 : \texttt{num}}$$

$$\tau\text{–}\texttt{EDiv}\ \frac{\Gamma \vdash e_1 : \texttt{num} \qquad \Gamma \vdash e_2 : \texttt{num}}{\Gamma \vdash e_1 \ / \ e_2 : \texttt{num}} \qquad \tau\text{–}\texttt{EMod}\ \frac{\Gamma \vdash e_1 : \texttt{num} \qquad \Gamma \vdash e_2 : \texttt{num}}{\Gamma \vdash e_1 \ \% \ e_2 : \texttt{num}}$$

$$\tau\text{–}\texttt{EConcat}\ \frac{\Gamma \vdash e_1 : \texttt{str} \qquad \Gamma \vdash e_2 : \texttt{str}}{\Gamma \vdash e_1 \texttt{ ++ } e_2 : \texttt{str}}$$

$$\tau\text{–}\texttt{EEq}\ \frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma \vdash e_2 : \tau_2 \qquad \tau_1 \equiv \tau_2}{\Gamma \vdash e_1 == e_2 : \texttt{bool}} \qquad \tau\text{–}\texttt{ELt}\ \frac{\Gamma \vdash e_1 : \texttt{num} \qquad \Gamma \vdash e_2 : \texttt{num}}{\Gamma \vdash e_1 < e_2 : \texttt{bool}}$$

$$\tau\text{–}\texttt{ESeq}\ \frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1 \ ; \ e_2 : \tau_2} \qquad \tau\text{–}\texttt{EIf}\ \frac{\Gamma \vdash e_0 : \texttt{bool} \qquad \Gamma \vdash e_1 : \tau_1 \qquad \Gamma \vdash e_2 : \tau_2 \qquad \tau_1 \equiv \tau_2}{\Gamma \vdash \texttt{if}\ (e_0)\ e_1\ \texttt{else}\ e_2 : \tau_1}$$

$$\tau\text{–}\texttt{EVal}\ \frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma[x : \tau_1] \vdash e_2 : \tau_2}{\Gamma \vdash \texttt{val}\ x = e_1;\ e_2 : \tau_2} \qquad \tau\text{–}\texttt{EVal}_\tau\ \frac{\Gamma \vdash e_1 : \tau_1 \qquad \tau_0 \equiv \tau_1 \qquad \Gamma[x : \tau_0] \vdash e_2 : \tau_2}{\Gamma \vdash \texttt{val}\ x : \tau_0 = e_1;\ e_2 : \tau_2}$$

$$\tau\text{–}\texttt{EFun}\ \frac{\Gamma \vdash \tau_1 \quad \ldots \quad \Gamma \vdash \tau_n \qquad \Gamma[x_1 : \tau_1, \ldots, x_n : \tau_n] \vdash e : \tau}{\Gamma \vdash \lambda(x_1 : \tau_1, \ldots, x_n : \tau_n).e : (\tau_1, \ldots, \tau_n) \to \tau}$$

$$\tau\text{–}\texttt{EApp}\ \frac{\begin{array}{c} \Gamma \vdash \tau_1 \quad \ldots \quad \Gamma \vdash \tau_m \\ \Gamma \vdash e_0 : [\alpha_1, \ldots, \alpha_m](\tau'_1, \ldots, \tau'_n) \to \tau' \qquad \Gamma \vdash e_1 : \tau''_1 \quad \ldots \quad \Gamma \vdash e_n : \tau''_n \\ \tau''_1 \equiv \tau'_1[\alpha_1 \leftarrow \tau_1, \ldots, \alpha_m \leftarrow \tau_m] \quad \ldots \quad \tau''_n \equiv \tau'_n[\alpha_1 \leftarrow \tau_1, \ldots, \alpha_m \leftarrow \tau_m] \end{array}}{\Gamma \vdash e_0[\tau_1, \ldots, \tau_m](e_1, \ldots, e_n) : \tau'[\alpha_1 \leftarrow \tau_1, \ldots, \alpha_m \leftarrow \tau_m]}$$

$$\tau\text{–}\mathtt{ERecDefs}\ \frac{\Gamma \vdash d_1 \vdash \Gamma_1 \quad \ldots \quad \Gamma_{n-1} \vdash d_n \vdash \Gamma_n \qquad \Gamma_n \vDash d_1 \quad \ldots \quad \Gamma_n \vDash d_n \qquad \Gamma_n \vdash e : \tau \qquad \Gamma \vdash \tau}{\Gamma \vdash d_1; \ldots; d_n; e : \tau}$$

