Universidade de Brasília
Instituto de Ciências Exatas

Departamento de Ciência da Computação

117536 - Lógica Computacional Turma: A

Relatório sobre **Radix Sort**

Alberto Tavares Duarte Neto - 18/0011707

3 de novembro de 2019

1 Introdução

Este é o relatório do trabalho da disciplina Lógica Computacional 1, com o objetivo de estudar o assistente de provas PVS através da realização da verificação formal da corretude do algoritmo de ordenação Radixsort.

Para a implementação do radix sort é necessário outro algoritmo de ordenação. Neste trabalho foi utilizado o mergesort para tal, cujas algumas propriedades foram provadas em semestres anteriores por outras turmas.

2 Apresentação do Problema e Solução

radix sort é um algoritmo de ordenação que... (completar)

Em outras palavras, utilizando as definições dadas, devemos provar:

1. permutations[T](merge_sort(l), l)
2. permutations[T](radixsort(l), l)

3. is_sorted?[T, lex](radixsort(l))

onde l é uma lista do tipo abstrato T.

2.1 Questão 1

Nesta questão devemos provar que o merge sort é uma permutação da lista.

2.1.1 Lemas Utilizados

- merge_occurence
- length_prefix
- length_suffix
- app_prefix_sum
- occurrences_of_app

2.1.2 Solução

A estratégia de prova utilizada foi realizar uma indução no tamanho da lista, utilizando o fato de que, ao expandir a definição de merge sort, temos o merge do prefixo e do sufixo da lista que são de fato menores que a lista, se encaixando na hipótese de indução.

Temos então o seguinte sequente ao expandir as definições de permutations e merge sort:

```
merge_sort_is_permutation :

[-1]  FORALL (y: list[T]):
        length(y) < length(x) IMPLIES
         (FORALL (x: T): occurrences(merge_sort(y))(x) = occurrences(y)(x))
  |-------
{1}   IF length(x) <= 1 THEN occurrences(x)(x_1)
      ELSE occurrences(merge(merge_sort(prefix(x, floor(length(x) / 2))),
                             merge_sort(suffix(x, floor(length(x) / 2)))))
                      (x_1)
      ENDIF
       = occurrences(x)(x_1)
```

Figura 1:

Onde x é uma lista com elementos do tipo T, e x_1 é um elemento do tipo T.

O caso onde

$$length(x) <= 1 \tag{1}$$

é trivial, então trabalharemos o caso

$$length(x) > 1 \tag{2}$$

Pelo lema (botar numero) nós temos:

$$occurrences(merge(l1, l2))(x) = occurences(l1)(x) + occurrences(l2)(x) \tag{3}$$

Podemos instanciar este com o prefixo e suffixo de l nas listas l1 e l2, respectivamente, e substituir a igualdade no consequente.

Assim obtemos:

```
merge_sort_is_permutation.2 :

{-1}  FORALL (y: list[T]):
        length(y) < length(x) IMPLIES
         (FORALL (x: T): occurrences(merge_sort(y))(x) = occurrences(y)(x))
  |-------
{1}   occurrences(merge_sort(prefix(x, floor(length(x) / 2))))(x_1) +
       occurrences(merge_sort(suffix(x, floor(length(x) / 2))))(x_1)
       = occurrences(x)(x_1)
```

Figura 2:

Neste sequente fica clara a utilização da hipótese de indução; é fácil ver, utilizando os lemas (n e m), que para uma lista de tamanho maior que 1 seu prefixo e sufixo terão tamanho menor que a lista original.

Portanto, aplicando a hipótese indutiva ao prefixo e sufixo, temos que as ocorrências do merge sort do prefixo e sufixo são iguais às ocorrências do prefixo e sufixo, respectivamente, e aplicando isto ao consequente:

```
merge_sort_is_permutation.2 :

  |-------
{1}   occurrences(prefix(x, floor(length(x) / 2)))(x_1) +
       occurrences(suffix(x, floor(length(x) / 2)))(x_1)
       = occurrences(x)(x_1)
```

Figura 3:

Por fim devemos utilizar dois lemas: (n) e (m). Desta forma tempos que uma lista x é igual ao append de seu prefixo e sufixo; e as ocorrências de um append de duas listas l1 e l2 são iguais às da soma das ocorrências de l1 e l2.

```
merge_sort_is_permutation.2 :

{-1}  occurrences(append(prefix(x, floor(length(x) / 2)),
                         suffix(x, floor(length(x) / 2))))
                 (x_1)
       =
       occurrences(prefix(x, floor(length(x) / 2)))(x_1) +
        occurrences(suffix(x, floor(length(x) / 2)))(x_1)
[-2]  x =
       append(prefix(x, floor(length(x) / 2)),
              suffix(x, floor(length(x) / 2)))
  |-------
[1]   occurrences(prefix(x, floor(length(x) / 2)))(x_1) +
       occurrences(suffix(x, floor(length(x) / 2)))(x_1)
       = occurrences(x)(x_1)
```

Figura 4:

Podemos aplicar estes dois fatos à equação no consequente, terminando a prova.

