

ATFAE – TRFAE with Algebraic Data Types

1 INTRODUCTION

ATFAE is a toy language for the [COSE212](#) course at Korea University. ATFAE stands for an extension of the [TRFAE](#) language with **algebraic data types**, and it supports the following features:

- **number (integer) values**
- **boolean values** (true and false)
- **arithmetic operators**: negation (-), addition (+), subtraction (-), multiplication (*), division (/), and modulo (%)
- **arithmetic comparison operators**: equality (== and !=) and relational (<, >, <=, and >=)
- **first-class functions**
- **recursive functions** (def)
- **conditionals** (if-else)
- **logical operators**: conjunction (&&), disjunction (||), and negation (!)
- **static type checking**
- **algebraic data types** (enum) and **pattern matching** (match)

This document is the specification of ATFAE. 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 ATFAE.

2 CONCRETE SYNTAX

The concrete syntax of ATFAE 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. 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>
<keyword>  ::= "true" | "false" | "def" | "if" | "else" | "val" | "enum"
           | "case" | "match" | "Number" | "Boolean"
<id>       ::= <idstart> <idcont>* butnot <keyword>
// expressions
<expr> ::= <number> | "true" | "false" | <uop> <expr> | <expr> <bop> <expr>
       | "(" <expr> ")" | "{" <expr> "}"
       | "val" <id> "=" <expr> ";"? <expr> | <id> | <params> "=" <expr>
       | "def" <id> <params> ":" <type> "=" <expr> ";"? <expr>
       | <expr> "(" " " ")" | <expr> "(" <expr> [ " " <expr> ]* ")"
       | "if" "(" <expr> ")" <expr> "else" <expr>
       | "enum" <id> "{" [ <variant> ";"? ]+ "}" ";"? <expr>
       | <expr> "match" "{" [ <mcase> ";"? ]+ "}"
```