$$\tau\text{–}\mathtt{EMatch}\ \frac{\begin{array}{c}\Gamma \vdash e : t[\tau_1, \ldots, \tau_m] \qquad \Gamma(t) = [\alpha_1, \ldots, \alpha_m]\{x_1(\tau_{1,1}, \ldots, \tau_{1,m_1}) + \ldots + x_n(\tau_{n,1}, \ldots, \tau_{n,m_n})\} \\ \forall 1 \leq i \leq n.\ \Gamma_i = \Gamma[x_{i,1} : \tau_{i,1}[\alpha_1 \leftarrow \tau_1, \ldots, \alpha_m \leftarrow \tau_m], \ldots, x_{i,m_i} : \tau_{i,m_i}[\alpha_1 \leftarrow \tau_1, \ldots, \alpha_m \leftarrow \tau_m]] \\ \Gamma_1 \vdash e_1 : \tau_1 \qquad \ldots \qquad \Gamma_n \vdash e_n : \tau_n \qquad \tau_1 \equiv \ldots \equiv \tau_n\end{array}}{\Gamma \vdash e\ \mathtt{match}\ \{\ \mathtt{case}\ x_1(x_{1,1}, \ldots, x_{1,m_1})\ \mathtt{=>}\ e_1;\ \ldots;\ \mathtt{case}\ x_n(x_{n,1}, \ldots, x_{n,m_n})\ \mathtt{=>}\ e_n\ \} : \tau_1}$$

$$\tau\text{–}\mathtt{EExit}\ \frac{\Gamma \vdash \tau \qquad \Gamma \vdash e : \mathtt{str}}{\Gamma \vdash \mathtt{exit}[\tau](e) : \tau}$$

the following type environment update rules for recursive definitions:

$$\boxed{\Gamma \vdash d \vdash \Gamma}$$

$$\mathtt{LazyVal}\ \frac{}{\Gamma \vdash \mathtt{lazy}\ x : \tau = e \vdash \Gamma[x : \tau]}$$

$$\mathtt{RecFun}\ \frac{}{\Gamma \vdash \mathtt{def}\ x[\alpha_1, \ldots, \alpha_m](x_1 : \tau_1, \ldots, x_n : \tau_n) : \tau = e \vdash \Gamma[x : [\alpha_1, \ldots, \alpha_m](\tau_1, \ldots, \tau_n) \to \tau]}$$

$$\mathtt{TypeDef}\ \frac{\begin{array}{c}t \notin \mathrm{Domain}(\Gamma) \qquad \Gamma_0 = \Gamma[t = [\alpha_1, \ldots, \alpha_m]\{x_1(\tau_{1,1}, \ldots, \tau_{1,m_1}) + \ldots + x_n(\tau_{n,1}, \ldots, \tau_{n,m_n})\}] \\ \Gamma_1 = \Gamma_0[x_1 : [\alpha_1, \ldots, \alpha_m](\tau_{1,1}, \ldots, \tau_{1,m_1}) \to t[\alpha_1, \ldots, \alpha_m]] \\ \ldots \qquad \Gamma_n = \Gamma_{n-1}[x_n : [\alpha_1, \ldots, \alpha_m](\tau_{n,1}, \ldots, \tau_{n,m_n}) \to t[\alpha_1, \ldots, \alpha_m]]\end{array}}{\Gamma \vdash \mathtt{enum}\ t[\alpha_1, \ldots, \alpha_m] \left\{ \begin{array}{l}\mathtt{case}\ x_1(x_{1,1} : \tau_{1,1}, \ldots, x_{1,m_1} : \tau_{1,m_1}); \\ \ldots \\ \mathtt{case}\ x_n(x_{n,1} : \tau_{n,1}, \ldots, x_{n,m_n} : \tau_{n,m_n})\end{array} \right\} \vdash \Gamma_n}$$

the following typing rules for recursive definitions:

$$\boxed{\Gamma \vDash d}$$

$$\mathtt{LazyVal}\ \frac{\Gamma \vdash \tau_0 \qquad \Gamma \vdash e_1 : \tau_1 \qquad \tau_0 \equiv \tau_1}{\Gamma \vDash \mathtt{lazy}\ x : \tau_0 = e_1}$$

