Arthur Andrade Ilario
NUSP: 10432515

Exercício Prático 1

Introdução

Essa atividade pretende colocar alguns dos conceitos explorados em sala de aula em prática. O objetivo é implementar um algoritmo de fecho reflexivo e transitivo de uma relação binária $R \subseteq A \times A$ sobre um conjunto finito A que é descrita por meio de um grafo direcionado. Dessa forma o problema pode ser dividido em duas etapas, encontrar o fecho reflexivo e encontrar o fecho transitivo.

Fecho reflexivo

O fecho reflexivo de uma relação binária sobre A é o menor subconjunto que contém R e possui propriedade reflexiva. Matematicamente, podemos enxergar o fecho como:

$$S = R \cup I_A, \quad I_A = \{(a,a)|a \in A\}$$

Ou seja, é a união entre a relação original e a relação identidade do conjunto A. Realizar essa operação (união com a identidade) é uma maneira simples de encontrar o fecho.

Fecho transitivo

O fecho reflexivo de uma relação binária sobre A é o menor subconjunto de A que contém R e possui propriedade transitiva. A propriedade transitiva garante que:

$$\forall a,b,c \in A : (a,b) \in R \land (b,c) \in R \Rightarrow (a,c) \in R$$

Para construir o fecho de R podemos ir incluindo novos pares transitivos, que podem ser construídos a partir da composição $R \circ R^i$ onde $R^i = R \circ R^{i-1}$ e $R^1 = R$. Assim temos uma construção recursiva do fecho transitivo. Matematicamente podemos resumir esse processo como:

$$S = \bigcup_{i=1}^{\infty} R^i$$

Na prática, só precisamos repetir o processo até que o resultado de uma nova iteração seja igual ao da anterior para obter o fecho transitivo.

Sobre o projeto

Os arquivos estão organizados em 2 diretórios principais: *lib* e *test*. O código fonte para o programa se encontra no arquivo */lib/fecho.ex* e a suíte de testes se encontra no arquivo */test/set_closure_test.exs*

1 Abordagem e implementação

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

Através de um único módulo, *SetClosure*, todas as operações necessárias para esse exercício seram realizadas. A abordagem utilizada foi pensada de maneira funcional: um método principal, *find_closure*, transforma a relação de entrada em uma relação que cumpre os critérios de um fecho reflexivo e transitivo. A transformação é realizada através de um *pipelining* da relação para a função *make_reflexive* e então para a função *make_transitive*.

```
def find_closure(relation, set) do
  relation |> make_reflexive(set) |> make_transitive
end
```

No trecho de código acima é possível observar claramente o fluxo de transformação pelo qual *relation* passa.

1.1 Estrutura de entrada

No enunciado do exercício uma representação como grafo direcionado é imposta. Porém o grafo pode ser representado de mais de uma forma (usaremos como exemplo a relação binária $R = \{(1,2),(2,3),(3,3)\}$):

1. Uma lista de listas - uma matriz binária onde cada elemento representa a existência (ou não existência) de um par a_i, a_j na relação, onde i, j são respectivamente os índices da linha e da coluna na matriz, e a_n é o elemento no n-ésimo índice do conjunto A
2. Uma lista de pares - cada elemento na lista representa um par que a relação possuí
3. Um mapa de inteiros para listas - cada chave representa o primeiro elemento de um par na relação e a lista subsequente representa o conjunto de números que ocupam a posição de segundo elemento no par

```
    1: [[false, true, false],
        [false, false, true],
        [false, false, true]]
    2: [{1,2},{2,3},{3,3}]
    3: %{1:[2], 2:[3], 3:[3]}
```

Para esta atividade foi escolhida a representação "lista de pares". Essa representação é vantajosa na visualização dos elementos da relação e para operações de união e subtração de conjuntos (já que é possível abusar dos operadores ++ e − para realizá-las em estruturas nesse formato). Um ponto negativo dessa estrutura é que a ordem dos elementos importa, diferente do que acontece para conjuntos, por isso é necessário tomar cuidado adicional na hora de realizar comparações (nos testes por exemplo).

