Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Lógica Computacional 1 Turma: A

Relatório sobre o projeto:

Formalização de Propriedades do Algoritmo Radix Sort

Maria Júlia Dias Lima - 170151140

15 de novembro de 2019

1 Introdução

A formalização de provas de propriedades, no caso descrito, faz uso do cálculo de sequentes em lógica de primeira ordem, de modo a garantir a validade da prova.
No cálculo de sequentes, a prova é feita a partir da conclusão,na forma de derivação dos passos anteriores, usando as regras de inferência. O símbolo de derivabilidade, usado para indicar o que se deseja provar, separa as proposições (sucedentes) a direita, no formato de disjunções. À esquerda, temos as proposições (antecedentes) assumidas ou dadas, sendo elas as premissas, ou hipóteses, consideradas de forma conjuntiva, ou seja, o conjunto de todas elas deve provar o que é dado no sucedente.
Esse tipo de formalização é utilizada no sistema de verificação PVS, utilizado como ferramenta de prova dos teoremas propostos, incluindo, de forma básica, regras proposicionais e quantificadas, indução, reeescrita, simplificações aritméticas e igualdades, além de abstração de dados e simplificação

de modelos. Isso é feito a partir da linguagem de especificação do sistema, teorias pré-definidas, checagem de tipos, e utilização das propriedades anteriores.

2 Contextualização do Problema

O objetivo do projeto proposto é demonstrar, formalmente, a correção de uma versão do Radix Sort que utilize duas pré-ordens, do tipo << e <=, isso sobre um tipo não interpretado, nomeado T. Tais pré-ordens formam a ordenação lexicográfica.
O Radix Sort é um algoritmo que funciona de forma a ordenar do último ao primeiro itens de cada elemento dado, ordenando todos os itens de número x em todos os elementos, até atingir o primeiro de todos eles, na ordem escolhida. Diante desse fato, a estabilidade do algoritmo utilizado para ordenar cada coluna de elementos é essencial, pois necessitamos que registros apareçam na sequência ordenada na mesma ordem da inicial.
Para a atividade, o algoritmo Merge Sort é dado como o estável para tal uso. Temos disponível axioma de garantia dessa estabilidade nos arquivos dados, e tal escolha foi sustentada pela propriedade de ordenção do algoritmo de pré-ordens totais sobre um tipo não interpretado, exatamente da forma como foi definido o Radix Sort (do tipo <<, <=).
O Merge Sort basea-se no princípio de junções sucessivas de duas sequências ordenadas, formando apenas uma sequência ordenada. Esse processo é usualmente feito de forma recursiva, por esse método tornar a divisão e junção simplificadas.
As três questões abaixo foram propostas e solucionadas com base nos algoritmos descritos, com o objetivo de demonstrar formalmente algumas de suas propriedades.

3 Especificação do problema e explicação do método de solução

3.1 Questão 1

A questão de número um do projeto propõe que seja demonstrada a preservação do conteúdo lista ordenada no Merge Sort. Foi dado o predicado

'permutations', que garante a preservação da lista instanciada por ele. Temos, nessa questão, que tal predicado instancia a aplicação do Merge Sort em uma lista de conteúdo l.

SEQUENTE INICIAL:

merge_sort_is_permutation :

$|-------$
[1] $FORALL(l : list[T]) : permutations(merge\_sort(l), l)$

Para a solução do problema, foi utilizado um método de solução que consistiu em aplicar a indução forte disponível no PVS previamente, o que dividiu a prova em casos base e de indução.

CASO BASE:

merge_sort_is_permutation.1 :

{-1} $length(x) <= 1$
[−2] $FORALL(y : list[T])$:
$length(y) < length(x)$ IMPLIES permutations(merge_sort(y), y)
$|-------$

{1} $FORALL(l : list[T]) : permutations(merge\_sort(l), l)$

O caso base da indução não foi extensamente expandido, sendo resolvido apenas com as informações já atingidas, usando comandos simples de conclusão de prova por serem casos triviais (uso direto de especificações previamente concluídas).
Já o passo de indução necessitou de divisões em sua estrutura para ser provado.Foram necessárias divisões em 4 casos, no total, para o passo de indução.

