Formalização das Propriedades do Algoritmo *Radix Sort*

João Lucas Azevedo Yamin Rodrigues da Cunha
Mariana Alencar do Vale
Pedro Vitor Valença Mizuno
Vitor Ribeiro Custódio

22 de novembro de 2018

Resumo

Neste relatório será abordado o desenvolvimento e a elaboração das provas para a formalização das propriedades do algoritmo de ordenação de listas *radix sort*.

1 Introdução e Contextualização

Neste relatório, trataremos do desenvolvimento e formalização de algumas provas necessárias para provar um algoritmo de ordenação.

Primeiramente, algoritmo de ordenação pode ser definido como um código com o objetivo arranjar elementos em uma determinada ordem.

No caso, estudaremos o *radix sort*, algoritmo o qual ordena números naturais, em princípio, comparando-os pelo dígito de menor significância para, em seguida, 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. Já na seção 3 explicaremos como foram fechados os ramos das árvores, de forma que finalizassem as provas. Por fim, na seção 4, está contida a conclusão do relatório.

2 Metodologia e Desenvolvimento

Nesta seção, descreveremos o processo de prova, os comandos utilizados em cada questão e a explicação das soluções obtidas.

2.1 Especificação do problema

2.1.1 Primeira Questão

Foi solicitado, para esta prova, a demonstração da veracidade da conjectura denominada *d_digits_gt*, definida, em código PVS, como:

```
d_digits_gt : CONJECTURE
FORALL(n : nat):
10^(n_digits(n)) > n
```

Dado um número natural qualquer *n*, usamos a função *n_digits()* para calcular o número de algarismos que o compõe. A partir desse resultado, a conjectura afirma que 10 elevado a *n_digits(n)* é sempre maior que o próprio número *n*.

2.1.2 Segunda Questão

Nesta questão, a conjectura afirma que, dado quaisquer duas listas de naturais, as permutações da fusão de ambas as listas através de dois métodos diferentes — *append*, um método de concatenação simples, e *merge*, que ordena os elementos a partir de um dígito especificado — é igual. Ou seja, estes métodos preservam os elementos das listas, embora resultem em listas com ordens distintas. Conforme o trecho de código a seguir:

```
merge_permutes : CONJECTURE
FORALL(l1, l2 : list[nat], d : nat):
permutations(append(l1,l2),merge(l1,l2,d))
```

2.1.3 Terceira Questão

Nesta questão, deve ser demonstrada a estabilidade do algoritmo *radix sort*, ou seja, utilizar o algoritmo intermediário estável, no caso, *merge sort* para esse fim. Para um algoritmo de ordenação ser considerado estável, a posição relativa entre dois elementos iguais deve ser mantida após a ordenação. Conforme o trecho a seguir:

```
merge_sort_d_sorts : CONJECTURE
 FORALL(l : list[nat], d : nat) :
     is_sorted_ud?(l,d) =>
     is_sorted_ud?(merge_sort(l,d),d+1)
```

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 parte da questão 2.

2.2.1 Primeira Questão

Na árvore principal, iniciamos a prova com o comando **measure-induct+ "n" ("n")**), que aplica indução forte em '*n*', criando uma hipótese do lado esquerdo. Em seguida, com **expand "n_digits" 1**, expandimos a definição de *n_digits*. Com isso, surgem casos *IF ELSE* da definição. Utilizamos, então **lift-if** para distribuir '10^' para dentro dos casos e um **prop** para separar os casos em ramos distintos.

Na primeira árvore gerada pelo **prop**, bastou utilizar **grind** para finalizá-la. O motivo será detalhado na seção 3.

Já na segunda árvore, utilizamos **inst -1 "ndiv(x!1, 10)"** para instanciar o *FORALL* em [-1], equivalente ao $\forall e$ do Cálculo de Gentzen. Uma vez que não tenha mais o *FORALL*, aplicamos **prop** para separar uma implicação em dois casos, dividindo a árvore mais uma vez.

No ramo 2.1, primeiro aplicamos **expand "^" 2**, expandindo a definição da exponenciação. Isso abriu dois casos da exponenciação: quando o expoente é negativo e quando o expoente é positivo. Por isso, com um **assert**, removemos o caso negativo, já que estamos trabalhando com números naturais. Em seguida, mais uma vez, expandimos a exponenciação através de **expand "expt"**, tirando um '$+1$' que estava dentro da expressão e trazendo ele para fora, multiplicando, conforme a propriedade da exponenciação. Mais uma vez, com **expand "^"**, expandimos a definição de '^' presenta na hipótese, inserindo o '*expt*' na expressão. Através do **typepred "ndiv(x!1, 10)"**, são adicionados dois precedentes referentes ao '*ndiv*'.

