

# Lecture 21 – Algebraic Data Types (1)

## COSE212: Programming Languages

Jihyeok Park



2023 Fall

- **TFAE** – FAE with **type system**.
  - **Type Checker** and **Typing Rules**
  - Interpreter and Natural Semantics
- **TRFAE** – RFAE with **type system**.
  - **Type Checker** and **Typing Rules**
  - Interpreter and Natural Semantics

- **TFAE** – FAE with **type system**.
  - **Type Checker** and **Typing Rules**
  - Interpreter and Natural Semantics
- **TRFAE** – RFAE with **type system**.
  - **Type Checker** and **Typing Rules**
  - Interpreter and Natural Semantics
- Let's learn **algebraic data types (ADTs)** and **pattern matching**!

- **TFAE** – FAE with **type system**.
  - **Type Checker** and **Typing Rules**
  - Interpreter and Natural Semantics
- **TRFAE** – RFAE with **type system**.
  - **Type Checker** and **Typing Rules**
  - Interpreter and Natural Semantics
- Let's learn **algebraic data types (ADTs)** and **pattern matching**!
- **TAF AE** – TRFAE with **ADTs** and **pattern matching**.
  - **Interpreter** and **Natural Semantics**
  - **Type Checker** and **Typing Rules**

- **TFAE** – FAE with **type system**.
  - **Type Checker** and **Typing Rules**
  - Interpreter and Natural Semantics
- **TRFAE** – RFAE with **type system**.
  - **Type Checker** and **Typing Rules**
  - Interpreter and Natural Semantics
- Let's learn **algebraic data types (ADTs)** and **pattern matching**!
- **TFAE** – TRFAE with **ADTs** and **pattern matching**.
  - **Interpreter** and **Natural Semantics**
  - **Type Checker** and **Typing Rules**
- In this lecture, we will focus on **Interpreter** and **Natural Semantics**.

## 1. Algebraic Data Types (ADTs) and Pattern Matching

- Recall: Types

- Product Types

- Union Types

- Sum Types

- Algebraic Data Types (ADTs)

- Pattern Matching

## 2. TFAE – TRFAE with ADTs and Pattern Matching

- Concrete Syntax

- Abstract Syntax

## 3. Interpreter and Natural Semantics for TFAE

- Algebraic Data Types

- Function Application

- Pattern Matching

- Examples

## 1. Algebraic Data Types (ADTs) and Pattern Matching

Recall: Types

Product Types

Union Types

Sum Types

Algebraic Data Types (ADTs)

Pattern Matching

## 2. TFAE – TRFAE with ADTs and Pattern Matching

Concrete Syntax

Abstract Syntax

## 3. Interpreter and Natural Semantics for TFAE

Algebraic Data Types

Function Application

Pattern Matching

Examples

## Definition (Types)

A **type** is a set of values.

For example, the `Int`, `Boolean`, and `Int => Int` types are defined as the following sets of values in Scala.

$$\begin{aligned}\text{Int} &= \{n \in \mathbb{Z} \mid -2^{31} \leq n < 2^{31}\} \\ \text{Boolean} &= \{\text{true}, \text{false}\} \\ \text{Int} \Rightarrow \text{Int} &= \{f \mid f \text{ is a function from Int to Int}\}\end{aligned}$$

```
val n: Int = 42           // 42    : Int
val b: Boolean = n > 10    // true   : Boolean
def f(x: Int): Int = x + 1 // f     : Int => Int
f(42)                   // 43    : Int
```



## Definition (Types)

A **type** is a set of values.

For example, the `Int`, `Boolean`, and `Int => Int` types are defined as the following sets of values in Scala.

$$\begin{aligned}\text{Int} &= \{n \in \mathbb{Z} \mid -2^{31} \leq n < 2^{31}\} \\ \text{Boolean} &= \{\text{true}, \text{false}\} \\ \text{Int} \Rightarrow \text{Int} &= \{f \mid f \text{ is a function from Int to Int}\}\end{aligned}$$

```
val n: Int = 42           // 42    : Int
val b: Boolean = n > 10    // true  : Boolean
def f(x: Int): Int = x + 1 // f     : Int => Int
f(42)                   // 43    : Int
```

Is it possible to define a **new type** by **combining** existing types?

## Definition (Types)

A **type** is a set of values.

