Interpreter

Simon Yllmark

February 2024

1 Introduction

In this task I created an Interpreter for a small functional language which technically is a subset of the functional language Elixir. The interpreter will handle sequences of pattern matching expressions. The idea is that the interpreter should evaluate an object that is inserted as a parameter and then return an data structure, that is the simple idea.

2 Evaluation

When talking about evaluation I would like to refer to course material that our teacher discussed during one of our lessons and a snippet of it can be viewed in figure 1

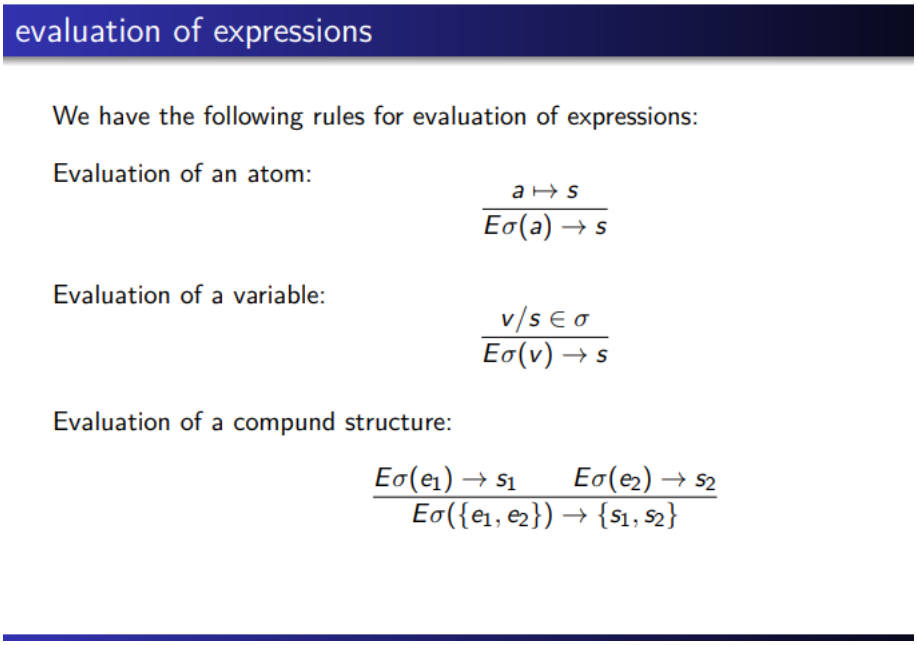

Figure 1: I know bad practise

Where he discussed the intrinsic nature between an expression and it's environment because it is kind of hard to evaluate an expression if the program have no source of references. What figure 1 describes is that the program can not evaluate an atom or an variable is there is no form of injective mapping like an environment implemented as a list of tuples. Then the figure 1 also state that if we want to evaluate a compound structure then the expressions by themselves must be evaluable.

2.1 The environment

The environment is thought of to be an abstract way of mapping two list of data structures to each other, as like the injective function $f(x) = y$ where x implies y. How the environment is implemented in the Interpreter as list containing tuples where one element point to another. This can get very complicated very quickly when we want to map multiple arguments in a function like $f(x, y, z) = a, b, c, d$ and thus the program have to create a data structure that can contain the result.

2.2 Expression

The evaluation of an expression will depend on the environment. For example if we would use one of the Interpreter's function give it an expression

and an environment like this

```
Eager.eval_expr({:var, :a}, [{:a,:b},{:c,:d}])
```

Then this is what it would return:

```
{:ok, :b}
```

So, it return a data structure in form of a tuple that contain atom :ok to say that the operation went smoothly and atom :b to say that there is an injective relationship between atom :a and atom :b which is specified in the environment. Here is some code that implement that:

```
  def eval_expr({:atm, id}, _) do {:ok, id} end
def eval_expr({:var, id}, env) do
  case Env.lookup(id, env) do
    nil ->
      :error
    {_, str} ->
      {:ok, str}
  end
end
def eval_expr({:cons, head, tail}, env) do
  case eval_expr(head, env) do
    :error ->
      :error
    {:ok, hs} ->
      case eval_expr(tail, env) do
        :error ->
          :error
      {:ok, ts} ->
        {:ok, {hs, ts}}
    end
  end
end
```

2.3 Pattern matching

Now we come to the part where we update the environment, this is not complicated at all. To explain this I would like to refer to lecture material found in canvas and a snippet of it can be viewed in figure 2. the literature describes three possible outcomes 1 where the pattern matching return the environment because the pattern matching have not updated the pattern in the environment, 2 the environment is updated is updated because a new pattern have been inserted into the program and thus the environment need to be updated and 3 the pattern matching fail because the environment

