Task 5 - Interpreter

Casper Kristiansson

2022-02-06

Introduction

Task five of the course Programming II consisted of implementing an interpreter for a functional language. The task included implementing an environment, evaluating expressions, pattern matching, sequences, case expressions, lambda expressions and named functions. The author will be discussing how each part was implemented and solved.

Method

The author stared of by watching the lectures on lambda and evaluation. The two lectures gave an introduction on how to evaluate and implement different parts of the function language. The author also made sure to thoroughly read the two documents on operational semantics and meta interpreter. In order to properly solve the task, the author wrote tests along the way to test and make sure that each part of the interpreter was working correctly.

Result

The author will be dividing up each implementation of the tasks into different sections and discuss how the solution works and how its implemented with the rest of the program. Because the program became quite only the most noteworthy parts will be mentioned.

The environment

The first task is to implement the environment which has the goal of mapping variables to data structures. To keep it simple the author, choose to represent it as a key-value pairs (tuples). The functions in the environment will be able to create a new, add, lookup and remove variables from the environment. The remove function should be able to remove multiple ids

from the environment at the same time, and the author solved this using the **Enum.member?** function.

```
def add(id, str, env) do [{id, str} | env] end
def lookup(_, []) do :nil end
def lookup(id, [{id, str} | _]) do {id, str} end
def lookup(id, [_ | env]) do lookup(id, env) end
def remove(_, []) do [] end
def remove(ids, [{id, str} | env]) do
  if Enum.member?(ids, id) do
    remove(ids, env)
  else
    [{id, str} | remove(ids, env)]
  end
end
```

Figure 1: Creating and managing the environment

Expressions

The expression function's goal is to evaluate data structure and create the correct function calls depending what type of expression it is. The evaluate expression will handle the following expressions: :atm, :var, :cons, :case, :lambda, :apply, :fun and lastly :call.

Figure two consists of the most basic evaluations where the goal is to return the id of an atom or the value of a variable which could be solved by calling the lookup function in the environment. If an evaluation is correct and it is able to find the correct variable a tuple of {*:ok, str*} is returned.

```
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
```

Figure 2: Evaluating the basic cases

The :cons atom is used to evaluate two different expressions which can be done by calling them one after the other. If everything worked correctly the expression will return the correct values of the atom or variable.

Figure three consists of the functions to handle the extension cases for the **case expression** and **lambda expression**. The case expressions includes the atom :case, pattern and a sequence. If the expression is valid, it gets evaluated with the help of the function evaluation clauses. The author also created extensions to handle expressions :apply, :fun, and :call.

```
def eval_expr({:case, expr, cls}, env) do
  case eval_expr(expr, env) do
    :error ->
      :error
    {:ok, str} ->
      eval_cls(cls, str, env)
  end
end
def eval_expr({:lambda, par, free, seq}, env) do
  case Env.closure(free, env) do
    :error ->
      :error
    closure ->
      {:ok, {:closure, par, seq, closure}}
  end
end
```

Figure 3: Extension expressions

Pattern Matching

The goal of the pattern matching is to match several types of atoms. The basic cases where the atom equal each other returns {*:ok, env*}. Beside from the two basic cases there are two other cases. The first case is when the evaluate match gets called with a var where it is required to check whenever if the environment contains the id. Depending on if it does and has the correct value the environment gets returned. This is because the environment is only allowed to contain one variable with the same name.

If a :cons structure is used, the evaluate match will divide and check the expressions after each other and make sure that both expressions are valid and match each other.

```
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
```

Figure 4: Evaluating pattern matching

Sequences

The sequences goal is to help evaluate matching expressions where the first element will be a patter matching expression and afterwards a list of expressions. The main function of sequences goal is to evaluate two different expression and afterwards match them together. Depending if the matching is correct the program will continue with the rest of the sequences.

Discussion

There are still a few parts of the interpreter that was not discussed or shown in the result, for example the functions extract variables and evaluate arguments. These functions are simple but still a vital part of the program. Another important part is the evaluate clauses which is an important part for the **case** expression. The interpreter is now able to evaluate basic expressions, cases, lambda functions and named functions. The named functions were implemented by using the apply function in elixir which is able to call named functions and add them to the program. It was implemented with the help of the :call evaluation expression and the :fun expression.