$$\mathtt{RecFun}\ \frac{\begin{array}{c}\alpha_1 \notin \mathrm{Domain}(\Gamma) \quad \ldots \quad \alpha_m \notin \mathrm{Domain}(\Gamma) \qquad \Gamma' = \Gamma[\alpha_1, \ldots, \alpha_m] \\ \Gamma' \vdash \tau_1 \quad \ldots \quad \Gamma' \vdash \tau_n \qquad \Gamma' \vdash \tau \qquad \Gamma'[x_1 : \tau_1, \ldots, x_n : \tau_n] \vdash e : \tau' \qquad \tau \equiv \tau'\end{array}}{\Gamma \vDash \mathtt{def}\ x[\alpha_1, \ldots, \alpha_m](x_1 : \tau_1, \ldots, x_n : \tau_n) : \tau = e}$$

$$\mathtt{TypeDef}\ \frac{\begin{array}{c}\alpha_1 \notin \mathrm{Domain}(\Gamma) \quad \ldots \quad \alpha_m \notin \mathrm{Domain}(\Gamma) \\ \Gamma' = \Gamma[\alpha_1, \ldots, \alpha_m] \qquad \Gamma' \vdash \tau_{1,1} \quad \ldots \quad \Gamma' \vdash \tau_{n,m_n}\end{array}}{\Gamma \vDash \mathtt{enum}\ t[\alpha_1, \ldots, \alpha_m] \left\{ \begin{array}{l}\mathtt{case}\ x_1(x_{1,1} : \tau_{1,1}, \ldots, x_{1,m_1} : \tau_{1,m_1}); \\ \ldots \\ \mathtt{case}\ x_n(x_{n,1} : \tau_{n,1}, \ldots, x_{n,m_n} : \tau_{n,m_n})\end{array} \right\}}$$

the following rules for well-formedness of types:

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

$$\frac{}{\Gamma \vdash \texttt{unit}} \qquad \frac{}{\Gamma \vdash \texttt{num}} \qquad \frac{}{\Gamma \vdash \texttt{bool}} \qquad \frac{}{\Gamma \vdash \texttt{str}}$$

$$\frac{\Gamma(t) = [\alpha_1, \ldots, \alpha_m]\{x_1(\tau_{1,1}, \ldots, \tau_{1,m_1}) + \ldots + x_n(\tau_{n,1}, \ldots, \tau_{n,m_n})\} \qquad \Gamma \vdash \tau_1 \quad \ldots \quad \Gamma \vdash \tau_m}{\Gamma \vdash t[\tau_1, \ldots, \tau_m]}$$

$$\frac{\alpha \in \text{Domain}(\Gamma)}{\Gamma \vdash \alpha} \qquad \frac{\Gamma' = \Gamma[\alpha_1, \ldots, \alpha_m] \qquad \Gamma' \vdash \tau_1 \quad \ldots \quad \Gamma' \vdash \tau_n \qquad \Gamma' \vdash \tau}{\Gamma \vdash [\alpha_1, \ldots, \alpha_m](\tau_1, \ldots, \tau_n) \to \tau}$$

and the following rules for type equivalence:

$$\boxed{\tau \equiv \tau}$$

$$\frac{}{\texttt{unit} \equiv \texttt{unit}} \qquad \frac{}{\texttt{num} \equiv \texttt{num}} \qquad \frac{}{\texttt{bool} \equiv \texttt{bool}} \qquad \frac{}{\texttt{str} \equiv \texttt{str}}$$

$$\frac{\tau_1 \equiv \tau'_1 \qquad \ldots \qquad \tau_m \equiv \tau'_m}{t[\tau_1, \ldots, \tau_m] \equiv t[\tau'_1, \ldots, \tau'_m]} \qquad \frac{}{\alpha \equiv \alpha}$$

$$\frac{\begin{array}{c}\tau_1 \equiv \tau'_1[\alpha'_1 \leftarrow \alpha_1, \ldots, \alpha'_m \leftarrow \alpha_m] \\ \ldots \qquad \tau_n \equiv \tau'_n[\alpha'_1 \leftarrow \alpha_1, \ldots, \alpha'_m \leftarrow \alpha_m] \qquad \tau \equiv \tau'[\alpha'_1 \leftarrow \alpha_1, \ldots, \alpha'_m \leftarrow \alpha_m]\end{array}}{[\alpha_1, \ldots, \alpha_m](\tau_1, \ldots, \tau_n \to \tau) \equiv [\alpha'_1, \ldots, \alpha'_m](\tau'_1, \ldots, \tau'_n \to \tau')}$$

