# Formalização de Unicidade de Respostas para Algoritmos de Ordenação

Logica Computacional 117366
Turma D

Lucas M. Chagas (12/0126643) e Pedro Henrique S. Perruci (14/0158596)

July 4, 2016

## Introdução

Ao longo do estudo das Estruturas de Dados, percebe-se que os *Algorítimos de Ordenação* têm importância fundamental na eficiência de processos. Alguns exemplos de métodos fundamentados em estruturas ordenadas, podemos citar a busca binária e algorítimos de união de listas. Como consequência de seus benefícios, métodos de ordenação são implementadas para as mais diversas estruturas de dados.

Sob o viés da lógica computacional, é essêncial garantir a unicidade de resultados dos métodos de ordenação. Dessa forma, pode-se provar que, para as mesmas entradas, uma mesma estrutura ordenada será obtida – independentemente do algorítimo escolhido. Támbem é coerente investigar que estruturas diferentes, portando os mesmos elementos, resultarão em estruturas ordenadas equivalentes.

O objetivo deste trabalho é justamente demonstrar a unicidade de respostas para os algorítimos de ordenação, implementados em listas e sequêcias finitas. Foram designados os métodos Quicksort e Bubble Sort voltados para o tipo abstrato *list*, Heapsort e Maxsort para o tipo *finite_sequences*. Como ferramenta, foi utilizado o assistente de provas PVS, associado às bibliotecas PVS de NASA Langley Research Center.

# 1 Questão 1

## 1.1 Especificação do Problema

A primeira questão do projeto consiste em demonstrar que o primeiro elemento de listas ordenadas é mínimo. Sua conjectura encontra-se expressa abaixo, no Algorithm 1.

**Algorithm 1** Conjectura da Questão 1.

min_is_sorted:
|— — — — —
[1] FORALL (l:list[nat], k : below[length(l)]) :
is_sorted?(l) => null?(l) OR nth(l,0) <= nth(l,k)

## 1.2 Solução Proposta

- *induct*("l")

A partir da indução na estrutura da lista l, a solução separa-se em dois ramos: base de indução e passo indutivo. A primeira resolve-se trivialmente, uma vez que a função de ordenação retorna verdadeiro para listas vazias. Por outro lado, o passo indutivo demanda demonstrar que, sabendo que o primeiro termo de uma lista $L$ é mínimo, mostrar que a propriedade se matêm quando adiciona-se um termo qualquer a $L$, sabendo que a lista resultante é ordenada. O sequênte resultante está expresso abaixo:

```
FORALL (j: below(length[L]):

    is_sorted(L) => null?(L) OR nth(L, 0) <= nth(L, j)
|- - - - -
FORALL (k: below(length[cons(L, a : nat]):

    is_sorted(cons(L, a : nat)) => null?(cons(L, a : nat)) OR
    nth(cons(L, a : nat), 0) <= nth(cons(L, a : nat), k)
```

- *induct*("k")

Resolve-se, então, realizar indução também na variável de contagem $k$. Note que sua função é servir como iterador para representar todos os n-ésimos valores da lista $cons(L, a)$. Logo, induzir este parâmetro significa mostrar que a propriedade é válida para a base indutiva $k = 0$ e, assumindo ser válida para $k = jb$, deve valer para $jb + 1$. Novamente, resolver para a base de indução é trivial.

- *inst*( $k = jb + 1$ )

Para solucionar o passo indutivo, decide-se instanciar $k$ com o valor $jb + 1$ e, assim, representar de forma equivalente os sequentes e consequentes. Esse passo dividiu a solução em diversos ramos.

- *split*

Utilizou-se o comando (*split*) para separar as implicações dos sequentes nos consequêntes. Os termos de cada sequente e consequente ficaram bem simplificados, no entanto, ampliou-se ainda mais a quantidade de ramos na demonstração.

- *expand* is_sorted? e *inst*( k = jb )

Cada um dos ramos de prova que foram encontrados tinham uma característica em comum: poderiam ser solucionados expandindo a definição de lista ordenada e instanciando sua variável de contagem com o termo jb.

- *expand* length e *expand* list2finseq[nat]

Por meio da expansão e simplificação destes dois termos, todos os ramos foram solucionados com sucesso. Acabava necessário a chamada do comando *assert* para validar as duas situações encontradas abaixo.

```
[-1] jb < length(L)
|- - - - -
[1] jb <= length(L) - 1
```

## 1.3 Considerações

A demonstração foi realizada com sucesso. Contudo, acredita-se que existam soluções mais simples para o problema — uma vez que axiomas muito semelhantes foram encontrados na solução dos diversos ramos. Recomenda-se a aventureiros futuros tentar simplificar a solução aqui constatada.

