Relatório Projeto 1

Grupo 5: Danilo Raposo, Felipe Batalha, Pedro Henrique

8 de Novembro 2019

1 Introdução

A utilização de sistemas assistentes de prova é extremamente importante para a Ciência da Computação. Através dessas ferramentas poderosas podemos testar propriedades matemáticas de algoritmos, testando a sua validade para diversos casos e garantindo que o funcionamento de uma solução é válido para todos os casos que possa encontrar, evitando inconsistências como comportamentos inesperados advindos de possibilidades não mapeadas.

Esse relatório tem como objetivo demonstrar a construção de provas através do PVS, utilizando para isso algoritmos conhecidos de ordenação: o *radix sort* e o *merge sort*, do qual a definição de *radix sort* depende. As provas a serem demonstradas abaixo não são necessariamente as mais elegantes ou eficientes, e foram obtidas em meio a familiarização com o assistente de provas. Diante disso,algumas das ramificações das provas poderiam ter sido evitadas e maior atenção será dada aos sequentes "principais" para o entendimento das formalizações,

2 Desenvolvimento

2.1 Questão 1

Para utilizarmos *merge sort* em *radix sort*, devemos nos certificar que o primeiro obtém uma permutação das listas que ordena, contendo as mesmas ocorrências da lista original na mesma quantidade. Isso consiste em provar uma conjectura, que diz que para toda lista l, a função **permutations(mergesort(l),l)** deve ser verdadeira.

O passo inicial para fazer isso é entender que o comprimento das listas em questão é fundamental, já que uma permutação de uma lista deve ter o mesmo comprimento que a mesma, sendo que o comprimento nesse caso é a quantidade de elementos. Podemos fazer isso através da indução forte, que no PVS é mediada pela função **measure-induct**, para a qual revelamos que faremos a indução a respeito do comprimento de l. Através desse passo obtemos que *merge sort* deve ser verdade também para todas as sublistas derivadas.

O passo seguinte foi instanciar através do comando **skeep** e depois expressar as permutações em termos da igualdade de ocorrências (occurrences) na lista ordenada e na original, obtendo uma relação de igualdade a qual utilizaremos posteriormente, e então expandimos o significado de *merge sort* no antecedente, o que evidencia mais uma função: **merge**, que une duas listas submetidas ao *merge sort*: o prefixo da lista e o sufixo da mesma,funções recursivas que dividem a lista (e sublistas resultantes) ao meio e as ordenam paralelamente. Utilizamos então o lema **merge occurence** através de **use***, que nos diz que as ocorrências da junção de duas listas são as mesmas que as obtidas através da soma entre elas e substituímos isso no sucedente (**replace**), de tal forma que removemos o merge da equação e temos ocorrências expressadas em termos da soma de seu prefixo e sufixo.

Instanciando o prefixo de l como uma sublista obtemos no antecedente (-1) que o número de ocorrências do prefixo é igual ao do *merge sort* do prefixo, **occurrences(mergesort(prefix(...)))(x1) = occurrences(prefix(...))(x1)**, o que é consistente com o que vimos até agora com respeito a l. Com isso podemos realizar uma substituição na soma das ocorrências de *merge sort*s do prefixo e sufixo do sucedente, eliminando o *merge sort* como na igualdade acima. O passo seguinte é fazer a mesma substituição para o prefixo, razão pela qual realizamos uma cópia do antecedente (-1). Feito isso, obtemos a relação **occurrences(prefix(...)))(x1) + occurrences(suffix(...))(x1)** no sucedente sempre que o comprimento for maior que da lista for maior que 1 como expresso pela relação, (quando lenght(x) menor ou igual a 1 a lista não pode ser mais dividida em prefixo e sufixo e deve ser reduzida a **occurrences(x)(x1)**, sinalizando o fim da recursão.

Agora podemos assumir o lema **occurrences of app** para utilizarmos a definição de append, uma função que concatena listas, tal que **occurrences(append(l1,l2)(x) = occurrences(l1)(x) + occurrences(l2)(x)**. Em seguida, substituimos o prefixo em l1 e o sufixo em l2, obtendo no antecedente uma relação de igualdade onde temos a mesma soma que a obtida no sucedente do parágrafo anterior, tornando possível uma substituição.

Para finalizar, fizemos o uso de mais um lema, chamado **app-prefix-sufix**. Esse lema nos mostra que concatenar o prefixo e o sufixo de uma lista l gera a própria lista. Instanciamos nesse lema o prefixo e sufixo sob os mesmos termos do sucedente, tal que **l = append(prefix(l,n),(suffix(l,n)))** se torna **l = append(prefix(x,floor(length(x)/2)),(suffix(x,floor(length(x)/2))))**, o que torna possível a substituição no sucedente, de tal maneira que **l = occurrences(append(prefix(x,floor(length(x)/2)),(suffix(x, floor(length(x)/2))))(x1) = occurrences(x)(x1)**. Como l e x podem ser intercambiáveis nesse caso o comando **assert** finaliza a questão uma vez que as pontas soltas geradas pelas hipóteses empregadas tenham sido amarradas ao longo das ramificações geradas durante a prova.