1.2 Implementação do fecho reflexivo

A função *make_reflexive* é a primeira e a mais simples pela qual a relação passa. Nela é implementado o fecho reflexivo.

```
def make_reflexive(relation, set) do
  found = count(relation,[])
  needed = set -- found
  fill(relation, needed)
end
```

Como discutido na introdução a maneira mais simples de fazer isso é realizando a união com a identidade do conjunto A. Para realizar a operação de união precisamos primeiro identificar os elementos repetidos entre os dois conjuntos (identidade e relação) e subtraí-lo do conjunto identidade, para então concatenar o resultado com a lista original de pares que representa a relação.

Uma maneira mais eficiente de fazer isso é montar uma lista contendo os inteiros que possuem um par (n, n) na relação, como é feito na função auxiliar *count*:

```
defp count([{i, j} | tail], acc) do
  if i == j do
    count(tail, [i] ++ acc)
  else
    count(tail, acc)
  end
end
defp count([], acc) do
  acc
end
```

A lista resultante é subtraída do conjunto sob o qual a relação existe, resultando com uma lista dos elementos que não possuem um par reflexivo na relação. A partir dela construímos os pares que complementam a relação na construção do fecho reflexivo:

```
defp fill(relation, [head | tail]) do
  fill([{head, head}] ++ relation, tail)
end
defp fill(relation, []) do
  relation
end
```

1.3 Implementação do fecho transitivo

A função *make_reflexive* é a segunda e mais complexa (principalmente pelo maior nível de recursão) pela qual a relação passa. Nela é implementado o fecho transitivo.

```
def make_transitive(relation) do
```

```
2      tree = build_tree(relation, %{})
3      transitive_closure = make_transitive(tree, Map.keys(tree), [])
4      transitive_closure ++ relation
5    end
6
7    defp make_transitive(tree, [head | tail], closure) do
8      {closure, tree} = dfs(tree[head], head, tree, closure)
9      make_transitive(tree, tail, closure)
10   end
11
12   defp make_transitive(_tree, [], closure) do
13     closure
14   end
```

Na introdução estudamos um modo de implementar o fecho transitivo que era através da união recursiva de auto-composições da relação R. Essa ideia ainda é usada aqui, mas é mascarada como uma busca em profundidade (*dfs*: Depth-First Search). A ideia deste algoritmo é construir uma estrutura auxiliar em forma de um mapa de listas (discutida na seção 1.1) usando a função *build_tree*, e percorrer cada elemento do mapa em profundidade, adicionando os elementos restantes na ordem em que são encontrados. A função da linha 7 é responsável por percorrer essa nova estrutura.

Vamos observar o funcionamento da função de busca em profundidade:

```
defp dfs([head | tail], current, tree, acc) do
  # Complete current tree with remaining sibling connections
  if Map.has_key?(tree, head) do
    remaining = tree[head] -- tree[current]
    tail = remaining ++ tail
    tree = %{tree | current => remaining ++ tree[current]}
    acc = Enum.map(remaining, &{current, &1}) ++ acc
    dfs(tail, current, tree, acc)
  else
    dfs(tail, current, tree, acc)
  end
end
defp dfs([], _current, tree, acc) do
  {acc, tree}
end
```

A função executa recursivamente até que a pilha(primeiro argumento da função) contendo os elementos da chave atual se esgote. Isso pode ser visto como percorrer todas as duplas da relação R em que a chave é o primeiro elemento da dupla.

A cada passo da recursão realizamos uma operação de união. Seja R_n o conjunto de todos os elementos que fazem par com n à esquerda, seja i a chave da lista atual e j um dos elementos da sua lista. Matematicamente temos:

$$S = R_j - R_i$$

Os elementos de S representam os elementos que precisam fazer par com i para constituir uma relação transitiva.

Na função apresentada, *remaining* representa o S, e a cada passo esses elementos são adicionados a pilha, que forma que esgotamos a busca de cada chave antes de prosseguir para a próxima (buscando em profundidade). Além disso atualizamos tanto a estrutura auxiliar (*tree*) quanto nosso conjunto solução (*acc*), representando os elementos restantes para formar um fecho.

Vale também explicar a utilização do cláusula *if* para garantir que não vamos acessar uma chave que não existe, já que a estrutura auxiliar foi construída apenas com contexto da relação e não tem conhecimento de elementos que pertencem ao conjunto mas não estão à esquerda de nenhum par.