For example, the `Int`, `Boolean`, and `Int => Int` types are defined as the following sets of values in Scala.

$$\begin{aligned}\text{Int} &= \{n \in \mathbb{Z} \mid -2^{31} \leq n < 2^{31}\} \\ \text{Boolean} &= \{\text{true}, \text{false}\} \\ \text{Int} \Rightarrow \text{Int} &= \{f \mid f \text{ is a function from Int to Int}\}\end{aligned}$$

```
val n: Int = 42           // 42    : Int
val b: Boolean = n > 10    // true  : Boolean
def f(x: Int): Int = x + 1 // f     : Int => Int
f(42)                    // 43    : Int
```

Is it possible to define a **new type** by **combining** existing types? **Yes!**

## Definition (Types)

A **type** is a set of values.

For example, the `Int`, `Boolean`, and `Int => Int` types are defined as the following sets of values in Scala.

$$\begin{aligned}\text{Int} &= \{n \in \mathbb{Z} \mid -2^{31} \leq n < 2^{31}\} \\ \text{Boolean} &= \{\text{true}, \text{false}\} \\ \text{Int} \Rightarrow \text{Int} &= \{f \mid f \text{ is a function from Int to Int}\}\end{aligned}$$

```
val n: Int = 42           // 42    : Int
val b: Boolean = n > 10    // true  : Boolean
def f(x: Int): Int = x + 1 // f     : Int => Int
f(42)                    // 43    : Int
```

Is it possible to define a **new type** by **combining** existing types? **Yes!**

Product Types, Union Types, Sum Types, and Algebraic Data Types!

## Definition (Product Types)

A **product type**  $(\tau_1, \dots, \tau_n)$  is a set of values of the form  $(v_1, \dots, v_n)$  where  $\tau_i$  is the type of  $v_i$  for  $1 \leq i \leq n$ .

## Definition (Product Types)

A **product type**  $(\tau_1, \dots, \tau_n)$  is a set of values of the form  $(v_1, \dots, v_n)$  where  $\tau_i$  is the type of  $v_i$  for  $1 \leq i \leq n$ .

It corresponds to the **Cartesian product** of sets:

$$(\tau_1, \dots, \tau_n) = \tau_1 \times \dots \times \tau_n$$

## Definition (Product Types)

A **product type**  $(\tau_1, \dots, \tau_n)$  is a set of values of the form  $(v_1, \dots, v_n)$  where  $\tau_i$  is the type of  $v_i$  for  $1 \leq i \leq n$ .

It corresponds to the **Cartesian product** of sets:

$$(\tau_1, \dots, \tau_n) = \tau_1 \times \dots \times \tau_n$$

For example, we can define product types in Scala as follows:

```
// A product type consisting of three different types
val triple: (Int, Boolean, String) = (42, true, "abc")

// A rectangle type with its width and height
type Rectangle = (Int, Int)
val rectangle: Rectangle = (10, 20)
val (w, h) = rectangle
val perimeter: Int = 2 * (w + h)           // 2 * (10 + 20) = 60
```

## Definition (Union Types)

A **union type**  $\tau_1 \mid \dots \mid \tau_n$  is a set of values whose type is one of  $\tau_1, \dots, \tau_n$ .

## Definition (Union Types)

A **union type**  $\tau_1 \mid \dots \mid \tau_n$  is a set of values whose type is one of  $\tau_1, \dots, \tau_n$ .

It corresponds to the **union** of sets:

$$\tau_1 \mid \dots \mid \tau_n = \tau_1 \cup \dots \cup \tau_n$$



## Definition (Union Types)

A **union type**  $\tau_1 \mid \dots \mid \tau_n$  is a set of values whose type is one of  $\tau_1, \dots, \tau_n$ .

It corresponds to the **union** of sets:

$$\tau_1 \mid \dots \mid \tau_n = \tau_1 \cup \dots \cup \tau_n$$

For example, we can define union types in Scala as follows:

```
val a: Int | Boolean | String = 42
val b: Int | Boolean | String = true
val c: Int | Boolean | String = "abc"

type Square = Int           // A square type
type Triangle = Int         // A equilateral triangle type
val x: Square | Traingle = 42 // Is this a square or a triangle?
```

## Definition (Union Types)

A **union type**  $\tau_1 \mid \dots \mid \tau_n$  is a set of values whose type is one of  $\tau_1, \dots, \tau_n$ .