5 SEMANTICS

We use the following notations in the semantics:

$$\text{Environments} \qquad \sigma \ \in \mathbb{X} \xrightarrow{\text{fin}} \mathbb{V} \quad (\texttt{Env})$$

$$\begin{array}{llll} \text{Values} \quad \mathbb{V} \ni v ::= () & (\texttt{UnitV}) & \mid \langle \lambda x.(e, \ldots, e), \sigma \rangle & (\texttt{CloV}) \\ \mid n & (\texttt{NumV}) & \mid \langle\!\langle e, \sigma \rangle\!\rangle & (\texttt{ExprV}) \\ \mid b & (\texttt{BoolV}) & \mid \langle x \rangle & (\texttt{ConstrV}) \\ \mid s & (\texttt{StrV}) & \mid x(v^*) & (\texttt{VariantV}) \end{array}$$

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

$$\boxed{\sigma \vdash e \Rightarrow v}$$

$$\texttt{EUnit}\ \frac{}{\sigma \vdash () \Rightarrow ()} \qquad \texttt{ENum}\ \frac{}{\sigma \vdash n \Rightarrow n} \qquad \texttt{EBool}\ \frac{}{\sigma \vdash b \Rightarrow b}$$

$$\texttt{EId}\ \frac{x \in \mathrm{Domain}(\sigma) \qquad \sigma(x) \neq \langle\!\langle e, \sigma \rangle\!\rangle}{\sigma \vdash x \Rightarrow \sigma(x)} \qquad \texttt{EId}_{\texttt{lazy}}\ \frac{x \in \mathrm{Domain}(\sigma) \qquad \sigma(x) = \langle\!\langle e, \sigma' \rangle\!\rangle \qquad \sigma' \vdash e \Rightarrow v}{\sigma \vdash x \Rightarrow v}$$

$$\texttt{EStr}\ \frac{}{\sigma \vdash s \Rightarrow s} \qquad \texttt{EAdd}\ \frac{\sigma \vdash e_1 \Rightarrow n_1 \qquad \sigma \vdash e_2 \Rightarrow n_2}{\sigma \vdash e_1 \texttt{ + } e_2 \Rightarrow n_1 + n_2} \qquad \texttt{EMul}\ \frac{\sigma \vdash e_1 \Rightarrow n_1 \qquad \sigma \vdash e_2 \Rightarrow n_2}{\sigma \vdash e_1 \texttt{ * } e_2 \Rightarrow n_1 * n_2}$$

$$\texttt{EDiv}\ \frac{\sigma \vdash e_1 \Rightarrow n_1 \qquad \sigma \vdash e_2 \Rightarrow n_2 \qquad n_2 \neq 0}{\sigma \vdash e_1 \texttt{ / } e_2 \Rightarrow n_1 \texttt{ / } n_2} \qquad \texttt{EMod}\ \frac{\sigma \vdash e_1 \Rightarrow n_1 \qquad \sigma \vdash e_2 \Rightarrow n_2 \qquad n_2 \neq 0}{\sigma \vdash e_1 \texttt{ \% } e_2 \Rightarrow n_1 \texttt{ \% } n_2}$$

$$\texttt{EEq}\ \frac{\sigma \vdash e_1 \Rightarrow v_1 \qquad \sigma \vdash e_2 \Rightarrow v_2}{\sigma \vdash e_1 \texttt{ == } e_2 \Rightarrow \texttt{eq}(v_1, v_2)} \qquad \texttt{ELt}\ \frac{\sigma \vdash e_1 \Rightarrow n_1 \qquad \sigma \vdash e_2 \Rightarrow n_2}{\sigma \vdash e_1 \texttt{ < } e_2 \Rightarrow n_1 < n_2}$$

$$\texttt{EConcat}\ \frac{\sigma \vdash e_1 \Rightarrow s_1 \qquad \sigma \vdash e_2 \Rightarrow s_2}{\sigma \vdash e_1 \texttt{ ++ } e_2 \Rightarrow s_1 \texttt{ ++ } s_2} \qquad \texttt{ESeq}\ \frac{\sigma \vdash e_1 \Rightarrow v_1 \qquad \sigma \vdash e_2 \Rightarrow v_2}{\sigma \vdash e_1 \texttt{ ; } e_2 \Rightarrow v_2}$$

