Lecture 20 – Typing Recursive Functions

COSE212: Programming Languages

Jihyeok Park

PLRG

2023 Fall

Recall

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

Recall

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

- Let's learn how to apply **type system** to recursive functions.

Recall

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

- Let's learn how to apply **type system** to recursive functions.

- RFAE is an extension of FAE with
  1. **recursive functions**
  2. **conditional expressions**

Recall

- **TFAE** – FAE with **type system**.
  - **Type Checker** and **Typing Rules**
  - Interpreter and Natural Semantics
- Let's learn how to apply **type system** to recursive functions.
- RFAE is an extension of FAE with
  1. **recursive functions**
  2. **conditional expressions**
- TRFAE – RFAE with **type system**.
  - **Type Checker** and **Typing Rules**
  - Interpreter and Natural Semantics

Contents

1. Types for Recursive Functions
   - Recall: `mkRec` and Recursive Functions
   - `mkRec` in TFAE
2. TRFAE – RFAE with Type System
   - Concrete Syntax
   - Abstract Syntax
3. Type Checker and Typing Rules
   - Arithmetic Comparison Operators
   - Conditionals
   - Recursive Function Definitions

Contents

1. Types for Recursive Functions
   - Recall: `mkRec` and Recursive Functions
   - `mkRec` in TFAE
2. TRFAE – RFAE with Type System
   - Concrete Syntax
   - Abstract Syntax
3. Type Checker and Typing Rules
   - Arithmetic Comparison Operators
   - Conditionals
   - Recursive Function Definitions

Recall: `mkRec` and Recursive Functions

A **recursive function** is a function that calls itself, and it is useful for **iterative processes** on **inductive data structures**.

Recall: `mkRec` and Recursive Functions

A **recursive function** is a function that calls itself, and it is useful for **iterative processes** on **inductive data structures**.

Let's define a **recursive function** `sum` that computes the sum of integers from 1 to n in Scala:

```
def sum(n: Int): Int =
  if (n < 1) 0          // base case
  else n + sum(n - 1)   // recursive case

sum(10) // 10 + 9 + 8 + 7 + 6 + 5 + 4 + 3 + 2 + 1 + 0 = 55
```

Recall: `mkRec` and Recursive Functions

Recall: `mkRec` and Recursive Functions

We learned two ways to support recursion functions:

Recall: mkRec and Recursive Functions

We learned two ways to support recursion functions:

1. by introducing a **helper function** called mkRec in FAE as follows:

```
/* FAE */
val mkRec = body => {
  val fX = fY => {
    val f = x => fY(fY)(x);
    body(f)
  };
  fX(fX)
};
val sum = mkRec(sum => n => if (n < 1) 0 else n + sum(n + -1)); sum(10)
```

Recall: mkRec and Recursive Functions

We learned two ways to support recursion functions:

1. by introducing a **helper function** called `mkRec` in FAE as follows:

```
/* FAE */
val mkRec = body => {
  val fX = fY => {
    val f = x => fY(fY)(x);
    body(f)
  };
  fX(fX)
};
val sum = mkRec(sum => n => if (n < 1) 0 else n + sum(n + -1)); sum(10)
```

or 2. by adding **new syntax** for recursive functions in RFAE:

```
/* RFAE */
def sum(n) = if (n < 1) 0 else n + sum(n + -1); sum(10)
```

Recall: mkRec and Recursive Functions

We learned two ways to support recursion functions:

1. by introducing a **helper function** called `mkRec` in FAE as follows:

```
/* FAE */
val mkRec = body => {
  val fX = fY => {
    val f = x => fY(fY)(x);
    body(f)
  };
  fX(fX)
};
val sum = mkRec(sum => n => if (n < 1) 0 else n + sum(n + -1)); sum(10)
```

or 2. by adding **new syntax** for recursive functions in RFAE:

```
/* RFAE */
def sum(n) = if (n < 1) 0 else n + sum(n + -1); sum(10)
```

Can we define `mkRec` in TFAE?