It corresponds to the **union** of sets:

$$\tau_1 \mid \dots \mid \tau_n = \tau_1 \cup \dots \cup \tau_n$$

For example, we can define union types in Scala as follows:

```
val a: Int | Boolean | String = 42
val b: Int | Boolean | String = true
val c: Int | Boolean | String = "abc"

type Square = Int           // A square type
type Triangle = Int         // A equilateral triangle type
val x: Square | Triangle = 42 // Is this a square or a triangle?
```

How can we **discriminate** between a square and a triangle?

## Definition (Union Types)

A **union type**  $\tau_1 \mid \dots \mid \tau_n$  is a set of values whose type is one of  $\tau_1, \dots, \tau_n$ .

It corresponds to the **union** of sets:

$$\tau_1 \mid \dots \mid \tau_n = \tau_1 \cup \dots \cup \tau_n$$

For example, we can define union types in Scala as follows:

```
val a: Int | Boolean | String = 42
val b: Int | Boolean | String = true
val c: Int | Boolean | String = "abc"

type Square = Int           // A square type
type Triangle = Int         // A equilateral triangle type
val x: Square | Traingle = 42 // Is this a square or a triangle?
```

How can we **discriminate** between a square and a triangle? **Sum types!**

## Definition (Sum Types)

A **sum type**  $x_1(\tau_1) + \dots + x_n(\tau_n)$  consists of **variants**  $x_i(\tau_i)$  for  $1 \leq i \leq n$ . For each variant  $x_i(\tau_i)$ ,  $x_i$  is the **constructor**, a function that takes a value  $v$  of type  $\tau_i$  and generates a value  $x_i(v)$  of the sum type.

## Definition (Sum Types)

A **sum type**  $x_1(\tau_1) + \dots + x_n(\tau_n)$  consists of **variants**  $x_i(\tau_i)$  for  $1 \leq i \leq n$ . For each variant  $x_i(\tau_i)$ ,  $x_i$  is the **constructor**, a function that takes a value  $v$  of type  $\tau_i$  and generates a value  $x_i(v)$  of the sum type.

It corresponds to a **tagged union** of sets:

$$x_1(\tau_1) + \dots + x_n(\tau_n) = \{x_i(v) \mid \exists 1 \leq i \leq n. \text{ s.t. } v \in \tau_i\}$$

## Definition (Sum Types)

A **sum type**  $x_1(\tau_1) + \dots + x_n(\tau_n)$  consists of **variants**  $x_i(\tau_i)$  for  $1 \leq i \leq n$ . For each variant  $x_i(\tau_i)$ ,  $x_i$  is the **constructor**, a function that takes a value  $v$  of type  $\tau_i$  and generates a value  $x_i(v)$  of the sum type.

It corresponds to a **tagged union** of sets:

$$x_1(\tau_1) + \dots + x_n(\tau_n) = \{x_i(v) \mid \exists 1 \leq i \leq n. \text{ s.t. } v \in \tau_i\}$$

For example, we can define **sum types** in Scala as follows:

```
case class Square(side: Int)
case class Triangle(side: Int)
type Shape = Square | Triangle
val x: Shape = Square(42)      // It is a square
val y: Shape = Triangle(42)    // It is a triangle
```

Now, we can **discriminate** between a square and a triangle!

## Definition (Sum Types)

A **sum type**  $x_1(\tau_1) + \dots + x_n(\tau_n)$  consists of **variants**  $x_i(\tau_i)$  for  $1 \leq i \leq n$ . For each variant  $x_i(\tau_i)$ ,  $x_i$  is the **constructor**, a function that takes a value  $v$  of type  $\tau_i$  and generates a value  $x_i(v)$  of the sum type.

```
case class Square(side: Int)           // A variant for squares
case class Triangle(side: Int)         // A variant for triangles
type Shape = Square | Triangle
val x: Square | Triangle = Square(42)  // It is a square
val y: Square | Triangle = Triangle(42) // It is a triangle

// `Square` is a constructor that takes an `Int` and generates a `Shape`
Square: Int => Shape

// `Square(42)` is a `Shape` value generated by `Square` constructor
Square(42): Shape
```

## Definition (Algebraic Data Types (ADTs))

An **algebraic data type**  $x_1(\tau_{1,1}, \dots, \tau_{1,m_1}) + \dots + x_n(\tau_{n,1}, \dots, \tau_{n,m_n})$  is a **recursive sum type** of **product types**.