$$\texttt{EIf}_T\ \frac{\sigma \vdash e_0 \Rightarrow \texttt{true} \qquad \sigma \vdash e_1 \Rightarrow v_1}{\sigma \vdash \texttt{if } (e_0)\ e_1 \texttt{ else } e_2 \Rightarrow v_1} \qquad \texttt{EIf}_F\ \frac{\sigma \vdash e_0 \Rightarrow \texttt{false} \qquad \sigma \vdash e_2 \Rightarrow v_2}{\sigma \vdash \texttt{if } (e_0)\ e_1 \texttt{ else } e_2 \Rightarrow v_2}$$

$$\texttt{EVal}\ \frac{\sigma \vdash e_1 \Rightarrow v_1 \qquad \sigma[x \mapsto v_1] \vdash e_2 \Rightarrow v_2}{\sigma \vdash \texttt{val } x \texttt{ = } e_1\texttt{; } e_2 \Rightarrow v_2} \qquad \texttt{EVal}_\tau\ \frac{\sigma \vdash e_1 \Rightarrow v_1 \qquad \sigma[x \mapsto v_1] \vdash e_2 \Rightarrow v_2}{\sigma \vdash \texttt{val } x \texttt{:} \tau_0 \texttt{ = } e_1\texttt{; } e_2 \Rightarrow v_2}$$

$$\texttt{EFun}\ \frac{}{\sigma \vdash \lambda(x_1 \texttt{:} \tau_1, \ldots, x_n \texttt{:} \tau_n).e \Rightarrow \langle \lambda(x_1, \ldots, x_n).e, \sigma \rangle}$$

$$\texttt{EApp}_\lambda\ \frac{\begin{array}{c}\sigma \vdash e_0 \Rightarrow \langle \lambda(x_1, \ldots, x_n).e, \sigma' \rangle \\ \sigma \vdash e_1 \Rightarrow v_1 \quad \ldots \quad \sigma \vdash e_n \Rightarrow v_n \qquad \sigma'[x_1 \mapsto v_1, \ldots, x_n \mapsto v_n] \vdash e \Rightarrow v\end{array}}{\sigma \vdash e_0[\tau_1, \ldots, \tau_m](e_1, \ldots, e_n) \Rightarrow v}$$

$$\texttt{EApp}_{\langle - \rangle}\ \frac{\sigma \vdash e_0 \Rightarrow \langle x \rangle \qquad \sigma \vdash e_1 \Rightarrow v_1 \qquad \ldots \qquad \sigma \vdash e_n \Rightarrow v_n}{\sigma \vdash e_0[\tau_1, \ldots, \tau_m](e_1, \ldots, e_n) \Rightarrow x(v_1, \ldots, v_n)}$$

$$\texttt{ERecDefs}\ \frac{(\sigma, \sigma_n) \vdash d_1 \vdash \sigma_1 \qquad \ldots \qquad (\sigma_{n-1}, \sigma_n) \vdash d_n \vdash \sigma_n \qquad \sigma_n \vdash e \Rightarrow v}{\sigma \vdash d_1; \ldots; d_n; e \Rightarrow v}$$

$$\texttt{EMatch}\ \frac{1 \leq i \leq n \qquad \sigma \vdash e \Rightarrow x_i(v_1, \ldots, v_{m_i}) \qquad \sigma[x_{i,1} \mapsto v_1, \ldots, x_{i,m_i} \mapsto v_{m_i}] \vdash e_i \Rightarrow v}{\sigma \vdash e \texttt{ match } \{ \texttt{ case } x_1(x_{1,1}, \ldots, x_{1,m_1}) \texttt{ => } e_1\texttt{; } \ldots\texttt{; } \texttt{case } x_n(x_{n,1}, \ldots, x_{n,m_n}) \texttt{ => } e_n \texttt{ } \} \Rightarrow v}$$

with the following environment update rules for recursive definitions:

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

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

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

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

the following auxiliary function:

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

$$\texttt{eq}((), ()) = \texttt{true} \qquad \texttt{eq}(n, n') = (n = n') \qquad \texttt{eq}(b, b') = (b = b') \qquad \texttt{eq}(s, s') = (s = s')$$
$$\texttt{eq}(x(v_1, \ldots, v_n), x'(v'_1, \ldots, v'_n)) = (x = x') \wedge \texttt{eq}(v_1, v'_1) \wedge \ldots \wedge \texttt{eq}(v_n, v'_n)$$
$$\texttt{eq}(\_, \_) = \texttt{false}(\text{otherwise})$$