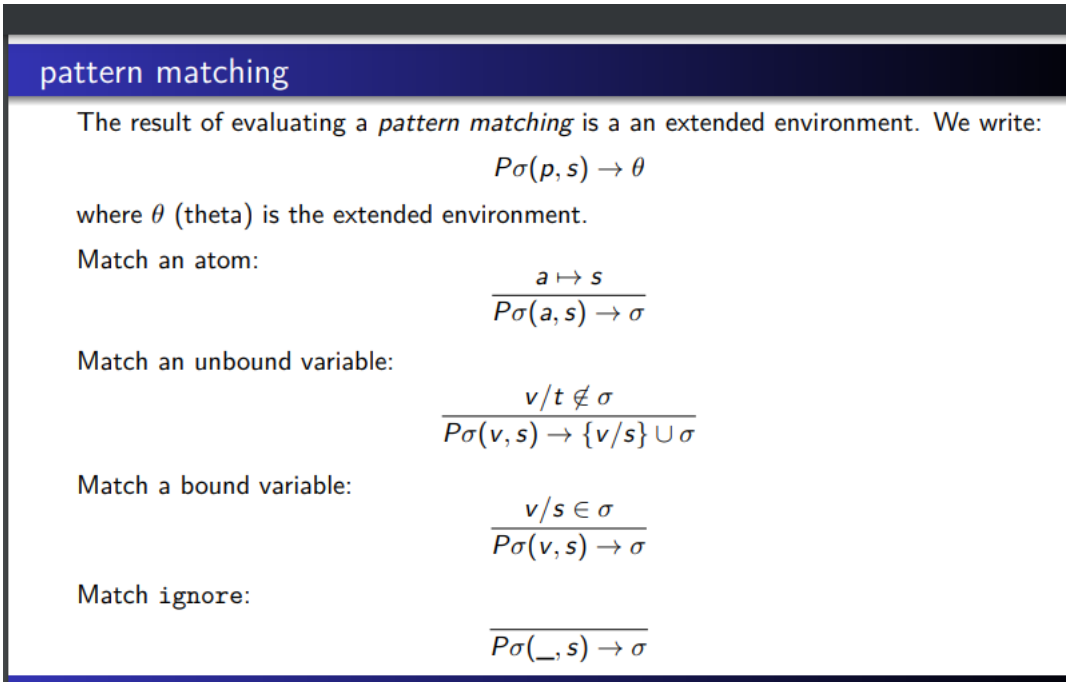

Figure 2: I know I'm at it again

does not contain an injective mapping for an atom or a pattern matching for a variable is already declared in the environment that is different from the variable specified in the pattern matching. Here is some code that implement that:

```
    def eval_match(:ignore, _, env) do {:ok, env} end
  def eval_match({:atm, id}, id, env) do {:ok, env} end
  def eval_match({:var, id}, str, env) do
    case Env.lookup(id, env) do
      :nil ->
        {:ok, Env.add(id, str, env)}
      {_, ^str} ->
        {:ok, env}
      {_, _} ->
        :fail
    end
  end
  def eval_match({:cons, hp, tp}, {head, tail}, env) do
    case eval_match(hp, head, env) do
      :fail ->
        :fail
      {:ok, env} ->
```

```
        eval_match(tp, tail, env)
    end
  end
  def eval_match(_, _, _) do :fail end
```

3 Sequences

I feel that it would be good to add in this report the part of the program where i sequence of statements can be inputted and evaluated, for example the Elixir code:

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

Which translated to iex terminal language is this:

```
x = :a
y = [x | :b]
[_, z] = y
z
```

Then using different atoms the program can then identify which operations that are taking place.

4 Extensions

Here comes the part where we teach the Interpreter to handle case expressions, lambda expressions and named functions. What things boil down to is those cases are to be represented as a sequence of statement which could for example could look like this:

```
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}]}
]
```

In this example we have a list of statements and atoms are used to represent the operation that is supposed to occur and we have variable and atoms that should be evaluated.

4.1 Case expressions and named functions

A case expression consists of an expression and a list of clauses where each clause is a pattern and a sequence, a clause is simply a tuple with the key word :clause. The program then need to evaluate case expression by taking a case expression containing a list of :clause tuples and evaluate them. Then if the program would evaluate this sequence seq:

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

Then the program would return this tuple:

```
{:ok, :yes}
```

This is because :var :x is assigned to :atm :a and and in the case expression :x then get reassigned to the :atm :yes.

4.2 Lambda expressions

To solve this task The Interpreter needed a new data structure called closure. A closure simply consists of a sequence of parameter variables, a sequence expression together with an environment, one of the purposes of the closure is to separate the free variables and the variables that are bound to a function. Here is an example of an sequence contain the atom operation lambda:

```
seq = [{:match, {:var, :x}, {:atm, :a}},
{:match, {:var, :f},
{:lambda, [:y], [:x], [{:cons, {:var, :x}, {:var, :y}}]}},
{:apply, {:var, :f}, [{:atm, :b}]}
]
```

What the program will do is to evaluate the sequence, create a closure for it to and break it down to it's :apply, :lambda, :match and etc part where the program execute operation on a case by case basis. This is the result:

```
{:ok, {:a, :b}}
```

4.3 Named functions

These named functions will be implemented on a case by case bases. This is then supposed to be similar to handling programs in a computer. For each named function the Interpreter will have a Elixir function that returns our representation of the parameters and its sequence. Then the look up of those functions will be done by the new Prgm module.

```
      def eval_expr({:fun, id}, _) do
    {par, seq} = apply(Prgm, id, [])
    {:ok, {:closure, par, seq, []}}
  end
  defmodule Prgm do
  def append() do
    {[:x, :y],
      [{:case, {:var, :x},
        [{:clause, {:atm, []}, [{:var, :y}]},
          {:clause, {:cons, {:var, :hd}, {:var, :tl}},
            [{:cons,
              {:var, :hd},
              {:apply, {:fun, :append}, [{:var, :tl}, {:var, :y}]}}]
          }]
        }]
      }
  end
end
```