

# Lecture 22 – Algebraic Data Types (2)

## COSE212: Programming Languages

Jihyeok Park



2024 Fall

- 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**

- 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**
- In this lecture, we will discuss on **Type Checker** and **Typing Rules**.

```
/* 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.

```
/* 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**.

```
/* 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.

```
/* 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.

## 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)



## 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)

Let's ❶ design **typing rules** of ATFAE 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:

```
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.

Let's ❶ design **typing rules** of ATFAE 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:

```
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**  $\Gamma$  as a mapping from variable names to their types.

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

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

However, we need additional information in type environments about new types defined by **algebraic data types** (ADTs).

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

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

However, we need additional information in type environments about new types defined by **algebraic data types** (ADTs).

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

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

and sum types are **commutative**:

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

However, we need additional information in type environments about new types defined by **algebraic data types** (ADTs).

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

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

and sum types are **commutative**:

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

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

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

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

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  $\Gamma$  (or `tenv`) as follows:

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

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



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

It is a well-typed ATFAE expression.

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

It is a well-typed ATFAE expression.

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

How about this?

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

It is a well-typed ATFAE expression.

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

How about this? **No!**

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

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

It is a well-typed ATFAE expression.

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

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

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

$$\overline{\Gamma \vdash \text{num}}$$

$$\overline{\Gamma \vdash \text{bool}}$$

$$\frac{\Gamma \vdash \tau_1 \quad \dots \quad \Gamma \vdash \tau_n \quad \Gamma \vdash \tau}{\Gamma \vdash (\tau_1, \dots, \tau_n) \rightarrow \tau}$$

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

```
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(tn) =>
    if (!tenv.tys.contains(tn)) error(s"invalid type name: $tn")
    NameT(tn)
```

```
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.addVars(params.map(p => p.name -> p.ty)))
  ArrowT(ptys, rty)
```

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

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

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

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

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

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

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

```
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}")
```

$$\tau\text{-App} \frac{\Gamma \vdash e_0 : (\tau_1, \dots, \tau_n) \rightarrow \tau \quad \Gamma \vdash e_1 : \tau_1 \quad \dots \quad \Gamma \vdash e_n : \tau_n}{\Gamma \vdash e_0(e_1, \dots, e_n) : \tau}$$

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



```
def typeCheck(expr: Expr, tenv: TypeEnv): Type = expr match
  ...
  case TypeDef(tn, ws, body) =>
    ???
```

$$\tau\text{-TypeDef} \frac{\text{???}}{\Gamma \vdash \text{enum } t \left\{ \begin{array}{l} \text{case } x_1(\tau_{1,1}, \dots, \tau_{1,m_1}) \\ \dots \\ \text{case } x_n(\tau_{n,1}, \dots, \tau_{n,m_n}) \end{array} \right\}; e : \text{???}}$$

```
def typeCheck(expr: Expr, tenv: TypeEnv): Type = expr match
...
case TypeDef(tn, ws, body) =>
  val newTEEnv = tenv.addType(tn, ws.map(w => w.name -> w.ptys).toMap)
  ???
```

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

First, we need to add the **type information** of the new ADT whose type name is  $t$  and its variants to the type environment.

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

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

Then, we need to check the **well-formedness** of the parameter types of variants of the new ADT.

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

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

Then, we need to check the **well-formedness** of the parameter types of variants of the new ADT.

Note that we use  $\Gamma'$  instead of  $\Gamma$  in the well-formedness check to support the **recursive** use of the type name  $t$  in the parameter types.

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

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

Finally, we need to check the type of the **body** expression with the extended type environment with the types of **constructors**  $x_1, \dots, x_n$ .

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

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

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

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

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

```
def typeCheck(expr: Expr, tenv: TypeEnv): Type = expr match
  ...
  case Match(expr, cs) => ???
```

$$\tau\text{-Match} \frac{\text{???}}{\Gamma \vdash e \text{ match } \left\{ \begin{array}{l} \text{case } x_1(x_{1,1}, \dots, x_{1,m_1}) \Rightarrow e_1 \\ \dots \\ \text{case } x_n(x_{n,1}, \dots, x_{n,m_n}) \Rightarrow e_n \end{array} \right\} : \text{???}}$$



```
def typeCheck(expr: Expr, tenv: TypeEnv): Type = expr match
...
case Match(expr, cs) => typeCheck(expr, tenv) match
  case NameT(t) =>
    ???
  case _ => error("not a variant")
```

$$\tau\text{-Match} \frac{\Gamma \vdash e : t \quad ???}{\Gamma \vdash e \text{ match } \left\{ \begin{array}{l} \text{case } x_1(x_{1,1}, \dots, x_{1,m_1}) \Rightarrow e_1 \\ \dots \\ \text{case } x_n(x_{n,1}, \dots, x_{n,m_n}) \Rightarrow e_n \end{array} \right\} : ???}$$

