Lecture 21 – Algebraic Data Types (1)
COSE212: Programming Languages

Jihyeok Park

⚫ PLRG

2023 Fall

PLRG

- **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**!

- **TAFAE** – 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**.

Contents

Contents

Recall: 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.

$$\text{Int} = \{n \in \mathbb{Z} \mid -2^{31} \le n < 2^{31}\}$$
$$\text{Boolean} = \{\text{true}, \text{false}\}$$
$$\text{Int => Int} = \{f \mid f \text{ is a function from Int to Int}\}$$

```scala
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!

Product Types

Definition (Product Types)

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

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

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

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

```scala
// 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
```

Union Types

> **Definition (Union Types)**
>
> A **union type** $\tau_1 \mid \ldots \mid \tau_n$ is a set of values whose type is one of $\tau_1, \ldots, \tau_n$.

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

$$\tau_1 \mid \ldots \mid \tau_n = \tau_1 \cup \ldots \cup \tau_n$$

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

```scala
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**!

Sum Types

Definition (Sum Types)

A **sum type** $x_1(\tau_1) + \ldots + x_n(\tau_n)$ 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 τ_i and generates a value $x_i(v)$ of the sum type.

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

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

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

```scala
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!

Sum Types

Definition (Sum Types)

A **sum type** $x_1(\tau_1) + \ldots + x_n(\tau_n)$ 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 τ_i and generates a value $x_i(v)$ of the sum type.

```scala
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
```
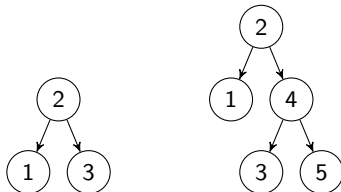
Algebraic Data Types (ADTs)

Definition (Algebraic Data Types (ADTs))

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

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

```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)))
```

Pattern Matching

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:

```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
```

Algebraic Data Types

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

- Scala

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

- Haskell

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

- Rust

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

- OCaml

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

- ...

Contents

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

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

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

1. **algebraic data types (ADTs)**
2. **pattern matching**
3. **type variables**

Concrete Syntax

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

1. **algebraic data types (ADTs)**
2. **pattern matching**
3. **type variables**

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

```
// expressions
<expr> ::= ...
         | "enum" <id> "{" [ <variant> ";"? ]+ "}"
         | <expr> "match" "{" [ <mcase> ";"? ]+ "}"
// variants
<variant> ::= <id> "(" ")" | <id> "(" <type> [ "," <type> ]* ")"
// match cases
<mcase> ::= "case" <id> "(" ")" "=>" <expr>
          | "case" <id> "(" <id> [ "," <id> ]* ")" "=>" <expr>
// types
<type> ::= ...
         | <id>          // type variable
```

Abstract Syntax

Expressions $\mathbb{E} \ni e ::= \dots$
$$| \text{ enum } t \ \{ \ [\text{case } x(\tau^*)]^* \ \}; \ e \qquad (\texttt{TypeDef})$$
$$| \ e \text{ match } \{ \ [\text{case } x(x^*) \text{ => } e]^* \ \} \quad (\texttt{Match})$$

Types $\mathbb{T} \ni \tau ::= \dots$
$$| \ t \qquad\qquad\qquad\qquad\qquad (\texttt{VarT})$$

```
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 VarT(name: String)
```

Abstract Syntax

```
/* TAFAE */
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(VarT("Tree"), NumT, VarT("Tree")))
  ),
  Match(App(Id("Node"), List(...)), List(
      MatchCase("Leaf", List("v"), Id("v")),
      MatchCase("Node", List("l", "v", "r"), Id("v")))))
```

Contents

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 $\mathbb{V} \ni v ::= n$ (NumV) $| \langle x \rangle$ (ConstrV)
                  $| b$ (BoolV) $| x(v^*)$ (VariantV)
                  $| \langle \lambda x.(e, \ldots, e), \sigma \rangle$ (CloV)

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

Algebraic Data Types

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

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

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

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

Algebraic Data Types

```
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}")
```

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

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

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

Pattern Matching

```scala
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)) =>
        arityCheck(ps.length, vs.length)
        interp(b, env ++ (ps zip vs))
      case None => error(s"no such case: $wname")
    case v => error(s"not a variant: ${v.str}")
```

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

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

Pattern Matching

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

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

```
/* TAFAE */
enum Tree {
  case Leaf(Number)
  case Node(Tree, Number, Tree)
}
Node(Leaf(1), 2, Leaf(3)) 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

PLRG

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

$$\text{App}_{\langle - \rangle} \frac{\text{Id} \dfrac{C \in \text{Domain}(\sigma_1)}{\sigma_1 \vdash C \Rightarrow \langle C \rangle} \quad \text{Num} \dfrac{}{\sigma_1 \vdash 42 \Rightarrow 42}}{\sigma_1 \vdash C(42) \Rightarrow C(42)} \qquad \text{Lt} \frac{\text{Id} \dfrac{n \in \text{Domain}(\sigma_2)}{\sigma_2 \vdash n \Rightarrow 42} \quad \text{Num} \dfrac{}{\sigma_2 \vdash 0 \Rightarrow 0}}{\sigma_2 \vdash n < 0 \Rightarrow \text{false}}$$

$$\text{TypeDef} \cfrac{\text{Match} \cfrac{\sigma_1 \vdash C(42) \text{ match } \left\{ \begin{array}{l} \text{case } B(b) \text{ => } b \\ \text{case } C(n) \text{ => } n < 0 \end{array} \right\} \Rightarrow \text{false}}{\varnothing \vdash \text{enum } A \left\{ \begin{array}{l} \text{case } B(\text{bool}) \\ \text{case } C(\text{num}) \end{array} \right\}; \; C(42) \text{ match } \left\{ \begin{array}{l} \text{case } B(b) \text{ => } b \\ \text{case } C(n) \text{ => } n < 0 \end{array} \right\} \Rightarrow \text{false}}}$$

where

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

Example 2   **PLRG**

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 **TAFAE** because **algebraic data types** are
recursive types:

```
/* TAFAE */
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 TAFAE:

```
/* TAFAE */
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 TAFAE:

```
/* TAFAE */
enum List:
  case Nil
  case Cons(head: Number, tail: List)
Cons(1, Cons(2, Cons(3, Nil)))            // (1, 2, 3)
```

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

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

Summary

1. Algebraic Data Types (ADTs) and Pattern Matching
    Recall: Types
    Product Types
    Union Types
    Sum Types
    Algebraic Data Types (ADTs)
    Pattern Matching

2. TAFAE – TRFAE with ADTs and Pattern Matching
    Concrete Syntax
    Abstract Syntax

3. Interpreter and Reduction Semantics for TAFAE
    Algebraic Data Types
    Function Application
    Pattern Matching
    Examples

- Algebraic Data Types (2)

Jihyeok Park
jihyeok_park@korea.ac.kr
https://plrg.korea.ac.kr