Lecture 22 – Algebraic Data Types (2)

COSE212: Programming Languages

Jihyeok Park

PLRG

2023 Fall

Recall

- A way to define new types by combining existing types:
  - product type
  - union type
  - sum type (tagged union type)
  - **algebraic data type** (recursive sum type of product types)

- **ATFAE** – TRFAE with **ADTs** and **pattern matching**.
  - **Interpreter** and **Natural Semantics**
  - **Type Checker** and **Typing Rules**

Recall

- A way to define new types by combining existing types:
  - product type
  - union type
  - sum type (tagged union type)
  - **algebraic data type** (recursive sum type of product types)
- **ATFAE** – TRFAE with **ADTs** and **pattern matching**.
  - **Interpreter** and **Natural Semantics**
  - **Type Checker** and **Typing Rules**
- Minor changes in the previous lecture:
  - TAFAE to ATFAE
  - **type variables** to **type names**
- In this lecture, we will discuss on **Type Checker** and **Typing Rules**.

Recall: Natural Semantics of ATFAE

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

The natural semantics of ATFAE ignores all the types.

`Leaf` and `Node` are not types but **variant names**.

`Leaf` and `Node` are **constructors** that take lists of values and produce **variant values** by adding their variant names as tags.

A **pattern matching** expression takes a **variant value** and finds the first match case whose name is equal to the variant name of the value.

Contents

Contents

Type Checker and Typing Rules

Let's ❶ design **typing rules** of TRFAE to define when an expression is well-typed in the form of:

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

and ❷ implement a **type checker** in Scala according to typing rules:

```scala
def typeCheck(expr: Expr, tenv: TypeEnv): Type = ???
```

The type checker returns the **type** of e if it is well-typed, or rejects it and throws a **type error** otherwise.

Similar to TRFAE, we will keep track of the **variable types** using a **type environment** Γ as a mapping from variable names to their types.

$$\text{Type Environments} \qquad \Gamma \in \mathbb{X} \xrightarrow{\text{fin}} \mathbb{T} \quad (\texttt{TypeEnv})$$

```scala
type TypeEnv = Map[String, Type]
```

Type Environment for ADTs

However, we need additional information about newly defined types by ADTs in type environments!

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

$$\Gamma(t) = x_1(\tau_{1,1}, \ldots, \tau_{1,m_1}) + \ldots + x_n(\tau_{n,1}, \ldots, \tau_{n,m_n})$$

Type Environment for ADTs

However, we need additional information about newly defined types by ADTs in type environments!

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

$$\Gamma(t) = x_1(\tau_{1,1}, \ldots, \tau_{1,m_1}) + \ldots + x_n(\tau_{n,1}, \ldots, \tau_{n,m_n})$$

whose variants are **commutative**. For example,

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

Type Environment for ADTs

However, we need additional information about newly defined types by ADTs in type environments!

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

$$\Gamma(t) = x_1(\tau_{1,1}, \ldots, \tau_{1,m_1}) + \ldots + x_n(\tau_{n,1}, \ldots, \tau_{n,m_n})$$

whose variants are **commutative**. For example,

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

```scala
case class TypeEnv(
  vars: Map[String, Type] = Map(),
  tys: Map[String, Map[String, List[Type]]] = Map()) {
  def +(pair: (String, Type)): TypeEnv =
    TypeEnv(vars + pair, tys)
  def ++(pairs: Iterable[(String, Type)]): TypeEnv =
    TypeEnv(vars ++ pairs, tys)
  def addTy(tname: String, ws: Map[String, List[Type]]): TypeEnv =
    TypeEnv(vars, tys + (tname -> ws))
}
```

Type Environment for ADTs

For example, consider the following an ADT for binary trees:

```
/* ATFAE */
enum Tree {
  case Leaf(Number)
  case Node(Tree, Number, Tree)
} ...
```

Type Environment for ADTs

For example, consider the following an ADT for binary trees:

```
/* ATFAE */
enum Tree {
  case Leaf(Number)
  case Node(Tree, Number, Tree)
} ...
```