```

// operators
<uop> ::= "-" | "!"
<bop> ::= "+" | "-" | "*" | "/" | "%" | "&&" | "||"
        | "==" | "!=" | "<" | "<=" | ">" | ">="
// function parameters
<params> ::= "(" ">" | "(" <param> [ ",", <param> ]* ">"
<param> ::= <id> ":" <type>
// variants
<variant> ::= "case" <id> "(" ">"
            | "case" <id> "(" <type> [ ",", <type> ]* ">"
// match cases
<mcase> ::= "case" <id> "(" ">" "=>" <expr>
            | "case" <id> "(" <id> [ ",", <id> ]* ">" "=>" <expr>
// types
<type> ::= "(" <type> ">" | "Number" | "Boolean" | <type> "=>" <type> | <id>

```

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	match	8	

3 ABSTRACT SYNTAX

The abstract syntax of ATFAE is defined as follows:

Expressions	$\mathbb{E} \ni e ::= n$	(Num)	val $x = e$; e	(Val)
	b	(Bool)	x	(Id)
	$e + e$	(Add)	$\lambda([x:\tau]^*).e$	(Fun)
	$e \times e$	(Mul)	def $x([x:\tau]^*): \tau = e$; e	(Rec)
	$e \div e$	(Div)	$e(e^*)$	(App)
	$e \bmod e$	(Mod)	if (e) e else e	(If)
	$e = e$	(Eq)	enum $t \{ [\text{case } x(\tau^*)]^* \}$; e	(TypeDef)
	$e < e$	(Lt)	$e \text{ match } \{ [\text{case } x(x^*) \Rightarrow e]^* \}$	(Match)

Types	$\mathbb{T} \ni \tau ::= \text{num}$	(NumT)	Numbers	$n \in \mathbb{Z}$	(BigInt)
	bool	(BoolT)	Identifiers	$x \in \mathbb{X}$	(String)
	$(\tau^*) \rightarrow \tau$	(ArrowT)	Booleans	$b \in \mathbb{B} = \{\text{true}, \text{false}\}$	(Boolean)
	t	(NameT)	Type Names	$t \in \mathbb{X}_t$	(String)

The types or semantics of the remaining cases are defined with the following desugaring rules:

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

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

4 TYPE SYSTEM

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

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

whose variants are commutative. For example,

$$A = B(\text{bool}) + C(\text{num}) \quad \text{equivalent to} \quad A = C(\text{num}) + B(\text{bool})$$

In the type system, type checking is defined with the following typing rules:

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

$$\begin{array}{c}
\tau\text{-Num} \frac{}{\Gamma \vdash n : \text{num}} \quad \tau\text{-Bool} \frac{}{\Gamma \vdash b : \text{bool}} \\
\\
\tau\text{-Add} \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{-Mul} \frac{\Gamma \vdash e_1 : \text{num} \quad \Gamma \vdash e_2 : \text{num}}{\Gamma \vdash e_1 \times e_2 : \text{num}} \\
\\
\tau\text{-Div} \frac{\Gamma \vdash e_1 : \text{num} \quad \Gamma \vdash e_2 : \text{num}}{\Gamma \vdash e_1 \div e_2 : \text{num}} \quad \tau\text{-Mod} \frac{\Gamma \vdash e_1 : \text{num} \quad \Gamma \vdash e_2 : \text{num}}{\Gamma \vdash e_1 \bmod e_2 : \text{num}} \\
\\
\tau\text{-Eq} \frac{\Gamma \vdash e_1 : \text{num} \quad \Gamma \vdash e_2 : \text{num}}{\Gamma \vdash e_1 = e_2 : \text{bool}} \quad \tau\text{-Lt} \frac{\Gamma \vdash e_1 : \text{num} \quad \Gamma \vdash e_2 : \text{num}}{\Gamma \vdash e_1 < e_2 : \text{bool}} \\
\\
\tau\text{-Val} \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{-Id} \frac{x \in \text{Domain}(\Gamma)}{\Gamma \vdash x : \Gamma(x)} \\
\\
\tau\text{-Fun} \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{-Rec} \frac{\Gamma \vdash \tau_1 \quad \dots \quad \Gamma \vdash \tau_n \quad \Gamma \vdash \tau \quad \Gamma[x_0 : (\tau_1, \dots, \tau_n) \rightarrow \tau, x_1 : \tau_1, \dots, x_n : \tau_n] \vdash e : \tau \quad \Gamma[x_0 : (\tau_1, \dots, \tau_n) \rightarrow \tau] \vdash e' : \tau'}{\Gamma \vdash \text{def } x_0(x_1 : \tau_1, \dots, x_n : \tau_n) : \tau = e; e' : \tau'} \\
\\
\tau\text{-App} \frac{\Gamma \vdash e_0 : (\tau_1, \dots, \tau_n) \rightarrow \tau \quad \Gamma \vdash e_1 : \tau_1 \quad \dots \quad \Gamma \vdash e_n : \tau_n}{\Gamma \vdash e_0(e_1, \dots, e_n) : \tau}
\end{array}$$

$$\begin{array}{c}
\tau\text{-If} \frac{\Gamma \vdash e_0 : \text{bool} \quad \Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash \text{if } (e_0) e_1 \text{ else } e_2 : \tau} \\
\\
