Projeto de Formalização de Propriedades do Algoritmo Radix Sort

Vinicius Sugimoto

November 22, 2018

1 Introdução

O projeto consiste em provar três propriedades do algoritmo Radix Sort: para todo natural n, $10^{n\_digits(n)} > n$, em que *n_digits(n)* é a quantidade de dígitos do número n; para toda lista l^1, l^2, a lista l^3 resultante do *merge* de l^1 com l^2 preserva os elementos da lista original, isto é, todo elemento em l^1 está em l^3 e todo elemento em l^2 está em l^3; para toda lista l e para todo natural d, se a lista l está ordenada ***até*** o dígito d, então a lista l' resultante do *merge_sort* de l no dígito d está ordenada até o dígito $d + 1$. O objetivo é provar essas propriedades com lógica computacional com a ajuda do programa assistente de provas *PVS*. Os resultados deste grupo em específico foram um pouco acima do esperado e ainda assim insatisfatórios.

2 Projeto

2.1 Questão 1

$$\forall n \in \mathbb{N}, 10^{n\_digits(n)} > n$$

Nessa, assim como nas questões seguintes, tentaremos uma prova por indução forte:

$$\forall y \in \mathbb{N},\ y < k \rightarrow n\_digits(y) > y \Rightarrow 10^{n\_digits(k)} > k$$

Aqui é onde começamos de fato a provar e é onde encontramos a verdadeira dificuldade dessa questão: nos familiarizarmos com o PVS – não que a questão seja simples, mas essa adaptação se mostrou um grande desafio – e seus comandos.

Como o objetivo principal não era nos especializarmos com o PVS, mas sermos apresentados a ele, precisamos saber de poucos de seus comandos, sendo esses os principais usados: *assert*, *grind*, *expand* e *replace*, sendo suas funções aplicar as regras, a lógica, definida nas bibliotecas do PVS na parte atual da prova para simplificar ou concluir a prova, aplicar algumas funções das bibliotecas do PVS para simplificar ou concluir a prova, expandir uma função por sua definição, substituir uma fórmula em outra quando forem iguais, respectivamente.

As minhas primeiras tentativas nessa prova foram aplicar comandos que modificassem alguma coisa na prova, pois não tinha ideia de como provar nem familiaridade com o PVS. Felizmente, depois de muitas falhas, eu entendia mais ou menos como utilizar os comandos mencionados.

O momento de virada dessa questão foi quando eu percebi, olhando o arquivo das questões, que a função *n_digits()* tinha um *if*, era definida em dois casos. Então eu a expandi e dividi a árvore de prova em dois ramos. A partir daí a prova foi acontecendo. Claro, às vezes emperrava em alguma parte, mas com a pouco experiência das falhas até o momento e a pouca noção que eu já tinha, foi relativamente simples terminar.

Então a prova da questão terminou de forma simples. Pelo menos é o que eu tinha pensado, mas como foi apontado pelo Professor Flávio L. C. Moura durante a apresentação do projeto, no final de um dos ramos, utilizei um *assert* para concluir a prova, pois olhando o estado daquele ramo naquele momento, sem pensar muito, as hipóteses pareciam fazer sentido para concluir a prova sem nenhuma outra modificação, pensei que um assert concluiria – e concluiu –, mas quando questionado sobre o porquê do *assert* concluir a prova, não conseguir mostrar toda a obviedade que tinha pensado.

Seguindo a sugestão do professor e intrigado por não entender o *assert*, em casa, tentei entender o que aquele *assert* fazia:

$$x = 10 \cdot ndiv(x, 10) + rem(10)(x)$$
$$ndiv(x, 10) \leq x$$
$$10 \cdot k > ndiv(x, 10)$$
$$|-----------------$$
$$x < 10$$
$$100 \cdot k > x$$

Fazendo algumas mudanças, cheguei em um ponto que dividi uma hipótese em dois casos, um dos casos satisfazia a prova, já o outro, não consegui desenvolver para uma solução:

$$x = 10 \cdot ndiv(x, 10) + rem(10)(x) \geq ndiv(x, 10)$$
$$10 \cdot (10 \cdot k) > 10 \cdot ndiv(x, 10)$$
$$|-----------------$$
$$x < 10$$
$$10 \cdot (10 \cdot k) > x$$

se $rem(10)(x) = 0$ então

$$x = 10 \cdot ndiv(x, 10) \Rightarrow 10 \cdot (10 \cdot k) > 10 \cdot ndiv(x, 10) \leftrightarrow 10 \cdot (10 \cdot k) > x$$

se $rem(10)(x) \neq 0$ então

???