We can add the type information of the Tree ADT to an existing type environment Γ (or tenv) as follows:

$$\Gamma[\mathtt{Tree} = \mathtt{Leaf}(\mathtt{num}) + \mathtt{Node}(\mathtt{Tree}, \mathtt{num}, \mathtt{Tree})]$$

```
val newTEnv = tenv.addTy(NameT("Tree"), Map(
  "Leaf" -> List(NumT),
  "Node" -> List(NameT("Tree"), NumT, NameT("Tree"))
))
```

Well-Formedness of Types

```
/* ATFAE */
enum Tree {
  case Leaf(Number)
  case Node(Tree, Number, Tree)
}
def f(t: Tree): Int = t
...
```

It is a well-typed ATFAE expression.

Well-Formedness of Types

```
/* ATFAE */
enum Tree {
  case Leaf(Number)
  case Node(Tree, Number, Tree)
}
def f(t: Tree): Int = t
...
```

It is a well-typed ATFAE expression.

```
/* ATFAE */
def f(t: Tree): Int = t
...
```

How about this?

Well-Formedness of Types

```
/* ATFAE */
enum Tree {
  case Leaf(Number)
  case Node(Tree, Number, Tree)
}
def f(t: Tree): Int = t
...
```

It is a well-typed ATFAE expression.

```
/* ATFAE */
def f(t: Tree): Int = t
...
```

How about this? **No!**

It is **syntactically correct** but the `Tree` type is **not defined**.

Well-Formedness of Types

```
/* ATFAE */
enum Tree {
  case Leaf(Number)
  case Node(Tree, Number, Tree)
}
def f(t: Tree): Int = t
...
```

It is a well-typed ATFAE expression.

```
/* ATFAE */
def f(t: Tree): Int = t
...
```

How about this? **No!**

It is **syntactically correct** but the `Tree` type is **not defined**.

We need to check the **well-formedness** of types with **type environment**.

Well-Formedness of Types

We need to check the **well-formedness** of types with **type environment**:

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

$$\frac{}{\Gamma \vdash \texttt{num}} \qquad \frac{}{\Gamma \vdash \texttt{bool}} \qquad \frac{\Gamma \vdash \tau_1 \quad \dots \quad \Gamma \vdash \tau_n \qquad \Gamma \vdash \tau}{\Gamma \vdash (\tau_1, \dots, \tau_n) \to \tau}$$

$$\frac{\Gamma(t) = x_1(\tau_{1,1}, \dots, \tau_{1,m_1}) + \dots + x_n(\tau_{n,1}, \dots, \tau_{n,m_n})}{\Gamma \vdash t}$$

```
def mustValid(ty: Type, tenv: TypeEnv): Type = ty match
  case NumT => NumT
  case BoolT => BoolT
  case ArrowT(ptys, rty) =>
    ArrowT(ptys.map(mustValid(_, tenv)), mustValid(rty, tenv))
  case NameT(x) =>
    if (!tenv.tys.contains(x)) error(s"free type variable: $x")
    NameT(x)
```

Function Definition

```scala
def typeCheck(expr: Expr, tenv: TypeEnv): Type = expr match
  ...
  case Fun(params, body) =>
    val ptys = params.map(_.ty)
    for (pty <- ptys) mustValid(pty, tenv)
    val rty = typeCheck(body, tenv ++ params.map(p => p.name -> p.ty))
    ArrowT(ptys, rty)
```

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

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

We need to check the **well-formedness** of parameter types.

Recursive Function Definition

```
def typeCheck(expr: Expr, tenv: TypeEnv): Type = expr match
  ...
  case Rec(f, params, rty, body, scope) =>
    val ptys = params.map(_.ty)
    for (pty <- ptys) mustValid(pty, tenv)
    mustValid(rty, tenv)
    val fty = ArrowT(ptys, rty)
    val newTEnv = tenv + (f -> fty) ++ params.map(p => p.name -> p.ty)
    typeCheck(scope, tenv + (f -> fty))
```