Recall: mkRec and Recursive Functions

We learned two ways to support recursion functions:

1. by introducing a **helper function** called mkRec in FAE as follows:

```
/* FAE */
val mkRec = body => {
  val fX = fY => {
    val f = x => fY(fY)(x);
    body(f)
  };
  fX(fX)
};
val sum = mkRec(sum => n => if (n < 1) 0 else n + sum(n + -1)); sum(10)
```

or 2. by adding **new syntax** for recursive functions in RFAE:

```
/* RFAE */
def sum(n) = if (n < 1) 0 else n + sum(n + -1); sum(10)
```

Can we define mkRec in TFAE? **No!** Let's see why.

mkRec in TFAE

```
/* TFAE */
val mkRec = (body: ???) => {
  val fX = (fY: ???) => {
    val f = (x: ???) => fY(fY)(x);
    body(f)
  };
  fX(fX)
};
val sum = mkRec((sum: ???) => (n: ???) =>
  if (n < 1) 0
  else n + sum(n + -1));
sum(10)
```

Let's fill out the parts of ??? for type annotations one by one.

mkRec in TFAE

```
/* TFAE */
val mkRec = (body: ???) => {
  val fX = (fY: ???) => {
    val f = (x: ???) => fY(fY)(x);
    body(f)
  };
  fX(fX)
};
val sum = mkRec((sum: ???) => (n: Number) =>
  if (n < 1) 0
  else n + sum(n + -1));
sum(10)
```

Let's fill out the parts of ??? for type annotations one by one.

mkRec in TFAE

```
/* TFAE */
val mkRec = (body: ???) => {
  val fX = (fY: ???) => {
    val f = (x: ???) => 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)
```

Let's fill out the parts of ??? for type annotations one by one.

mkRec in TFAE

```
/* TFAE */
val mkRec = (body: (Number => Number) => Number => Number) => {
  val fX = (fY: ???) => {
    val f = (x: ???) => 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)
```

Let's fill out the parts of ??? for type annotations one by one.

mkRec in TFAE

```
/* TFAE */
val mkRec = (body: (Number => Number) => Number => Number) => {
  val fX = (fY: ???) => {
    val f = (x: ???) => fY(fY)(x);
    body(f)                             // f: Number => Number
  };
  fX(fX)
};
val sum = mkRec((sum: Number => Number) => (n: Number) =>
  if (n < 1) 0
  else n + sum(n + -1));
sum(10)
```

Let's fill out the parts of ??? for type annotations one by one.

mkRec in TFAE

```
/* TFAE */
val mkRec = (body: (Number => Number) => Number => Number) => {
  val fX = (fY: ???) => {
    val f = (x: Number) => fY(fY)(x);
    body(f)                                 // f: Number => Number
  };
  fX(fX)
};
val sum = mkRec((sum: Number => Number) => (n: Number) =>
  if (n < 1) 0
  else n + sum(n + -1));
sum(10)
```

Let's fill out the parts of ??? for type annotations one by one.

mkRec in TFAE

```
/* TFAE */
val mkRec = (body: (Number => Number) => Number => Number) => {
  val fX = (fY: ???) => {
    val f = (x: Number) => fY(fY)(x);   // fY(fY): Number => Number
    body(f)                             // f: Number => Number
  };
  fX(fX)
};
val sum = mkRec((sum: Number => Number) => (n: Number) =>
  if (n < 1) 0
  else n + sum(n + -1));
sum(10)
```

Let's fill out the parts of ??? for type annotations one by one.

mkRec in TFAE

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

Let's fill out the parts of ??? for type annotations one by one.

Let T be the type of fY.

mkRec in TFAE

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

Let's fill out the parts of ??? for type annotations one by one.

Let `T` be the type of `fY`.

Then, `T` should be equal to `T => Number => Number`.

mkRec in TFAE

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

Let's fill out the parts of ??? for type annotations one by one.

Let `T` be the type of `fY`.

Then, `T` should be equal to `T => Number => Number`.

mkRec in TFAE

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

Let's fill out the parts of ??? for type annotations one by one.

Let `T` be the type of `fY`.

Then, `T` should be equal to `T => Number => Number`.

mkRec in TFAE

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

Let's fill out the parts of ??? for type annotations one by one.

Let T be the type of fY.

Then, T should be equal to T => Number => Number.

mkRec in TFAE

PLRG

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

Let's fill out the parts of ??? for type annotations one by one.

