Terceiro Exercício Programa

Eric Rodrigues Pires, Mateus Nakajo de Mendonça

Nesse relatório, criamos um simulador de autômatos finitos determinísticos e não-determinísticos em Elixir.

***Palavras-chave*—Elixir, programação funcional, autômato finito determinístico, autômato finito não-determinístico.**

I. Introdução

NESTE exercício, tivemos como objetivo construir um programa capaz de determinar se uma cadeia pertence a uma linguagem aceitada por um autômato finito determinístico ou por um autômato finito não-determinístico. Um autômato finito determinístico *M* é definido como:

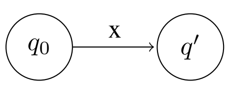
$$M = (Q, \Sigma, \delta, q_0, F) \tag{1}$$

onde Q é o conjunto de estados, Σ é o alfabeto de entrada ($\epsilon \notin \Sigma$), δ é a função de transição de estados, q_0 é o estado inicial, e F é o conjunto de estados de aceitação.

A função de transição de estados é da forma:

$$\delta(q, x) = q' \tag{2}$$

Ela indica para dados estado atual *q* e símbolo de entrada *x*, o próximo estado será *q'*. Visualmente, podemos representar essa transição por:



Uma cadeia é aceita por um autômato finito determinístico se ao final de todas as transições de estados, o estado final for de aceitação.

Um autômato finito não-determinístico é definido da mesma forma, mas a função de transição de estados é diferente. Podem existir transições da forma:

$$\delta(q, x) = \{q_1, q_2, ..., q_k\} \tag{3}$$

Ou seja, se o estado atual for *q* e o símbolo de entrada *x*, o próximo estado pode ser qualquer $q_i \in \{q_1, q_2, ..., q_k\}$. Além disso, podem existir transições da forma:

$$\delta(q, \epsilon) = q' \tag{4}$$

Onde ϵ é a cadeia vazia.

É possível provar que para todo autômato finito não-determinístico existe um autômato finito determinístico que define a mesma linguagem [5].

II. Desenvolvimento

Usamos a linguagem de programação Elixir para desenvolver o algoritmo. Por se tratar de uma linguagem funcional, escrevemos os *loops* como recursões de cauda. Utilizamos várias funções da biblioteca padrão `Enum`, como `map/2`, `reduce/3`, `sort/1`, `filter/2`, `all?/2`, `count/1`, dentre outras. Essas funções foram muito úteis para manipular dados em listas e simplificaram bastante o código.

1) Modelagem

Ao longo do nosso programa, os autômatos são representados por uma estrutura `%Automaton{}` composta por uma função de transição, um estado inicial e estados de aceitação. As transições são representadas por tuplas de três elementos: estado atual, símbolo de entrada e próximo estado. Transições com ϵ são representadas pelo símbolo *nil*. As cadeias são representadas por listas de símbolos. Uma cadeia vazia é representada por uma lista vazia.

As duas funções principais do nosso programa são `dfa_generates_word?/2` e `nfa_generates_word?/2`. A primeira retorna se uma dada cadeia pertence a linguagem gerada por um autômato determinístico finito dado. A segunda faz a mesma coisa mas para um autômato não-determinístico. Ela faz uso internamente da função `convert_nfa_to_dfa/1`, que recebe um autômato não-determinístico finito e o converte para um determinístico.

2) Algoritmo

A lógica para determinar se uma cadeia pertence à linguagem gerada por um autômato finito determinístico foi a seguinte:

1) A partir do estado inicial $q0$, achar o próximo estado para o primeiro símbolo da cadeia de entrada.
2) Repetir esse processo até consumir toda a cadeia
3) Verificar se o estado final é um estado de aceitação.

Devido à simplicidade deste algoritmo e que cada passo é determinado apenas pelos valores atuais, utilizamos uma função `Enum.reduce/3` para realizar o cálculo do estado final.

Para o caso de autômatos finitos não-determinísticos, primeiramente os convertemos em determinísticos e em seguida aplicamos o algoritmo mostrado acima.

