Universidade de Brasília
Instituto de Ciências Exatas

Departamento de Ciência da Computação

Lógica Computacional 1

Formalização de propriedades do algortimo *Radix sort*

Manuela Matos Correia de Souza - 160135281
Gustavo Macedo de Carvalho - 170058867

November 18, 2019

1 Introdução

O presente trabalho tem como objetivo formalizar propriedades dos algoritmos de ordenação Merge sort e Radix Sort.

Na seção 2, explicamos a prova da propriedade de que Merge Sort gera uma permutação da lista na qual é aplicado. Na seção 3, provamos a mesma propriedade para o algortimo Radix Sort. Por último, na seção 4, mostramos que Radix Sort realmente ordena uma lista segundo a ordem estabelecida.

Todas essas provas foram construídas usando o assistente de prova PVS e, em sua íntegra, possuem mais detalhes do que os mencionados no trabalho. Esses detalhes foram omitidos por uma questão de fluidez e dinamicidade do texto, mas podem ser vistos nos arquivos de prova disponibilizados.

2 Merge sort gera uma permutação

Queremos provar que $mergesort(l)$ retorna uma lista que é permutação de l. Como mergesort é um algoritmo recursivo, é natural tentar utilizar indução como estratégia de prova dessa propriedade. Quanto ao tipo de indução, também é natural escolher a indução forte sobre o comprimento de l, pois a recursão de merge sort se dá em listas cujo comprimento é metade do comprimento da lista original. Uma vez decidida a estratégia de prova, expandimos a definição de

$mergesort(l)$ para tentar utilizar a hipótese de indução e provar a propriedade. A partir daí, temos que analisar dois casos derivados da definição: o caso em que o comprimento de l é menor ou igual a um e o caso em que é maior que um.

No primeiro caso, $mergesort(l)$ retorna a própria lista l, por definição. Daí, só precisamos mostrar que l é uma permutação de si mesma, o que segue da definição de permutação.

No segundo caso, o comprimento de l é maior que um, e então $mergesort(l)$ retorna a função $merge$ aplicada a $mergesort(l_1)$ e $mergesort(l_2)$, onde l_1 e l_2 são as sublistas correspondentes a primeira metade e segunda metade de l, respectivamente. Como a hipótese de indução nos diz que a propriedade vale para qualquer lista de comprimento menor que o comprimento de l, podemos usar como hipótese que $mergesort(l_1)$ é permutação de l_1 e $mergesort(l_2)$ é permutação de l_2. No entanto, antes de podermos usar essas hipóteses, precisamos mostrar que os comprimentos de l_1 e l_2 são, de fato, menores que o comprimento de l. Isso foi feito nos lemas "length_suff_menor" e "length_prefix_menor", acrescentados à teoria mergesort. As provas desses lemas seguem imediatamente dos lemas "length_suffix" e "length_prefix" e por isso não serão detalhadas.

Nesse momento, temos como hipóteses as afirmações sobre $mergesort(l_1)$ e $mergesort(l_2)$, e queremos mostrar uma propriedade sobre $merge(mergesort(l_1), mergesort(l_2))$. Precisamos, portanto, estabelecer uma relação entre $merge$ e as hipóteses.

Essa relação surge de maneira indireta, usando a propriedade transitiva da permutação. Por um lado, temos o lema "merge_is_permutation", donde segue que $merge(mergesort(l_1), mergesort(l_2))$ é permutação de $append(mergesort(l_1), mergesort(l_2))$. Por outro lado, temos o lema "permutations_of_appends", que, acrescido das hipóteses, nos diz que $append(mergesort(l_1), mergesort(l_2))$ é permutação de $append(l_1, l_2)$. Pela transitividade da permutação, $merge(mergesort(l_1), mergesort(l_2))$ é permutação de $append(l_1, l_2)$. Para completar a prova, o lema "app_prefix_suffix" diz que $append(l_1, l_2) = l$, e portanto $merge(mergesort(l_1), mergesort(l_2))$ é permutação de l. Isso completa a prova de que merge sort gera uma permutação da lista na qual é aplicado.

Observação: a prova feita no pvs usa o lema "perm_app_mergesort", que foi acrescentado (e provado) à teoria. No entanto, esse lema é apenas uma instanciação do lema "permutations_of_appends", que já existia no arquivo original. Por isso, foi escolhido na explicação da prova usar o lema original.

3 Radix sort gera uma permutação

A prova dessa questão segue da propriedade da questão um e da transitividade da permutação.

Por definição, radix sort aplica merge sort a uma lista l duas vezes consecutivas, utilizando em cada vez uma pré-ordem diferente. Pela propriedade de permutação de merge sort, temos que $mergesort[T, \ll](mergesort[T, \leq](l))$ é permutação de $mergesort[T, \leq](l)$. Novamente por essa propriedade, temos que $mergesort[T, \leq](l)$ é permutação de l. Assim, pela transitividade da per-

mutação, segue que $mergesort[T, \ll](mergesort[T, \leq](l))$ é permutação de l, como queríamos provar.

