Meta-Interpreter -Seminar 2

Has-san Nazar

February 6, 2015

1 Introduction

In this assignment the task is to create a Meta Interpreter for a small functional language. A meta interpreter is informally a program that interprets and evaluates a program and its language.Furthermore, An interpreter for a language similar or identical to its own implementation language is called meta-interpreter. An example for this is the GCC compiler for C/C++ which is itself written in the C language. Thus we would call this a Meta-Interpreter. A Meta interpreter is also called a self interpreter because of the fact that it interprets code from a language that itself is made from. The task is to create a limited functional program within Erlang and a Meta Interpreter to evaluate and interpret the functional subset of the Erlang language defined. With a limited functional program we mean a subset of the already existing Erlang language at which the created Meta Interpreter will work on.

The Interpreter should be made to handle simple expressions such as variable assignment and atoms. Also the program should be able to evaluate pattern matching. To then make the program a bit more complex, we are asked to develope a Case handler for the Interpreter. Which is basically a way to handle simple case functions for the language defined. This report will walk through the steps for creating this Interpreter and discussions on how the various parts were put together will, in detail, be explained in the report below.

2 Approach

2.1 Environment

To start building our Subset language from erlang we firstly need to define a sort of datastructure that will map our language variables to that data structure. For this we need to define 3 functions:

The functions defined were:

1. new() This function would return a new empty datastructure or "environment to the user
2. add(Id, Str, Env) This function adds a value[Str] to a variable[Id] and stores it in the environment[Env]
3. lookup(Id, Env) This function looks up the environment[Env] it the variable[Id] exists, and returns it if found.

The implementation was very simple and we used the lists module to add and look up the list. The full implementation of env.erl module is given below:

```
new()->  %return an empty enviroment
    [].
add(Id,Str,Env) -> %add a key-value tuple in the enviroment
    A = [{Id,Str}],
lists:append(Env, A).
    lookup(Id,Env) ->
    lists:keyfind(Id, 1, Env).
```

With this implementation done we can now go forth to create the actual Interpreter for our language.

2.2 Expressions

Now that our environment is set up, we can start working with our target programming language. The target is to be able to handle simple exressions that should be evaluated to data structures. By using a meta interpreter it is easy to implement such expression handling because the instructions of the interpreted language can be mapped directly to the data structures defined on the meta interpreter. The expression handling will be following the **BNF below:**

⟨*expression*⟩ ::= ⟨*atom*⟩
| ⟨*variable*⟩
| ⟨*cons*⟩

⟨*cons*⟩ ::= '' ⟨*expression*⟩ ',' ⟨*expression*⟩ ''

Also an expression is equivalent to a atom or a variable or a cons where cons is equivalent to an expression—- So our program needs to be able to differentiate between an atom, a variable and a "cons" cell.

Implementation in Erlang:

```
eval_expr({atm, Id}, _) ->% Vår Expression är en Atom
   {ok, Id}; % Då retunerar vi endast Atomnamnet

eval_expr({var, Id}, Env) ->%
   case env:lookup(Id, Env) of % Finns vår Variable i vår miljö?
      false ->  % Nej tydligen inte!
         error;
      {Id, Val} ->% Ja den finns
         {ok, Val} % Då retunerar vi svaret tillbaka
   end;

eval_expr({cons, {var, X}, {atm, B}}, Env) -> %
   case eval_expr({var, X}, Env) of % Finns Head i vår miljö?
      error ->% Nej
         error;
      {ok, A} ->% Ja den finns
         case eval_expr({atm, B}, Env) of %Titta om b finns?
```

```
            error ->
               error;
            {ok, B} ->
               {ok, {A,B}} % Retunera {A,B}
         end
   end;
```

2.3 Pattern Matching

Now that the expression handling is implemented, we need to be able to handle pattern matches, that is, to take a pattern and a data structure and a environment and either return ok, Env where the Env is the extended environment or the atom fail.