# 2 Questão 2

## 2.1 Especificação do Problema

A segunda questão do projeto consiste em demonstrar que quick_sort(l) e bubblesort(l) são funcionalmente equivalentes, utilizando os principais resultados da unicidade. Sua conjectura encontra-se expressa abaixo, no Algorithm 2.

**Algorithm 2** Conjectura da Questão 2.

func_eqQB:
|— — — — — —
[1] FORALL (l: list[nat]): quick_sort(l) = bubblesort(l)

## 2.2 Solução Proposta

- *lemma* "quick_sort_works"
- *lemma* "bubblesort_works"
- *lemma* "unicity_sorted_lists"

Para formalizar, usamos os lemas "quick_sort_works", "bubblesort_works" e "unicity_sorted_lists".

- *inst* (quicksort(l) bubblesort(l) l)
- *inst* (l)
- *inst* (l)

Instanciamos na fórmula -1 quick_sort(l) e bubblesort(l) para verificar a permutação entre quick_sort(l) e bubblesort(l) e verificar se as listas são sorteadas. Na fórmula -2 e -3 foi instanciado l para que possamos validar o que foi levantado pela fórmula -1.

- *split*

Primeiramente, dividimos a conjunção em 4 subárvores, sendo que 3 dos 4 itens são trivialmente verdade.

No caso onde não é trivial, tivemos que provar que as permutações de l, bubblesort(l) e quick_sort(l), l são transitivas.

- *lemma* "permutations_is_transitive"

Para provar isso, tivemos que chamar o lemma "permutations_is_transitive".

- *inst* (quick_sort(l) l bubblesort(l))

Instanciamos na fórmula -1 quick_sort(l), l e bubblesort(l), onde chegamos em três subárvores, sendo todas trivias, completando a prova.

### 2.3 Considerações

Foi utilizado alguns conceitos da Álgebra, como a transitividade, para resolver a questão.

# 3 Questão 3

## 3.1 Especificação do Problema

Na questão 3, assim como na questão 2, provamos que maxsort(s) e heapsort(s) são funcionalmente equivalentes, utilizando os principais resultados da unicidade. Sua conjectura encontra-se expressa abaixo, no Algorithm 3.

**Algorithm 3** Conjectura da Questão 3.

```
func_eqMH :
|— — — — —
[1] FORALL (s: finite_sequence[nat]): maxsort(s) = heapsort(s)
```

## 3.2 Solução Proposta

- *lemma* “maxsort_works”
- *lemma* “heapsort_works”
- *lemma* “unicity_sorted_seqs”

Para formalizar, usamos os lemas "maxsort_works", "heapsort_works" e "unicity_sorted_seqs".

- *inst* (maxsort(s) heapsort(s) s)
- *inst* (s)
- *inst* (s)

Instanciamos na fórmula -1 maxsort(s) e heapsort(s) para verificar a permutação entre maxsort(s) e heapsort(s) e verificar se as listas são sorteadas.

- *split*

Primeiramente, dividimos a conjunção em 4 subárvores. 3 dos 4 itens são triviais. No caso onde não é trivial, tivemos que provar que as permutações de s, maxsort(s) e s, heapsort(s) são transitivas e que a permutação(s, maxsort(s)) e permutação (maxsort(s),s) são simétricas.

- *lemma* “permutations_equiv”

Para provar isso, tivemos que chamar o lemma "permutations_equiv".

- *expand* “symmetric”

Tivemos que expandir a fórmula de symmetric para verificar a simetria da permutação(s, maxsort(s)). Obtemos, assim, a simetria da permutação(s, maxsort(s)) e permutação(maxsort(s), s).

- *expand* “transitive”

Expandimos também a fórmula de transitive para verificar a transitividade entre a permutação(maxsort(s), s) e permutação (s, heapsort(s)).

Com a expansão, dividimos a conjunção e assim chegamos na conclusão da prova.

## 3.3 Considerações

Assim como na questão 2, foram utilizados principios da Álgebra, transitividade e simetria, para resolver a questão.

# 4 Questão 4

## 4.1 Especificação do Problema

A questão 4 consiste em demonstrar a unicidade de resultados entre as funções quick_sort e maxsort — a primeira elaborada para listas e a segunda para sequências finitas. utilizar bem as transformações $list2finseq$ e $finseq2list$ e suas prpriedades será fundamental para a prova. A conjectura para a questão encontra-se expressa abaixo.

