Trabalho de Formalização de Prova de Algoritmos

Caio Bernardon Nascif Kaawi Massucato 16/0115001
Artur Filgueiras Scheiba Zorron 18/0013696

17 de novembro de 2019

Resumo

Neste relatório será abordado o desenvolvimento de provas para a formalização das propriedades dos algoritmos de ordenação chamados *merge sort* e *radix sort*.

1 Introdução e Contextualização

Neste relatório, trataremos do desenvolvimento e formalização de algumas provas necessárias para provar um algoritmo de ordenação, que nesse caso são o *merge-sort* e o *radix-sort*.

Primeiramente, um algoritmo de ordenação pode ser definido como um código com o objetivo de arranjar elementos em uma determinada ordem. O propósito de um algoritmo de ordenação é facilitar o acesso aos dados.

No caso, estudaremos o*merge sort*, que é baseado no conceito de "Dividir e Conquistar". Nele os dados são divididos no ponto médio e ordenados a partir dessas divisões, em seguida ele une os sub-arranjos em um único conjunto ordenado. Veremos também o *radix sort*, algoritmo o qual ordena números naturais, em princípio, comparando-os pelo dígito de menor significância para depois compará-los pelos próximo dígito, repetindo esse processo até alcançar o de maior significância, finalizando a ordenação.

Na seção 2 serão descritos os problemas e a explicação dos passos utilizados nas provas e também citados alguns dos pontos chave para sua formalização. Por fim, na seção 3, está contida a conclusão do problema.

2 Metodologia e Desenvolvimento

Nesta seção, descreveremos o processo de prova em cada questão e explicaremos as soluções obtidas.

2.1 Especificação do problema

2.1.1 Questão 1

Foi solicitado, para esta prova, a demonstração de que o algoritmo de *merge sort* retorna uma permutação da lista passada como entrada.

```
merge_sort_is_permutation: CONJECTURE
FORALL (l : list[T]) :
    permutations(merge_sort(l), l)
```

Esse enunciado conjectura que para toda lista ”l”com elementos do tipo ”T”, o algoritmo retorna uma permutação para essas listas.

2.1.2 Segunda Questão

Nesta questão, a conjectura afirma que o radix sort é um algoritmo de ordenação estável. Isso quer dizer que este algoritmo preserva a ordem de registros de chaves iguais. Isto é, se tais registros aparecem na sequência ordenada na mesma ordem em que estão na sequência inicial.

```
radixsort_permutes : CONJECTURE
FORALL(l : list[T]): permutations[T](radixsort(l), l)
```

2.1.3 Terceira Questão

Nesta questão, a conjectura afirma que o radix sort realmente ordena utilizando propriedades do merge sort e propriedaes de ordenação estável.

```
radixsort_sorts : CONJECTURE
    FORALL(l : list[T]):
        is_sorted?[T, lex](radixsort(l))
```

Encontramos muitos problemas na solução desse exercício, principalmente pela propriedade lexicográfica.

2.2 Explicação das soluções

Nessa seção estarão descritos os comandos utilizados para a elaboração da prova no PVS, bem como sua explicação para as questões 1 e 2.

2.2.1 Primeira Questão

Para começar a formalização da nossa prova, utilizamos o comando *measure-induct "length(l)l"*, que aplica indução forte no tamanho (*length*) da lista, dessa forma nos vimos em frente a dois quantificadores universais $FORALL$, tomando assim o próximo passo como a introdução de uma constante de skolem, afim de nos livrarmos de um desses quantificadores e em seguida utilizar o comando *flatten* para uma simplificação disjuntiva, para que pudéssemos separar e utilizar parte do enunciado como um antecedente no nosso sequente. Em sequência, expandimos a definição de *merge_sort* e utilizamos comandos para levar os condicionais ao ”*top level*”da fórmula e o prop para separarmos esses casos em subárvores diferentes.

A primeira subárvore é trivial pois temos como hipótese que o *length(l)*- onde ”l”é uma lista - é menor ou igual a 1 ($length(l) <= 1$), logo a prova é trivial pois pela definição de permutação isso é óbvio.

A segunda subárvore, por sua vez, exige mais trabalho e possui muitos passos repetidos e triviais, que talvez não nos ajudarão a compreender a questão, portanto, decidimos explicar de forma mais sucinta para favorecer a comunicação. Caso não compreenda as simplificações que serão feitas na explicação, consulte a árvore de prova.

Agora ao que interessa, a subárvore possui a seguinte cara:

```
[-1] FORALL (y: list[T]):
        length(y) < length(l1)
        IMPLIES permutations(merge_sort(y),y)
|-------
{1} length(l1) <= 1
{2} permutations(merge(merge_sort(prefix(l1, floor(length(l1)
                          /2))),
                          merge_sort(suffix(l1, floor(length(l1)

                          /2)))), l1)
```

Note que em 1 temos como sucedente uma fórmula que tínhamos como antecedente na subárvore trivial. Bom, sabendo disso, vale também ressaltar que nosso objetivo ao iniciarmos a prova para essa segunda subárvore foi de tentar utilizar da hipótese de indução e de outros lemas conhecidos para realizar substituições, até que tivéssemos antecedentes que pudessem nos levar à um axioma proposicional no fim da prova.

