Lógica Computacional 1
Descrição do Projeto
Equivalência entre noções distintas de permutação

11 de agosto de 2021

Profs. Flávio L. C. de Moura e Mauricio Ayala-Rincón
Estagiário de Docência: Gabriel Ferreira Silva

1 Introdução

A noção de permutação é essencial na prova da correção de algoritmos de ordenação. De fato, dizemos que um algoritmo de ordenação é correto se sempre retorna uma permutação da lista de entrada que esteja ordenada. Intuitivamente, dizemos que duas listas são permutações uma da outra quando possuem os mesmos elementos, mas não necessariamente na mesma ordem. Assim, se *num_oc x l* denota o número de ocorrências de x em l então a lista l_1 é uma permutação da lista l_2 se *num_oc x* l_1 = *num_oc x* l_2, para todo x. A função recursiva que conta o número de ocorrências de um elemento em uma lista é definida em Coq como a seguir:

```
Fixpoint num_oc n l  :=
  match l with
    | nil => 0
    | h :: tl =>
      if n =? h then S(num_oc n tl) else  num_oc n tl
  end.
```

Chamaremos de `permutation` a noção de permutação baseada na contagem do número de ocorrências de um elemento:

```
Definition permutation l l' := forall n:nat, num_oc n l = num_oc n l'.
```

Esta definição foi utilizada na formalização da correção do algoritmo `mergesort`[1]:

```
Function mergesort (l: list nat) {measure length l}:=
  match l with
  | nil => nil
  | hd :: nil => l
  | hd :: tail =>
     let n := length(l) / 2 in
     let l1 := firstn n l in
     let l2 := skipn n l in
     let sorted_l1 := mergesort(l1) in
     let sorted_l2 := mergesort(l2) in
     merge (sorted_l1, sorted_l2)
  end.
```

onde a função `merge` é definida a seguir:

[1] `https://github.com/ensino-unb/correcao-mergesort`

```
Function merge (p: list nat * list nat) {measure len p} :=
match p with
  | (nil, l2) => l2
  | (l1, nil) => l1
  | ((hd1 :: tl1) as l1, (hd2 :: tl2) as l2) =>
if hd1 <=? hd2 then hd1 :: merge (tl1, l2)
      else hd2 :: merge (l1, tl2)
   end.
```

e o teorema que garante que a lista retornada pelo algoritmo é uma permutação da lista de entrada:

```
Theorem mergesort_is_perm: forall l, permutation l (mergesort l).
```

No entanto, existem outras formas de definir o mesmo conceito. Por exemplo, você seria capaz de apresentar uma definição indutiva para a noção de permutação? As regras de inferência a seguir fazem isto.

$$\frac{}{\texttt{perm}\ l\ l}\ (\texttt{perm\_refl}) \qquad \frac{\texttt{perm}\ l\ l'}{\texttt{perm}\ (x :: l)\ (x :: l')}\ (\texttt{perm\_hd})$$

$$\frac{\texttt{perm}\ l\ l'}{\texttt{perm}\ (x :: y :: l)\ (y :: x :: l)}\ (\texttt{perm\_swap}) \qquad \frac{\texttt{perm}\ l\ l' \qquad \texttt{perm}\ l'\ l''}{\texttt{perm}\ l\ l''}\ (\texttt{perm\_trans})$$

A regra *perm_refl* é um axioma que estabelece a reflexividade do predicado *perm*. Já a regra *perm_hd* estabelece que se l e l' são permutações, então ao inserirmos os mesmos elementos na cabeça de ambas as listas continuamos obtendo permutações. Analogamente, a regra *perm_swap* nos diz que se l e l' são permutações então $x :: y :: l$ e $y :: x :: l'$ continuam sendo permutações uma da outra. Por fim, a regra *perm_trans* estabelece a transitividade deste predicado. A definição correspondente em Coq é dada a seguir:

```
Inductive perm: list nat -> list nat -> Prop :=
| perm_refl: forall l, perm l l
| perm_hd: forall x l l', perm l l' -> perm (x::l) (x::l')
| perm_swap: forall x y l l', perm l l' -> perm (x::y::l) (y::x::l')
| perm_trans: forall l1 l2 l3, perm l1 l2 -> perm l2 l3 -> perm l1 l3.
```

Esta noção foi utilizada na formalização da correção do algoritmo de ordenação por inserção[2]:

```
Fixpoint ord_insercao l :=
 match l with
   | nil => nil
   | h :: tl => insere h (ord_insercao tl)
 end.
```

onde a função `insere` é definida como a seguir:

[2] https://github.com/ensino-unb/correcao_ord_insercao

```
Fixpoint insere (n:nat) (l: list nat) :=
 match l with
 | nil => n :: nil
 | h :: tl => if n <=? h then (n :: l)
            else (h :: (insere n tl))
                     end.

Theorem ord_insercao_perm: forall l, perm l (ord_insercao l).
```

A pergunta que surge naturalmente neste contexto é: estas duas noções de permutação coincidem? O presente tem como objetivo responder afirmativamente a esta questão. Para isto, teremos uma prova do seguinte teorema ao final do projeto:

Teorema 1 *Sejam l e l' listas quaisquer. Então* (`perm` l l') *se, e somente se,* (`permutation` l l').

2 Descrição do Projeto

A prova do Teorema 1 será dividida em duas provas menores:

Lema 1 *Sejam l e l' listas quaisquer. Se* (`perm` l l') *então* (`permutation` l l').

e

Lema 2 *Sejam l e l' listas quaisquer. Se* (`permutation` l l') *então* (`perm` l l').

Nas subseções a seguir, detalharemos os passos para completarmos estas provas.