Sequente do passo de indução, que diz que se o tamanho de y ¡ x, implicando mergesort(y) é permutação de y, então a permutação da expansão de mergesort(x) é uma permutação de x e o tamanho de x <= 1:

merge_sort_is_permutation.2 :

[−1] $FORALL(y : list[T])$:
$length(y) < length(x)$ IMPLIES $permutations(merge\_sort(y), y)$
|−−−−−−−
{1} $length(x) <= 1$
{2}$permutations(merge(merge\_sort(prefix(x, floor(length(x)/2))),$
$merge\_sort(suffix(x, floor(length(x)/2)))))$

Foram utilizados 6 lemas auxiliares nas provas referentes a tais passos:

1)Elementos são preservados,segundo a definição de Merge Sort.
2)O Merge Sort preserva o tamanho da lista.
3)Os elementos são preservados na anexagem de duas listas.
4)O prefixo dado de uma lista l, nomeado n, tem o tamanho que n tem na lista l.
5)O sufixo da lista l listada em (4), com relação ao prefixo dado, tem tamanho total de l menos o tamanho do prefixo, n.
6)Uma lsita l é igual a junção de dois prefixos, nos formatos dados em (4) e (5).

Para a solução do primeiro deles, foram utilizados os lemas (1) e o (6). Foi usada a prova de que um sequente com antecedente vazio e sucedente dado como sendo o fato de que as ocorrências de prefixo e sufixo já descritos são iguais à da lista total. Tal resultado foi obtido encontrando uma tautologia no sucedente, podendo, assim, concluir a validade da prova.
Primeiro sequente do primeiro ramo, no qual se deseja provar que as ocorrências de elementos no prefixo de l + a ocorrência no mergesort do sufixo é igual a ocorrência total na lista (preservação das ocorrências):

merge_sort_is_permutation.2.1.1 :

{-1} $FORALL(y : list[T])$:
$occurrences(merge\_sort(suffix(x, floor(length(x)/2))))(x\_1) =$
$occurrences(suffix(x, floor(length(x)/2)))(x\_1)$
|−−−−−−−
[1] $length(x) <= 1$
[2] $occurrences(prefix(x, floor(length(x)/2)))(x\_1)+$
$occurrences(merge\_sort(suffix(x, floor(length(x)/2))))$

$= occurrences(x)(x\_1)$

Para o segundo caso, utilizou-se os lemas (2) e (5).Foi resolvido utilizando uma contradição nos sucedentes dados, devido ao fato de termos uma lista x de tamanho <= a 1 e seu sufixo ter tamanho menor que 0, o que é impossível, chegando a um absurdo. Diante disso, podemos concuir que temos uma prova válida.
Segundo sequente do primeiro ramo, no qual se deseja provar que as ocorrências de elementos no prefixo de l + a ocorrência no mergesort do sufixo é igual a ocorrência total na lista (preservação das ocorrências) e que o tamanho dos sufixo é menor que o da lista:

merge_sort_is_permutation.2.1.2 :

$|-------$
{1} $length(suffix(x, floor(length(x)/2))) < length(x)$
[2] $length(x) <= 1$
[3] $occurrences(prefix(x, floor(length(x)/2)))(x\_1)+$
$occurrences(merge\_sort(suffix(x, floor(length(x)/2))))$
$= occurrences(x)(x\_1)$

Para o terceiro caso, foi utilizada uma prova de raciocínio similar a segunda, usando apenas o lema (4), pois se referia ao prefixo e não ao sufixo, mas possuia formato similar e, portanto, teve uma solução similar.
Primeiro sequente do segundo ramo, com dado a preservaçao das ocorrencias no merge sort do sufixo, a provar a preservação das ocorrências do merge sort do prefixo + do sufixo, e ambos tem tamanho menor que a lista: que o da lista:

merge_sort_is_permutation.2.2.1 :

{-1} $FORALL(x\_1 : T) :$
$occurrences(merge\_sort(suffix(x, floor(length(x)/2))))(x\_1) =$
$occurrences(suffix(x, floor(length(x)/2)))(x\_1)$
$|-------$
[1] $length(prefix(x, floor(length(x)/2))) < length(x)$
[2] $length(x) <= 1$
[3] $occurrences(prefix(x, floor(length(x)/2)))(x\_1)+$

$occurrences(merge\_sort(suffix(x, floor(length(x)/2))))$
$= occurrences(x)(x\_1)$

Para o quarto caso, usou-se os lemas (1), (4), (5). Foi resolvido no formato de encontrar uma contradição, e a encontrada diz respeito ao fato de que o tamanho da lista é dado como menor ou igual a 1 e os tamanhos do prefixo e do sufixo são menores que o da lista original, o que torna pelo menos um deles menor que 0.Isso não é possível e, portanto, temos um absurdo de conclusão, demonstrando a validade da prova.
Segundo sequente do segundo ramo, no qual se deseja provar a preservação das ocorrências do merge sort do prefixo + do sufixo, e que ambos tem tamanho menor que a lista:

merge_sort_is_permutation.2.2.2 :