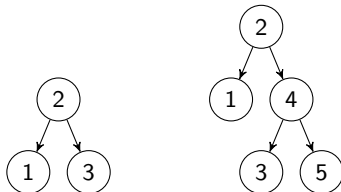


## Definition (Algebraic Data Types (ADTs))

An **algebraic data type**  $x_1(\tau_{1,1}, \dots, \tau_{1,m_1}) + \dots + x_n(\tau_{n,1}, \dots, \tau_{n,m_n})$  is a **recursive sum type** of **product types**.

For example, we can define **algebraic data type** for trees in Scala:

```
enum Tree:  
  case Leaf(v: Int)  
  case Node(l: Tree, v: Int, r: Tree)  
  
val t1: Tree = Node(Leaf(1), 2, Leaf(3))  
val t2: Tree = Node(Leaf(1), 2, Node(Leaf(3), 4, Leaf(5)))
```



## Definition (Pattern matching)

We can use **pattern matching** for algebraic data types to identify which variant of the sum type a value belongs to and extract the data it contains.

## Definition (Pattern matching)

We can use **pattern matching** for algebraic data types to identify which variant of the sum type a value belongs to and extract the data it contains.

For example, we can define a function `sum` that sums all the values in a tree using pattern matching (`match`) on the `Tree` type in Scala:

```
enum Tree:
  case Leaf(v: Int)
  case Node(l: Tree, v: Int, r: Tree)

def sum(t: Tree): Int = t match
  case Leaf(v)          => v
  case Node(l, v, r) => sum(l) + v + sum(r)

sum(Node(Leaf(1), 2, Leaf(3)))           // 6
sum(Node(Leaf(1), 2, Node(Leaf(3), 4, Leaf(5)))) // 15
```

Many functional languages support **algebraic data types**:

- Scala

```
enum Tree { Leaf(v: Int), Node(l: Tree, v: Int, r: Tree) }
```

- Haskell

```
data Tree = Leaf Int | Node Tree Int Tree
```

- Rust

```
enum Tree { Leaf(i32), Node(Tree, i32, Tree) }
```

- OCaml

```
type tree = Leaf of int | Node of tree * int * tree
```

- ...

## 1. Algebraic Data Types (ADTs) and Pattern Matching

Recall: Types

Product Types

Union Types

Sum Types

Algebraic Data Types (ADTs)

Pattern Matching

## 2. TFAE – TRFAE with ADTs and Pattern Matching

Concrete Syntax

Abstract Syntax

## 3. Interpreter and Natural Semantics for TFAE

Algebraic Data Types

Function Application

Pattern Matching

Examples

# TAF AE – TRFAE with ADTs and Pattern Matching PLRG

Now, let's extend TRFAE into TAF AE to support **algebraic data types** and **pattern matching**. (Assume that TRFAE supports multiple arguments for functions.)

```
/* TAF AE */  
enum Tree {  
  case Leaf(Number)  
  case Node(Tree, Number, Tree)  
}  
Leaf(42) match {  
  case Leaf(v)      => v  
  case Node(l, v, r) => v  
}
```

For TAF AE, we need to extend **expressions** of TRFAE with

- 1 algebraic data types (ADTs)
- 2 pattern matching
- 3 type names

For TAFAE, we need to extend **expressions** of TRFAE with

- ① **algebraic data types (ADTs)**
- ② **pattern matching**
- ③ **type names**

For TFAE, we need to extend **expressions** of TRFAE with

- 1 algebraic data types (ADTs)
- 2 pattern matching
- 3 type names

We can extend the **concrete syntax** of FAE as follows:

```
// expressions
<expr> ::= ...
    | "enum" <id> "{" [ <variant> ";"? ]+ "}" ";"? <expr>
    | <expr> "match" "{" [ <mcase> ";"? ]+ "}"

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

// match cases
<mcase> ::= "case" <id> "(" " )" "=" <expr>
    | "case" <id> "(" <id> [ "," <id> ]* ")" "=" <expr>

// types
<type> ::= ... | <id>           // type names
```



