Interpretator

Kenan Dizdarevic

30 Januari 2023

Inledning

Vi skall i denna uppgift implementera en interpretator för ett litet funktionellt språk. För att lyckas implementera en interpretator behöver vi lite förkunskaper.

En interpretators syfte är att exekvera kod utan en kompilator. Istället för att omvandla hela koden till maskinkod och exekvera den på CPU-nivå, exekverar man koden rad för rad eller varje uttryck för sig.

För att vi skall kunna implementera en fungerande interpretator i Elixir behöver vi ha koll på termer, datastrukturer och mönster. En annan mycket viktig kunskap är satslogik. Vi skall i grund och botten implementera de givna satserna i Elixir. Har man koll på satslogiken är detta en relativt simpel uppgift.

1 Implementering

1.1 Omgivning

För att interpretatorn skall fungera behöver vi implementera en omgivning. Syftet med omgivningen är att vi skall kunna mappa från variabler till datastrukturer.

Vi gör ett antagande där vi antar att omgivningen kommer vara liten. Det är en människa som kommer skriva koden, således kommer den inte vara enorm. Vi väljer att implementera omgivningen som en lista av *nyckel-värde tupler*.

Omgivningen är relativt enkel att implementera. Vi har möjligheten att skapa en ny omgivning, funktionen `add/3` returnerar en ny omgivning med en ny bindning där ett variabel-id är bundet till en struktur. Funktionen `lookup/2` returnerar bindningen baserat på variabel-id. Funktionen `remove/2` returnerar en ny omgivning där alla bindningar av variabel-id är borttagna.

1.2 Uttryck

I vår implementering skall uttryck evalueras till datastrukturer. Viktigt att veta är att en atom är representerad som Elixirs egna atom. Strukturen {`:cons, uttryck, uttryck`} representeras av en Elixir tupel. Vi skall nu implementera funktionen `eval_expr/2` som har ett uttryck och en omgivning som argument. Funktionen skall returnera {`:ok, datastruktur`} om uttrycket går att evaluera. Om uttrycket inte går att evaluera skall funktionen returnera `:error`.

Det är relativt simpelt att evaluera en atom eller en variabel. När vi evaluerar en variabel utnyttjar vi funktionen `lookup/2` som vi skapade. Sedan utför vi mönstermatchning och returnerar den associerade datastrukturen.

Vi skall nu studera satslogiken och implementera evaluering av en sammansatt struktur. Satsen ser ut som följande:

$$\frac{E\sigma(e_1)\to s_1 \quad E\sigma(e_2)\to s_2}{E\sigma(\{e_1, e_2\}) \to \{s_1, s_2\}}$$

Först evaluerar vi uttrycket e_1 i en omgivning av σ. Om resultatet är en struktur s_1 som finns i σ, gör vi samma sak med e_2. Om det andra fallet också stämmer har vi evaluerat en funktion i en omgivning av σ. Satsen beskriver alltså en process för att kontrollera om en funktion är korrekt evaluerad i en omgivning av σ. Koden för denna sats ser ut som följande:

```
def eval_expr({:cons, e1, e2}, env) do
    case eval_expr(e1, env) do
      :error ->
        :error
      {:ok, s1} ->
        case eval_expr(e2, env) do
          :error ->
            :error
          {:ok, s2} ->
            {:ok, {s1, s2}}
        end
    end
  end
```

Vi har en tydlig förståelse för satslogiken, vilket innebär att koden är enkel att förstå. Vi har två fall precis som i satsen. I det första fallet evaluerar vi e_1 i vår omgivning. Det två utfall vi har är att antigen mappar e_1 till en struktur i vår omgivning eller inte. Om uttrycket inte mappar till en struktur returnerar vi `:error` annars utför vi den andra evalueringen. Den kan ha exakt samma fall som föregående evaluering, i det fallet att det lyckas returnerar vi tupeln {`:ok, {s1, s2}`}. För att utföra evalueringen av de två uttrycken anropar vi `eval_expr/2` rekursivt.