$|-------$
$\{1\}\ length(suffix(x, floor(length(x)/2))) < length(x)$
$[2]\ length(prefix(x, floor(length(x)/2))) < length(x)$
$[3]\ length(x) <= 1$
$[4]\ occurrences(merge\_sort(prefix(x, floor(length(x)/2))))(x\_1)+$
$occurrences(merge\_sort(suffix(x, floor(length(x)/2))))(x\_1)+$
$= occurrences(x)(x_1)$

3.2 Questão 2

A questão 2 propõe a prova de que o Radix Sort de uma lista l é uma permutação dessa lista.
No sequente abaixo temos um sequente da prova, no qual é utilizado um 7º lema, que diz que a aplicação do Merge Sort em uma lista é uma permutação da lista, que é exatamente similar ao que se deseja provar: o Merge Sort é uma permutação do Merge Sort de uma lista. Isso foi possível instanciar no antecedente, substituindo o (l) pelo merge sort[T, <=](l). Com isso, manteve-se as ordens da lista especificadas na definição de Radix Sort, visíveis no consequente do sequente abaixo.
Sequente da prova:

radixsort_permutes :

{-1} $FORALL(l : list[T]) : permutations(merge\_sort(l), l)$
$|---------$
[1] $permutations[T](merge\_sort[T, <<])(merge\_sort[T, <=](l), l)$

3.3 Questão 3

A questão de número 3 propõe demonstrar que o Radix Sort definido gera uma lista ordenada na ordem lexicográfica dada, usando menor menor e menor igual.
Sequente inicial da prova:

radixsort_sorts :

$|---------$
[1] $FORALL(l : list[T]) : is\_sorted?[T, lex](radixsort(l))$

A prova foi desenvolvida expandindo lema de ordenação e a ordem lexicográfica, e dividindo no caso de o n-ésimo elemento em uma lista ordenada pelo Radix Sort ser maior ou igual ao de k+1, na ordem lexicografica. Os casos encontrados foram os seguintes, que foram desenvolvidos.
No primeiro deles, usou-se os lemas de definição de que o Merge Sort ordena uma lista, que o merge sort conserva a lista e a estabilidade do algoritmo, e o que define o Merge Sort como monotone. A partir disso, obteve-se a prova de que o i-ésimo elemento da lista é menor ou igual a um j-ésimo, dado que o i menor que j.
Sequente 1 do caso dado:

radixsort_sorts.1 :

{-1} $nth(radixsort(l), 1 + k) << nth(radixsort(l), k)$
[−2] $k <= length(radixsort(l)) - 2$
$|---------$
[1] $(nth(radixsort(l), k) << nth(radixsort(l), 1 + k)$ AND
NOT $(nth(radixsort(l), k + 1) << nth(radixsort(l), k))$
OR
$((nth(radixsort(l), k) << nth(radixsort(l), 1 + k)$ AND
$nth(radixsort(l), 1 + k) << nth(radixsort(l), k))$

AND $nth(radisxsort(l), k) <= nth(radixsort(l), 1 + k))$

No segundo sequente, obteve-se provas por expansões e utilizações do lema que garante a ordenação do Merge Sort, sucessivamente e com repetições. O terceiro e último sequente, foi um TCC , já previamente gerado automatiamente pelo sistema, que teve a prova fechada de forma trivial.
Sequente 2 do caso dado:

radixsort_sorts.2 :

$[-1]\ k <= length(radixsort(l) - 2)$
$|-------$
$\{1\}\ nth(radixsort(l), 1+k) << nth(radixsort(l), k)$ [2] $(nth(radixsort(l), k) << nth(radixsort(l), 1 + k)$ AND
NOT $(nth(radixsort(l), 1 + k) << nth(radixsort(l), k)))$
OR
$((nth(radixsort(l), k) << nth(radixsort(l), 1 + k)$ AND
$nth(radixsort(l), 1 + k) << nth(radixsort(l), k))$
AND $nth(radixsort(l), k) <= nth(radixsort(l), 1 + k))$

4 Conclusão

Diante da conclusão do projeto, foi possível aprender apropriadamente o uso dos comandos do PVS, como ferramenta no auxílio de provas, o que demonstra conhecimento do conteúdo abordado na disciplina e capacidade de aplicação, além de demonstrar a prova do que foi proposto, formalmente.

5 Referências

Mauricio Ayala-Rincón, Flávio L.C. de Moura - Applied Logic for Computer Scientists. Computational Deduction and Formal Proofs-Springer (2017)
http://logitext.mit.edu/logitext.fcgi/tutorial
Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein - Introduction to algorithms-The MIT Press (2001)
https://pvs.csl.sri.com/