**Algorithm 4** Conjectura da Questão 4.

func_eqQM :
|— — — — —
[1] FORALL (l: list[nat], s: finite_sequence[nat]):
list2finseq(l)=s IMPLIES
list2finseq[nat](quick_sort(l)) = maxsort(s)

## 4.2 Solução Proposta

O sequente da conjectura pode ser simplificado pela chamada do comando (*skeep*). Fica então, mais explícita a situação de nossa prova.

```
[-1] list2finseq(l) = s
|- - - - -
[1] list2finseq( quick_sort(l) ) = maxsort(s)
```

A solução encontrada pode ser dividida em três etapas principais. Seu detalhamento, será feito nas subseções logo em seguida.

1. Mostrar que ambos os métodos de ordenação são funcionais;
2. Provar a unicidade entre os métodos, de forma semelhante a realizada nas questões 2 e 3;
3. Concluir sua equivalência por meio das propriedades de transformações de tipos abstratos.

### 4.2.1 Parte 1 — Mostrar que ambos os métodos de ordenação são funcionais

- *lemma* quick_sort_works, *inst* “l”
- *lemma* maxsort_works, *inst* “s”
- *lemma* is_sorted_iff_sorted, *inst* “quick_sort(l)”

O objetivo principal dessa etapa é reunir sequentes valiosos para a demonstração a partir da funcionalidade dos métodos de ordenação. Para tanto, é necessário convocar os lemas expressos acima. O último deles, *is_sorted_iff_sorted*, garante que se uma lista é ordenada, sua sequência equivalente também encontra-se ordenada.

### 4.2.2 Parte 2 — Provar a unicidade entre os métodos

- *lemma* unicity_sorted_seqs
- *inst*( "list2finseq( quick_sort(l)" "maxsort(s)" )

Semelhantemente ao realizado nas questões 2 e 3, realizou-se a chamada do lema da unicidade em sequências. Instanciá-los nos termos das estruturas ordenadas quick_sort(l) e maxsort(s) foi essencial para encontrar sub-árvores triviais e encaminhar bem a demonstração. Em seguida, resta-nos provar que os mesmos elementos constam na lista ordenada e sua sequência correspondente ordenado. O sequente em seguida expressa tal situação:

```
[-1] permutations( s, maxsort(s) )
[-2] permutations( quick_sort(l), l )
[-3] list2finseq(l) = s
|- - - - -
[1] permutations( list2fiseq( quicksort(l) ), maxsort(s) )
[2] list2finseq( quick_sort(l) ) = maxsort(s)
```

- *lemma* permutations_equiv
- *expand* transitive?
- *inst* "list2finseq( quick_sort(l) )" "list2finseq( l )" "maxsort(s)"

A propriedade de transitividade de sequências é convocada para mostrar que permutação dos termos [-1], [-2], associados à [-3], equivalem à permutação [1]. Foi instanciada de forma a obter soluções triviais para sub-árvores. Resta-nos provar o seguinte sequente para concluir a prova:

```
[-1] permutations( quick_sort(l), l )
|- - - - -
[1] permutations( list2fiseq( quicksort(l) ), list2fiseq( l )
```

### 4.2.3 Parte 3 — Concluir equivalência por meio das propriedades de transformações de tipos abstratos

- *lemma* perm_fsq_iff_perm_list
- *inst* "quick_sort(l)" "l"

É evidente nossa prova já está praticamente concluída. A última alteração nessesária é mostrar que a transformação *list2finseq* não interfere na permutação de seus elementos, representada pelo lema convocado.

- *lemma* fs2l_l2fs_is_id, *inst* “l”
- *lemma* fs2l_l2fs_is_id, *inst* “quick_sort(l)”

Por fim, a dupla chamada desse lema prova a equivalência das transformações *list2finseq* e *finseq2list*, que acabam resultando da instanciação do lema do item anterior.

- *flatten*
- *split*
- *assert*

A prova é finalmente concluída a partir da simplificação dos lemas convocados e uma chamada do comando (*assert*).

## Conclusão

Neste projeto, demostramos que o primeiro elemento de listas ordenadas é mínimo, provamos que quick_sort(l) e bubblesort(l) são funcionalmente equivalentes, provamos também que maxsort(s) e heapsort(s) são funcionalmente equivalentes, e demonstramos a unicidade de resultados entre as funções quick_sort e maxsort — a primeira elaborada para listas e a segunda para sequências finitas.

Com a realização do projeto, pudemos, com sucesso, verificar a unicidade entre os algoritmos de ordenação. Foi possível observar as equivalências entre os diversos comandos do PVS com as regras do cálculo de sequêntes que vimos no decorrer da disciplina.