

MiniScala – A Small Subset of Scala Language

1 INTRODUCTION

MiniScala is a toy language for the COSE212 course at Korea University. MiniScala is inspired by the Scala programming language.¹ The name MiniScala indicates that it is a minimal subset of the Scala language, 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 MiniScala. 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 MiniScala.

2 CONCRETE SYNTAX

The concrete syntax of MiniScala 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>* "\"
```

¹<https://www.scala-lang.org/>

```

<keyword> ::= "Any" | "Boolean" | "Nothing" | "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" "(" <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> ")"
           | "Any" | "Nothing" | "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 (\Rightarrow) operator is right-associative. For expressions, the precedence and associativity of operators are defined as follows:

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

3 ABSTRACT SYNTAX

The abstract syntax of MiniScala 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]^? = 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}(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	$\mathbb{T} \ni \tau ::= \top$	(AnyT)	$ \text{num}$	(NumT)	$ t[\tau^*]$	(IdT)
	$ \perp$	(NothingT)	$ \text{bool}$	(BoolT)	$ \alpha$	(IdT)
	$ \text{unit}$	(UnitT)	$ \text{str}$	(StrT)	$ [\alpha^*](\tau^*) \rightarrow \tau$	(ArrowT)

Identifiers $x \in \mathbb{X}$ (String)

Numbers $n \in \mathbb{Z}$ (BigInt)

Booleans $b \in \mathbb{B}$ (Boolean)

Strings

Type Names

Type Variables

$s \in \mathbb{S}$ (String)

$t \in \mathbb{X}_t$ (String)

$\alpha \in \mathbb{X}_\alpha$ (String)

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:

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

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

4 TYPE SYSTEM

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

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

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:

$$\begin{array}{c}
 \boxed{\Gamma \vdash e : \tau} \\
 \\
 \begin{array}{c}
 \tau\text{-EUnit} \frac{}{\Gamma \vdash () : \text{unit}} \quad \tau\text{-EAdd} \frac{\Gamma \vdash e_1 : \tau_1 \quad \tau_1 <: \text{num} \quad \Gamma \vdash e_2 : \tau_2 \quad \tau_2 <: \text{num}}{\Gamma \vdash e_1 + e_2 : \text{num}} \\
 \\
 \tau\text{-EEnum} \frac{}{\Gamma \vdash n : \text{num}} \quad \tau\text{-EMul} \frac{\Gamma \vdash e_1 : \tau_1 \quad \tau_1 <: \text{num} \quad \Gamma \vdash e_2 : \tau_2 \quad \tau_2 <: \text{num}}{\Gamma \vdash e_1 * e_2 : \text{num}} \\
 \\
 \tau\text{-EBool} \frac{}{\Gamma \vdash b : \text{bool}} \quad \tau\text{-EDiv} \frac{\Gamma \vdash e_1 : \tau_1 \quad \tau_1 <: \text{num} \quad \Gamma \vdash e_2 : \tau_2 \quad \tau_2 <: \text{num}}{\Gamma \vdash e_1 / e_2 : \text{num}} \\
 \\
 \tau\text{-EStr} \frac{}{\Gamma \vdash s : \text{str}} \quad \tau\text{-EMod} \frac{\Gamma \vdash e_1 : \tau_1 \quad \tau_1 <: \text{num} \quad \Gamma \vdash e_2 : \tau_2 \quad \tau_2 <: \text{num}}{\Gamma \vdash e_1 \% e_2 : \text{num}} \\
 \\
 \tau\text{-EId} \frac{x \in \text{Domain}(\Gamma)}{\Gamma \vdash x : \Gamma(x)} \quad \tau\text{-EConcat} \frac{\Gamma \vdash e_1 : \tau_1 \quad \tau_1 <: \text{str} \quad \Gamma \vdash e_2 : \tau_2 \quad \tau_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}{\Gamma \vdash e_1 == e_2 : \text{bool}} \quad \tau\text{-ELt} \frac{\Gamma \vdash e_1 : \tau_1 \quad \tau_1 <: \text{num} \quad \Gamma \vdash e_2 : \tau_2 \quad \tau_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 : \tau_0 \quad \tau_0 <: \text{bool} \quad \Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \text{if } (e_0) e_1 \text{ else } e_2 : \tau_1 \sqcup \tau_2} \\
 \\
 \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_1 <: \tau_0 \quad \Gamma[x : \tau_0] \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 \dots \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 \dots \Gamma \vdash e_n : \tau''_n \quad \tau''_1 <: \tau'_1[\alpha_1 \leftarrow \tau_1, \dots, \alpha_m \leftarrow \tau_m] \dots \tau''_n <: \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'[\alpha_1 \leftarrow \tau_1, \dots, \alpha_m \leftarrow \tau_m]}
 \end{array}
 \end{array}$$