Let `T` be the type of `fY`.

Then, `T` should be equal to `T => Number => Number`.

mkRec in TFAE

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

Let's fill out the parts of ??? for type annotations one by one.

Let T be the type of fY.

Then, T should be equal to T => Number => Number.

mkRec in TFAE

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

Let's fill out the parts of ??? for type annotations one by one.

Let `T` be the type of `fY`.

Then, `T` should be equal to `T => Number => Number`.

We cannot define such **recursive type** in TFAE.

mkRec in TFAE

Since Scala supports **recursive types**, we can define mkRec as follows:[1]

```
type Number = BigInt
case class T(self: T => Number => Number) // T = T => Number => Number
val mkRec = (body: (Number => Number) => Number => Number) => {
  val fX = (fY: T) => {
    val f = (x: Number) => fY.self(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)
```

[1]This code is given by 최민석 and 최용욱 and slightly modified. Thanks!

mkRec in TFAE

Since Scala supports **recursive types**, we can define mkRec as follows:[1]

```
import scala.language.implicitConversions
given Conversion[T, T => Number => Number] = _.self
given Conversion[T => Number => Number, T] = T(_)
type Number = BigInt
case class T(self: T => Number => Number) // T = T => Number => Number
val mkRec = (body: (Number => Number) => Number => Number) => {
  val fX = (fY: T) => {
    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)
```

[1]This code is given by 최민석 and 최용욱 and slightly modified. Thanks!

mkRec in TFAE

Since Scala supports **recursive types**, we can define mkRec as follows:[1]

```
import scala.language.implicitConversions
given Conversion[T, T => Number => Number] = _.self
given Conversion[T => Number => Number, T] = T(_)
type Number = BigInt
case class T(self: T => Number => Number) // T = T => Number => Number
val mkRec = (body: (Number => Number) => Number => Number) => {
  val fX = (fY: T) => {
    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)
```

However, we cannot do it in TFAE.

[1]This code is given by 최민석 and 최용욱 and slightly modified. Thanks!

mkRec in TFAE

Since Scala supports **recursive types**, we can define mkRec as follows:[1]

```
import scala.language.implicitConversions
given Conversion[T, T => Number => Number] = _.self
given Conversion[T => Number => Number, T] = T(_)
type Number = BigInt
case class T(self: T => Number => Number) // T = T => Number => Number
val mkRec = (body: (Number => Number) => Number => Number) => {
  val fX = (fY: T) => {
    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)
```

However, we cannot do it in TFAE. So, let's add **type system** to RFAE!

[1] This code is given by 최민석 and 최용욱 and slightly modified. Thanks!

Contents

1. Types for Recursive Functions
   - Recall: `mkRec` and Recursive Functions
   - `mkRec` in TFAE

2. TRFAE – RFAE with Type System
   - Concrete Syntax
   - Abstract Syntax

3. Type Checker and Typing Rules
   - Arithmetic Comparison Operators
   - Conditionals
   - Recursive Function Definitions

TRFAE – RFAE with Type System

Before defining TRFAE, guess the type of the following RFAE expressions:

```
/* RFAE */ def f(n) = n; f
```

TRFAE – RFAE with Type System

Before defining TRFAE, guess the type of the following RFAE expressions:

```
/* RFAE */ def f(n) = n; f
```

Without type annotation for parameter `n`, we cannot guess its type.

TRFAE – RFAE with Type System

Before defining TRFAE, guess the type of the following RFAE expressions:

```
/* RFAE */ def f(n) = n; f
```

Without type annotation for parameter `n`, we cannot guess its type.

```
/* RFAE */ def f(n: Number) = n; f
```

With type annotation for parameter `n`, we can guess its type.

TRFAE – RFAE with Type System

Before defining TRFAE, guess the type of the following RFAE expressions:

```
/* RFAE */ def f(n) = n; f
```

Without type annotation for parameter `n`, we cannot guess its type.

```
/* RFAE */ def f(n: Number) = n; f
```

With type annotation for parameter `n`, we can guess its type.