First, we need to check the type of the **matched expression**  $e$  and ensure that it is a **type name**.

```
def typeCheck(expr: Expr, tenv: TypeEnv): Type = expr match
...
case Match(expr, cs) => typeCheck(expr, tenv) match
  case NameT(t) =>
    val tmap = tenv.tys.getOrElse(t, error(s"unknown type: $t"))
    mustValidMatch(t, cs, tmap)
    ???
  case _ => error("not a variant")
```

$$\tau\text{-Match} \frac{\Gamma \vdash e : t \quad \Gamma(t) = x_1(\tau_{1,1}, \dots, \tau_{1,m_1}) + \dots + x_n(\tau_{n,1}, \dots, \tau_{n,m_n})}{\Gamma \vdash e \text{ match } \left\{ \begin{array}{l} \text{case } x_1(x_{1,1}, \dots, x_{1,m_1}) \Rightarrow e_1 \\ \dots \\ \text{case } x_n(x_{n,1}, \dots, x_{n,m_n}) \Rightarrow e_n \end{array} \right\} : ???} \quad ???$$

Then, we need to 1) look up the **type information** of the type name  $t$  in the type environment  $\Gamma$  and 2) check the **validity** of the match cases.

The following Scala code is an implementation of the `mustValidMatch` function that checks the validity of the match cases:

```
def mustValidMatch(
  t: String,
  cs: List[MatchCase],
  tmap: Map[String, List[Type]],
): Unit =
  val xs = cs.map(_.name)
  val ys = tmap.keySet
  for (x <- xs if xs.count(_ == x) > 1) error(s"duplicate case $x for $t")
  for (x <- xs if !ys.contains(x))      error(s"unknown case $x for $t")
  for (y <- ys if !xs.contains(y))      error(s"inexhaustive case $y for $t")
  for {
    MatchCase(x, ps, _) <- cs
    n = tmap(x).size
    m = ps.size
    if n != m
  } error(s"arity mismatch ($n != $m) in case $x for $t")
```

```
def typeCheck(expr: Expr, tenv: TypeEnv): Type = expr match
...
case Match(expr, cs) => typeCheck(expr, tenv) match
  case NameT(t) =>
    val tmap = tenv.tys.getOrElse(t, error(s"unknown type: $t"))
    mustValidMatch(t, cs, tmap)
    val tys = for (MatchCase(x, ps, b) <- cs)
      yield typeCheck(b, tenv.addVars((ps zip tmap(x))))
    ???
  case _ => error("not a variant")
```

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

Now, we need to check the type of the **body** expressions  $e_i$  with the type environment  $\Gamma_i$  extended with the parameter types of the match cases.

```
def typeCheck(expr: Expr, tenv: TypeEnv): Type = expr match
...
case Match(expr, cs) => typeCheck(expr, tenv) match
  case NameT(t) =>
    val tmap = tenv.tys.getOrElse(t, error(s"unknown type: $t"))
    mustValidMatch(t, cs, tmap)
    val tys = for (MatchCase(x, ps, b) <- cs)
      yield typeCheck(b, tenv.addVars((ps zip tmap(x))))
    tys.reduce((lty, rty) => { mustSame(lty, rty); lty })
  case _ => error("not a variant")
```

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

Finally, all the **body** expressions  $e_i$  should have the **same type**  $\tau$ , which is the type of the whole match expression.

## 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)

## 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.

## 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                       // Number
```



## 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                      // Number
```

It throws a **type error** when evaluating `true + 1` at run-time while this expression is **well-typed** (i.e., **unsound type system**).

## 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                      // Number
```

It throws a **type error** when evaluating `true + 1` at run-time while this expression is **well-typed** (i.e., **unsound type system**).

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

## 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                      // Number
```

It throws a **type error** when evaluating `true + 1` at run-time while this expression is **well-typed** (i.e., **unsound 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**!

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

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

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 }  
}  
enum A { case X(Boolean) }           // f: A => Number  
f(X(true)) + 1                       // X: Boolean => A  
                                     // Number
```

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

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 }  
}  
enum A { case X(Boolean) }           // f: A => Number  
f(X(true)) + 1                       // X: Boolean => A  
                                     // Number
```

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

Unfortunately, it throws a **type error** when evaluating `true + 1` at run-time while this expression is **well-typed** (i.e., **unsound type system**).

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                       // Number
```

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

Unfortunately, it throws a **type error** when evaluating `true + 1` at run-time while this expression is **well-typed** (i.e., **unsound type system**).

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

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 }  
}  
enum A { case X(Boolean) }             // f: A => Number  
f(X(true)) + 1                         // X: Boolean => A  
                                        // Number
```

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

Unfortunately, it throws a **type error** when evaluating `true + 1` at run-time while this expression is **well-typed** (i.e., **unsound type system**).

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!



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

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

## 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)

<https://github.com/ku-plrg-classroom/docs/tree/main/cose212/atfae>

- Please see above document 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.

- Parametric Polymorphism

Jihyeok Park

[jihyeok\\_park@korea.ac.kr](mailto:jihyeok_park@korea.ac.kr)

<https://plrg.korea.ac.kr>