# Lazy Evaluation

## Computer Language Processing 2018 Final Report

Andrea Veneziano Ludo Hoffstetter
EPFL
{andrea.veneziano, ludo.hoffstetter}@epfl.ch

## 1. Introduction

During the semester, we developed an **interpreter** and a **compiler** for a functional language, similar to Scala, called **Amy**. Even though only the compiler, by definition, performs code generation, they share most of the components:

- A **lexer**, which converts the input text, made of characters, into a list of tokens, abstracting away all useless information.
- A **parser**, which, in our implementation, takes a sequence of tokens and first transforms them into parse trees and only then converts them into abstract syntax trees (ASTs).
- A **name analyzer**, which tasks are:
  1. Rejecting programs that do not follow Amy naming rules,
  2. Assigning unique identifiers to every name (f.ex. variable declarations),
  3. Populating the symbol table, effectively binding uses to declarations.
- A **type checker**, which checks that programs adhere to Amy typing rules. It can prevent certain errors, such as additions between an integer and a string.
- A **code generator**, which generates WebAssembly bytecode from the ASTs.

Our task for this final lab was to change the evaluation strategy of the interpreter from eager to **lazy evaluation**. This change entails some modifications in *Interpreter.scala*.

Lazy evaluation allows us to support infinite streams, as well as to avoid repeated evaluations and useless computations, although less experienced programmers could find it less intuitive.

## 2. Examples

In order to fully explain the functionality of our extension, we provide several examples that highlight the **average uses** and the **corner cases** in the directory *examples/lazy*.

Let's start with an example that shows how **infinite streams** should behave with lazy evaluation:

```
def powersOfTwo(start: Int): List =
      Cons(start, powersOfTwo(start * 2))

print(take(powersOfTwo(1), 5))
```

This code leads to an endless loop with eager evaluation, however the output is *[1, 2, 4, 8, 16]* with lazy evaluation because instead of an infinite recursion, *powerOfTwo(1)* evaluates only what is needed (i.e the 5 first elements of the stream).

As we said before, lazy evaluation avoids **useless computations**:

```
val x = (print(42); 0)
val y = x + 1
print("Ok!")
```

Only *Ok!* is printed after the execution of this code. None of the values are used, thus they are not evaluated and *print(42)* is never reached.

The same property is applied to **function arguments**:

```
def f(x: Int): Unit = {
    print("Ok!")
}

val y: Int = (print("Hello"); 2)
f(y)
```

The function *f* requires an argument but never uses it. Thus it is never evaluated since it is not needed and the output is *Ok!*.

The behavior of lazy evaluation with **pattern matching** also has to be defined:

```
val l: List = Cons(1, Cons(2, Cons(error(), Nil())))
// No error

l match {
    case Nil() => ()
    case Cons(h, t) => print(h)
}
// Still no error

l match {
    case Nil() => ()
    case Cons(h1, Cons(h2, Cons(h3, _))) =>
      // Still no error
      print(h3)
      // The evaluation of the third element of l
      // is forced and the error is thrown
}
```

The evaluation logic of the pattern matching is very similar to the function call one. After the first *match*, the error is not thrown since only the first element is evaluated and printed. The second *match* forces the evaluation of the error, therefore the program stops.

We said earlier that lazy evaluation is useful in order to avoid **repeated evaluations**, here is an example:

```
val y: Int = (print("Hello"); 2)
print(y)
print(y)
```

The first *print* statement forces the evaluation of *y*, thus *Hello* and *2* are printed. The second *print* attempts to force the evaluation of *y* but since it was already computed earlier, the cached value is returned and the total output is *Hello 2 2*.

## 3. Implementation

In this section we will first briefly discuss what is lazy evaluation made of, and then we will expose how we changed the interpreter.

### 3.1 Theoretical Background

Lazy evaluation defers expressions' evaluation until their results are needed and uses memoization to avoid repeating it.

