Lecture 13 – Lazy Evaluation
COSE212: Programming Languages

Jihyeok Park

◭ PLRG

2023 Fall

Recall

- We learned two different evaluation strategies, **call-by-value** and **call-by-reference**, in the previous lecture.
    - **Call-by-value** (CBV) eagerly evaluates the arguments and passes the evaluated values to the function.
    - **Call-by-reference** (CBR) passes the references (i.e., addresses) of the arguments to the function.

- In this lecture, we will learn another evaluation strategy called **lazy evaluation** while the previous two are called **eager evaluation**.
    - **Call-by-name** (CBN)
    - **Call-by-need** (CBN')

- **LFAE** – FAE with Lazy Evaluation
    - Interpreter and Natural Semantics

Contents

Contents

So far, all the languages we have defined are based on the **eager evaluation** strategy; all the expressions are eagerly evaluated regardless of whether they are really needed or not.

Consider two FAE expressions (division is supported):

```
/* FAE */
val a = 1 + 2;
val b = 5 / 0;      // runtime error: division by zero
a * 3
```

```
/* FAE */
val f = a => b => a * 3;
f(1 + 2)(5 / 0)     // runtime error: division by zero
```

Is it possible to **delay** the evaluation until its result is really needed? **Yes!**

This is called **lazy evaluation**.

For example, Scala supports **lazy evaluation** for immutable variables with the `lazy` keyword and parameters with the `=>` notation.

```scala
val a = 1 + 2
lazy val b = 5 / 0
a * 3              // 9
```

```scala
def f(a: Int, b: => Int): Int = a * 3
f(1 + 2, 5 / 0)    // 9
```

Now, the value `5 / 0` for the variable `b` and the argument `5 / 0` for the parameter `b` are not evaluated because they are not really needed.

```scala
  ... def getOrElse[B >: A](default: => B): B = ...
 env.getOrElse(x, error(s"free identifier: $x"))
```

The `getOrElse` method of `Option` type is also defined with the lazy evaluation for the default value.

Lazy Evaluation

Many programming languages support **lazy evaluation** for many reasons.

- **Conditional Expressions** or **Short-circuit Operators**

```
if (true) 42 else (5/0) // 42      -- 5/0 is not evaluated
false && (5/0)<1         // false  -- (5/0)<1 is not evaluated
```

- **Optimization**: It could optimize the performance by avoiding unnecessary computations.

```
def f(x: Int, y: =>Int): Int = if (x < 0) 0 else x * y
f(42, complex(...))     // 0       -- complex(...) is not evaluated
```

- **Infinite Data Structures**: It makes it possible to define and manipulate infinite data structures.

```
val nats: LazyList[BigInt] = 0 #:: nats.map(_ + 1)
nats.take(10).toList      // List(0, 1, 2, 3, 4, 5, 6, 7, 8, 9)
```

Contents

LFAE – FAE with Lazy Evaluation

Now, let's extend FAE into LFAE to support **lazy evaluation**. (Assume that val is supported in FAE as syntactic sugar.)

```
/* LFAE */
val a = 1 + 2;
val b = c + 3;
a * 3    // 9
```

```
/* LFAE */
val f = a => b => a * 3;
f(1 + 2)(c + 3)    // 9
```

For LFAE, we don't have to extend any syntax.

Let's focus on how to extend the semantics of FAE to support **lazy evaluation**.

While there are diverse ways to define the lazy evaluation semantics, we will define **call-by-name** (CBN) semantics for LFAE.

It delays the evaluation of values of val and arguments of function applications as much as possible until they are really needed.

For LFAE, 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 **expression values** for lazy evaluation.

$$
\begin{aligned}
\text{Values} \quad \mathbb{V} \ni v ::= \ & n & (\texttt{NumV}) \\
| \ & \langle \lambda x.e, \sigma \rangle & (\texttt{CloV}) \\
| \ & \langle\!\langle e, \sigma \rangle\!\rangle & (\texttt{ExprV})
\end{aligned}
$$

```
enum Value:
  case NumV(n: BigInt)
  case CloV(p: String, b: Expr, e: Env)
  case ExprV(e: Expr, env: Env)  // for lazy evaluation
```

Function Application

```
def interp(expr: Expr, env: Env): Value = expr match
  ...
  case App(f, e) => interp(f, env) match
    case CloV(p, b, fenv) => interp(b, fenv + (p -> interp(e, env)))
    case v                => error(s"not a function: ${v.str}")
```

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

$$\text{App} \; \frac{\sigma \vdash e_0 \Rightarrow \langle \lambda x.e_2, \sigma' \rangle \qquad \sigma \vdash e_1 \Rightarrow v_1 \qquad \sigma'[x \mapsto v_1] \vdash e_2 \Rightarrow v_2}{\sigma \vdash e_0(e_1) \Rightarrow v_2}$$

We want to **delay** the evaluation of the **argument expression** e_1 as much as possible until it is really needed.

