Universidade de Brasília
Instituto de Ciências Exatas

Departamento de Ciência da Computação

117366 - Lógica Computacional 1
Turma: A

Formalização de Propriedades do Algoritmo Radix Sort utilizando o Assistente de Provas PVS

André Marques - 150005491
Leo Akira - 180137531
Rodrigo Navarro - 150147376

18 de novembro de 2019

1 Introdução

Este relatório tem como base o projeto final da disciplina Lógica Computacional 1, do segundo semestre de 2019, lecionada pelo professor Flávio Moura. O objetivo deste projeto é estudar e aplicar a ementa teórica da disciplina, com foco na corretude do algoritmo de ordenação *Radix Sort*. Para a realização das provas, foi utilizado o assistente de provas Prototype Verification System (PVS).

2 Apresentação do Problema

Algoritmos de ordenação são essenciais tanto em ciência da computação como em qualquer outra área que armazena dados. A possibilidade de acessar os dados de forma mais eficiente, seja ela alfabética, numérica, lexicográfica, etc é uma das razões mais importantes para a necessidade de bons algoritmos de ordenação. Dessa forma, a procura por algoritmos cada vez mais eficientes resultou em várias versões, vários refinamentos e vários algoritmos que solucionam problemas diferentes, mas com a mesma ideia, a ordenação. Para a realização deste projeto foram focados os algoritmos *Radix Sort*, que por sua vez utiliza o algoritmo *Merge Sort*.

De forma resumida, o *merge sort* segue o princípio de Dividir e Conquistar, isto é, ele Divide (o problema em vários subproblemas e resolve-os através de recursividade) e Conquista (após cada subproblema ser resolvido ocorre a "conquista", que é a união (*merge)* das resoluções dos subproblemas).

O *radix sort* ordena itens identificados por chaves únicas. Essas chaves podem ser uma *string* ou número e a função do *radix sort* é ordená-las de forma lexicográfica.

3 Questões

3.1 Questão 1

Nesta primeira questão precisávamos provar que o algoritmo *merge sort* gera uma lista que é uma permutação da lista de entrada. Como método inicial de prova, utilizaremos uma indução a partir da medida *length* de nossa lista *l*, com isso obtemos nossa hipótese de indução:

```
FORALL (y:list[T]):
  length(y) < length(x) IMPLIES permutations(merge_sort(y), y)
```

Tendo que provar `permutations(merge_sort(x), x)`

A partir disso, exploramos a definição de *merge sort*, onde obtemos dois casos, um onde a *length* da lista x é menor ou igual a 1, que é trivial pois o *merge sort* de tal lista sempre retorna a mesma, e outro onde a lista é maior, onde temos que provar:

```
permutations(merge(merge_sort(prefix(x, floor(length(x) / 2))),
                   merge_sort(suffix(x, floor(length(x) / 2)))), x)
```

Visando tal, começaremos simplificando esta fórmula, que podemos fazê-lo utilizando o lema "*Merge Occurrence*", precisando apenas expandir a definição de *permutations* para obter uma fórmula próxima à instanciação do lema.

Assim, podemos utilizar a hipótese da indução primeiramente instanciando a lista y como **"prefix(x, floor(length(x) / 2))"** e depois como **"suffix(x, floor(length(x) / 2))"**, ambas geram um *branch* adicional onde precisamos provar que a *length* da instanciação é menor que a de x, para tal, basta instanciarmos os lemas "*Length Prefix*" e "*Length Suffix*" e substitui-los respectivamente em cada prova.

Agora no *branch* principal, as instanciações nos permitem aplicar que as *occurrences* de um elemento em uma lista onde aplicamos o *merge_sort* são iguais às *occurrences* do mesmo elemente na lista original, reduzindo a fórmula à:

```
occurrences(prefix(x, floor(length(x) / 2)))(x_1) +
   occurences(suffix(x, floor(length(x) / 2)))(x_1)
   = occurrences(x)(x_1)
```

Então, para finalizarmos, basta aplicar respectivamente os lemas "*Occurrences of App*" e "*App Prefix Suffix*", cujas substituições no lado esquerdo da igualdade geram o lado direito, resultando em uma tautologia.