Expressions	$\mathbb{E} \ni e ::= \dots$	
	$\text{enum } t \{ [\text{case } x(\tau^*)]^* \}; e$	(TypeDef)
	$e \text{ match } \{ [\text{case } x(x^*) \Rightarrow e]^* \}$	(Match)
Types	$\mathbb{T} \ni \tau ::= \dots$	
	$t$	(NameT)
Type Names	$t \in \mathbb{X}_t$	(String)

```

enum Expr:
  ...
  case TypeDef(name: String, varts: List[Variant], body: Expr)
  case Match(expr: Expr, mcases: List[MatchCase])

case class Variant(name: String, ptys: List[Type]):
case class MatchCase(name: String, params: List[String], body: Expr):

enum Type:
  ...
  case NameT(name: String)
    
```

```
/* TAF AE */  
enum Tree {  
  case Leaf(Number)  
  case Node(Tree, Number, Tree)  
}  
Leaf(42) match {  
  case Leaf(v)      => v  
  case Node(l, v, r) => v  
}
```

will be parsed to the following abstract syntax tree (AST) in Scala:

```
TypeDef("Tree", List(  
  Variant("Leaf", List(NumT)),  
  Variant("Node", List(NameT("Tree"), NumT, NameT("Tree")))  
),  
Match(App(Id("Leaf"), List(Num(42))), List(  
  MatchCase("Leaf", List("v"), Id("v")),  
  MatchCase("Node", List("l", "v", "r"), Id("v")))))
```

## 1. Algebraic Data Types (ADTs) and Pattern Matching

Recall: Types

Product Types

Union Types

Sum Types

Algebraic Data Types (ADTs)

Pattern Matching

## 2. TFAE – TRFAE with ADTs and Pattern Matching

Concrete Syntax

Abstract Syntax

## 3. Interpreter and Natural Semantics for TFAE

Algebraic Data Types

Function Application

Pattern Matching

Examples

For TAFAE, we need to 1) implement the **interpreter** with environments:

```
def interp(expr: Expr, env: Env): Value = ???
```

and 2) define the **natural semantics** with environments:

$$\sigma \vdash e \Rightarrow v$$

For TAFAE, we need to 1) implement the **interpreter** with environments:

```
def interp(expr: Expr, env: Env): Value = ???
```

and 2) define the **natural semantics** with environments:

$$\sigma \vdash e \Rightarrow v$$

with a new kind of values called **constructor values** and **variant values**:

Values	$\forall \ni v ::= n$	(NumV)	$\langle x \rangle$	(ConstrV)
	$b$	(BoolV)	$x(v^*)$	(VariantV)
	$\langle \lambda x.(e, \dots, e), \sigma \rangle$	(CloV)		

```
enum Value:
  ...
  case ConstrV(name: String)
  case VariantV(name: String, values: List[Value])
```

```
def interp(expr: Expr, env: Env): Value = expr match
...
case TypeDef(_, ws, body) =>
  interp(body, env ++ ws.map(w => w.name -> ConstrV(w.name)))
```

$$\sigma \vdash e \Rightarrow v$$

$$\text{TypeDef} \frac{\sigma[x_1 \mapsto \langle x_1 \rangle, \dots, x_n \mapsto \langle x_n \rangle] \vdash e \Rightarrow v}{\sigma \vdash \text{enum } t \left\{ \begin{array}{l} \text{case } x_1(\tau_{1,1}, \dots, \tau_{1,m_1}) \\ \dots \\ \text{case } x_n(\tau_{n,1}, \dots, \tau_{n,m_n}) \end{array} \right\}; e \Rightarrow v}$$

```
/* TAFAE */
enum Tree { case Leaf(Number); case Node(Tree, Number, Tree) }
Leaf(42) match { case Leaf(v) => v; case Node(l, v, r) => v }
```

```
def interp(expr: Expr, env: Env): Value = expr match
...
case App(f, es) => interp(f, env) match
  case CloV(ps, b, fenv) => ...
  case ConstrV(name) => VariantV(name, es.map(interp(_, env)))
  case v                => error(s"not a function: ${v.str}")
```