Let's define an expression value $\langle\langle e, \sigma \rangle\rangle$.

Function Application

```
def interp(expr: Expr, env: Env): Value = expr match
  ...
  case App(f, e) => interp(f, env) match
    case CloV(p, b, fenv) => interp(b, fenv + (p -> ExprV(e, env)))
    case v                => error(s"not a function: ${v.str}")
```

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

$$\text{App} \frac{\sigma \vdash e_0 \Rightarrow \langle \lambda x.e_2, \sigma' \rangle \qquad \sigma'[x \mapsto \langle\!\langle e_1, \sigma \rangle\!\rangle] \vdash e_2 \Rightarrow v_2}{\sigma \vdash e_0(e_1) \Rightarrow v_2}$$

```
/* LFAE */
val f = x => x + 1; f(1)
```

Unfortunately, in this expression, f has an expression value $\langle\!\langle \lambda x.x + 1, \sigma \rangle\!\rangle$ rather than a closure value. It means that we need to evaluate the expression value $\langle\!\langle e, \sigma \rangle\!\rangle$ to get a closure value.

Let's define the **strict evaluation** for values to get its real value, a number or a closure, rather than an expression value.

$$\boxed{v \Downarrow v}$$

$$\text{StrictNum} \; \frac{}{n \Downarrow n} \qquad \text{StrictClo} \; \frac{}{\langle \lambda x.e, \sigma \rangle \Downarrow \langle \lambda x.e, \sigma \rangle}$$

$$\text{StrictExpr} \; \frac{\sigma \vdash e \Rightarrow v \qquad v \Downarrow v'}{\langle\!\langle e, \sigma \rangle\!\rangle \Downarrow v'}$$

```
def strict(v: Value): Value = v match
  case ExprV(e, env) => strict(interp(e, env))
  case _             => v
```

Function Application

```
def interp(expr: Expr, env: Env): Value = expr match
  ...
  case App(f, e) => interp(f, env) match
    case CloV(p, b, fenv) => interp(b, fenv + (p -> ExprV(e, env)))
    case v                => error(s"not a function: ${v.str}")
```

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

$$\text{App} \ \frac{\sigma \vdash e_0 \Rightarrow \langle \lambda x.e_2, \sigma' \rangle \qquad \sigma'[x \mapsto \langle\!\langle e_1, \sigma \rangle\!\rangle] \vdash e_2 \Rightarrow v_2}{\sigma \vdash e_0(e_1) \Rightarrow v_2}$$

Now, let's get the real value of function expression e_0 by using the strict evaluation of values to handle if the function expression evaluates to an expression value.

```
def interp(expr: Expr, env: Env): Value = expr match
  ...
  case App(f, e) => strict(interp(f, env)) match
    case CloV(p, b, fenv) => interp(b, fenv + (p -> ExprV(e, env)))
    case v                => error(s"not a function: ${v.str}")
```

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

$$\text{App} \frac{\sigma \vdash e_0 \Rightarrow v_0 \qquad v_0 \Downarrow \langle \lambda x.e_2, \sigma' \rangle \qquad \sigma'[x \mapsto \langle\!\langle e_1, \sigma \rangle\!\rangle] \vdash e_2 \Rightarrow v_2}{\sigma \vdash e_0(e_1) \Rightarrow v_2}$$

Now, let's get the real value of function expression e_0 by using the strict evaluation of values to handle if the function expression evaluates to an expression value.