Para iniciarmos nossa prova, expandimos a definição de permutations em 2 e introduzimos um lema "merge_occurrence"para podermos instanciar as definições do nosso sucedente 2 nessa fórmula, uma vez que ela possui o "merge"em sua composição. Após utilizarmos das definições para substituição, escondemos essa fórmula com *hide*, uma vez que ela já nos serviu como poderia.

A partir desse momento, nos deparamos com os conceitos de prefixo e sufixo de lista em nosso sucedente, logo precisávamos de algum artifício para introduzí-los em nosso antecedente ou então alguma forma de remover esses conceitos. Dessa forma, utilizamos dos lemas chamados "*length_suffix*"e "*length_prefix*"para substituirmos suas definições no nosso sequente e depois escondemos ambos os antecedentes.

Ao fim desses passos, encontramo-nos com o seguinte sequente:

```
|-------
[1] length(l1) <= 1
[2] occurrences(suffix(l1, floor(length(l1) / 2)))(x) +
occurrences(prefix(l1, floor(length(l1) / 2)))(x) =
occurrences(l1)(x)
```

Muito progresso havia sido feito, embora agora não tínhamos nenhuma hipótese sobrando e a prova não havia sido concluída. Esse passo exigiu muito estudo e a ajuda de alguns companheiros, até que enfim encontramos o lema "*occurrences_prefix_suffix*", que tem o seguinte formato:

```
FORALL (l: list[T], x: T):
        occurrences(l)(x) =
          occurrences(suffix(l, floor(length(l) / 2)))(x) +
            occurrences(prefix(l, floor(length(l) / 2)))(x)
```

Note as semelhanças entre o sucedente [2] apresentado no trecho anterior e a definição apresentada acima. Era o que nos faltava. Após a introdução desse lema, a solução foi trivial: removemos o quantificador universal com uma instanciação, substituímos o resultado em nosso sucedente [2] e, finalmente, chegamos em nosso axioma proposicional:

```
{-1} occurrences(l1)(x) =
       occurrences(suffix(l1, floor(length(l1) / 2)))(x) +
       occurrences(prefix(l1, floor(length(l1) / 2)))(x)
|-------
[1] length(l1) <= 1
[2] occurrences(suffix(l1, floor(length(l1) / 2)))(x) +
       occurrences(prefix(l1, floor(length(l1) / 2)))(x)
       = occurrences(l1)(x)
```

É obvio notar que -1 e [2] são logicamente iguais, portanto, [2] é uma verdade e que, dessa forma, podemos utilizar o comando "propax"para provarmos um sequente da forma:
$S \vdash ..., true, ...,$

2.2.2 Segunda Questão

Para chegarmos na solução da questão 2, fizemos primeiro uma solução um tanto quanto equivocada, a qual no dia da apresentação recebemos a instrução de refinarmos essa prova. O equívoco mencionado se deu por termos utilizado, assim como na questão 1, uma indução forte pelo "measure_induct", mas não utilizamos a hipótese de indução. Obviamente não faz sentido, uma vez que, como verá a seguir, é possível solucionar sem indução.

Apesar de não termos utilizado essa solução como definitiva, acredito que seja válido mencionar que essa solução agregou muito ao grupo, uma vez que tivemos que introduzir um novo lema e prová-lo:

```
permutation_permute : LEMMA
  FORALL(l1, l2, l3 : list[T]) :
  (permutations(l1, l2) AND permutations(l2, l3))
  IMPLIES permutations(l1, l3)
```

Na prova definitiva dessa questão, começamos dando o comando *(skeep)* para retirar o quantificador universal. Depois, demos o comando *(expand "radixsort")* e o comando *expand "permutations"* para colocar a definição deles dentro da nossa prova.

Após isso, introduzimos dois lemas, que utilizam a teoria de preordens das listas:
(lemma "merge_sort_is_permutation[T, <<]" e
(lemma "merge_sort_is_permutation[T, <=]").

Colocamos alguns *(inst?)* para instanciar constantes e retirar quantificadores universais. Quando aparecia algum comando não expandido, nossa ação era expandir ele e instanciar suas constantes, como fizemos com *(expand "permutations")* e com os comandos *(inst?)* e então utilizar essas definições para substuições em outras fórmulas. Com esse processo conseguimos chegar em uma prova bem simplificada, então demos o comando *(assert)* para simplificar a prova e assim foi provada a conjectura da questão 2.

3 Conclusão

Como conclusão, deixo registrado que foi um trabalho bem suado onde encontramos muitas dificuldades em utilizar a ferramenta PVS por ter pouca informação na Internet sobre ela. Porém, nosso trabalho em equipe foi o essencial, o conteúdo do projeto nos ensinou como colocar em prática todo o conhecimento teórico absorvido em sala de aula e nas provas, tal qual era, em nossa opinião, abstrato até então.

Conseguimos nos organizar para desenvolvermos juntos as soluções e chegarmos nas provas. Apesar de não termos completado a prova para a questão 3, estudamos bastante para nossas tentativas e acreditamos que essa persistência nas questões 1 e 2 fizeram a diferença. Foi um projeto bem interessante onde os dois integrantes sofreram muito e aprenderam muito também.

Referências

[1] Mauricio Ayala-Rincón and Flávio L.C. de Moura, *Applied Logic for Computer Scientists*. Springer, 2017.

[2] PVS Documentation — Disponível em: `http://pvs.csl.sri.com/documentation.shtml`.

[3] PVS Prover Guide — Disponível em: `https://pvs.csl.sri.com/doc/pvs-prover-guide.pdf`