Arthur Andrade Ilario
NUSP: 10432515

Exercício Prático 3

PCS3556
13 de Março de 2021

Introdução

Essa atividade coloca alguns dos conceitos explorados em sala de aula em prática. O objetivo é implementar um algoritmo de simulação de autômato determinístico e não-determinístico.

Sobre o artigo de referência

Muitos dos métodos e abordagens usados neste exercício programa se inspirarem no conteúdo presente no artigo de referência[1]. Com isso, a abordagem para definir um autômato foi a de um "processo que codifica o sistema de transições inteiro e especifica uma unidade de comunicação".

Assim existem 2 programas principais, um responsável por representar o sistema de transições e executá-las e outro para representar uma unidade de comunicação, responsável por gerenciar a fita de entrada.

Sobre o projeto

Os arquivos estão organizados em 2 diretórios principais: *lib* e *test*. O código fonte para o programa se encontra dividido em três arquivos diferentes: */lib/automata.ex*, responsável por auxiliar na criação de autômatos e facilitar testes, */lib/state_machine.ex*, responsável por implementar a macro que gerará os métodos que codificam o sistema de transições da máquina de estado, e */lib/tape.ex*, responsável por implementar a unidade de comunicação que o estado da fita. A suíte de testes se encontra nos arquivo */test/tape_test.exs* e */test/automata_test.exs*.

1 Abordagem e implementação

Para realizar as tarefas computacionais propostas, foi utilizada a linguagem funcional Elixir (1.11.2).

Como citado nas seções anteriores, o programa possui 3 módulos diferentes, um responsável pela unidade de comunicação, outra para implementar o autômato e um para auxiliar nos testes com o autômato.

1.1 Unidade de comunicação

A unidade de comunicação é responsável por gerenciar o estado da fita. A interface e estrutura de dados adotada para esse módulo tentou se aproximar ao máximo da utilizada no artigo de referência. Dessa forma, temos quatro métodos principais:

- *init* - Inicializa a fita t
- *at* - Retorna uma cópia do elemento no cabeçote da fita
- *reconfig!* - Move o cabeçote para a direita
- *contents* - Retorna todo conteúdo da fita

O módulo é implementado usando um *Agent*, isto é, uma estrutura própria da linguagem que gerencia o estado de uma variável em um processo paralelo. Isso é uma ótima representação para uma unidade de comunicação, que deve apenas servir para gerenciar o estado da fita, que nesse caso é a nossa variável.

1.1.1 O método init

O método init talvez seja o mais essencial do módulo, já que ele inicializa o agente e permite então que todas as outras funções sejam executadas:

```
1   @doc """
2   Inicializa a fita t.
3   A fita tem o formato [left, right], onde:
4   left: elementos a esquerda do cabe ote
5   right: elementos a direita do cabe ote
6   O cabe ote estah posicionado no primeiro elemento da lista right
7   """
8   @spec init([String.t()]) :: on_start()
9   def init(content) do
10    # $ indica o comeco da fita
11    Agent.start_link(fn -> [["$"],content] end)
12  end
13  @spec init([String.t()], [String.t()]) :: on_start()
14  def init(left, right) do
15    # $ indica o comeco da fita
16    Agent.start_link(fn -> [left,right] end)
17  end
```

Embora a documentação da função já dê uma boa ideia do que está acontecendo, cabe explicar algumas coisas: a função *start_ link/1* recebe uma função que inicializa o estado do agente, isto é função que apenas retorna o valor inicial, sem receber qualquer argumento, e o retorno dessa função é um par :ok, PID, que indica a criação do agente foi bem sucedida e o identificador do seu processo (dentro da BEAM VM) que usaremos para alterar e resgatar o estado do agente. Além disso podemos ver que existem duas funções init, a primeira delas converte uma lista de símbolos(strings) no formato usual da fita e a segunda inicializa uma

fita a partir do formato usual (essa segunda função será útil mais adiante para clonar fitas e executar testes)

1.1.2 O método at

O método at retorna o elemento em que o cabeçote está atualmente, isto é, o primeiro elemento da lista da direita.

```
1   @doc """
2   Retorna uma copia do elemento lido pelo cabecote da fita
3   """
4   @spec at(pid) :: String.t()
5   def at(tape) do
6     # Retorna "$" para indicar o fim da fita
7     Agent.get(tape, &read_head(&1))
8   end
9   defp read_head([_head | tail]) do
10    t = hd(tail)
11    if t == [] do
12      "$"
13    else
14      hd(t)
15    end
16  end
```