2.2 Questão 2

A nossa segunda questão é mais um passo na direção de provar o radix sort. Assim como ocorreu para o merge sort, queremos provar a conjectura que diz que radix sort permuta qualquer lista que submetermos a ele, ou seja, para toda lista: **permutations[T](radixsort(l),l)**. Expandimos então o significado de permutations como fizemos na primeira questão, retratando as em termos de suas ocorrências e após realizar **skeep** duas vezes seguidas temos **occurrences(radixsort(l))(x) = occurrences(l)(x)**. Expandindo radix sort vemos que sua definição implementa o merge sort seguindo duas pré-ordens totais distintas: **occurrences(mergesort(mergesort[T,<=](l)))(x) = occurrences(mergesort[T,<=](l)))(x)**

Como já provamos que merge sort é a permutação das listas que recebe, podemos então utilizar tal lema com antecendente. e expressá-lo também em função de suas ocorrências expandindo o significado de permutação que temos e instanciando as mesmas pré-ordens que temos no sucedente para então fazemos uma substituição no mesmo, obtendo **occurrences(mergesort[T,<=](l)))(x) = occurrences(l)(x)**.

Mais uma vez fazemos uso do lema que provamos na primeira questão, e o expandimos em termo de suas ocorrências. Uma instanciação automática (**inst?**) nos permite então mostrar que as ocorrências do merge sort seguindo a pré-ordem definida é permutação da lista assim como na primeira questão, uma vez que **occurrences(mergesort[T,<=](l)))(x) = occurrences(l)(x)** e **occurrences(mergesort(l)))(x) = occurrences(l)(x)**, podemos dizer que **occurrences(mergesort[T,<=](l)))(x) = occurrences(mergesort(l)))(x) = occurrences(l)(x)** , que pode ser escrito como **occurrences(mergesort[T,<=](l)))(x) = permutations(mergesort(l),l))**, que já provamos ser verdade.

2.3 Questão 3

Por fim, temos que provar que a permutação do radix sort também é um sort seguindo a ordem lexicográfica, como definida no trabalho. Portanto temos a conjectura inicial, para todas listas, **is_sorted?[T,lex](radixsort(l))**. Naturalmente, fazemos as etapas iniciais de instanciar e expandir radix sort, chegando em **is_sorted?[T, lex](merge_sort[T, <<](merge_sort[T, <=](l)))**, aonde podemos aplicar o lema de que todas as listas estão ordenadas ao passar por merge sort, **FORALL (l: list[T]): is_sorted?(merge_sort(l))**. Instanciando esta lista com ”merge_sort[T, <=](l)”, aproximamos o constatado no lema ao estado da sequente na árvore, o que é desejável para prová-la.

Neste ponto do desenvolvimento utilizamos da expansão de ”is_sorted?” e ”lex”, para tentar provar a questão utilizando o lema anterior, gerando proposições muito extensas com ”AND” e ”OR” seguidos, devido a expansão de ”lex”. Para lidar com toda a proposição atual, fazemos **skeep** e alguns **split**; alguns galhos gerados já estão resolvidos, outros são resolvidos com **inst?** seguido de **grind**, restando apenas um galho com [1] **nth(merge_sort[T, <<](merge_sort[T, <=](l)), k) <= nth(merge_sort[T, <<](merge_sort[T, <=](l)), 1 + k)**. Tal expressão é próxima de um dos fatos na sequente, mas não suficientemente, por isso trazemos o lema que merge sort é conservativo, e fazemos **inst** e **split** nele. A maioria dos galhos gerados são já resolvidos ou resolvíveis com **grind**, restando apenas 1 galho.

A proposição [1] mencionada anteriormente se mantém, e o lema da conservatividade agora conta **EXISTS (i, j: below[lenght(merge_sort(”...”))]): i <j AND ”...” AND ”...”**. Na nossa solução aplicamos **skolem -1 (”a” ”b”)** e **flatten -1** e após algumas tentativas falhas de **replace -2 1** e **replace -3 1**, utilizamos então o lema ”is_sorted_implies_monotone”, ou seja, a definição mais genérica de ordenação. Instanciamos com **inst -1 ”merge_sort[T, <=](l)”**, e após um último **split**, algumas instanciações simples e o uso novamente do lema "merge_sort_is_sorted", que consta que merge sort é ordenado, resolvemos [1] e a árvore fica completa.

3 Conclusão

Conseguimos completar as três árvores com sucesso. Este trabalho foi muito produtivo em termos de entender como usar o PVS e quão potente é para a tarefa de auxiliar em provas formais.

Foi também interessante observar quão importante foi o uso de lemas para as conclusões das questões, e como resultados de questões anteriores foram reutilizados em outras questões. Cada questão serviu como um aprendizado útil para a próxima. O desenvolvimento da terceira questão provavelmente seria consideravelmente mais lento e improdutivo se não tivéssemos feito a 1 e 2.

References

[1] S. Owre N. Shankar J. M. Rushby D. W. J. Stringer-Calvert. *PVS Language Reference*. 2001.
[1]