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

$$\tau\texttt{-Rec}\ \frac{\begin{array}{c}\Gamma \vdash \tau_1 \quad \ldots \quad \Gamma \vdash \tau_n \qquad \Gamma \vdash \tau \\ \Gamma[x_0 : (\tau_1, \ldots, \tau_n) \to \tau, x_1 : \tau_1, \ldots, x_n : \tau_n] \vdash e : \tau \\ \Gamma[x_0 : (\tau_1, \ldots, \tau_n) \to \tau] \vdash e' : \tau'\end{array}}{\Gamma \vdash \texttt{def}\ x_0(x_1 : \tau_1, \ldots, x_n : \tau_n) : \tau = e;\ e' : \tau'}$$

We need to check the **well-formedness** of parameter and return types.

Function Application

```scala
def typeCheck(expr: Expr, tenv: TypeEnv): Type = expr match
  ...
  case App(fun, args) => typeCheck(fun, tenv) match
    case ArrowT(ptys, retTy) =>
      if (ptys.length != args.length) error("arity mismatch")
      (ptys zip args).map((p, a) => mustSame(typeCheck(a, tenv), p))
      retTy
    case ty => error(s"not a function type: ${ty.str}")
```

$$\tau\text{-App}\ \frac{\Gamma \vdash e_0 : (\tau_1, \ldots, \tau_n) \to \tau \qquad \Gamma \vdash e_1 : \tau_1 \qquad \ldots \qquad \Gamma \vdash e_n : \tau_n}{\Gamma \vdash e_0(e_1, \ldots, e_n) : \tau}$$

No change in the type checking for **function application**.

Algebraic Data Types

```scala
def typeCheck(expr: Expr, tenv: TypeEnv): Type = expr match
  ...
  case TypeDef(tname, ws, body) =>
    val newTEnv = tenv.addTy(tname, ws.map(w => w.name -> w.ptys).toMap)
    for (w <- ws; pty <- w.ptys) mustValid(pty, newTEnv)
    typeCheck(
      body,
      newTEnv ++ ws.map(w => w.name -> ArrowT(w.ptys, NameT(tname)))
    )
```

$$\Gamma' = \Gamma[t = x_1(\tau_{1,1}, \ldots, \tau_{1,m_1}) + \ldots + x_n(\tau_{n,1}, \ldots, \tau_{n,m_n})]$$

$$\tau\text{--TypeDef}\ \frac{\Gamma' \vdash \tau_{1,1} \quad \ldots \quad \Gamma' \vdash \tau_{n,m_n} \qquad \Gamma' \left[\begin{array}{l} x_1 : (\tau_{1,1}, \ldots, \tau_{1,m_1}) \to t, \\ \ldots, \\ x_n : (\tau_{n,1}, \ldots, \tau_{n,m_n}) \to t \end{array} \right] \vdash e : \tau}{\Gamma \vdash \text{enum } t \left\{ \begin{array}{l} \text{case } x_1(\tau_{1,1}, \ldots, \tau_{1,m_1}) \\ \ldots \\ \text{case } x_n(\tau_{n,1}, \ldots, \tau_{n,m_n}) \end{array} \right\} ;\ e : \tau}$$

Algebraic Data Types

```
def typeCheck(expr: Expr, tenv: TypeEnv): Type = expr match
  ...
  case TypeDef(tname, ws, body) =>
    val newTEnv = tenv.addTy(tname, ws.map(w => w.name -> w.ptys).toMap)
    for (w <- ws; pty <- w.ptys) mustValid(pty, newTEnv)
    typeCheck(
      body,
      newTEnv ++ ws.map(w => w.name -> ArrowT(w.ptys, NameT(tname)))
    )
```

$$\tau\text{-}\texttt{TypeDef}\ \frac{\begin{array}{c}\Gamma' = \Gamma[t = x_1(\tau_{1,1}, \dots, \tau_{1,m_1}) + \dots + x_n(\tau_{n,1}, \dots, \tau_{n,m_n})] \\ \Gamma' \vdash \tau_{1,1} \quad \dots \quad \Gamma' \vdash \tau_{n,m_n} \qquad \Gamma' \left[\begin{array}{l} x_1 : (\tau_{1,1}, \dots, \tau_{1,m_1}) \to t, \\ \dots, \\ x_n : (\tau_{n,1}, \dots, \tau_{n,m_n}) \to t \end{array}\right] \vdash e : \tau\end{array}}{\Gamma \vdash \texttt{enum}\ t \left\{\begin{array}{l} \texttt{case}\ x_1(\tau_{1,1}, \dots, \tau_{1,m_1}) \\ \dots \\ \texttt{case}\ x_n(\tau_{n,1}, \dots, \tau_{n,m_n}) \end{array}\right\};\ e : \tau}$$