2.1 Questão 1

A primeira questão do projeto consiste em provar diretamente o Lema , cujo enunciado em Coq é dado como a seguir:

```
Lemma perm_to_permutation: forall l l', perm l l' -> permutation l l'.
```

Este lema pode ser provado por indução na hipótese de que a lista `l` é uma permutação da lista `l'`, ou seja, indução na hipótese `perm l l'`. Como o predicado `perm` é definido indutivamente, esta prova consiste em considerar separadamente cada uma das regras que compõem esta definição. Assim, teremos quatro casos a considerar:

1. `perm l l'` é gerada pela regra `perm_refl`, e portanto `l` e `l'` são a mesma lista.
2. `perm l l'` é gerada pela regra `perm_hd`, e portanto `l` e `l'` são permutações que têm o mesmo elemento como cabeça das listas.
3. `perm l l'` é gerada pela regra `perm_swap`, e portanto `l` tem a forma `x :: y :: l1` e `l'` tem a forma `y :: x :: l2`, e `l1` é uma permutação de `l2`.
4. Por fim, se `perm l l'` é gerada pela regra `perm_trans` então existe uma lista `l''` tal que `perm l l''` e `perm l'' l'`.

A prova de cada um destes casos, como você poderá constatar, é consequência quase direta das hipóteses de indução.

2.2 Questão 2

A prova de que `perm` implica em `permutation`, ou seja, de que se a lista `l` é uma permutação da lista `l'` segundo o predicado `perm` então `l` é também uma permutação da lista `l'` segundo o predicado `permutation`, não foi uma tarefa tão desafiadora, concorda? Como veremos, a prova do Lema 2 será bem mais sofisticada porque agora a nossa hipótese não é mais definida indutivamente. Vamos iniciar com um exercício relativamente simples: se `permutation nil l` então `l = nil`. Ou seja, se a lista `l` é uma permutação da lista vazia então `l` tem que ser a lista vazia:

```
Lemma permutation_nil: forall l, permutation nil l -> l = nil.
```

Como a definição do predicado `permutation` não é indutiva, não podemos usar a mesma estratégia de prova da questão anterior, mas note que a estrutura de listas é definida indutivamente, e portanto podemos aplicar indução em `l`.

2.3 Questão 3

No arquivo de formalização, apresentamos diversos lemas de caráter mais técnico que nos permitirão concluir a prova do Lema 2. Selecionamos aqui um dos mais importantes:

```
Lemma num_occ_cons: forall l x n, num_oc x l = S n -> exists l1 l2,
                    l = l1 ++ x :: l2 /\ num_oc x (l1 ++ l2) = n.
```

Este lema nos diz que se a lista `l` possui pelo menos uma ocorrência do elemento `x`, então `l` pode ser escrita na forma `l1 ++ x :: l2`, onde `l1` e `l2` são listas que possuem uma ocorrência a menos de `x` do que `l`, ou seja, `num_oc x (l1 ++ l2) = n`. Este lema também pode ser provado por indução em `l`.

2.4 Questão 4

Por fim, podemos resolver o lema 2 que aparece no arquivo de formalização como a seguir:

```
Lemma permutation_to_perm: forall l l', permutation l l' -> perm l l'.
```

Podemos iniciar a prova usando indução em `l`. O caso em que `l` é a lista vazia pode ser resolvido facilmente com o apoio do lema provado na questão 2. Quando `l` tem a forma `(a :: l1)`, então uma inspeção na estrutura da lista `l'` se faz necessária. Podemos então prosseguir usando indução em `l'`, mas apenas uma análise de casos (`case l'`) é suficiente.

3 Etapas de desenvolvimento do Projeto

Os alunos deverão definir grupos de trabalho limitados a três membros até o dia 12 de agosto de 2021. Os grupos serão organizados via GitHub Classroom da seguinte forma:

- Os alunos combinam entre si quem fará parte de cada grupo;
- Um dos elementos do grupo vai acessar o link `https://classroom.github.com/g/qMdebPu-` e criar o grupo (clicando no botão "Create team");

- Depois que o grupo foi criado, os outros membros do grupo devem acessar o mesmo link e clicar no botão "Join" do grupo correspondente.

Importante! Siga rigorosamente os passos acima para a formação dos grupo, e observe que depois que os grupos foram formados, não será mais possível a migração para outros grupos.

O projeto será dividido em duas etapas como segue:

- **Verificação da Formalização** (20 pontos): Os grupos deverão ter finalizado todas as provas do arquivo `PermEquiv.v`.
- **Entrega do Relatório Final** (10 pontos): Cada grupo de trabalho deverá entregar um relatório inédito (em formato pdf), limitado a oito páginas (12 pts, A4, espaçamento simples) até o dia **17 de outubro de 2021** com o seguinte conteúdo:

  1. Introdução e contextualização do problema;
  2. Explicação da soluções;
  3. Especificação do problema e explicação do método de solução;
  4. Descrição da formalização;
  5. Conclusões;
  6. Referências.

  Uma cópia deste relatório deve ficar no repositório do projeto no GitHub.

- Até o final do dia 17 de outubro de 2021, os arquivos `PermEquiv.v` e o pdf com o relatório final devem estar atualizados no repositório do GitHub!

Referências

[ARdM17] M. Ayala-Rincón and F.L.C. de Moura. *Applied Logic for Computer Scientists - computational deduction and formal proofs*. UTiCS, Springer, 2017.

[BvG99] S. Baase and A. van Gelder. *Computer Algorithms — Introduction to Design and Analysis*. Addison-Wesley, 1999.

[CLRS09] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Electrical Engineering and Computer Science Series. MIT press, third edition, 2009.