How about this?

```
/* RFAE */ def f(n: Number) = f(n); f
```

TRFAE – RFAE with Type System

Before defining TRFAE, guess the type of the following RFAE expressions:

```
/* RFAE */ def f(n) = n; f
```

Without type annotation for parameter `n`, we cannot guess its type.

```
/* RFAE */ def f(n: Number) = n; f
```

With type annotation for parameter `n`, we can guess its type.

How about this?

```
/* RFAE */ def f(n: Number) = f(n); f
```

Unfortunately, its return type is not clear and actually can be any type.

TRFAE – RFAE with Type System

PLRG

Before defining TRFAE, guess the type of the following RFAE expressions:

```
/* RFAE */ def f(n) = n; f
```

Without type annotation for parameter `n`, we cannot guess its type.

```
/* RFAE */ def f(n: Number) = n; f
```

With type annotation for parameter `n`, we can guess its type.

How about this?

```
/* RFAE */ def f(n: Number) = f(n); f
```

Unfortunately, its return type is not clear and actually can be any type.

So, we need **type annotation** for both parameters and return types.

```
/* RFAE */ def f(n: Number): Number = f(n); f
```

TRFAE – RFAE with Type System

Now, let's extend RFAE into TRFAE with **type system**.

```
/* TRFAE */
def sum(n: Number): Number = {
  if (n < 1) 0
  else n + sum(n + -1)
};
sum(10) // 55
```

```
/* TRFAE */
def fib(n: Number): Number = {
  if (n < 2) n
  else fib(n + -1) + fib(n + -2)
};
fib(7) // 13
```

TRFAE – RFAE with Type System

Now, let's extend RFAE into TRFAE with **type system**.

```
/* TRFAE */
def sum(n: Number): Number = {
  if (n < 1) 0
  else n + sum(n + -1)
};
sum(10) // 55
```

```
/* TRFAE */
def fib(n: Number): Number = {
  if (n < 2) n
  else fib(n + -1) + fib(n + -2)
};
fib(7) // 13
```

For TRFAE, we need to consider the **type system** of the following cases:

1. **arithmetic comparison operators**
2. **conditionals**
3. **recursive function definitions**

Concrete Syntax

PLRG

We need to add following concrete syntax from RFAE for TRFAE:

1. **type annotations** for **recursive function definitions**
2. **types** (number, boolean, and arrow types)

```
// expressions
<expr> ::= ...
         | <expr> "<" <expr>
         | "if" "(" <expr> ")" <expr> "else" <expr>
         | "def" <id> "(" <id> ":" <type> ")" ":" <type>
           "=" <expr> ";" <expr>

// types
<type> ::= "(" <type> ")"        // only for precedence
         | "Number"              // number type
         | "Boolean"             // boolean type
         | <type> "=>" <type>    // arrow type
```

Abstract Syntax

Similarly, we can define the **abstract syntax** of TRFAE as follows:

Expressions

$$\begin{array}{rll} \mathbb{E} \ni e ::= & \ldots & \\ & \mid e < e & (\texttt{Lt}) \\ & \mid \texttt{if}\ (e)\ e\ \texttt{else}\ e & (\texttt{If}) \\ & \mid \texttt{def}\ x(x\texttt{:}\tau)\texttt{:}\tau\texttt{=}e;\ e & (\texttt{Rec}) \end{array}$$

Types

$$\begin{array}{rll} \mathbb{T} \ni \tau ::= & \texttt{num} & (\texttt{NumT}) \\ & \mid \texttt{bool} & (\texttt{BoolT}) \\ & \mid \tau \to \tau & (\texttt{ArrowT}) \end{array}$$

Abstract Syntax

Similarly, we can define the **abstract syntax** of TRFAE as follows:

Expressions

$$
\begin{array}{rrll}
\mathbb{E} \ni e & ::= & \ldots & \\
& | & e < e & (\texttt{Lt}) \\
& | & \texttt{if}\ (e)\ e\ \texttt{else}\ e & (\texttt{If}) \\
& | & \texttt{def}\ x(x\texttt{:}\tau)\texttt{:}\tau\texttt{=}e;\ e & (\texttt{Rec})
\end{array}
$$