It is indeed **type unsound**, and we will fix it later.

Pattern Matching

```
def typeCheck(expr: Expr, tenv: TypeEnv): Type = expr match
  ...
  case Match(expr, cs) => typeCheck(expr, tenv) match
    case NameT(tname) =>
      val ts = tenv.tys.getOrElse(tname, error(s"unknown type: $tname"))
      val xs = cs.map(_.name).toSet
      if (ts.keySet != xs || xs.size != cs.length) error("invalid case")
      cs.map {
        case MatchCase(x, ps, b) => typeCheck(b, tenv ++ (ps zip ts(x)))
      }.reduce((lty, rty) => { mustSame(lty, rty); lty })
    case _ => error("not a variant")
```

$$\tau\texttt{-Match}\ \frac{\Gamma \vdash e : t \qquad \Gamma(t) = x_1(\tau_{1,1}, \ldots, \tau_{1,m_1}) + \ldots + x_n(\tau_{n,1}, \ldots, \tau_{n,m_n}) \\ \forall 1 \leq i \leq n.\ \Gamma_i = \Gamma[x_{i,1} : \tau_{i,1}, \ldots, x_{i,m_i} : \tau_{i,m_i}] \qquad \Gamma_1 \vdash e_1 : \tau \quad \ldots \quad \Gamma_n \vdash e_n : \tau}{\Gamma \vdash e\ \texttt{match} \left\{ \begin{array}{l} \texttt{case}\ x_1(x_{1,1}, \ldots, x_{1,m_1})\ \texttt{=>}\ e_1 \\ \ldots \\ \texttt{case}\ x_n(x_{n,1}, \ldots, x_{n,m_n})\ \texttt{=>}\ e_n \end{array} \right\} : \tau}$$

Contents

Recall: Type Soundness

Definition (Type Soundness)

A **type system** is **sound** if it guarantees that a **well-typed** program will **never** cause a **type error** at run-time.

Recall: Type Soundness

Definition (Type Soundness)

A **type system** is **sound** if it guarantees that a **well-typed** program will **never** cause a **type error** at run-time.

Consider the following ATFAE expression:

```
/* ATFAE */
enum A { case X(Number) }                          // X: Number => A
val f = (a: A) = a match { case X(n) => n }  // f: A => Number
enum A { case X(Boolean) }                         // X: Boolean => A
f(X(true)) + 1                                      // TypeError: true + 1
```

Recall: Type Soundness

Definition (Type Soundness)

A **type system** is **sound** if it guarantees that a **well-typed** program will **never** cause a **type error** at run-time.

Consider the following ATFAE expression:

```
/* ATFAE */
enum A { case X(Number) }                        // X: Number => A
val f = (a: A) = a match { case X(n) => n }      // f: A => Number
enum A { case X(Boolean) }                       // X: Boolean => A
f(X(true)) + 1                                   // TypeError: true + 1
```

It throws a type error when evaluating `true + 1` at run-time while this expression is **well-typed** according to the type system.

Recall: Type Soundness

PLRG

Definition (Type Soundness)

A **type system** is **sound** if it guarantees that a **well-typed** program will **never** cause a **type error** at run-time.

Consider the following ATFAE expression:

```
/* ATFAE */
enum A { case X(Number) }                         // X: Number => A
val f = (a: A) = a match { case X(n) => n }       // f: A => Number
enum A { case X(Boolean) }                        // X: Boolean => A
f(X(true)) + 1                                    // TypeError: true + 1
```

It throws a type error when evaluating `true + 1` at run-time while this expression is **well-typed** according to the type system.