A função possui um funcionamento simples, ela acessa o agente usando o seu PID, e executa a função *read_head/1* nele para extrair o estado no formato desejado, ou seja, apenas o primeiro elemento da lista da direita. Caso a lista esteja vazia, a função deve retornar o caractere "$"que representa o fim da fita

1.1.3 O método reconfig!

O método reconfig! atualiza a posição do cabeçote da fita, movendo uma célula para a direita. Isso equivale a mover o primeiro da lista da direita para o ultimo elemento da lista na esquerda:

```
1   @doc """
2   Move o cabecote para direita
3   """
4   @spec reconfig!(pid) :: atom
5   def reconfig!(tape) do
6     # Gera um erro caso a fita ja esteja vazia
7     Agent.update(tape, &move_right!(&1))
```

```
8   end
9   defp move_right!([head | tail]) do
10    [head ++ [hd(hd(tail))], tl(hd(tail))]
11  end
```

A função *move_right!* é executada para atualizar o estado do agente, recebendo o estado anterior e transformando no próximo estado. A exclamação ao final do nome da função indica que este método pode gerar um erro. Essa exceção acontece pois a função não verifica se já está no fim da fita, e caso não haja células restantes, não há como mover o cabeçote, retornando um erro.

1.1.4 O método contents

Esse método é o mais simples todos, retornando apenas o estado atual do agente, sem qualquer formatação

```
1   @doc """
2   Retorna todo conteudo da fita no formato [left, right]
3   """
4   @spec contents(pid) :: [[String.t()]]
5   def contents(tape) do
6     Agent.get(tape, &(&1))
7   end
```

Entretanto, em conjunto com a função *init/2*, esse método facilita a criação de uma fita nova a partir de uma fita que já existe, procedimento muito importante na execução de autômatos não-determinísticos.

1.2 Máquina de estados

A máquina de estados é responsável por codificar todo sistema de transição que especifica o autômato. Esse módulo funciona injetando um conjunto de métodos necessários para a execução do autômato, através de uma macro.

1.2.1 Macros e a *using* macro

Macros são estruturas especiais em Elixir que permitem que, durante a compilação, trechos de códigos sejam gerados programaticamente, é um exemplo de metaprogramação, e se aproxima do que um compilador faz, isto é, um programa que gera programas.

Esse módulo define uma macro do tipo *using*, que quando usada em um módulo, modifica ele injetando suas propriedades e os métodos definidos nela. Dessa forma, como será mostrado

nas próximas seções, esse módulo pode ser visto como um compilador de máquina de estados, que transforma uma descrição da máquina num módulo em elixir que a represente.

1.2.2 Inicialização e chamada da macro

A macro é inicializada recebendo 3 parâmetros que vão definir a máquina de estados: o estado inicial, a lista de estados finais e a lista de transições.

```
  defmacro __using__(opts) do
    # Le todos os parametros
    initial = Keyword.get(opts, :initial)
    final = Keyword.get(opts, :final)
    transitions = Keyword.get(opts, :transitions)
```

Um exemplo de chamada para essa macro pode ser vista no exemplo abaixo:

```
def MySimpleAutomata do
  use StateMachine, initial: :q1, final: [:q3],
   transitions: [
     q1: {"a", [:q2]},
     q1: {"b", [:q3]},
     q2: {"a", [:q2]},
     q2: {"b", [:q3]}
   ]
end
```

Para representar os estados da máquina foi escolhida a estrutura de dados atom do elixir, já que essas estruturas representam valores únicos e que não podem ser confundidos, por exemplo, com uma string. Uma máquina de estados pode possuir mais de um estado final, dessa forma usamos uma lista de estados (atoms) para representá-los. Finalmente, para representar as transições, usamos uma lista de Keywords, isto é, para cada transição temos o estado de origem, o símbolo de transição e a lista de próximos estados(para uma máquina de estados não determinística por exemplo).