4 Radix sort ordena

Queremos mostrar que radix sort ordena uma lista segundo a ordem "lex". Seja $l = radixsort[T, lex](l')$. Por definição, a lista l está ordenada (segundo a ordem "lex") quando, $\forall k$ tal que $0 \leqslant k < length(l)$, se supormos $k < length(l) - 2$ então uma das seguintes situações ocorre:

- $nth(l, k)$ é menor que $nth(l, k+1)$ segundo a ordem ($\ll$), e a recíproca não é verdadeira. Em outras palavras, $nth(l, k)$ é **estritamente** menor que $nth(l, k+1)$.
- $nth(l, k)$ é menor que $nth(l, k+1)$ segundo a ordem ($\ll$), e a recíproca é verdadeira. Isso significa que eles são iguais pela ordem ($\ll$). Além disso, na ordem ($\leq$), $nth(l, k)$ é menor que $nth(l, k+1)$.

onde $nth(l, x)$ corresponde ao elemento de l na posição x.

Podemos considerar cada caso separadamente se fizermos suposições sobre a relação de $nth(l, k+1)$ com $nth(l, k)$. Supondo que $nth(l, k+1)$ é menor que $nth(l, k)$ na ordem ($\ll$), conseguimos provar o segundo caso. Agora, se supormos que $nth(l, k+1)$ não é menor que $nth(l, k)$, temos dois casos para serem analisados, uma vez que isso pode acontecer por diferentes motivos. O primeiro motivo é que $k+1$ pode ser maior ou igual a $length(l)$. Nesse caso, a função nth não estaria definida, então qualquer afirmação sobre $nth(l, k+1)$ é verdadeira por vacuidade. O segundo motivo é se, de fato, $nth(l, k+1)$ não for menor que $nth(l, k)$ na ordem ($\ll$). Em ambas situações, essa suposição nos permite provar o primeiro caso. A seguir, vamos detalhar as provas de cada um desses ramos.

4.1 Se $nth(l, k+1)$ não é menor que $nth(l, k)$ na ordem ($\ll$) por vacuidade

Aqui temos como hipóteses

$$k < length(l) - 2 \tag{1}$$

$$\neg\ (nth(l, k+1) \ll nth(l, k)) \tag{2}$$

$$k + 1 \geq length(l) \tag{3}$$

De (1) e (3) chegamos ao absurdo, e daí podemos derivar qualquer coisa. Portanto, esse caso está provado.

4.2 Se $nth(l, k+1)$ não é menor que $nth(l, k)$ na ordem $(\ll)$

Aqui temos como hipóteses

$$k < length(l) - 2 \tag{4}$$

$$\neg\ (nth(l, k+1) \ll nth(l, k)) \tag{5}$$

e queremos mostrar

$$\neg\ (nth(l, k+1) \ll nth(l, k))\ \ \wedge\ \ nth(l, k+1) \ll nth(l, k) \tag{6}$$

A primeira parte de (6) já é dada na hipótese, então só precisamos mostrar a segunda parte. Lembrando que

$$l = radixsort[T, lex](l') = mergesort[T, \ll](mergesort[T, \leq](l'))$$

podemos usar o lema "merge_sort_is_sorted" instanciado com $mergesort[T, \leq](l')$. Esse lema nos diz que $mergesort[T, \ll](mergesort[T, \leq](l'))$ está ordenado na ordem $(\ll)$, isto é, que $\forall k$ tal que $0 \leq k < length(l)$, se supormos $k < length(l) - 2$, então $nth(mergesort[T, \ll](mergesort[T, \leq](l')), k) \ll nth(mergesort[T, \ll](mergesort[T, \leq](l')), k+1)$. Como esse lema é enunciado para todo $0 \leq k < length(l)$, precisamos analisar duas situações: quando $k < length(l) - 2$ e quando isso não acontece.

No primeiro caso, a prova está concluida, porque como já tínhamos nas hipóteses $k < length(l) - 2$, o lema nos dá a parte que faltava do consequente por *modus ponens*, isto é, que $nth(l, k+1) \ll nth(l, k)$.

No segundo caso, se $k \geq length(l)$, então chegamos em um absurdo, pois já tínhamos como hipótese $k < length(l) - 2$. Do absurdo podemos derivar qualquer coisa, e portanto a afirmação também está provada.

Isso completa a demonstração do teorema no caso em que $nth(l, k+1)$ não é menor que $nth(l, k)$ na ordem $(\ll)$.