2.2 Questão 2

2.2.1 Lemas Utilizados

- permutations_is_transitive
- merge_sort_is_permutation

2.2.2 Solução

A prova desta questão é simples e segue diretamente da questão anterior. Utilizando o lema, agora provado, de que merge sort permuta a lista l nós podemos realizar duas instâncias deste:

```
radixsort_permutes :

{-1}  permutations(merge_sort(merge_sort[T, <=](l)), merge_sort[T, <=](l))
[-2]  permutations(merge_sort(l), l)
  |-------
[1]   permutations[T](merge_sort[T, <<](merge_sort[T, <=](l)), l)
```

Figura 5:

Então utilizando a transitividade da permutação, que nos é garantida pelo lema (n), obtemos o resultado

$$permutations(merge\_sort[T, <<](merge\_sort[T, <=](l)), l) \tag{4}$$

que é exatamente o que queríamos provar.

2.3 Questão 3

Queremos mostrar que radixsort está ordenado segundo a ordem lexicográfica lex. A ideia da demonstração é utilizar o fato de que (...)

2.3.1 Lemas Utilizados

- merge_sort_is_sorted
- merge_sort_is_conservative
- is_sorted_implies_monotone

2.3.2 Solução

Após expandir as definições de radixsort, is_sorted? e lex, temos:

```
radixsort_sorts :

{-1}  k <= length(merge_sort[T, <<](merge_sort[T, <=](l))) - 2
  |-------
{1}   lex(nth(merge_sort[T, <<](merge_sort[T, <=](l)), k),
          nth(merge_sort[T, <<](merge_sort[T, <=](l)), 1 + k))
```

Figura 6:

ou seja, se k e 1 + k são índices da lista então os elementos com estes índices estão ordenados segundo a ordem lexicográfica.

Para proceder com a prova podemos utilizar o lema da ordenação do merge sort com a ordem ¡¡; com isso podemos mostrar que, ao aplicar radix sort a uma lista, o elemento do índice k é menor que o elemento de índice 1 + k segundo esta ordem.

Portanto, aplicando simplificações proposicionais, obtemos:

```
radixsort_sorts :

[-1]  nth(merge_sort(merge_sort[T, <=](l)), k) <<
       nth(merge_sort(merge_sort[T, <=](l)), 1 + k)
[-2]  k <= length(merge_sort[T, <<](merge_sort[T, <=](l))) - 2
{-3}  (nth(merge_sort[T, <<](merge_sort[T, <=](l)), 1 + k) <<
        nth(merge_sort[T, <<](merge_sort[T, <=](l)), k))
  |-------
{1}   nth(merge_sort[T, <<](merge_sort[T, <=](l)), k) <=
       nth(merge_sort[T, <<](merge_sort[T, <=](l)), 1 + k)
```

Figura 7:

A fórmula -3, que estava no consequente, foi para o antecedente negado após a simplificação proposicional, e com isso temos que os elementos de índice k e 1 + k são iguais segundo a ordem ¡¡.

Para que estes elementos estejam ordenados segundo a ordem ¡=, devemos ter que o merge sort é um algoritmo de ordenação estável; ou seja, se dois elementos são iguais então eles mantêm sua ordem relativa após a aplicação do algoritmo.

Utilizando o lema da estabilidade do merge sort temos, então:

```
radixsort_sorts :

{-1}  i < j
  |-------
{1}   nth(merge_sort[T, <=](l), i) <= nth(merge_sort[T, <=](l), j)
```

Figura 8:

É fácil de imaginar que, se i ¡ j, o elemento de índice i é menor segundo ¡¡ que o elemento de índice j após aplicarmos merge sort com a ordem ¡= na lista l.

Isso é garantido por lema, de modo que este tem de hipótese que a lista l estar ordenada mas temos, também por lema, que a lista retornada pelo merge sort é ordenada.

Então a demonstração está concluída.

3 Conclusão