1.2.3 Geração de transições

Uma função de transição recebe o estado atual e o símbolo lido da fita e retorna o próximo estado (ou o conjunto de próximos estados) para aquela entrada. Para cada transição que a macro recebe como parâmetro é criada uma função que a represente. Por exemplo:

```
  # Para a seguinte transicao
  t = {q1: {"a"}, [:q2]}
  # Eh criada a seguinte funcao
  def transition(:q1, "a"), do: [:q2]
```

Essa geração é feita dentro da macro nas seguintes linhas:

```
1  defmacro __using__(opts) do
2     ...
3     # Constroi as funcoes de transicao
4     transitions = Enum.map(transitions, &build_transition(&1))
5     ...
6     quote do
7       ...
8       # Insere as funcoes especificas que especificam o automato
9       unquote_splicing(transitions)
10      ...
11      def transition(_, _), do: :error
12    end
13 end
14
15 # Para cada transicao cria uma funcao que retorna o proximo estado
      para
16 # um conjunto (estado atual, transicao) de entrada
17  defp build_transition({state, {symbol, next_state}}) do
18    quote do
19      def transition(unquote(state),unquote(symbol)), do: unquote(
           next_state)
20    end
21  end
```

Dentro da definição da macro vemos duas partes importantes. Primeiro na linha 4, geramos uma lista de funções, uma pra cada transição. Em seguida, dentro da construção do código que será injetado pela macro (estrutura quote do) a função *unquote_splicing/1* é usada para expandir a lista contendo as transições. A função *transition(_,_), do: :error* serve para os casos em que um estado não possui transição (como numa máquina não determinística, em que a sequência não deve ser aceitado caso não seja possível continuar consumindo a fita).

A função *build_transition/1* é a que realmente gera as funções de transição, lendo o estado de origem, o símbolo de transição e os estados de destino.

1.2.4 Geração de estados finais

Os estados finais são construídos como meras verificações booleanas, isto é, se o estado for final, a função retorna um valor verdadeiro, caso contrário, falso.

```
1  defmacro __using__(opts) do
2     ...
3     # Constroi as funcoes de estado final
```

```
4       final = Enum.map(final, &build_final(&1))
5       quote do
6         ...
7         unquote_splicing(final)
8         ...
9         def final?(_), do: false
10      end
11    end
12
13    # Constroi as funcoes que representam estados finais
14    defp build_final(state) do
15      quote do
16        def final?(unquote(state)), do: true
17      end
18    end
```

Similar a geração de transições, primeiro as funções são construídas e depois injetadas na macro. A função responsável por criar as funções finais é a *build_final/1*. A partir de um estado final ela gera a função *final?/1* que sempre retorna um valor verdadeiro para o estado desejado.

1.2.5 Execução da máquina de estados

Os métodos que permitem a execução da máquina de estados em cima de uma fita também são definidos nessa macro

```
1       # Roda o automato com uma cadeia, iniciando pelo estado inicial
2       def run(tape) do
3         run(tape, [unquote(initial)])
4       end
5       defp run(_, :error) do
6         false
7       end
8       defp run(tape, states) do
9         # Le o simbolo atual
10        t = Tape.at(tape)
11
12        # Se for o final da fita, confere se estah num estado de
              aceitacao
13        if t == "$" do
14          Enum.any?(states, &final?(&1))
15        else
16          # Move o cabecote
```

```
17            Tape.reconfig!(tape)
18
19            # Caso houver mais de um caminho possivel, executa ate que
                 um aceite a fita
20            # ou esgotar a fita
21            if length(states) > 1 do
22              Enum.any?(states, &clone_run(tape, &1, t))
23            else
24              run(tape, transition(hd(states), t))
25            end
26          end
27        end
28
29        # Clona o estado da fita atual e roda uma nova simulacao
30        defp clone_run(tape, state, t) do
31          [left, right] = Tape.contents(tape)
32          {:ok, tape} = Tape.init(left, right)
33          run(tape, transition(state, t))
34        end
35      end
```

A primeira função *run/1* é a que deve ser chamada para executar a máquina de estados, os próximos métodos são internos ao módulo. Esse método deve receber apenas a referencia para ser usada na leitura da fita e inicia a execução da máquina no estado inicial

Em seguida temos o método essencial *run/2*. Ele recebe tanto o estado atual da máquina quanto o conjunto de próximos estados(para uma máquina não determinística por exemplo). Primeiro é realizada a leitura do cabeçote da fita, caso seja o fim da fita, a função checa se o estado atual é algum dos estados finais, usando a função final?/1, caso contrário, move o cabeçote e continua realiza uma recursão para o próximo estado da máquina com aquela transição. Se a máquina for não-determinística (isto é, se houver mais de um possível próximo estado), a função verifica se pelo menos um dos possíveis caminhos aceita a sequência, chamando a função *clone_run/3*.