In order to achieve this, we used **thunks**, parameterless closures that cache the value they wrap upon evaluation.

### 3.2 Implementation Details

We defined our thunk with memoization as another possible Value, **LazyValue**, in the interpreter. This choice allows us to use it in place of another Value, for example when passing arguments to functions/constructors calls or when we have to modify the environment binding identifiers to values (the *locals* map) in correspondence of a val declaration.

```
type Lazy = () => Value

case class LazyValue(private val f: Lazy) extends Value {
    val expr: Lazy = {
      var evaluated: Boolean = false
      var value: Value = UnitValue
      () => {
        if (!evaluated) {
          value = f()
          evaluated = true
        }
        value
      }
    }

    def apply(): Value = expr()
  }
```

We also defined an implicit conversion from Value to Lazy to ease the use of LazyValue.

```
implicit def v2l(v: => Value): Lazy = () => v
```

The code above represents the additional structure we needed; now we will see the actual changes to the pre-existing interpreter code.

The first major modification is that the value bound to a **val** is wrapped into a thunk, and the corresponding LazyValue is added to *locals*.

```
case Let(df, value, body) =>
 val newLocals = locals + (df.name ->
                           LazyValue(interpret(value)))
 interpret(body)(newLocals)
```

The second important modification is related to **variables** usage. If a LazyValue local is used during execution, it **forces** the evaluation of the wrapped Value and returns it.

```
case Variable(name) =>
 val v = locals(name)
 v match {
  case l: LazyValue => l()
```

```
  case _ => v
 }
```

The third modification is about the **arguments** passed to a function or a constructor **call**, which are now wrapped in a thunk, because they should be evaluated only when/if used.

```
case Call(qname, args) =>
 val interpretedArgs = args.map(arg =>
                              LazyValue(interpret(arg)))
 ...
```

The last modification concerns **pattern matching**. The first two changes we made almost guarantee that it only evaluates expressions as much as needed. We just need to add, to ensure correctness, a case inside *matchesPattern* which takes care of LazyValue. This is particularly useful to avoid MatchError when we work with nested CaseClasses; since each one is lazy, we need to evaluate it in order to perform the recursive call to *matchesPattern* which is needed to match each argument.

```
case Match(scrut, cases) =>
 ...
 def matchesPattern(v: Value, pat: Pattern):
        Option[List[(Identifier, Value)]] = {
    ((v, pat): @unchecked) match {
        ...
        case (l@LazyValue(_), _) =>
                matchesPattern(l(), pat)
        ...
 }
```

Finally, it is worth mentioning the small adjustment we made to the ***builtIns*** map, whose values are now functions taking a list of LazyValue as arguments (since every function call receives LazyValues as arguments, as defined in the Call case). For this reason, the head of the list is now evaluated before its usage.

```
val builtIns: Map[(String, String),
                  List[LazyValue] => Value] = Map(
 ("Std", "printInt") ->
    { args => println(args.head().asInt); UnitValue },
 ("Std", "printString") ->
    { args => println(args.head().asString); UnitValue },
 ...
 ("Std", "intToString") ->
    { args => StringValue(args.head().asInt.toString) },
 ("Std", "digitToString") ->
    { args => StringValue(args.head().asInt.toString) }
)
```

## 4. Possible Extensions

A natural extension of our work could be implementing lazy evaluation in the Amy ***compiler***. In fact, we could even support both lazy and eager evaluation strategies. However, this would require substantial changes in almost every component of the compiler.

## References


- *https://www.quora.com/How-is-lazy-evaluation-implemented-in-functional-programming-languages*
- *https://www.codementor.io/agustinchiappeberrini/lazy-evaluation-and-javascript-a5m7g8gs3*
- *http://matt.might.net/articles/implementing-laziness/*
- *https://en.m.wikibooks.org/wiki/Haskell/Laziness#Thunks_and_Weak_head_normal_form*