Types

$$
\begin{array}{rrll}
\mathbb{T} \ni \tau & ::= & \texttt{num} & (\texttt{NumT}) \\
& | & \texttt{bool} & (\texttt{BoolT}) \\
& | & \tau \to \tau & (\texttt{ArrowT})
\end{array}
$$

We can define the abstract syntax of TRFAE in Scala as follows:

```
enum Expr:
  ...
  case Lt(left: Expr, right: Expr)
  case If(cond: Expr, thenExpr: Expr, elseExpr: Expr)
  case Rec(x: String, p: String, pty: Type, rty: Type, b: Expr, s: Expr)
enum Type:
  case NumT
  case BoolT
  case ArrowT(paramTy: Type, retTy: Type)
```

Contents

1. Types for Recursive Functions
   - Recall: `mkRec` and Recursive Functions
   - `mkRec` in TFAE

2. TRFAE – RFAE with Type System
   - Concrete Syntax
   - Abstract Syntax

3. Type Checker and Typing Rules
   - Arithmetic Comparison Operators
   - Conditionals
   - Recursive Function Definitions

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.

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 TFAE, 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]
```

Arithmetic Comparison Operators

```
def typeCheck(expr: Expr, tenv: TypeEnv): Type = expr match
  ...
  case Lt(left, right) =>
    mustSame(typeCheck(left, tenv), NumT)
    mustSame(typeCheck(right, tenv), NumT)
    BoolT
```

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

$$\tau\text{–Lt}\ \frac{\Gamma \vdash e_1 : \texttt{num} \qquad \Gamma \vdash e_2 : \texttt{num}}{\Gamma \vdash e_1 < e_2 : \texttt{bool}}$$

Type checker should do

1. check the types of e_1 and e_2 are `num` in Γ
2. return `bool` as the type of $e_1 < e_2$

Conditionals

```scala
def typeCheck(expr: Expr, tenv: TypeEnv): Type = expr match
  ...
  case If(cond, thenExpr, elseExpr) => ???
```

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

$$\tau\text{–If}\ \frac{???}{\Gamma \vdash \texttt{if}\ (e_0)\ e_1\ \texttt{else}\ e_2 : ???}$$

Conditionals

```scala
def typeCheck(expr: Expr, tenv: TypeEnv): Type = expr match
  ...
  case If(cond, thenExpr, elseExpr) => ???
```

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

$$\tau\text{-If}\ \frac{???}{\Gamma \vdash \texttt{if}\ (e_0)\ e_1\ \texttt{else}\ e_2 : ???}$$

Let's think about the types of the following TRFAE expressions:

```
if (true) 1 else 2
```

Conditionals

```
def typeCheck(expr: Expr, tenv: TypeEnv): Type = expr match
  ...
  case If(cond, thenExpr, elseExpr) => ???
```

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

$$\tau\text{–If}\ \frac{???}{\Gamma \vdash \texttt{if}\ (e_0)\ e_1\ \texttt{else}\ e_2 : ???}$$

Let's think about the types of the following TRFAE expressions:

`if (true) 1 else 2` should be `Number`

Conditionals

```
def typeCheck(expr: Expr, tenv: TypeEnv): Type = expr match
  ...
  case If(cond, thenExpr, elseExpr) => ???
```

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

$$\tau\text{–If}\ \frac{???}{\Gamma \vdash \texttt{if}\ (e_0)\ e_1\ \texttt{else}\ e_2 : ???}$$

Let's think about the types of the following TRFAE expressions:

`if (true) 1 else 2` should be `Number`

`if (true) 1 else true`

Conditionals

PLRG

```
def typeCheck(expr: Expr, tenv: TypeEnv): Type = expr match
  ...
  case If(cond, thenExpr, elseExpr) => ???
```

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

$$\tau\text{–If}\ \frac{???}{\Gamma \vdash \texttt{if}\ (e_0)\ e_1\ \texttt{else}\ e_2 : ???}$$

Let's think about the types of the following TRFAE expressions:

`if (true) 1 else 2`	should be	`Number`
`if (true) 1 else true`	might be	`Number`?

Conditionals