It happens because the **same type name** `A` is defined twice and **shadows** the previous one with **different types** for its **variants**.

Recall: Type Soundness

Definition (Type Soundness)

A **type system** is **sound** if it guarantees that a **well-typed** program will **never** cause a **type error** at run-time.

Consider the following ATFAE expression:

```
/* ATFAE */
enum A { case X(Number) }                         // X: Number => A
val f = (a: A) = a match { case X(n) => n }       // f: A => Number
enum A { case X(Boolean) }                        // X: Boolean => A
f(X(true)) + 1                                    // TypeError: true + 1
```

It throws a type error when evaluating `true + 1` at run-time while this expression is **well-typed** according to the type system.

It happens because the **same type name** `A` is defined twice and **shadows** the previous one with **different types** for its **variants**.

Let's **forbid** the redefinition of **same type name** in the scope of **ADTs**!

Algebraic Data Types - Revised (1)

```scala
def typeCheck(expr: Expr, tenv: TypeEnv): Type = expr match
  ...
  case TypeDef(tname, ws, body) =>
    if (tenv.tys.contains(tname)) error(s"already defined type: $tname")
    val newTEnv = tenv.addTy(tname, ws.map(w => w.name -> w.ptys).toMap)
    for (w <- ws; pty <- w.ptys) mustValid(pty, newTEnv)
    typeCheck(
      body,
      newTEnv ++ ws.map(w => w.name -> ArrowT(w.ptys, NameT(tname)))
    )
```

$$\tau\text{–TypeDef}\ \frac{\begin{array}{c}\Gamma' = \Gamma[t = x_1(\tau_{1,1},\ldots,\tau_{1,m_1}) + \ldots + x_n(\tau_{n,1},\ldots,\tau_{n,m_n})] \\ t \notin \mathsf{Domain}(\Gamma) \qquad \Gamma' \vdash \tau_{1,1} \quad \ldots \quad \Gamma' \vdash \tau_{n,m_n} \\ \Gamma'\left[\begin{array}{l} x_1 : (\tau_{1,1},\ldots,\tau_{1,m_1}) \to t, \\ \ldots, \\ x_n : (\tau_{n,1},\ldots,\tau_{n,m_n}) \to t \end{array}\right] \vdash e : \tau\end{array}}{\Gamma \vdash \mathsf{enum}\ t \left\{\begin{array}{l} \mathsf{case}\ x_1(\tau_{1,1},\ldots,\tau_{1,m_1}) \\ \ldots \\ \mathsf{case}\ x_n(\tau_{n,1},\ldots,\tau_{n,m_n}) \end{array}\right\};\ e : \tau}$$

Algebraic Data Types - Revised (1)

Now, consider the following another ATFAE expression:

```
/* ATFAE */
val f = {
  enum A { case X(Number) }                               // X: Number => A
  (a: A) => a match { case X(n) => n }
}                                                         // f: A => Number
enum A { case X(Boolean) }                                // X: Boolean => A
f(X(true)) + 1                                            // TypeError: true + 1
```

Algebraic Data Types - Revised (1)

Now, consider the following another ATFAE expression:

```
/* ATFAE */
val f = {
  enum A { case X(Number) }                            // X: Number => A
  (a: A) => a match { case X(n) => n }
}                                                      // f: A => Number
enum A { case X(Boolean) }                             // X: Boolean => A
f(X(true)) + 1                                         // TypeError: true + 1
```

It throws a type error when evaluating `true + 1` at run-time while this expression is **well-typed** according to the type system.

Algebraic Data Types - Revised (1)

Now, consider the following another ATFAE expression:

```
/* ATFAE */
val f = {
  enum A { case X(Number) }                               // X: Number => A
  (a: A) => a match { case X(n) => n }
}                                                          // f: A => Number
enum A { case X(Boolean) }                                 // X: Boolean => A
f(X(true)) + 1                                             // TypeError: true + 1
```

It throws a type error when evaluating `true + 1` at run-time while this expression is **well-typed** according to the type system.

