Primeiro Exercício Programa

Eric Rodrigues Pires, Mateus Nakajo de Mendonça

Nesse relatório, demonstramos como pudemos desenvolver um algoritmo de fecho reflexivo e transitivo de uma relação binária através dos conceitos aprendidos em aula.

***Palavras-chave*—Elixir, relação binária, fecho, reflexão, transitividade, recursão.**

I. Introdução

Esse exercício programa tem como objetivo determinar o fecho transitivo e reflexivo de uma relação dada. O fecho transitivo e reflexivo de uma relação é o menor conjunto tal que sua união com a relação resulta em uma relação transitiva e reflexiva. Usaremos a linguagem de programação Elixir para seu desenvolvimento.

A linguagem de programação Elixir é uma linguagem de propósito geral funcional, dinâmica e que roda na máquina virtual Erlang. Com Elixir, é possível criar aplicações tolerantes a falhas e distribuídas. Nesse projeto, usamos o ferramenta Mix, que facilita a criação, gerenciamento e teste de projetos escritos em Elixir. A linguagem tem preocupação especial com a documentação do código. Existem nativamente funções de acesso e geração de documentação e os exemplos são testados tais como o código, a fim de evitar inconsistência.

Ao longo deste projeto, usaremos recursos da linguagem como *pattern mathing* e recursão. *Pattern mathing* é um mecanismo para extrair valores de uma estrutura de dados que satisfaçam um padrão. Recursão – no contexto de programação – é o processo de usar uma função para definir a si própria.

II. Desenvolvimento

Para este projeto, decidimos evitar o uso de quaisquer funções da biblioteca padrão do Elixir. Alternativamente, como será apresentado na seção "Algoritmo", optamos por manualmente desenvolver funções auxiliares pelos princípios de recursão e *pattern matching*.

1) Modelagem

Criamos o nosso módulo do Elixir, `RTC`, que irá criar um fecho reflexivo e transitivo (em inglês, "*reflexive transitive closure*") a partir de uma relação binária R e um conjunto A tal que $R \subseteq A \times A$. Para isso, precisamos definir como será a modelagem de relações binárias e conjuntos no nosso sistema.

Para fins de simplicidade, todas as estruturas serão representadas por listas, apesar de matematicamente não terem ordem. Portanto, o algoritmo não poderá fazer distinção entre dois conjuntos com mesmos elementos em ordens diferentes.

Assim, decidimos utilizar uma lista de elementos para o conjunto, uma lista de duplas ordenadas para a relação binária, e qualquer símbolo do Elixir como elementos – embora utilizaremos apenas átomos nos nossos testes. Por exemplo:

```
relation = [{:a, :b}, {:b, :c}]
set = [:a, :b, :c]
```

2) Algoritmo

A ideia que tivemos para o funcionamento do algoritmo é a de simplesmente varrer, um a um, os elementos do conjunto A. Para cada elemento, varreremos a relação de entrada, e verificaremos as tuplas onde o primeiro equivale ao elemento iterado atualmente, verificando então os elementos atingidos a partir desse. Para evitar loops infinitos (por exemplo, em um grafo com ciclos), interromperemos as iterações quando retornarmos a algum valor alcançado anteriormente. Em pseudo-código, podemos descrever isso como:

```
fecho_reflexivo_transitivo(relação, conjunto):
 fecho = []
 para cada elem do conjunto:
  fecho += obter_elementos_atingidos(elem, [elem])
 retornar fecho

obter_elementos_atingidos(atual, caminho):
 resultado = []
 para cada dupla da relação:
  {x, y} = dupla
  se x == atual:
   se y está em caminho ou y não está no conjunto:
    continue
   senão:
    resultado += obter_elementos_atingidos(
      atual, caminho ++ [y])
 resultado += {caminho[0], atual}
 retornar resultado
```

Esse código obterá todos os elementos alcançáveis para cada elemento do conjunto e os unirá em uma única lista "fecho" a ser retornada. Em outras palavras, retorna o fecho transitivo da relação. Como a lista ainda inclui {elem, elem} para todos os elementos do conjunto, o fecho será também reflexivo.