1.3 Mönstermatchning

Att evaluera ett uttryck är relativt enkelt. Det blir genast mer komplext att evaluera ett mönstermatchande uttryck. Vi skall implementera funktionen `eval_match/3` som tar emot ett mönster, en datastruktur och en omgivning. Funktionen returnerar antingen atomen `:fail` eller {`:ok, omgivning`} om mönstermatchningen lyckas. Omgivningen som returneras är den utökade omgivningen där vi har lagt till den nya strukturen.

Vi har ett fall där vi lägger till en ny variabel till vår omgivning. Där kollar vi först om variabeln finns i vår omgivning med `lookup/2`. Det kan uppstå två fall, variabeln finns i omgivningen eller inte. Om den finns returnerar vi samma omgivning annars lägger vi till variabeln i omgivningen och returnerar den utökade omgivningen.

Vi skall återigen studera satslogiken men denna gång för mönstermatchning med ett sammansatt uttryck. Vi har följande sats:

$$\frac{P\sigma(p_1, s_1) \rightarrow \sigma' \quad \wedge \quad P\sigma'(p_2, s_2) \rightarrow \theta}{P\sigma(\{p_1, p_2\}, \{s_1, s_2\}) \rightarrow \theta}$$

Vi skall göra mönstermatchning i omgivningen σ av uttrycket e_1 och datastrukturen s_1. Om detta lyckas sker mönstermatchningen med e_2 och s_2 skall ske i den nya omgivningen σ', om detta fungerar returneras den utökade omgivningen θ.

Koden som vi implementerade för denna sats är följande:

```
def eval_match({:cons, p1, p2}, {s1, s2}, env) do
    case eval_match(p1, s1, env) do
      :fail ->
        :fail
      {:ok, env} ->
        eval_match(p2, s2, env)
    end
end
```

Funktionens argument är ett mönster, en datastruktur och en omgivning. I vårt första fall anropar vi `eval_match/3` rekursivt på första mönstret och första strukturen. Om detta lyckas genomför vi ytterligare en mönstermatchning men på det andra mönstret och den andra datastrukturen. Det viktiga att ha koll på är att omgivningen vid det andra rekursiva anropet är vår nya omgivning σ'. När alla rekursiva anrop är klara erhåller vi vår utökade omgivning θ.

1.4 Sekvenser

Uttryck och mönstermatchning leder oss in på sekvenser. Med hjälp av våra tidigare implementeringar skall vi nu implementera så att vi kan evaluera

en sekvens. En sekvens respresenteras som en lista där de första elementen är mönstermatchande uttryck och de sista är vanliga uttryck. Evalueringen börjar med en tom omgivning som utökas medan vi fortsätter evaluera listan.

Varje mönstermatchande uttryck evalueras i två steg. Det högra uttrycket evalueras och returnerar en datastruktur. Det vänstra uttrycket matchas mot datastrukturen, vilket resulterar i en utökad miljö. En viktig del av detta är att innan vi kan utföra mönstermatchningen måste vi skapa en ny omgivning som är likadan som den befintliga. Med undantaget att vi har tagit bort bindningar för alla variabler som vi hittar i mönstret.

Vi skall implementera tre nya funktioner `eval_seq/2`, `eval_scope/2` och `extract_vars/1`. Funktionen `extract_vars/1` skall returnera en lista av alla variabler i det givna mönstret.

Vi skall nu titta närmare på ett fall av `eval_seq/2`. Koden är följande:

```
def eval_seq([{:match, ptr, exp} | tail], env) do
    case eval_expr(exp, env) do
      :error ->
        :error
      {:ok, str} ->
        env = eval_scope(ptr, env)
        case eval_match(ptr, str, env) do
          :fail ->
            :error
          {:ok, env} ->
            eval_seq(tail, env)
        end
    end
end
```

`eval_seq/2` tar emot en sekvens och en omgivning. Sedan evaluerar vi det första uttrycket i listan. Om detta fungerar anropar vi `eval_scope/2` vars syfte är att ta bort bindningen som ges av `ptr` i vår omgivning. Sedan anropar vi `eval_match/3` med det givna mönstret och den nya omgivningen. Slutligen gör vi ett rekursivt anrop på `eval_seq/2` med nästa uttryck i listan. Detta sker tills vi har evaluerat hela sekvensen.