```scala
def typeCheck(expr: Expr, tenv: TypeEnv): Type = expr match
  ...
  case If(cond, thenExpr, elseExpr) => ???
```

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

$$\tau\text{–If}\ \frac{???}{\Gamma \vdash \texttt{if}\ (e_0)\ e_1\ \texttt{else}\ e_2 : ???}$$

Let's think about the types of the following TRFAE expressions:

`if (true) 1 else 2`	should be	`Number`
`if (true) 1 else true`	might be	`Number`?
`(x: Boolean) => if (x) 1 else x`		

Conditionals

```
def typeCheck(expr: Expr, tenv: TypeEnv): Type = expr match
  ...
  case If(cond, thenExpr, elseExpr) => ???
```

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

$$\tau\text{–If}\ \frac{???}{\Gamma \vdash \texttt{if}\ (e_0)\ e_1\ \texttt{else}\ e_2 : ???}$$

Let's think about the types of the following TRFAE expressions:

`if (true) 1 else 2`	should be	`Number`
`if (true) 1 else true`	might be	`Number`?
`(x: Boolean) => if (x) 1 else x`	cannot have a type	

Conditionals

PLRG

```
def typeCheck(expr: Expr, tenv: TypeEnv): Type = expr match
  ...
  case If(cond, thenExpr, elseExpr) => ???
```

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

$$\tau\text{-If}\ \frac{???}{\Gamma \vdash \texttt{if}\ (e_0)\ e_1\ \texttt{else}\ e_2 : ???}$$

Let's think about the types of the following TRFAE expressions:

`if (true) 1 else 2`	should be	`Number`
`if (true) 1 else true`	might be	`Number`?
`(x: Boolean) => if (x) 1 else x`	cannot have a type	

Type checker cannot know the **actual value** of **condition expression**.

Conditionals

PLRG

```
def typeCheck(expr: Expr, tenv: TypeEnv): Type = expr match
  ...
  case If(cond, thenExpr, elseExpr) => ???
```

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

$$\tau\text{-If}\ \frac{???}{\Gamma \vdash \texttt{if}\ (e_0)\ e_1\ \texttt{else}\ e_2 : ???}$$

Let's think about the types of the following TRFAE:

`if (true) 1 else 2`	should be `Number`
`if (true) 1 else true`	REJECT
`(x: Boolean) => if (x) 1 else x`	REJECT

Type checker cannot know the **actual value** of **condition expression**.

Let's accept only if **both types** of then- and else-expressions are **same**.

Conditionals

PLRG

```
def typeCheck(expr: Expr, tenv: TypeEnv): Type = expr match
  ...
  case If(cond, thenExpr, elseExpr) =>
    mustSame(typeCheck(cond, tenv), BoolT)
    val thenTy = typeCheck(thenExpr, tenv)
    val elseTy = typeCheck(elseExpr, tenv)
    mustSame(thenTy, elseTy)
    thenTy
```

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

$$\tau\text{–If}\ \frac{\Gamma \vdash e_0 : \texttt{bool} \qquad \Gamma \vdash e_1 : \tau \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash \texttt{if}\ (e_0)\ e_1\ \texttt{else}\ e_2 : \tau}$$

Type checker should do

1. check the type of e_0 is $\texttt{bool}$ in Γ
2. check the types of e_1 and e_2 are equal in Γ
3. return the type of e_1 (or e_2)

Recursive Function Definitions

```
def interp(expr: Expr, env: Env): Value = expr match
  ...
  case Rec(f, p, pty, rty, body, scope) =>
    ???
```

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

$$\tau\text{–Rec}\ \frac{???}{\Gamma \vdash \texttt{def}\ x_0(x_1 : \tau_1) : \tau_2 = e_2;\ e_3 : ???}$$

Recursive Function Definitions

```
def interp(expr: Expr, env: Env): Value = expr match
  ...
  case Rec(f, p, pty, rty, body, scope) =>

    mustSame(typeCheck(body, ???), rty)
    ???
```

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

$$\tau\texttt{-Rec}\ \frac{???\vdash e_2 : \tau_2 \qquad ???}{\Gamma \vdash \texttt{def}\ x_0(x_1 : \tau_1) : \tau_2 = e_2;\ e_3 : ???}$$

Type checker should do

1. check the type of e_2 is τ_2 in ???
2. ???

Recursive Function Definitions