Há, porém, três detalhes no código que desenvolvemos que diferem deste pseudo-código:

a. *Será utilizada recursão ao invés de iteração*. Isto é simples; dado que todos os elementos iterados (a relação e o conjunto) são listas, podemos iterar utilizando pattern matching:

```
iterar(lista):
 case lista do
  [] -> []
  [head | tail] -> funcao(head) ++ iterar(tail)
   # funcao(x) retornará uma lista parcial
 end
```

b. *Será utilizado pattern matching ao invés de condições/funções*. Especificamente, realizaremos as seguintes trocas:

```
{x, y} = dupla
se x == atual:
 ...
# Trocar por:
case dupla do
 {^atual, y} -> ...
end
# ---------------------- #
se y está em caminho:
 ...
# Trocar por:
se contém(caminho, y):
 ...

defp contém(lista, elem) do
 case lista do
  [] -> false
  [^elem | _] -> true
  [_ | tail] -> contém(tail, elem)
 end
end
# ---------------------- #
caminho[0]
# Trocar por:
primeiro(caminho)

defp primeiro(lista) do
 case lista do
   [head | _] -> head
 end
end
```

c. *Queremos retornar um fecho ordenado e com elementos únicos*. Atualmente, como apresentado acima, o código retornará um fecho cujas duplas variam em ordem e quantidade de acordo com a relação de entrada. Por exemplo, haverá múltiplos elementos {:a, :b} se houver múltiplos caminhos de :a até :b; e a ordem das tuplas {:a, _} depende da ordem delas na relação original. Além disso, a ordem do conjunto modifica também quais conjuntos de duplas aparecem primeiro. Para minimizar os efeitos que a ordem da relação e do conjunto impõe, o resultado final será ordenado e terá duplicatas removidas. Existem funções para isso na biblioteca padrão do Elixir, mas desejamos garantir o uso de recursão. Para isso, criamos duas funções auxiliares simples, unique() e sort(), sobre as quais o resultado final será aplicado antes de ser retornado ao usuário. Assim, o resultado do fecho reflexivo transitivo será previsível.

Desta forma, aprofundando-se sobre o pseudo-código e com a sintaxe do Elixir, criamos a função `RTC.of(relação, conjunto)`, que realiza o fecho reflexivo transitivo da relação sobre o conjunto.

3) Testes

Criamos os casos de teste do nosso programa, com entradas e saídas esperadas. A ideia é cobrir o máximo de condições de borda para verificar o funcionamento total do nosso módulo. Sendo assim, definimos as seguintes propriedades para várias entradas:

- *Relação ou conjunto vazio*. Para relação vazia, será retornado o fecho simétrico sobre o conjunto. Para conjunto vazio, o fecho será sempre vazio.
- *Dupla da relação com elemento fora do conjunto*. Se o elemento estiver do lado esquerdo da dupla, não iteraremos sobre o elemento nunca. Se estiver do lado direito, precisamos ignorar esta dupla.
- *Ciclos na relação*. Este caso pode gerar loops infinitos, então verificaremos que ele roda.
- *Múltiplos caminhos a um mesmo elemento*. O caso em que podemos chegar a um único elemento por múltiplos caminhos pode gerar elementos duplicados.
- *Grafos de relação desconexos*. Deve gerar o mesmo valor de duas chamadas desconexas do fecho.
- *Relação já reflexiva/transitiva*. A reflexividade/transitividade não pode ser perdida.
- *Elemento inalcançável por outro*. Não haverá transitividade do segundo elemento ao primeiro.
- *Um único elemento no conjunto*. O fecho será a bijeção do elemento sobre ele mesmo.
- *Tipos diferentes de elementos no conjunto*. A única coisa que levará os tipos em consideração é a ordenação no final. Tirando isso, o valor dos elementos não possui importância.

Construímos os testes tentando cobrir estas propriedades e outros casos que nos ocorreram, de relações sem condições de borda.

III. Resultados

Ao término desse exercício programa, conseguimos desenvolver um algoritmo de fecho reflexivo e transitivo, em `rtc/lib/rtc.ex`. Os casos de teste foram criados no arquivo `rtc/test/rtc_test.exs`, e preveem várias situações as quais o programa responderá. Executando o comando `mix test` na pasta `rtc`, todos os testes passaram, corroborando o correto funcionamento do programa.

IV. Conclusão

Esse exercício programa nos permitiu exercitar conceitos importantes de matemática discreta. Usamos a definição de relações transitivas e reflexivas para criar o algoritmo para calcular o fecho. A linguagem Elixir foi uma ótima ferramenta para essa tarefa. Ela nos deu ferramentas de programação funcional (*pattern matting*, recursão) que foram adequadas para a resolução do problema. Aprendemos muito sobre programação funcional com esse exercício. O novo paradigma nos forçou a abandonar estruturas de programação como *loop* e variáveis mutáveis, que são bastante comuns em linguagem imperativas. No lugar delas, pensamos em recursão e imutabilidade. Esse aprendizado será valioso para projetos futuros.

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.