3.2 Questão 2

O objetivo desta questão é que provemos que o *radix sort* de uma lista qualquer l do tipo T é uma permutação dessa mesma lista. Sabe-se que o *radix sort* dessa lista l é um *merge sort* da pré ordem `<<` de outro *merge sort* de pré ordem `<=` da lista. Logo, para a resolução dessa questão foi usada como lema a resolução da questão anterior, que nos possibilita a chamada de *merge sort* com os argumentos `[T, <<] e [T, <=]`.

Ao expandir a função *radixsort*, nota-se que o *radixsort* precisa da função *mergesort*. Logo, como explicado no início desta subseção, foi usado o lema provado na questão anterior 'merge_sort_is_permutation' com argumento `[T, <<]`

para começar do lado esquerdo da proposição. Ao aplicar este lema, expandimos `permutations` e nota-se que a definição de `permutations (occurrences)` é aplicada ao nosso problema.

```
   {-1} FORALL (: list[T]):
        FORALL (x: T): occurrences(merge_sort(l))(x) = occurrences(l)(x)
I----------------------
{1} FORALL (l: list[T]):
       FORALL (x: T):
         occurrences(merge_sort[T, <<](merge_sort[T, <=](l)))(x) =
            occurrences(l)(x)
```

Agora, aplicamos as regras de sequente de Gentzen com o comando `skeep`, que aplica repetidamente o comando `skolem` mantendo o nome das variáveis da fórmula. Aplicamos agora o comando `inst?` para istanciar as variáveis e repetimos os dois útimos comandos para remover os `FORALL` do antecedente e do sucedente.

Com os `FORALL` removidos, agora queremos substituir o argumento do antecedente e aplicá-lo no sucedente, que é o que queremos provar. Logo, aplicamos o comando `replaces` para substituir o argumento `[T, <<] por [T, <=]`, simplificando assim em apenas uma fórmula.

```
   I---------------------
{1}  occurrences(merge_sort[T, <=](l))(x) = occurrences(l)(x)
```

Agora que *mergesort* tem como argumento `[T, <=]`, podemos aplicar o mesmo lema aplicado anteriormente, alterando apenas o argumento.

```
(lemma "merge_sort_is_permutation[T, <=]")
```

Repetimos o processo ao expandir `permutations` e aplicar o comando `inst?` para remover os `FORALL` e simplificar a fórmular, de forma que a instanciação prova a questão.

3.3 Questão 3

Aqui, queremos mostrar que ao aplicar o *Radix Sort* numa lista *l* do tipo *T*, este criará uma lista ordenada segundo a ordem lexicográfica *lex*, já definida. Como visto na questão 2, que o *Radix Sort* utiliza o *Merge Sort*, os lemas necessários para a realização desta prova são os seguintes:

- merge_sort_is_sorted;
- merge_sort_is_conservative;
- is_sorted_implies_monotone.

A necessidade pelos dois primeiros lemas é trivial visto que precisa-se "chamar"os lemas que definem que o *Merge Sort* cria uma lista *l'* realmente ordenada e que essa lista tem seus elementos conservados, ou seja, *length(l) = length(l')*. A dificuldade aqui encontrada foi na utilização do terceiro lema. Várias tentativas de prova foram utilizadas neste exercício, porém o grupo não foi possível realizar uma simplificação suficiente nos antecedentes para que a prova em si fosse concretizada.

4 Conclusão

A realização deste projeto foi bastante importante para cimentar o conteúdo teórico de uma forma mais prática, visto que o assistente de provas PVS obriga um tipo de prova passo a passo, o que implica o mínimo de conhecimento teórico para a sua concretização. Apesar do grupo não ter completado o último exercício do projeto, este contato com a ferramenta PVS foi bastante útil para todos os elementos do grupo.

5 Referências

1. M. Ayala-Rincón and F. L. C. de Moura. *Applied Logic for Computer Scientists - Computational Deduction and Formal Proofs.* Undergraduate Topics in Computer Science. Springer, 2017.

2. T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. Introduction to Algorithms. MIT Electrical Engineering and Computer Science Series. MIT press, second edition, 2001.

3. Wikipédia - Radix Sort. Disponível em https://pt.wikipedia.org/wiki/Radix_sort

4. Wikipédia - Merge Sort. Diponível em https://pt.wikipedia.org/wiki/Merge_sort

5. Relatório Grupo Radix Sort da Disciplina PAA 2019/01