Since the second `A` type does not shadow the first one, the type system allows the definition of the second `A` type.

Algebraic Data Types - Revised (1)

Now, consider the following another ATFAE expression:

```
/* ATFAE */
val f = {
  enum A { case X(Number) }                          // X: Number => A
  (a: A) => a match { case X(n) => n }
}                                                    // f: A => Number
enum A { case X(Boolean) }                           // X: Boolean => A
f(X(true)) + 1                                       // TypeError: true + 1
```

It throws a type error when evaluating `true + 1` at run-time while this expression is **well-typed** according to the type system.

Since the second `A` type does not shadow the first one, the type system allows the definition of the second `A` type.

It happens because the first `A` type escapes its scope and is still visible in the scope of the second `A` type.

Algebraic Data Types - Revised (1)

Now, consider the following another ATFAE expression:

```
/* ATFAE */
val f = {
  enum A { case X(Number) }                     // X: Number => A
  (a: A) => a match { case X(n) => n }
}                                               // f: A => Number
enum A { case X(Boolean) }                      // X: Boolean => A
f(X(true)) + 1                                  // TypeError: true + 1
```

It throws a type error when evaluating `true + 1` at run-time while this expression is **well-typed** according to the type system.

Since the second `A` type does not shadow the first one, the type system allows the definition of the second `A` type.

It happens because the first `A` type escapes its scope and is still visible in the scope of the second `A` type.

Let's **forbid** the escape of **ADTs** from their scope!

Algebraic Data Types - Revised (2)

```
def typeCheck(expr: Expr, tenv: TypeEnv): Type = expr match
  ...
  case TypeDef(tname, ws, body) =>
    if (tenv.tys.contains(tname)) error(s"already defined type: $tname")
    val newTEnv = tenv.addTy(tname, ws.map(w => w.name -> w.ptys).toMap)
    for (w <- ws; pty <- w.ptys) mustValid(pty, newTEnv)
    mustValid(typeCheck(
      body,
      newTEnv ++ ws.map(w => w.name -> ArrowT(w.ptys, NameT(tname)))
    ), tenv)
```

$$\tau\text{–}\texttt{TypeDef}\ \frac{\begin{array}{c}\Gamma' = \Gamma[t = x_1(\tau_{1,1}, \ldots, \tau_{1,m_1}) + \ldots + x_n(\tau_{n,1}, \ldots, \tau_{n,m_n})] \\ t \notin \mathrm{Domain}(\Gamma) \qquad \Gamma' \vdash \tau_{1,1} \quad \ldots \quad \Gamma' \vdash \tau_{n,m_n} \\ \Gamma'\left[\begin{array}{l} x_1 : (\tau_{1,1}, \ldots, \tau_{1,m_1}) \to t, \\ \ldots, \\ x_n : (\tau_{n,1}, \ldots, \tau_{n,m_n}) \to t \end{array}\right] \vdash e : \tau \qquad \boxed{\Gamma \vdash \tau}\end{array}}{\Gamma \vdash \text{enum } t \left\{\begin{array}{l} \text{case } x_1(\tau_{1,1}, \ldots, \tau_{1,m_1}) \\ \ldots \\ \text{case } x_n(\tau_{n,1}, \ldots, \tau_{n,m_n}) \end{array}\right\};\ e : \tau}$$

Exercise #12

- Please see this document[1] on GitHub.
  - Implement `typeCheck` function.
  - Implement `interp` function.
- It is just an exercise, and you **don't need to submit** anything.
- However, some exam questions might be related to this exercise.

[1] https://github.com/ku-plrg-classroom/docs/tree/main/cose212/atfae.

Summary

1. Type Checker and Typing Rules
   - Type Environment for ADTs
   - Well-Formedness of Types
   - (Recursive) Function Definition and Application
   - Algebraic Data Types
   - Pattern Matching

2. Type Soundness of ATFAE
   - Recall: Type Soundness
   - Algebraic Data Types - Revised (1)
   - Algebraic Data Types - Revised (2)

Next Lecture

- Parametric Polymorphism

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