Addition and Multiplication

```scala
type BOp = (BigInt, BigInt) => BigInt
def numBOp(x: String)(op: BOp)(l: Value, r: Value): Value =
  (strict(l), strict(r)) match
    case (NumV(l), NumV(r)) => NumV(op(l, r))
    case (l, r) => error(s"invalid operation: ${l.str} $x ${r.str}")

val numAdd: (Value, Value) => Value = numBOp("+")(_ + _)
val numMul: (Value, Value) => Value = numBOp("*")(_ * _)

def interp(expr: Expr, env: Env): Value = expr match
  ...
  case Add(l, r) => numAdd(interp(l, env), interp(r, env))
  case Mul(l, r) => numMul(interp(l, env), interp(r, env))
```

$$\text{Add} \quad \frac{\sigma \vdash e_1 \Rightarrow v_1 \qquad v_1 \Downarrow n_1 \qquad \sigma \vdash e_2 \Rightarrow v_2 \qquad v_2 \Downarrow n_2}{\sigma \vdash e_1 + e_2 \Rightarrow n_1 + n_2}$$

$$\text{Mul} \quad \frac{\sigma \vdash e_1 \Rightarrow v_1 \qquad v_1 \Downarrow n_1 \qquad \sigma \vdash e_2 \Rightarrow v_2 \qquad v_2 \Downarrow n_2}{\sigma \vdash e_1 \times e_2 \Rightarrow n_1 \times n_2}$$

Contents

PLRG

In Scala, lazy function parameters are evaluated using **call-by-name** evaluation strategy but lazy values are evaluated using **call-by-need**.

Call-by-Name (CBN) evaluation strategy evaluates delayed expressions **multiple times** if they are used multiple times:

```scala
def inc(x: Int): Int = { println("inc"); x + 1 }
def mul5(x: => Int): Int = x + x + x + x + x
mul5(inc(1))        // 10 and prints "inc" 5 times
```

Call-by-Need (CBN') evaluation strategy is a **memoized** version of CBN, which evaluates delayed expressions only **once** at the first time they are used and then **reuses** the result:

```scala
def inc(x: Int): Int = { println("inc"); x + 1 }
lazy val x: Int = inc(1)
x + x + x + x + x   // 10 and prints "inc" only once
```

Call-by-Name (CBN) vs. Call-by-Need (CBN')

PLRG

In purely functional languages, CBN' is **equivalent** to CBN and only has **performance benefits** because it avoids unnecessary re-evaluations.

However, with **mutation**, CBN' is **not equivalent** to CBN because it evaluates function arguments **only once** the first time they are used, and thus, it may lead to **different** results:

```
var y: Int = 2
def f(x: Int): Int = x + y
def g(z: => Int): Int = { z; y = 5; z }   // Call-by-Name
g(f(1))                                    // 1 + 5 = 6
```

```
var y: Int = 2
def f(x: Int): Int = x + y
lazy val z: Int = f(1)                     // Call-by-Need
z; y = 5; z                                // 1 + 2 = 3
```

Interpreter with Call-by-Need

```scala
enum Value:
  ...
  case ExprV(e: Expr, env: Env, var value: Option[Value]) // for caching

def strict(v: Value): Value = v match
  case ev @ ExprV(e, env, v) => v match
    case Some(cache) => cache        // reuse cached value
    case None =>                     // the first use
      val cache = interp(e, env)     // evaluate the expression
      ev.value = Some(cache)         // cache the value
      cache                          // return the value
  case _ => v

def interp(expr: Expr, env: Env): Value = expr match
  ...
  case App(f, e) => strict(interp(f, env)) match
    // initialize `value` with `None` to denote no caching
    case CloV(p,b,fenv) => interp(b, fenv + (p -> ExprV(e, env, None)))
    case v              => error(s"not a function: ${v.str}")
```

Midterm Exam

- Midterm exam will be given in class.
- **Date:** 13:30-14:45 (1 hour 15 minutes), October 25 (Wed.).
- **Location:** 535, Asan Science Building (아산이학관)
- **Coverage:** Lectures 1 – 13
- **Format:** 7–9 questions with closed book and closed notes
  - Fill in the blank in a Scala code snippet.
  - Define a syntax or semantics of extended language features.
  - Write the evaluation results of given expressions.
  - Yes/No questions about concepts in programming languages.
  - etc.

Summary

1. Lazy Evaluation

2. LFAE – FAE with Lazy Evaluation
   Interpreter and Natural Semantics
   Function Application
   Addition and Multiplication

3. Call-by-Name (CBN) vs. Call-by-Need (CBN')
   Interpreter with Call-by-Need

- Continuations

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