The Basic Meta-Interpreter

William Frid

Spring 2024

Introduction

When developing software you often test the code to debug and simply verify that it's working as intended as a whole. This can be done in different ways, such as; compiling and running, or using an interpreter. With the second one being the most common. *Note that there exists "interpreted languages" (such as JavaScript[in most cases]).*[1]

Interpreters come in different forms, for example, those who run the whole code at once (as if the code was first compiled then run, but at a lower rate), and those who run line by line. The first one is a great place to start if one wishes to learn how these work as they can be tricky to get a grasp of at first. This is precisely the goal of this report. To provide a deeper understanding of interpreters and functional code languages. More precisely, it covers the main parts of a meta-interpreter (a.k.a. self-interpreter) built using Elixir.

As this report belongs to an assignment for the course **ID1019** at KTH Royal Institute of Technology, it will also include comments about the steps taken. For a better understanding of the content of this report, I'd recommend that you read it together with the code written and documentation for it. The files are available in a repository over at GitHub.com (URL: https://github.com/williamfridh/ID1019_VT19_TCOMK_Programming_2).

Implementation

The first step of interpretation is the evaluation of *sequences* (or simply "code lines"). These "code lines" were in the written code handled by the function `eval_seq/2`. Its purpose is to iterate over each line (will be called `match` henceforth) in the provided code and at the same time provide a list-based environment holding values of variables.

For a better understanding, let's use the example sequence seen in the snippet below as it tests most parts of the program. This *sequence* of

[1] JavaScript. Mozilla. 2023. https://developer.mozilla.org/en-US/docs/Web/JavaScript (2024-01-31).

`matches` would be read one line at a time by `eval_seq/2` (first line being `{:match, {:var, :x}, {:atm,:a}}`) that would start by passing the second argument (`{:atm, :a}` a.k.a. the *expression*) to `eval_expr/2` and receive the response `{:ok, :a}`.

```
seq = [{:match, {:var, :x}, {:atm,:a}},
    {:match, {:var, :y}, {:cons, {:var, :x}, {:atm, :b}}},
    {:match, {:cons, :ignore, {:var, :z}}, {:var, :y}},
    {:var, :z}]
```

Next, a copy of the environment without the variables in the first argument (a.k.a. *pattern*) is created and used as an argument alongside the response, and the first argument (a.k.a. *expression*) to call `eval_match/3`. This is a function that performs pattern matching, which compares if the two first arguments can be combined and looks up or adds any value needed in the environment (make variable `:x` point to atom `:a` in this case). It returns an updated environment `{:ok, [:x, :a]}` (can be compared to `x = :a;`) and passes this to itself (recursion) to evaluate the next `match`.

Note that using function overloading `eval_match/3` *supports different types of arguments, that is for instance* `:ignore`*,* `:atm`*, and* `:cons` *as seen in line three of the sequence.*

Now assume the program has reached the second line of the *sequence*. Once again the expression needs to be sent for evaluation first. As the expression consists of a `:cons` (tuple), it will be caught by a different `eval_seq/2`. It elevates the head first, then the tail if no errors pop up. And finally, combine these two results into a new tuple holding the results. That is, `{:ok, {:x, :b}}` which is then matched with the variable `:y` (can be compared to `y = :a, :b;`).

The last two lines are not different. The third `match` is evaluated and will be interpreted as `{_, z} = y;` and the fourth `match` gets interpreted as simply `z`. Combining these four results, we get what's in the snippet below which would (if inserted into Elixir) yield `{:ok, :b}`.

```
x = :a; y = {:a, :b}; {_, z} = y; z
```

Extensions

Case Expressions

Assume you have the *sequence* as seen below. Reading it gives us the understanding that it first saves the atom `:a` to the variable `x`, then using a `case` compares that variable to two different `clauses`. And that the result is `{:ok, :yes}`.

```
seq = [{:match, {:var, :x}, {:atm, :a}},
    {:case, {:var, :x},
        [{:clause, {:atm, :b}, [{:atm, :ops}]},
         {:clause, {:atm, :a}, [{:atm, :yes}]}
    ]}
]
```

But how does it get to that stage? The trick here is to catch the `case` with a unique `eval_seq/2` that forwards the data to `eval_cls/3` that compares each `clause` until a match between the given variable (`a`) and the second argument in the `clause` (*pattern*) is found. This behavior is well described in the snippet below. Note how the function sends the matching `clause's` *sequence* for evaluation.

```
def eval_cls([], _, _) do :error end
def eval_cls([{:clause, ptr, seq} | cls], exprval, env) do
    case eval_match(ptr, exprval, env) do
        :fail ->
            eval_cls(cls, exprval, env)
        {:ok, env} ->
            eval_seq(seq, env)
    end
end
```

Lambda Expressions

What about lambda expressions then? Well, they are simpler than they look as most of the logic needed is already written by this stage. The first would be to catch the `lambda` with `eval_seq/2`. There a new environment for the lambda function would be generated as passing the whole environment containing variables not used would slow down the program. Next, a `closure` is returned (can be seen as a program inside the program).

Then an `apply` is used that has two arguments: a variable pointing to the `clause` and an environment of argument. Inside this function, the lambda function is fetched via `eval_expr/2` and the arguments are paired to the parameters. Finally, the *sequence* (logic inside the lambda function) is sent with the new environment (containing the paired data) for evaluation. Thus the `apply` is the call to the lambda function.

```
def eval_expr({:apply, expr, args}, env) do
    case eval_expr(expr, env) do
        :error -> :error
        {:ok, {:closure, par, seq, closure}} ->
            case eval_args(args, env) do
                :error -> :error
```

```
                    {:ok, strs} ->
                        env = Env.args(par, strs, closure)
                        eval_seq(seq, env)
                end
        end
    end
```

Named Functions

Using named functions instead of lambda functions isn't much different. The only additional function needed is a `eval_expr/2` that loads in a saved function and returns its *sequence* inside a `closure`. This was the final part of the assignment where the final test was to run the *sequence* seen below. Upon doing so, You'd receive the result `{:ok, {:a, {:b, {:c, {:d, []}}}}}`.

```
seq = [{:match, {:var, :x},
        {:cons, {:atm, :a}, {:cons, {:atm, :b}, {:atm, []}}}},
    {:match, {:var, :y},
        {:cons, {:atm, :c}, {:cons, {:atm, :d}, {:atm, []}}}},
    {:apply, {:fun, :append}, [{:var, :x}, {:var, :y}]}]
```

Discussion

As the interpreter written for this report is - as mentioned in the introduction - a meta-interpreter and a simple at that, it's clear to say that there's more to it than covered in this report (not to mention the given page limit of 4). However, it still grants a better understanding and gives a foundation to work from if one wishes.

Conclusion

Creating meta-interpreters requires a deep understanding of how programming languages are built and well-defined terms to keep track of the development. After having developed one however, continuing to other languages or expanding the written interpreter should be straightforward as it's mostly about seeing the patterns.

Seeing how much additional logic is required for an interpreter like this, it's easy to tell that the interpreted code runs slower than pure Elixir code. Running benchmarks is however not part of this assignment and speed is not the goal.

Lastly, the knowledge about functional programming received through this assignment can be used to learn how to write macros (used with caution).