A função *clone_run/3* por sua vez clona a fita, combinando os métodos contents e init, e executa a simulação com o próximo estado que lhe foi dado.

2 Testes

Os testes são divididos em dois grupos: os testes da unidade de comunicação e os testes do autômato. Os testes da unidade de comunicação se encontram no arquivo */test/tape_test.exs* e os testes do automato se encontram no arquivo */test/automata_test.exs*.

2.1 Testes da unidade de comunicação

Quatro testes diferentes são realizados com a unidade de comunicação:

1. Teste de existência do módulo: apenas verifica se o módulo existe, e portanto, se foi importado corretamente
2. Teste de criação de fita: testa se a fita é criada no formato esperado, ou seja, uma lista com duas listas, uma contendo os elementos a esquerda do cabeçote(inicialmente apenas "$", que indica o começo da fita) e outra contendo os elementos a direita do cabeçote.
3. Teste de leitura da fita: verifica se o item lido pela função *at/1* é o item correto.
4. Teste de movimentação: verifica se o cabeçote se move como esperado, ou seja, se após a chamada da função *reconfig!/1* o primeiro elemento da lista da direita se torna o último da lista da esquerda.

2.2 Testes de autômato

Para realizar os testes do autômato o módulo Automata foi usado pra auxiliar na construção da máquina de estados

```
1  defmodule Automata do
2    @type transition :: {atom, {String.t(), atom}}
3    @type config :: {atom, [transition], [atom]}
4    @spec builder(config) :: boolean
5    def builder({init_state, transitions, final_states}) do
6      random_seed = :rand.uniform(100000)
7      module_name = String.to_atom("AutomataStateMachineModule#{
       random_seed}")
8      ast = quote do
9        use StateMachine, initial: unquote(init_state), final: unquote(
             final_states), transitions: unquote(transitions)
10     end
11     Module.create(module_name, ast, Macro.Env.location(__ENV__))
12     module_name
13   end
14 end
```

Esse módulo recebe os 3 parâmetros que a macro receberia, e compila durante a execução do programa um novo módulo, dinamicamente nomeado, que representa o autômato em questão.

Foram executados 5 testes com esse módulo:

1. Teste de existência do módulo: apenas verifica se o módulo existe, e portanto, se foi importado corretamente
2. Teste de estado final: testa se a função final? retorna um valor verdadeiro para qualquer um dos estados finais.
3. Teste de autômato determinístico: verifica se ao criar um autômato determinístico ele aceita uma cadeia que deveria aceitar e rejeita uma que deveria rejeitar.
4. Teste de autômato não-determinístico: verifica se ao criar um autômato não-determinístico ele aceita uma cadeia que deveria aceitar e rejeita uma que deveria rejeitar.
5. Teste de rejeição por ausência de transições: verifica se uma cadeia é rejeitada quando o estado atual não tem como consumir mais uma entrada.

3 Próximos passos

Ao executar a suíte de testes é possível observar que os tempos de execução dos testes giram em torna de 30ms. Esse tempo mais elevado ocorre devo ao processo de compilação do módulo que ocorre durante a execução do teste. Uma solução para acelerar a execução dos testes seria compilar um módulo de teste para cada caso de teste, usando diretamente a macro e não o módulo auxiliar Automata.

Além disso, este programa não implementa transições ϵ. Para implementá-las seria necessário adicionar mais uma transição para ser testada antes de chamar a função Enum.any(). Infelizmente isso pioraria a execução de recursão de cauda. Além disso, existe a possibilita de alguma dessas transições levar a um caminho redundante e infinito, no qual o programa nunca terminaria. Uma outra solução seria transformar o automato não-determinístico com transições ϵ em um sem transições ϵ, eliminando a necessidade de tratar esse caso.

Referências

[1] C. Wagenknecht, D. P. Friedman, *Teaching Nondeterministic and Universal Automata using Scheme*, Computer Science Department, Indiana University. 2 de Outubro de 1996