$$\sigma \vdash e \Rightarrow v$$

$$\text{App}_{(-)} \frac{\sigma \vdash e_0 \Rightarrow \langle x \rangle \quad \sigma \vdash e_1 \Rightarrow v_1 \quad \dots \quad \sigma \vdash e_n \Rightarrow v_n}{\sigma \vdash e_0(e_1, \dots, e_n) \Rightarrow x(v_1, \dots, v_n)}$$

```
/* TAFAE */
enum Tree { case Leaf(Number); case Node(Tree, Number, Tree) }
Leaf(42) match { case Leaf(v) => v; case Node(l, v, r) => v }
```

```
def interp(expr: Expr, env: Env): Value = expr match
...
case Match(expr, cases) => interp(expr, env) match
  case VariantV(wname, vs) => cases.find(_.name == wname) match
    case Some(MatchCase(_, ps, b)) =>
      if (ps.length != vs.length) error("arity mismatch")
      interp(b, env ++ (ps zip vs))
    case None => error(s"no such case: $wname")
  case v => error(s"not a variant: ${v.str}")
```

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

$$\text{Match} \frac{1 \leq i \leq n \quad \sigma \vdash e \Rightarrow x_i(v_1, \dots, v_{m_i}) \quad \forall j < i. x_j \neq x_i \quad \sigma[x_{i,1} \mapsto v_1, \dots, x_{i,m_i} \mapsto v_{m_i}] \vdash e_i \Rightarrow v}{\sigma \vdash e \text{ match } \left\{ \begin{array}{l} \text{case } x_1(x_{1,1}, \dots, x_{1,m_1}) \Rightarrow e_1 \\ \dots \\ \text{case } x_n(x_{n,1}, \dots, x_{n,m_n}) \Rightarrow e_n \end{array} \right\} \Rightarrow v}$$



There exists an **order** between the match cases: **first match wins!**

$$\begin{array}{c}
 1 \leq i \leq n \quad \sigma \vdash e \Rightarrow x_i(v_1, \dots, v_{m_i}) \quad \forall j < i. x_j \neq x_i \\
 \text{Match} \frac{\sigma[x_{i,1} \mapsto v_1, \dots, x_{i,m_i} \mapsto v_{m_i}] \vdash e_i \Rightarrow v}{\sigma \vdash e \text{ match } \left\{ \begin{array}{l} \text{case } x_1(x_{1,1}, \dots, x_{1,m_1}) \Rightarrow e_1 \\ \dots \\ \text{case } x_n(x_{n,1}, \dots, x_{n,m_n}) \Rightarrow e_n \end{array} \right\} \Rightarrow v}
 \end{array}$$

```

/* TAFAE */
enum Tree {
  case Leaf(Number)
  case Node(Tree, Number, Tree)
}
def f(t: Tree): Number = t match {
  case Leaf(v)          => v
  case Leaf(v)          => v + 1      // ignored
  case Node(l, v, r)    => v
  case Node(l, v, r)    => v + 1      // ignored
}; ...
    
```

# Example 1

```
/* TAFAE */  
enum A { case B(Boolean); case C(Number) }  
C(42) match { case B(b) => b; case C(n) => n < 0 }
```

$$\begin{array}{c} \text{Id } \frac{C \in \text{Domain}(\sigma_1)}{\sigma_1 \vdash C \Rightarrow \langle C \rangle} \quad \text{Num } \frac{}{\sigma_1 \vdash 42 \Rightarrow 42} \quad \text{Id } \frac{n \in \text{Domain}(\sigma_2)}{\sigma_2 \vdash n \Rightarrow 42} \quad \text{Num } \frac{}{\sigma_2 \vdash 0 \Rightarrow 0} \\ \text{App}_{\langle - \rangle} \frac{}{\sigma_1 \vdash C(42) \Rightarrow C(42)} \quad \text{Lt } \frac{}{\sigma_2 \vdash n < 0 \Rightarrow \text{false}} \\ \text{Match } \frac{}{\sigma_1 \vdash C(42) \text{ match } \left\{ \begin{array}{l} \text{case } B(b) \Rightarrow b \\ \text{case } C(n) \Rightarrow n < 0 \end{array} \right\} \Rightarrow \text{false}} \\ \text{TypeDef } \frac{}{\emptyset \vdash \text{enum } A \left\{ \begin{array}{l} \text{case } B(\text{bool}) \\ \text{case } C(\text{num}) \end{array} \right\}; C(42) \text{ match } \left\{ \begin{array}{l} \text{case } B(b) \Rightarrow b \\ \text{case } C(n) \Rightarrow n < 0 \end{array} \right\} \Rightarrow \text{false}}$$

where

$$\begin{aligned} \sigma_1 &= [B \mapsto \langle B \rangle, C \mapsto \langle C \rangle] \\ \sigma_2 &= \sigma_2[n \mapsto 42] \end{aligned}$$