Discutindo com outros alunos que fizeram o trabalho, ouvi que o *assert* considera só um desses casos – o que dá certo – e conclui a prova. Depois disso decidi parar de tentar resolver esse detalhe para prosseguir para outros problemas: o próximo estava na questão dois.

2.2 Questão 2

$$\forall y_1 \in lista[nat], y_2 \in lista[nat] :$$

$$\forall d \in \mathbb{N} :$$

$$length(y_1) + length(y_2) < length(x_1) + length(x_2) \rightarrow$$

$$permutations(append(y_1, y_2), merge(x_1, x_2, d))$$

$$| - - - - - - - - - - - - - - - -$$

$$\forall d \in \mathbb{N} :$$

$$permutations(append(x_1, x_2), merge(x_1, x_2, d))$$

Nessa questão, depois de aplicar a indução forte, começo a prova e diferentemente da primeira questão não fiquei muito emperrado no começo dela.

Depois de removermos os $\forall$'s de cima e expandirmos o *merge*, que tem alguns casos, tínhamos quatro ramos: com a lista x_1 vazia, com a lista x_2 vazia, com nenhuma lista vazia e o d-ésimo dígito da cabeça de x_1 menor ou igual que d-ésimo dígito da cabeça de x_2 e o caso contrário a todos esses.

Nos dois primeiros casos, com pelo menos uma das listas vazia, queríamos chegar em $permutations(append(x_1, x_2), append(x_1, x_2))$ para provar, mas isso parece óbvio (a lista resultante de apensar x_1 com x_2 faz parte das permutações da lista resultante de apensar x_1 com x_2), então tentei usar um *assert* mas não deu certo, percebi que tinha que usar um *grind* e completei esses ramos.

No terceiro ramo não parecia tão óbvio mas tentei usar um *grind* e completou a prova.

Dado que eu tinha provado todos os outros três ramos com um *grind*, pensei em tentar no último ramo. Infelizmente não deu certo. Tínhamos:

$$\forall y_1, y_2 \in lista[nat], \forall d \in \mathbb{N} :$$

$$length(y_1) + length(y_2) < length(x_1) + length(x_2) \rightarrow$$

$$permutations(append(y_1, y_2), merge(y_1, y_2, d))$$

$$| - - - - - - - - - - - - - - - -$$

$$permutations(append(x_1, x_2), cons(car(x_2), merge(x_1, cdr(x_2), d)))$$

Para aproximar a parte de cima com a de baixo e remover os $\forall$'s, instanciei y_1 com x_1, y_2 com $cdr(x_2)$ e d com d:

$$length(x_1) + length(cdr(x_2)) < length(x_1) + length(x_2) \rightarrow$$

$$permutations(append(x_1, cdr(x_2)), merge(x_1, cdr(x_2), d))$$

$$| - - - - - - - - - - - - - - - -$$

$$permutations(append(x_1, x_2), cons(car(x_2), merge(x_1, cdr(x_2), d)))$$

Depois disso eu fiquei meio travado, tentei aplicar vários comando mas sem saber mesmo o que fazer. Depois pensei que eu deveria expandir alguma das funções. Tentei algumas até que expandi a *permutations* e cheguei em dois ramos: um eu consegui provar, mas outro estava me dando mais trabalho ainda. Depois de muito tentar, discutindo com colegas que disseram ter conseguido provar, perguntei como fizeram. Eles usaram um lema que era muito parecido com o que queríamos chegar na parte de baixo para completar a prova. Era a solução que faltava. Seguindo o meu mesmo pensamento de tentar aproxima a parte de cima à de baixo, instanciava as variáveis do lema e com um pouco mais de trabalho consegui chegar ao final da prova.