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

$$\tau\text{-EExit} \frac{\Gamma \vdash e : \tau}{\Gamma \vdash \text{exit}(e) : \perp}$$

the following type environment update rules for recursive definitions:

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

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

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

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

the following typing rules for recursive definitions:

$$\boxed{\Gamma \models d}$$

$$\text{LazyVal} \frac{\Gamma \vdash \tau_0 \quad \Gamma \vdash e_1 : \tau_1 \quad \tau_1 <: \tau_0}{\Gamma \models \text{lazy } x : \tau_0 = e_1}$$

$$\text{RecFun} \frac{\alpha_1 \notin \text{Domain}(\Gamma) \quad \dots \quad \alpha_m \notin \text{Domain}(\Gamma) \quad \Gamma' = \Gamma[\alpha_1, \dots, \alpha_m] \quad \Gamma' \vdash \tau_1 \quad \dots \quad \Gamma' \vdash \tau_n \quad \Gamma' \vdash \tau \quad \Gamma'[x_1 : \tau_1, \dots, x_n : \tau_n] \vdash e : \tau' \quad \tau' <: \tau}{\Gamma \vdash \text{def } x[\alpha_1, \dots, \alpha_m](x_1 : \tau_1, \dots, x_n : \tau_n) : \tau = e}$$

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

the following rules for well-formedness of types:

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

$$\frac{}{\Gamma \vdash \top} \quad \frac{}{\Gamma \vdash \perp} \quad \frac{}{\Gamma \vdash \text{unit}} \quad \frac{}{\Gamma \vdash \text{num}} \quad \frac{}{\Gamma \vdash \text{bool}} \quad \frac{}{\Gamma \vdash \text{str}}$$

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

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

the following rules for subtyping relation:

$$\boxed{\tau <: \tau}$$

$$\frac{}{\perp <: \tau} \quad \frac{}{\tau <: \top} \quad \frac{}{\tau <: \tau} \quad \frac{\tau_1 <: \tau'_1 \quad \dots \quad \tau_m <: \tau'_m}{t[\tau_1, \dots, \tau_m] <: t[\tau'_1, \dots, \tau'_m]}$$

$$\frac{\forall 1 \leq i \leq n. \tau'_i[\alpha'_1 \leftarrow \alpha_1, \dots, \alpha'_m \leftarrow \alpha_m] <: \tau_i \quad \tau <: \tau'[\alpha'_1 \leftarrow \alpha_1, \dots, \alpha'_m \leftarrow \alpha_m]}{[\alpha_1, \dots, \alpha_m](\tau_1, \dots, \tau_n) \rightarrow \tau <: [\alpha'_1, \dots, \alpha'_m](\tau'_1, \dots, \tau'_n) \rightarrow \tau'}$$

and the following rules for join (\sqcup) and meet (\sqcap) operations on types:

$$\boxed{\tau \sqcup \tau = \tau}$$

$$\frac{\tau <: \tau' \quad \tau' <: \tau}{\tau \sqcup \tau' = \tau'} \quad \frac{\tau' <: \tau}{\tau \sqcup \tau' = \tau} \quad \frac{\tau[\tau_1, \dots, \tau_n] \sqcup t[\tau'_1, \dots, \tau'_n] = t[\tau_1 \sqcup \tau'_1, \dots, \tau_n \sqcup \tau'_n]}{\tau = [\alpha_1, \dots, \alpha_m](\tau_1, \dots, \tau_n) \rightarrow \tau_0 \quad \tau' = [\alpha'_1, \dots, \alpha'_m](\tau'_1, \dots, \tau'_n) \rightarrow \tau'_0}$$