\tau\text{-TypeDef} \frac{\begin{array}{c} \Gamma' = \Gamma[x_1(\tau_{1,1}, \dots, \tau_{1,m_1}) + \dots + x_n(\tau_{n,1}, \dots, \tau_{n,m_n})] \\ t \notin \text{Domain}(\Gamma) \quad \Gamma' \vdash \tau_{1,1} \quad \dots \quad \Gamma' \vdash \tau_{n,m_n} \end{array} \quad \frac{\Gamma'[x_1 : (\tau_{1,1}, \dots, \tau_{1,m_1}) \rightarrow t, \dots, x_n : (\tau_{n,1}, \dots, \tau_{n,m_n}) \rightarrow t] \vdash e : \tau \quad \Gamma \vdash \tau}{\Gamma \vdash \text{enum } t \{ \text{case } x_1(\tau_{1,1}, \dots, \tau_{1,m_1}); \dots; \text{case } x_n(\tau_{n,1}, \dots, \tau_{n,m_n}) \}; e : \tau}}{\Gamma \vdash e : t \quad \Gamma(t) = x_1(\tau_{1,1}, \dots, \tau_{1,m_1}) + \dots + x_n(\tau_{n,1}, \dots, \tau_{n,m_n})} \\
\tau\text{-Match} \frac{\forall 1 \leq i \leq n. \Gamma_i = \Gamma[x_{i,1} : \tau_{i,1}, \dots, x_{i,m_i} : \tau_{i,m_i}] \quad \Gamma_1 \vdash e_1 : \tau \quad \dots \quad \Gamma_n \vdash e_n : \tau}{\Gamma \vdash e \text{ match } \{ \text{case } x_1(x_{1,1}, \dots, x_{1,m_1}) \Rightarrow e_1; \dots; \text{case } x_n(x_{n,1}, \dots, x_{n,m_n}) \Rightarrow e_n \} : \tau}
\end{array}$$

and the following rules for well-formedness of types:

$$\begin{array}{c}
\boxed{\Gamma \vdash \tau} \\
\\
\frac{}{\Gamma \vdash \text{num}} \quad \frac{}{\Gamma \vdash \text{bool}} \quad \frac{\Gamma \vdash \tau_1 \quad \dots \quad \Gamma \vdash \tau_n \quad \Gamma \vdash \tau}{\Gamma \vdash (\tau_1, \dots, \tau_n) \rightarrow \tau} \\
\\
\frac{\Gamma(t) = x_1(\tau_{1,1}, \dots, \tau_{1,m_1}) + \dots + x_n(\tau_{n,1}, \dots, \tau_{n,m_n})}{\Gamma \vdash t}
\end{array}$$

5 SEMANTICS

We use the following notations in the semantics:

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

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

$$\begin{array}{c}
\boxed{\sigma \vdash e \Rightarrow v} \\
\\
\text{Num} \frac{}{\sigma \vdash n \Rightarrow n} \quad \text{Bool} \frac{}{\sigma \vdash b \Rightarrow b} \\
\\
\text{Add} \frac{\sigma \vdash e_1 \Rightarrow n_1 \quad \sigma \vdash e_2 \Rightarrow n_2}{\sigma \vdash e_1 + e_2 \Rightarrow n_1 + n_2} \quad \text{Mul} \frac{\sigma \vdash e_1 \Rightarrow n_1 \quad \sigma \vdash e_2 \Rightarrow n_2}{\sigma \vdash e_1 \times e_2 \Rightarrow n_1 \times n_2} \\
\\
\text{Div} \frac{\sigma \vdash e_1 \Rightarrow n_1 \quad \sigma \vdash e_2 \Rightarrow n_2 \quad n_2 \neq 0}{\sigma \vdash e_1 \div e_2 \Rightarrow n_1 \div n_2} \quad \text{Mod} \frac{\sigma \vdash e_1 \Rightarrow n_1 \quad \sigma \vdash e_2 \Rightarrow n_2 \quad n_2 \neq 0}{\sigma \vdash e_1 \text{ mod } e_2 \Rightarrow n_1 \text{ mod } n_2} \\
\\
\text{Eq} \frac{\sigma \vdash e_1 \Rightarrow n_1 \quad \sigma \vdash e_2 \Rightarrow n_2}{\sigma \vdash e_1 = e_2 \Rightarrow n_1 = n_2}
\end{array}$$

$$\begin{array}{c}
\text{Lt} \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{Val} \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{Id} \frac{x \in \text{Domain}(\sigma)}{\sigma \vdash x \Rightarrow \sigma(x)} \\
\\
\text{Fun} \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{Rec} \frac{\sigma' = \sigma[x_0 \mapsto \langle \lambda(x_1, \dots, x_n).e, \sigma' \rangle] \quad \sigma' \vdash e' \Rightarrow v'}{\sigma \vdash \text{def } x_0(x_1:\tau_1, \dots, x_n:\tau_n):\tau = e; e' \Rightarrow v'} \\
\\
\text{App}_\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(e_1, \dots, e_n) \Rightarrow v} \\
\\
\text{App}_{(-)} \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(e_1, \dots, e_n) \Rightarrow x(v_1, \dots, v_n)} \\
\\
\text{If}_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{If}_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{TypeDef} \frac{\sigma[x_1 \mapsto \langle x_1 \rangle, \dots, x_n \mapsto \langle x_n \rangle] \vdash e \Rightarrow v}{\sigma \vdash \text{enum } t \{ \text{case } x_1(\tau_{1,1}, \dots, \tau_{1,m_1}); \dots; \text{case } x_n(\tau_{n,1}, \dots, \tau_{n,m_n}) \}; e \Rightarrow v} \\
\\
\text{Match} \frac{\sigma \vdash e \Rightarrow x_i(v_1, \dots, v_{m_i}) \quad \forall j < i. x_j \neq x_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}$$