Para converter o autômato finito não-determinístico em determinístico, aplicamos o seguinte algoritmo:

1) Determinar o alfabeto a partir das transições do autômato não-determinístico, ignorando transições vazias `nil`.
2) Calcular $\delta^*(q_0, \epsilon) = \{q_0, K\}$, ou seja a função de transição de estado estendida para o estado inicial e a cadeia vazia. $\{q_0, K\}$ será o estado inicial do autômato determinístico.
3) Localizar nas novas regras de transição os estados que podem ser alcançados e ainda não foram considerados.
4) Para cada estado $\{q_i, q_j, ..., q_m\}$, calcular $\delta^*(q_i, a) \cup \delta^*(q_j, a) \cup ... \cup \delta^*(q_m, a) = \{q'_k, q'_l, ..., q'_n\}$, e adicionar nas regras de transição.

5) Repetir os passos 3 e 4 para cada estado do autômato determinístico do lado direito das transições e símbolos do alfabeto, até não haver mais estados a serem adicionados no autômato determinístico.
6) Para cada estado $\{q_i, ..., q_j, ..., q_m\}$, ele será um estado de aceitação se algum q_j no autômato não-determinístico for de aceitação.

Outra alternativa para simular um autômato finito não-determinístico seria calcular todos os possíveis estados finais para a cadeia de entrada, e em seguida verificar se algum deles é um estado de aceitação. Esse processo é mais complexo do que no caso determinístico.

Sendo assim, avaliamos que é mais prático de um ponto de desenvolvimento fazer um pré-processamento na forma de converter o autômato finito não-determinístico para determinístico, e então todas as cadeias a serem analisadas para aquele autômato podem usar o algoritmo já desenvolvido.

3) Testes

Realizamos três tipos de testes. No primeiro, verificamos se o autômato determinístico aceita apenas as cadeias pertencentes à linguagem definida por ele, no caso, a linguagem da expressão regular $a^*|(a^*b^*c^*)^*$. Repetimos esse teste para um autômato não-determinístico que aceita cadeias da linguagem definida por $0^*|(01)^*$. No último teste, utilizando o conceito de testes baseados em propriedades, verificamos se a função de conversão de autômato não-determinístico para determinístico funciona corretamente – isto é, se os resultados do autômato não-determinístico são idênticos aos do determinístico.

III. Resultados

Os testes realizados bateram com o esperado, de acordo com a especificação do exercício. Assim, o algoritmo foi capaz de simular autômatos finitos determinísticos e não-determinísticos. A função de conversão de autômatos não-determinísticos para autômatos determinísticos também funcionou corretamente.

IV. Conclusão

Esse exercício programa nos permitiu ampliar e desenvolver nossos conceitos em matemática discreta sobre gramáticas, autômatos e pela linguagem Elixir, melhorar nossos conhecimentos de programação funcional. Quanto ao programa desenvolvido, avaliamos que conseguimos simular o funcionamento de autômatos finitos de forma precisa. Porém, para melhor eficiência, o ideal seria deixar a conversão de autônomo não-determinístico para determinístico na mão do usuário, para evitar o recálculo do autômato determinístico equivalente para cada cadeia nova a ser testada. Mesmo assim, os testes que criamos nos deram confiança a respeito da corretude do algoritmo implementado. Observamos também que o algoritmo para simular um autômato finito determinístico é bastante simples. Sua implementação foi facilitada pelos recursos de linguagem funcional do Elixir.

Referências

[1] ALMEIDA, Ulisses. *Learn Functional Programming with Elixir*. The Pragmatic Bookshelf, 2018.

[2] FORD, Neal. *Functional Thinking*. Disponível em: https://www.infoq.com/presentations/Functional-Thinking. Acesso em 7 de fevereiro de 2018.

[3] Elixir. *Introduction*. Disponível em: https://elixir-lang.org/getting-started/introduction.html. Acesso em 7 de fevereiro de 2018.

[4] Elixir. *Enum*. Disponível em: https://hexdocs.pm/elixir/Enum.html Acesso em 24 de fevereiro de 2018.

[5] BUSCH, Konstantin. *Nondeterministic Finite Automata*. Disponível em: http://www.csc.lsu.edu/ busch/courses/theorycomp/slides/NFA.ppt. Acesso em 20 de março de 2018.

[6] Elixir. *Structs*. Disponível em: https://elixir-lang.org/getting-started/structs.html Acesso em 22 de março de 2018.