

# Lazy Evaluation for AMY

## Compiler Construction '19 Final Report

Bradley Mathez, 260413

EPFL

{bradley.mathez}@epfl.ch

### 1. Introduction

The original goal of this project was to develop a compiler for the AMY language. AMY is a simple language and this project aimed to learn about compilers. For this we implemented the different parts during the previous labs.

The aim of this extension is to add the *lazy evaluation* feature. For example we can find that in Scala language and it allows to build infinite lists without evaluating the whole list. For this extension we are going to modify only the first part of the whole project which is the interpreter. An interpreter takes the input and interpret it line by line. It is slower but simpler to interpret than compile the input and then execute it.

In this report, some examples will be given to demonstrate when our extension could be useful and why. After that, we will explain our implementation in details and how we developed this extension. Finally some possibilities to further extend this project will be displayed.

### 2. Examples

In AMY we cannot define an infinite list and try to print the first 3 elements (see example below).

---

```
object InfiniteStream{
  def countFrom(start: Int): L.List = {
    L.Cons(start, countFrom(start + 1))
  }

  val l:L.List = countFrom(0);
  Std.printString(L.toString(L.take(countFrom(0), 3)))
}
```

---

The problem is that AMY is going to evaluate the whole list to store it in the variable *l* and it will break the stack with a stack overflow.

An other example could be to define a variable with an error in it like in the code below.

---

```
object myTest {
  val l2: L.List = L.Cons(1, L.Cons(error("lazy"), L.Nil()));
  ...
}
```

---

Using the interpreter without lazy evaluation is gonna evaluate the whole right part. Thus, we will obtain an error with the message "lazy". The same code with the lazy evaluation will only generate a thunk (see next section) with the right part as the expression store in it without evaluation.

### 3. Implementation

As we only have to add the lazy evaluation in the interpreter, we have to modify only the interpreter to implement this feature.

The core idea is to use Thunks instead of simple variable. A Thunk is composed of three elements as we could see in the definition below and extends from Value.

---

```
abstract class Value {
  ...
case class Thunk(
  var value: Option[Value],
  e: Expr,
  env: Map[Identifier, Value]) extends Value
  ...
}
```

---

value is the None if haven't evaluated the expression *e* otherwise it's simply the result of the evaluation of *e* using the environment *env*. *e* is the expression that is attributed to the Thunk. Finally, *env* is the environment in which the Thunk has been declared/created.

Then we duplicated the whole interpret function to create interpretLazy which interpret the least

possible things and generally returns a `Thunk` instead of an evaluated `Value`. For example we modify every simple operations like `Mul(lhs, rhs)`.

---

```
def interpretLazy(expr: Expr)(
  implicit locals: Map[Identifier, Value]): Value = {
...
case Minus(lhs, rhs) =>
  Thunk(None, Minus(lhs, rhs), locals)
case Times(lhs, rhs) =>
  Thunk(None, Times(lhs, rhs), locals)
...
}
```

---

The idea in the interpreter is to evaluate the least possible things and every evaluation has to be thought before done. For example if we take the previous example with the infinite list. We wouldn't evaluate everything to infinity. We would only evaluate the elements step by step. For example the `Call` token will be matched like this.

---

```
case Call(qname, args) =>
  if (isConstructor(qname)) {
    CaseClassValue(
      qname,
      args.map(e => Thunk(None, e, locals))
    )
  }
...
}
```

---

First of all we call `interpretLazy` at the beginning. In this function we always try to return a `Thunk` as we said previously. However with some token we have to evaluate the expression. For example with `Call`. Please refer our code to see in details the implementation (It wasn't possible to put everything in the little box of code of this report).

There are only two bridges from `interpretLazy` to `interpret`. The first one is with a call to a `builtIns` function and the second is in the condition of an `if-then-else` (ITE). These cases are the only ones where the evaluation is mandatory.

In the other direction (`interpret` to `interpretLazy`) there are three bridges. Two are from the case `Call(...)` and one is from case `Match(...)`. As we have to evaluate the least possible things that's normal to go from a `Call(...)` to the lazy interpreter (think about the infinite list example discussed previously. We don't want to evaluate the arguments infinitely but just the first "level"). For the bridge from the case `Match(...)` it's the same idea. If the pattern is a `Thunk` then we only want to evaluate the first "level".

For the rest you have to look at the code. We didn't have time to finish the report. However the code works well and is able to execute the proposed examples.

## 4. Possible Extensions

We finished the implementation but not the report. It compiles and correctly executes the code examples we thought about. We tried the infinite list and every others suggested in the extensions list. We also tried our own example and the interpreter evaluates them as it should. We haven't spotted any problems during my tests.

We could extend the lazy evaluation to the compiler. There are more challenges to do that as we have to think about each step of the compiler.