Lazy Evaluation for AMY

Compiler Construction '19 Final Report

Bradley Mathez
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.

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 is 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 evaluated the least possible things and every evaluations has to be thought

before done. For example if we takes 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))
        )
    }
    ...
```

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

4. Possible Extensions

I 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. I 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.