2.3 Questão 3

$$\forall y \in lista[nat], \forall d \in \mathbb{N}$$

$$length(y) < length(x) \rightarrow is\_sorted\_ud?(y, d) \Rightarrow$$

$$is\_sorted\_ud?(merge\_sort(y, d), d + 1)$$

$$| - - - - - - - - - - - - - - - -$$

$$\forall d \in \mathbb{N}$$

$$is\_sorted\_ud?(x, d) \Rightarrow is\_sorted\_ud?(merge\_sort(x, d), d), d + 1)$$

Comecei a questão três tentando a mesma coisa: aproximar a parte de cima com a de baixo instanciando as variáveis e removendo os $\forall$'s: y com x e d com d:

$$length(x) < length(x) \rightarrow is\_sorted\_ud?(x, d) \Rightarrow$$

$$is\_sorted\_ud?(merge\_sort(x, d), d + 1)$$

$$is\_sorted\_ud?(x, d)$$

$$| - - - - - - - - - - - - - - - -$$

$$is\_sorted\_ud?(x, d) \Rightarrow$$

$$is\_sorted\_ud?(merge\_sort(x, d), d), d + 1)$$

Achei estranho logo que acabou ficando $length(x) < length(x)$, porque isso seria falso, mas tentei prosseguir. Para remover a implicação de cima, usei o *split*, o que criou três ramos e dois foram provados pelos axiomas. O terceiro ramo que sobrou tinha na parte de baixo $length(x) < lengt(x)$:

$$is\_sorted\_ud?(x, d)$$

$$| - - - - - - - - - - - - - - - -$$

$$length(x) < lengt(x)$$

$$is\_sorted\_ud?(merge\_sort(x, d), d + 1)$$

A partir daí eu consegui desenvolver um pouco tentando de várias maneiras, mas não achei que era algo que daria muito futuro. O tempo acabou e não conseguir terminar essa questão.

Deixo, então, o resto ou a prova toda da questão três como exercício para o leitor (e para mim).

3 Resultados

3.1 Acertos

Tive bastante dificuldades mas sinto como uma vitória as questões que completei e pedaços que consegui fazer, apesar de insatisfatório, insuficiente. Acredito que eu consiga usar a ajuda do PVS para provar qualquer coisa que eu consiga provar no papel também.

3.2 Erros

Alguns pontos que o PVS "disse" que estava correto e me permitiram concluir as provas ainda assim ficaram um pouco confusos para mim, é uma grande ajuda, mas as vezes ele é mais rápido que eu para perceber as coisas (quase sempre)

4 Conclusão

Considero que o trabalho foi bastante difícil, mas não acho que foi teve uma especificação injusta, acho que dava para ter feito e bem feito, foi falta de preparação minha. Meus resultados incompletos e insatisfatórios mostram isso.

5 Extra - Uma solução errada para a Questão 3

Na verdade, é uma solução errada para qualquer prova, um dos grandes motivos dela ser errada: se você escrever um lema para provar provar a questão três que é exatamente o enunciado dela, você consegue provar facilmente usando ele.

Mas a prova ficará marcada com *incomplete*, pois o lema não está provado. Para resolver esse problema, escreva um outro lema que é igual ao anterior e use ele para provar o primeiro lema. Assim, o primeiro lema estara provado como *incomplete*.

Se antes, só com o primeiro lema, a prova estava provada 50%, digamos, pois faltava prova o lema, agora com o segundo lema está provada 75%, pois falta provar o lema do lema. E se criarmos um terceiro lema igual para provar o segundo, teríamos 87,5% da questão toda provada. Poderíamos fazer isso cada vez mais. Então seja $n \in \mathbb{N} > 2$, usamos o lema n para provar o lema $n-1$ e diminuímos em 50% a quantidade que falta a ser provada da questão. Agora se fizermos $n \to \infty$, a quantidade que falta a ser provada da questão $q \in \mathbb{R}$ tende a 0. Então no infinito essa questão – e qualquer outra – está provada. $\square$

Só uma piada para descontrair o final do semestre.

6 Referências

http://flaviomoura.mat.br/lc1.html