$$\frac{\forall 0 \leq i \leq n. \tau''_i = \tau'_i[\alpha'_1 \leftarrow \alpha_1, \dots, \alpha'_m \leftarrow \alpha_m]}{\tau \sqcup \tau' = [\alpha_1, \dots, \alpha_m](\tau_1 \sqcap \tau''_1, \dots, \tau_n \sqcap \tau''_n) \rightarrow \tau_0 \sqcup \tau''_0}$$

$$\boxed{\tau \sqcap \tau = \tau}$$

$$\frac{\tau <: \tau' \quad \tau' <: \tau}{\tau \sqcap \tau' = \tau} \quad \frac{\tau' <: \tau}{\tau \sqcap \tau' = \tau'} \quad \frac{\tau[\tau_1, \dots, \tau_n] \sqcap t[\tau'_1, \dots, \tau'_n] = t[\tau_1 \sqcap \tau'_1, \dots, \tau_n \sqcap \tau'_n]}{\tau = [\alpha_1, \dots, \alpha_m](\tau_1, \dots, \tau_n) \rightarrow \tau_0 \quad \tau' = [\alpha'_1, \dots, \alpha'_m](\tau'_1, \dots, \tau'_n) \rightarrow \tau'_0}$$

$$\frac{\forall 0 \leq i \leq n. \tau''_i = \tau'_i[\alpha'_1 \leftarrow \alpha_1, \dots, \alpha'_m \leftarrow \alpha_m]}{\tau \sqcap \tau' = [\alpha_1, \dots, \alpha_m](\tau_1 \sqcup \tau''_1, \dots, \tau_n \sqcup \tau''_n) \rightarrow \tau_0 \sqcap \tau''_0}$$

Note that the join (\sqcup) and meet (\sqcap) operations are always defined for any two types, and the result of the omitted cases is \top for join and \perp for meet operations.

5 SEMANTICS

We use the following notations in the semantics:

$$\text{Environments } \sigma \in \mathbb{X}^{\text{fin}} \rightarrow \mathbb{V} \quad (\text{Env})$$

Values	$\mathbb{V} \ni v ::= () \quad (\text{UnitV})$	$ \langle \lambda x.(e, \dots, e), \sigma \rangle \quad (\text{CloV})$
	$ n \quad (\text{NumV})$	$ \langle e, \sigma \rangle \quad (\text{ExprV})$
	$ b \quad (\text{BoolV})$	$ \langle x \rangle \quad (\text{ConstrV})$
	$ s \quad (\text{StrV})$	$ x(v^*) \quad (\text{VariantV})$

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

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

$$\begin{array}{c} \text{EUnit} \frac{}{\sigma \vdash () \Rightarrow ()} \quad \text{EAdd} \frac{x \in \text{Domain}(\sigma) \quad \sigma(x) \neq \langle e, \sigma' \rangle}{\sigma \vdash x \Rightarrow \sigma(x)} \quad \text{EMul} \frac{x \in \text{Domain}(\sigma) \quad \sigma(x) = \langle e, \sigma' \rangle \quad \sigma' \vdash e \Rightarrow v}{\sigma \vdash x \Rightarrow v} \\ \text{EId} \frac{}{\sigma \vdash s \Rightarrow s} \quad \text{ESeq} \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{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{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{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{EConcat} \frac{\sigma \vdash e_1 \Rightarrow s_1 \quad \sigma \vdash e_2 \Rightarrow s_2}{\sigma \vdash e_1 ++ e_2 \Rightarrow s_1 ++ s_2} \quad \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{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{EApp}_\lambda \frac{\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{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}_{\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)} \end{array}$$

$$\text{ERecDef} \frac{(\sigma, \sigma_n) \vdash d_1 \vdash \sigma_1 \quad \dots \quad (\sigma_{n-1}, \sigma_n) \vdash d_n \vdash \sigma_n \quad \sigma_n \vdash e \Rightarrow v}{\sigma \vdash d_1; \dots; d_n; e \Rightarrow v}$$

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

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:

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

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