2 Tillägg

2.1 Case uttryck

Vår interpretator har nu förmågan att hantera sekveneser av uttryck. Dessa uttryck är simpla därför skall vi utöka interpretatorn så att den kan hantera *case uttryck*. Interpretatorn skall utökas så att den har möjlighet att evaluera olika datastrukturer beroende på stadiet av exekveringen.

Ett case uttryck består av ett uttryck och en lista av klausuler där varje klausul är ett mönster och en sekvens. Ett case uttryck kommer att representeras som:

```
{:case, uttryck, klausul}
```

Vi skall implementera funktionen `eval_cls/3` som tar emot en lista av klausuler, en datastruktur och en omgivning. Koden för en del av denna funktion är följande:

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

Vi utför mönstermatchning på den första klausulen. Istället för att använda den givna omgivningen anropar vi `eval_scope/2` med mönstret och den givna omgivningen. Vi tar alltså bort bindningen som mönstret implicerar i vår omgivning. Om mönstermatchningen inte lyckas anropar vi `eval_cls/3` rekursivt tills vi lyckas. När vi väl har lyckats anropar vi `eval_seq/2` för att evaluera sekvensen.

2.2 Lambda uttryck

Ett *lambda uttryck* består av en sekvens parametervariabler, en sekvens fria variabler och en sekvens av uttryck. Ett lambda uttryck representeras i Elixir som:

```
{:lambda, parametervariabler, friaVariabler, omgivning}.
```

När vi evaluerar uttrycket måste vi utöka vår uppsättning av datastrukturer. Vi inför därför en datastruktur *closure* som ser ut som följande:

```
{:closure, parametervariabler, friaVariabler, omgivning}
```

För att kunna evaluera ett lambda uttryck måste vi utöka funktionerna som vår Env-modul har. Den måste ha möjligheten att skapa en ny omgivning från en lista av variabel-identifierare och en existerande miljö.

Denna utökning av omgivningen används i ett fall av `eval_expr/2` som behandlar lambda uttryck. Vi behöver endast utöka `eval_expr/2` med ett ytterligare fall. Denna funktion kan antingen returnera `:error` eller

```
{:ok, {:closure, parametervariabler, sekvens, klausul}}
```

I det fallet att en klausul returneras kommer vi att använda den i en funktion som tillämpar klausulen på ett uttryck som består av argument.

Detta tillämpas i ett annat fall av `eval_expr/2` som tar emot tupeln {`:apply, uttryck, argument`} och en omgivning. Om denna funktion returnerar en klausul kommer vi anropa `eval_args/2` som returnerar en lista av strukturerna. Denna lista kommer att användas när vi skapar nya bindningar till alla strukturer i klausulen. Detta implementeras också i Env-modulen.

Slutligen anropar vi `eval_seq/2` som tar emot sekvensen och den utökade omgivningen. När detta är evaluerat har vi evaluerat ett lambda uttryck.

2.3 Namngivna funktioner

Vi skall ge vår interpretator möjligheten att evaluera namnginva funktioner. Varje namngiven funktion kommer ha en Elixir funktion som returnerar representationen av argumenten och dess sekvens. Vi lägger till termen {`:fun, id`} som indikerar att det är en funktion.

Slutsats

Slutligen ser vi att det är fullt möjligt att konstruera en interpretator i Elixir. Det kräver dock en hel del förkunskap och varsam eftertanke. Men om man arbetar systematiskt med en förståelse för vad som skall lösas kommer man relativt lindrigt undan.

Vi började med att implementera en omgivning som är relativt simpel. Den kan modifiera bindningar på olika vis.

Sedan implementerade vi evaluering av uttryck och mönster. Där fick vi en djupare förståelse för hur omgivningen bör behandlas med hjälp av satslogiken. Dessa två implementeringar gav oss möjligheten att evaluera sekvenser. Evalueringen av sekvenser är en påbyggnad på det vi gjorde tidigare och gick smidigt att implementera.

Till sist implementerade vi case uttryck, lambda uttryck och namngivna funktioner. Uttrycken som vi arbeteade med innan detta var enkla. Där av utökade vi interpretatorn så att den kan evaluera olika datastrukturer beroende på stadiet av exekveringen.