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.

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

Lazy Evaluation

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.

Lazy Evaluation

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
```

Lazy Evaluation

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?

Lazy Evaluation

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**.

Lazy Evaluation

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

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

```
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.

Lazy Evaluation

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

- **Short-circuit Evaluation**: It could avoid unnecessary computations for boolean expressions.

```
ture  && ((5 / 0) < 1)        // error  -- division by zero
false && ((5 / 0) < 1)        // false  -- (5/0)<1 is not evaluated
true  || ((5 / 0) < 1)        // true   -- (5/0)<1 is not evaluated
false || ((5 / 0) < 1)        // error  -- division by zero
```

Most programming languages (e.g., C++, Java, Python, JavaScript, and Scala) support **short-circuit evaluation** for boolean expressions.

Lazy Evaluation

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

- **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(-7, complex(...))     // 0 -- complex(...) is not evaluated
```

Lazy Evaluation

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

- **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(-7, complex(...))     // 0 -- complex(...) is not evaluated
```

In fact, we already utilized lazy evaluation in our interpreter:

```
// The definition of `getOrElse` method in `Map`
def getOrElse[V1 >: V](key: K, default: => V1): V1 = ...

// The implementation of interpreter
def interp(expr: Expr, env: Env): Value = expr match
  ...
  case Id(x) => env.getOrElse(x, error(s"free identifier: $x"))
```

Lazy Evaluation

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

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

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

  - Haskell

```haskell
let nats = 0 : map (+1) nats
take 10 nats              -- [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Contents

LFAE – FAE with Lazy Evaluation

PLRG

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
```

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** rather than **eager evaluation**.

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

We want to **delay** the evaluation of **argument expressions** in function applications **as much as possible** until they are really needed.

Interpreter and Natural Semantics

For LFAE, we need to 1) implement the **interpreter** with environments:

```
def interp(expr: Expr, env: Env): Value = ???
```

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:

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

with a new kind of values called **expression values** for lazy evaluation.

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

```
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

```scala
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}")
```

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

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

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

PLRG

```
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}")
```

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

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

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}")
```

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

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

```
/* LFAE */
(f => f(1))(x => x+1) // error -- not a function (expression value in f)
```

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.

Strict Evaluation for Expression Values

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

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

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

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

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

Function Application

```scala
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}")
```

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

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

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.

Function Application

PLRG

```
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}")
```

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

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

Then, how to handle the identifier lookup and arithmetic operation?

Addition

```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("+")(_ + _)

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

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

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

For addition, we require actual values for both operands to perform addition. Thus, we need to perform strict evaluation for both operands.

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

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

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

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

Similarly, we need to perform strict evaluation for both operands for multiplication as well.

Identifier Lookup

```scala
def interp(expr: Expr, env: Env): Value = expr match
  case Id(x) => env.getOrElse(x, error(s"free identifier: $x"))
```

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

$$\texttt{Id}\ \frac{x \in \mathsf{Domain}(\sigma)}{\sigma \vdash x \Rightarrow \sigma(x)}$$

We will not perform strict evaluation for the value of identifier lookup because we can just pass the value without knowing its actual value.

```scala
/* LFAE */
(f => f(1))(x => x + 1) // 2
```

Now, it successfully evaluates to 2.

Contents

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

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')

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:

```scala
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
```

```scala
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 for Call-by-Need (CBN')

```
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 represent 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

- The 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 the syntax or semantics of extended language features.
  - Write the evaluation results of given expressions.
  - Yes/No questions about concepts in programming languages.
  - etc.
- Note that there is **no class** on **October 23 (Mon.)**.

Summary

1. Lazy Evaluation

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

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

Next Lecture

- Continuations

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