Meta Interpreter

Oscar Melin

2016-11-05

1 Uppgiften

Den här rapporten behandlar skapandet av en meta interpreter för ett minimalt funktionellt språk väldigt likt Erlang.

En interpreter är ett datorprogram som tar ett ett annat program som input och sedan tolkar och utför instruktionerna av detta program utan att först kompilera programmet till maskinkod. Vidare kallas en interpreter som är skriven i sammma eller ett liknande språk till språket som tolkas för en meta interpreter.

Uppgiften till fråga går ut på att implementera en meta interpreter kapabel att hantera variabeldeklarering, simpel mönstermatchning samt så ska den kunna tolka case of statements, likt de som finns i Erlang.

2 Ansats

2.1 Teori

Vårat nya språk som ska tolkas av meta interpretern kommer att bestå av två enheter vid tolkningsfasen, en miljö och en sekvens av kod som ännu ej har tolkats. Miljön är en kartläggning mellan variabler och deras tilldelade värde. Exempelvis så representerar vi miljön {X/qwer, Y/asdf} som {{'X', qwer}, {'Y', asdf}}. Medans en sekvens är en följd av mönstarmatchande uttryck och case expressions följt av ett enkelt uttryck.

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

Figur: En simpel sekvens.

En definieras som följande i **BNF:**

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

Sekvensens individuella beståndsdelar definieras senare i egna sektioner.

2.2 Praktik

Den praktiska implementeringen består av två huvudsakliga delar, implementerade av två moduler; **env.erl** och **eager.erl**.

2.2.1 env.erl

env.erl innehåller de operationer som krävs för att interagera med miljön, såsom möjligheten att lägga till variabler, se ifall en variabel existerar och om så är fallet, returnera variabelns värde.

```
new() ->
    [].

add(Id, Str, Env) ->
    Env ++ [{Id, Str}].

lookup(_, []) ->
    false;
lookup(Id, [{Id, Str} | _]) ->
    {Id, Str};
lookup(Id, [_ | R]) ->
    lookup(Id, R).
```

Exempelvis, följande funktionsanrop ska returnera {foo, 42}:

```
env:lookup(foo, env:add(foo, 42, env:new())).
```

2.2.2 eager.erl

eager.erl tolkar uttryck, mönstermatchar, utvärderar case expressions och räknar ut resultatet av en sekvens av dessa. Eftersom detta är en meta interpreter så representeras en atom och en cons-cell i detta språk av en Erlang atom och en Erlang cons-cell. Modulen består av tre funktioner.

eval_expr/2 tar ett uttryck och en miljö och returnerar antingen **{ok, S}** där S är en datastruktur, eller **error**. Vi har valt att uttrycka ett uttryck enligt nedan och således behöver **eval_expr/2** kunna skilja på dessa.

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

```
eval_expr({atm, Id}, _) ->
    {ok, Id};
eval_expr({var, Id}, Env) ->
    case env:lookup(Id, Env) of
        false ->
            error;
        {Id, Str} ->
            {ok, Str}
    end;
eval_expr({cons, {var, H}, {atm, T}}, Env) ->
    case eval_expr({var, H}, Env) of
        error ->
            error;
        {ok, V} ->
            {ok, {V, T}}
    end.
```

eval_match/3. Vi kommer även att behöva möjligheten att hantera mönstermatchningar, det vill säga, att jämföra ett mönster med en datastruktur och antingen returnera **{ok, Env}** där **Env** är den förlängda miljön eller atomen **fail**.

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

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

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

```
eval_match(ignore, _, Env) ->
    {ok, Env};
eval_match({atm, Id}, Id, Env) ->
    {ok, Env};
eval_match({var, Id}, V, Env) ->
    case env:lookup(Id, Env) of
        false ->
            {ok, env:add(Id, V, Env)}; % Add if not in environment.
        {Id, V} ->
            {ok, Env}; % Simply return if found in environment.
        {Id, _} ->
            fail % Var name already bound.
    end;
eval_match({cons, H, T}, {A , B}, Env)  ->
```

```
    case eval_match(H, A, Env) of % Match head first.
        fail ->
            fail;
        {ok, EnvNew} ->
            eval_match(T, B, EnvNew) % Match tail last.
    end;
eval_match(_, _, _) ->
    fail.
```

eval_seq/2 tolkar och en sekvens av mönstarmatchande uttryck och case expressions följt av ett enkelt uttryck. **{ok, Str}** där **Str** är värdet på det avslutade uttrycket eller atomen **error** returneras. Första delen av **eval_seq/2** som behandlar uttryck och matchningar är utelämnad för att spara plats.

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

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

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

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

```
    ...
eval_seq([{switch, _, []}, _], _) -> % No matches -> error.
        error;
eval_seq([{switch, Exp, [{statement, Ptr, Body} | R]} | Seq], Env) ->
    case eval_expr(Exp, Env) of
        error ->
            error;
        {ok, _} ->
            case eval_match(Exp, Ptr, Env) of
                fail -> % No match, jump to next clause.
                    eval_seq([{switch, Exp, R} | Seq], Env);
                {ok, _} -> % Match, evaluate the corresponding sequence
                           % and remaining sequence after case expression.
                    eval_seq([Body] ++ Seq, Env)
            end
    end.
```

3 Utvärdering och Sammanfattning

Med interfacefunktionen:

```
eval(Seq) ->
    eval_seq(Seq, env:new()).
```

Testar vi anropet:

```
eager:eval([
{match, {var, x}, {atm,a}},
{switch, {var, x},
[{statement, b, {match, {var, y}, {atm, ett}}},
{statement, a, {match, {var, y}, {atm, två}}}]},
{var, y}]).
```

Som motsvarar koden:

```
X = a,
case x of
    b -> Y = ett;
    a -> Y = två;
end,
Y
```

Returnerar:

```
{ok,två}
```

Till en början så fanns det en del problem endast relaterade till att förstå konceptet av vad en meta interpreter var och hur man kunde implementera en sådan rent praktiskt. Vidare så antar jag att denna lösning är aningen annorlunda med case expressions ej definierade under uttryck utan som en egen beståndsdel i ett uttryck. Ett första försök att ha case expressions definierat under <expression> ledde till en del praktiska problem samtidigt som jag fick en idé att lösa problemet som presenterat. Detta ledde till att case expressions vart definierade direkt under <sequence> .