## Example 2

In **TFAE**, we cannot define `mkRec` because of the lack of **recursive types** in the language:

```
/* TFAE */

val mkRec = (body: (Number => Number) => Number => Number) => {
  val fX = (fY: ???) => {
    val f = (x: Number) => fY(fY)(x);
    body(f)
  };
  fX(fX)
};

val sum = mkRec((sum: Number => Number) => (n: Number) =>
  if (n < 1) 0
  else n + sum(n + -1));

sum(10)
```

## Example 2

Now, we can define `mkRec` in **TAF AE** because **algebraic data types** are **recursive types**:

```
/* TAF AE */  
enum T { case T(T => Number => Number) }  
val mkRec = (body: (Number => Number) => Number => Number) => {  
  val fX = (fY: T) => {  
    val f = (x: Number) => fY match { case T(fZ) => fZ(fY)(x) };  
    body(f)  
  };  
  fX(T(fX))  
};  
val sum = mkRec((sum: Number => Number) => (n: Number) =>  
  if (n < 1) 0  
  else n + sum(n + -1));  
sum(10)
```

## Example 3

We can define abstract syntax of AE using ADTs in TAF AE:

```
/* TAF AE */  
enum Expr:  
  case Num(number: Number)  
  case Add(left: Expr, right: Expr)  
  case Mul(left: Expr, right: Expr)  
Add(Num(1), Mul(Num(2), Num(3)))      // 1 + 2 * 3
```

## Example 3

We can define abstract syntax of AE using ADTs in TAF AE:

```
/* TAF AE */  
enum Expr:  
  case Num(number: Number)  
  case Add(left: Expr, right: Expr)  
  case Mul(left: Expr, right: Expr)  
Add(Num(1), Mul(Num(2), Num(3)))           // 1 + 2 * 3
```

We can define list type as well using ADTs in TAF AE:

```
/* TAF AE */  
enum List:  
  case Nil  
  case Cons(head: Number, tail: List)  
Cons(1, Cons(2, Cons(3, Nil)))             // List of 1, 2, and 3
```

## Example 3

We can define abstract syntax of AE using ADTs in TAF AE:

```
/* TAF AE */
enum Expr:
  case Num(number: Number)
  case Add(left: Expr, right: Expr)
  case Mul(left: Expr, right: Expr)
Add(Num(1), Mul(Num(2), Num(3)))           // 1 + 2 * 3
```

We can define list type as well using ADTs in TAF AE:

```
/* TAF AE */
enum List:
  case Nil
  case Cons(head: Number, tail: List)
Cons(1, Cons(2, Cons(3, Nil)))             // List of 1, 2, and 3
```

However, it only works for **monomorphic** lists (i.e., lists of numbers)

## Example 3

We can define abstract syntax of AE using ADTs in TAF AE:

```
/* TAF AE */  
enum Expr:  
  case Num(number: Number)  
  case Add(left: Expr, right: Expr)  
  case Mul(left: Expr, right: Expr)  
Add(Num(1), Mul(Num(2), Num(3)))           // 1 + 2 * 3
```

We can define list type as well using ADTs in TAF AE:

```
/* TAF AE */  
enum List:  
  case Nil  
  case Cons(head: Number, tail: List)  
Cons(1, Cons(2, Cons(3, Nil)))             // List of 1, 2, and 3
```

However, it only works for **monomorphic** lists (i.e., lists of numbers)

We will learn **parametric polymorphism** later in this course.



## 1. Algebraic Data Types (ADTs) and Pattern Matching

- Recall: Types

- Product Types

- Union Types

- Sum Types

- Algebraic Data Types (ADTs)

- Pattern Matching

## 2. TFAE – TRFAE with ADTs and Pattern Matching

- Concrete Syntax

- Abstract Syntax

## 3. Interpreter and Natural Semantics for TFAE

- Algebraic Data Types

- Function Application

- Pattern Matching

- Examples

- Algebraic Data Types (2)

Jihyeok Park

`jihyeok_park@korea.ac.kr`

`https://plrg.korea.ac.kr`