4.3 Se $nth(l, k+1)$ é menor que $nth(l, k)$ na ordem $(\ll)$

Nesse ramo, temos como hipóteses

$$k < length(l) - 2 \tag{7}$$

$$(nth(l, k+1) \ll nth(l, k)) \tag{8}$$

e queremos mostrar que

$$[nth(l, k) \ll nth(l, k+1)]\ \ \wedge\ \ [nth(l, k+1) \ll nth(l, k)]\ \wedge \tag{9}$$

$$[nth(l, k) \leq nth(l, k+1)]$$

O primeiro termo do consequente deriva da mesma explicação da seção anterior; pelo lema "merge_sort_is_sorted" temos que l está ordenada na ordem $(\ll)$. O segundo termo é a utilização direta da hipótese (2). Por fim, falta mostrar a

terceira fórmula do consequente. Como já derivamos os primeiros dois termos do consequente, podemos usá-los na prova do terceiro. Queremos mostrar então que se dois elementos consecutivos são iguais na ordem ($\ll$), então o primeiro é menor que o segundo na ordem ($\leq$). Isso remete ao axioma de estabilidade do algoritmo merge sort. Se instanciamos o axioma com $l = mergesort[T, \leq]$, $m = k$ e $n = k+1$, obtemos uma implicação cujo lado esquerdo são os termos já derivados nesse ramo da prova: $nth(l, k) \ll nth(l, k + 1)$ e $nth(l, k + 1) \ll nth(l, k)$. Daí, por *modus ponens*, concluimos que existem $0 \leq i, j < length(l)$ tais que

$$i < j \tag{10}$$

$$nth(mergesort[T, \ll](mergesort[T, \leq](l')), k) = nth(mergesort[T, \leq](l'), i) \tag{11}$$

$$nth(mergesort[T, \ll](mergesort[T, \leq](l')), k + 1) = nth(mergesort[T, \leq](l'), j) \tag{12}$$

Tomando i' e j' como acima e substituindo (11) e (12) em no subtermo de (9) que queremos mostrar, chegamos a

$$nth(mergesort[T, \leq](l'), i') \leq nth(mergesort[T, \leq](l'), j')$$

onde $i' < j'$. Agora, pelo lema "is_sorted_implies_monotone", para qualquer $0 \leq j < length(l)$ e $0 \leq i < j$, temos que, se l está ordenada, então $nth(l, i) \leq nth(l, j)$. Isso é exatamente o que queremos mostrar e, portanto, só precisamos satisfazer a hipótese de que $mergesort[T, \leq](l')$ está ordenada. Mas isso segue imediatamente do lema "merge_sort_is_sorted", e então provamos o terceiro e último ramo da demonstração.

5 Conclusão

Com a realização deste trabalho, se tornou evidente a necessidade de se utilizar um assistente automatizado de provas quando se deseja provar alguma teoria.

Antes de estarmos matriculados nesta disciplina, todas as provas de teoria realizadas por nós em disciplinas anteriores foram realizadas com lápis e papel e envolviam uma quantidade muito menor de lemas e conjecturas do que a necessária para a solução das questões propostas neste projeto. Desta maneira, ao entrar em contato com a grande quantidade de sentenças teóricas presentes nos arquivos disponibilizados pelo professor e, dada a quantidade razoavelmente grande de passos executados na solução das questões 1 e 3, foi perceptível a importância do sistema PVS ao oferecer um ambiente que pudesse afirmar com segurança se uma ação poderia ou não ser tomada, assim como mostrar o que já havia sido feito e o que ainda se havia de fazer.

Caso a resolução das questões propostas no trabalho devesse ser feita inteiramente com lápis e papel, haveria uma grande chance de erros serem cometidos e

também a dificuldade em completar as provas seria muito maior, visto que seria bem mais difícil monitorar de forma organizada o progresso feito.

Naquilo que é concernente à resolução das questões propriamente ditas, pudemos perceber que é necessário avaliar as informações que temos à disposição e, a partir destas, montar uma estratégia para provar a teoria, ao invés de um simples *modus operandi* de tentativa e erro, pois, podemos encontrar um cenário a partir do qual se torna impossível prosseguir, de modo que se torna necessário desfazer vários passos realizados, além de que nem sempre é possível saber intutivamente qual o próximo passo a ser tomado, reforçando ainda mais a importância de ser guiado por uma estratégia estabelecida no início da prova.

Se existe algum ponto negativo que podemos afirmar sobre este projeto é que não sentimos que este tenha servido como um complemento à teoria estudada durante as aulas. Em verdade, a sensação é de que apenas uma pequena parte da teoria vista em sala foi de fato utilizada aqui, já que durante as aulas foram vistas regras fundamentais da lógica clássica e da lógica intuicionista e, neste trabalho, trabalhamos com teorias mais complexas estabelecidas nos arquivos disponibilizados. Uma possível solução para este problema pode ser trabalhar mais em sala de aula com provas que envolvam não apenas as regras fundamentais, porém com um conjunto de definições, lemas e conjecturas um pouco mais robusto.

No geral, avaliamos que este trabalho representou um aprendizado importante e também que este nos ensinou mais do que uma possível terceira prova teórica.

6 Referências bibliográficas

- PVS Prover Guide
- PVS System Guide