É interessante notar que a escolha da busca em profundidade é útil por utilizar uma pilha e não uma fila como a busca em largura, assim podemos expandir a pilha apenas adicionando um novo elemento em seu topo o que é bom para o esquema de listas ligadas que o elixir usa.

Finalmente, é importante entender que a função *Enum.map/2* percorre recursivamente um objeto enumerável, colocando cada elemento gerado a partir da função numa lista acumuladora.

2 Testes

Os testes para a função foram divididos em 3 conjuntos: testes do fecho reflexivo, testes do fecho transitivo e testes do fecho reflexivo e transitivo.

Para rodar a suíte de teste basta executar o comando *mix test* na pasta principal do projeto. Para adicionar novos testes, basta copiar o modelo dos testes já presentes no arquivo */test/set_closure_test.exs*

2.1 Primeiro conjunto

O primeiro conjunto de testes verifica se a função *make_reflexive* produz os resultados desejados para quaisquer entradas.

É importante notar que devido ao formato escolhido para a saída da função (vide seção 1.1) a ordem dos elementos importa. Para que isso não afete os testes, usaremos uma estratégia para nos ajudar: compararemos $S - R$, onde S é o resultado e R a relação original, com $I_A - R$, a relação identidade do conjunto base, já que $S = R \cup I_A \Rightarrow S - R = I_A - R$.

Testes realizados:

1. $R = \{(1,1),(1,4),(2,3),(4,5),(5,1)\} \quad A = \{1,2,3,4,5\}$
   $S = \{(1,1),(1,4),(2,2),(2,3),(3,3),(4,4),(4,5),(5,1),(5,5)\}$

2. $R = \{(1,1),(2,2),(3,3),(4,4),(5,5)\} \quad A = \{1,2,3,4,5\}$
   $S = \{(1,1),(2,2),(3,3),(4,4),(5,5)\}$

3. $R = \{(1,2),(2,3)\} \quad A = \{1,2,3,4\}$
   $S = \{(1,1),(1,2),(2,2),(2,3),(3,3),(4,4),(5,5)\}$

2.2 Segundo conjunto

O segundo conjunto de testes verifica se a função *make_transitive* produz os resultados desejados para quaisquer entradas.

É importante notar que devido ao formato escolhido para a saída da função (vide seção 1.1) a ordem dos elementos importa. Para que isso não afete os testes, usaremos uma estratégia para nos ajudar: primeiro verificaremos se a função modificou a entrada (caso isso fosse esperado) e se o resultado esperado subtraído do resultado calculado é igual ao conjunto vazio ($S_{esperado} - S = \emptyset$)

Testes realizados:

1. $R = \{(1,1),(1,4),(2,3),(4,5),(5,1)\} \quad A = \{1,2,3,4,5\}$
   $S = \{(5,5),(5,4),(4,4),(4,1),(1,5),(1,1),(1,4),(2,3),(4,5),(5,1)\}$

2. $R = \{(1,2),(2,3),(1,3)\} \quad A = \{1,2,3\}$
   $S = \{(1,2),(2,3),(1,3)\}$

2.3 Terceiro conjunto

O terceiro conjunto de testes verifica se a função *find_closure* produz os resultados desejados para quaisquer entradas, verificando também a integração das funções anteriores

É importante notar que devido ao formato escolhido para a saída da função (vide seção 1.1) a ordem dos elementos importa. Para que isso não afete os testes, usaremos uma estratégia para nos ajudar: primeiro verificaremos se a função modificou a entrada (caso isso fosse esperado) e se o resultado esperado subtraído do resultado calculado é igual ao conjunto vazio ($S_{esperado} - S = \emptyset$)

Testes realizados:

1. $R = \{(1,1),(1,4),(2,3),(4,5),(5,1)\} \quad A = \{1,2,3,4,5\}$
   $S = \{(5,5),(5,4),(4,4),(4,1),(1,5),(1,1),(1,4),(2,2),(2,3),(4,5),(5,1),(3,3)\}$

2. $R = \{(1,1),(1,2),(2,2),(2,3),(3,3),(1,3)\} \quad A = \{1,2,3\}$
   $S = \{(1,1),(1,2),(2,2),(2,3),(3,3),(1,3)\}$