⟨*pattern*⟩ ::= ⟨*atom*⟩
| ⟨*variable*⟩
| ⟨_⟩
| ⟨*cons-pattern*⟩

⟨*cons-pattern*⟩ ::= '' ⟨*pattern*⟩ ',' ⟨*pattern*⟩ ''

The pattern is equivalent to either a atom or a variable or a _(dont care operator) or a cons-pattern, where a cons pattern is ————————————

Implementation in Erlang:

```
eval_match(ignore, _, Env) ->
   {ok, Env};
eval_match({atm, _}, _, Env) ->
   {ok, Env};
eval_match({var, Id}, Str, Env) -> % Vi tittar här om en variable med angiven value finns
   case env:lookup(Id, Env) of
      false ->
         {ok, env:add(Id, Str, Env)}; % om nej lägg till
      {Id, Str} ->
         {ok, [env:lookup(Id, Env)]}; % om ja skicka tillbaka
      {Id, _} ->
         fail % Id av var finns i Env, Dock så har den ett annat värde
   end;

eval_match({cons, Head, Tail}, {A,B}, Env) -> % Vi skickar med 2 patterns och 2 strukturer
   case eval_match(Head , {A}, Env) of %Match
      fail ->
         fail;
      {ok, NewEnv} -> % Vi skickar med nya Miljön för att
         eval_match(Tail, {B}, NewEnv)  % Matcha Tail med B
   end;
eval_match(_, _, _) ->% Om inget annat körs så skickar vi fail.
   fail.
```

2.4 Sequence

This is basically a way to stuff a list of pattern matching expressions followed by a single expression. For example the sequence

```
X = foo, Y = nil, {Z,_} = {bar,nil}, {X,{Z,Y}}.
```

⟨*sequence*⟩ ::= ⟨*expression*⟩
| ⟨*match*⟩ ',' ⟨*sequence*⟩

⟨*match*⟩ ::= ⟨*pattern*⟩ '=' ⟨*expression*⟩

Implementation in Erlang:

```
eval(Seq) ->
eval_seq(Seq, []).

eval_seq([Exp], Env) ->
   eval_expr(Exp, Env);
eval_seq([{match, Ptr, Exp}|Seq], Env) ->
   case eval_expr(Exp, Env) of
   error ->
      error;
   {ok, Str} ->
      case eval_match(Ptr, Str, Env) of
         fail ->
            error;
         {ok, NewEnv} ->
            eval_seq(Seq,NewEnv)
      end
   end.
```

With this done we can now evaluate sequences of pattern matchings, build up the environment and then use expressions to return answers.

3 The Case handler

When we have a working interpreter and sublanguange with expression handling and pattern matching, we can move on to implement a case handler in our language. This is so that simple "if" statements can be used.

The case handler was implemented following the BNF below:

⟨*expression*⟩ ::= ⟨*case expression*⟩
| ⟨*cons*⟩
| ⟨*variable*⟩
| ⟨*atom*⟩

⟨*case expression*⟩ ::= 'case' ⟨*expression*⟩ 'of' ⟨*clauses*⟩ 'end'

⟨*clauses*⟩ ::= ⟨*clause*⟩ | ⟨*clause ';' ¡clauses*⟩

⟨*clause*⟩ ::= ⟨*pattern*⟩ '->' ⟨*sequence*⟩

Implementation in Erlang:

```
eval_expr({switch, Exp, [{clause, Ptr, Seq}|Rest]}, Env)->
   case eval_expr(Exp,Env) of
      error ->
         error;
      {ok, _} ->
         case eval_match(Exp, Ptr, Env) of
            fail->
               eval_expr({switch, Exp, Rest}, Env);
            {ok, [{_,_}]} ->
               eval_seq(Seq, Env)
         end
   end;
eval_expr({switch, _, []}, _)->
   error.
```

With this implementation we can evaluate "case of" expressions as well.
Evaluation of:

```
case z of
   {a,b} -> x=7;
   {c,d} -> y=12
end
```

..where z = ''a,b'' would be presented like this in our implementation

```
eager:eval_expr({switch, {var, z}, [{clause, {a,b}, [{match, {var,x},
{atm,7}},{var,x}]},{clause, {c,d}, [{match, {var,y},{atm,12}},{var,x}]}]}
, [{z,{a,b}}]).

Returns {ok, 7}
```

Which means that the variable x har now been given the value 7 in our environment. Which is to show that our case handler works as intended.