```
def interp(expr: Expr, env: Env): Value = expr match
  ...
  case Rec(f, p, pty, rty, body, scope) =>

    mustSame(typeCheck(body, tenv + (p -> pty)), rty)
    ???
```

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

$$\tau\text{−Rec}\ \frac{\Gamma[x_1 : \tau_1] \vdash e_2 : \tau_2 \qquad ???}{\Gamma \vdash \texttt{def}\ x_0(x_1 : \tau_1) : \tau_2 = e_2; e_3 : ???}$$

Type checker should do

1. check the type of e_2 is τ_2 in the type environment extended with type information for parameter $(x_1 : \tau_1)$
2. ???

Recursive Function Definitions

```
def interp(expr: Expr, env: Env): Value = expr match
  ...
  case Rec(f, p, pty, rty, body, scope) =>
    val fty = ArrowT(pty, rty)
    mustSame(typeCheck(body, tenv + (f -> fty) + (p -> pty)), rty)
    ???
```

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

$$\tau\text{-}\texttt{Rec}\ \frac{\Gamma[x_0 : \tau_1 \to \tau_2, x_1 : \tau_1] \vdash e_2 : \tau_2 \qquad ???}{\Gamma \vdash \texttt{def}\ x_0(x_1\!:\!\tau_1)\!:\!\tau_2\!=\!e_2;\ e_3 : ???}$$

Type checker should do

1. check the type of e_2 is τ_2 in the type environment extended with type information for function $(x_0 : \tau_1 \to \tau_2)$ and parameter $(x_1 : \tau_1)$
2. ???

Recursive Function Definitions

```
def interp(expr: Expr, env: Env): Value = expr match
  ...
  case Rec(f, p, pty, rty, body, scope) =>
    val fty = ArrowT(pty, rty)
    mustSame(typeCheck(body, tenv + (f -> fty) + (p -> pty)), rty)
    typeCheck(scope, ???)
```

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

$$\tau\text{-Rec}\ \frac{\Gamma[x_0 : \tau_1 \to \tau_2, x_1 : \tau_1] \vdash e_2 : \tau_2 \qquad ??? \vdash e_3 : \tau_3}{\Gamma \vdash \texttt{def}\ x_0(x_1{:}\tau_1){:}\tau_2{=}e_2; e_3 : \tau_3}$$

Type checker should do

1. check the type of e_2 is τ_2 in the type environment extended with type information for function $(x_0 : \tau_1 \to \tau_2)$ and parameter $(x_1 : \tau_1)$
2. return the type of e_3 in ???

Recursive Function Definitions

PLRG

```
def interp(expr: Expr, env: Env): Value = expr match
  ...
  case Rec(f, p, pty, rty, body, scope) =>
    val fty = ArrowT(pty, rty)
    mustSame(typeCheck(body, tenv + (f -> fty) + (p -> pty)), rty)
    typeCheck(scope, tenv + (f -> fty))
```

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

$$\tau\text{-}\mathtt{Rec}\ \frac{\Gamma[x_0 : \tau_1 \rightarrow \tau_2, x_1 : \tau_1] \vdash e_2 : \tau_2 \qquad \Gamma[x_0 : \tau_1 \rightarrow \tau_2] \vdash e_3 : \tau_3}{\Gamma \vdash \mathtt{def}\ x_0(x_1{:}\tau_1){:}\tau_2{=}e_2;\, e_3 : \tau_3}$$

Type checker should do

1. check the type of e_2 is τ_2 in the type environment extended with type information for function $(x_0 : \tau_1 \rightarrow \tau_2)$ and parameter $(x_1 : \tau_1)$
2. return the type of e_3 in the type environment extended with type information for function $(x_0 : \tau_1 \rightarrow \tau_2)$

Summary

1. Types for Recursive Functions
   - Recall: `mkRec` and Recursive Functions
   - `mkRec` in TFAE
2. TRFAE – RFAE with Type System
   - Concrete Syntax
   - Abstract Syntax
3. Type Checker and Typing Rules
   - Arithmetic Comparison Operators
   - Conditionals
   - Recursive Function Definitions

Exercise #11

- 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/trfae.

Next Lecture

- Algebraic Data Types (1)

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