A partir desta regra, a continuação do ramo foi alterada desde a apresentação do projeto. Desta vez, utilizou-se **replace -1** para, a partir da igualdade existente em [-1], substituir '$x!1$' pela expressão a que lhe era atribuída em todas suas ocorrências nas outras expressões. Escondemos expressões desnecessárias para o fechamento do ramo por meio de **hide (-1 -2 1)**, restando apenas [-3] e [2]. Em seguida, substituímos uma grande expressão por um nome menor utilizando **name-replace "expt10" "expt(10*n_digits (ndiv(rem(10)(x!1) + 10*ndiv(x!1, 10), 10)))"**, facilitando o entendimento para o PVS. Por fim, finalizamos o ramo por meio do **assert**.

Para a árvore 2.2, utilizamos **typepred "ndiv(x!1,10)"** para adicionar o predicado da função '*ndiv*' para, então, finalizarmos o ramo com um **assert**, conforme será explicado na seção 3.

2.2.2 Segunda Questão

Na árvore principal, iniciamos a prova com o comando **measure-induct+ "length(l1) + length(l2)" ("l1" "l2")**), que aplica indução forte em '$l1$' e '$l2$', usando o tamanho das duas listas como parâmetro. Em seguida, utilizamos **skeep** para retirar o $FORALL$ presente no consequente, de forma que seja possível trabalhar em cima do '*permutations*'. Usamos a função **expand "permutations"1** para aplicarmos a definição da função no consequente [1] para, então, utilizarmos outro skeep, retirando outro FORALL.

A função **rewrite "occurrences_of_app"** é chamada para reescrever todas as funções 'occurrences(append)' pela soma das 'occurrences' de cada lista envolvida. Então, utilizamos **expand "merge"1** para aplicar a definição de merge no consequente [1], gerando diversas expressões condicionais '*IF*' e '*ELSE*', assim, separando os quatro condicionais em quatro árvores por meio da função **prop**.

Para o primeiro caso (Árvore 1), utilizamos **expand "permutations"**-2 para aplicar a definição de permutations em [-2]. Em seguida, retiramos o FORALL por meio da instanciação dos elementos '$x!1$' e '$x!2$' com a função **inst -2 "x!1" "x!2"**, seguido de outro **inst -2 "d"** e separamos a implicação por meio de **split**, formando dois novos casos.

A árvore 1.1 foi resolvida por meio da função **inst -1 "x"**, a qual retirou o FORALL restante, instanciando o x e, por fim, a função **rewrite "occurrences_of_app-1** finaliza a árvore, como será explicado na seção 3. A árvore 1.2 também será resolvida por meio de **rewrite "occurrences_of_app"2**.

Para o segundo caso (Árvore 2), é utilizado **lemma "append_null"**, o qual adiciona às hipóteses o lema correspondente ao '*append_null*', a qual afirma que o '*append*' de uma lista não vazia com uma lista nula resulta na lista não vazia da entrada. Retiramos o FORALL por meio de **inst -1 "x!1"** e, por fim, consideramos dois novos subcasos devido ao uso de **case "x!2 = null"**.

A árvore 2.1 foi resolvida pela aplicação da definição de 'occurrences' no segundo caso da equação 1 por meio da função **expand "occurrences"1 2**. Foi finalizada, então,

por meio de um **assert**. Já a árvore 2.2 expande 'occurrences' em **expand "occurrences"2 3**. Escondemos as expressões [-2] e [1] com a função **hide-all-but (-2 1)** a fim facilitar a compreensão do PVS, e finalizamos com um **assert**.

Para o terceiro caso (Árvore 3), expandimos a definição de 'occurrences' na terceira ocorrência da expressão 1 por meio de **expand "occurrences"1 3**, abrindo casos condicionais, esses separados pelas funções **lift-if** e **prop**.

Para a árvore 3.1, repetimos o processo de expandir a definição de 'occurrences' por meio de **expand "occurrences"1 1**, utilizamos **assert** para resolver algum dos casos condicionais e, para retirarmos o FORALL, instanciamos a expressão [-3] por meio de **inst -3 "x!1" "x!2" "d"** e utilizamos outro **assert** para resolver outras condições. Apresentamos a definição de 'occurrences_of_app' nas hipóteses por meio de **lemma "occurreces_of_app"**, retiramos outro FORALL por outra instanciação **inst -1 "cdr(x!1)" "x!2" "x"** e, utilizando **prop**, dividimos a implicação em dois subcasos.

Subdividindo em dois casos, a árvore 3.1.1 tem 'permutations' expandida por meio de utilizando **expand "permutations"**, ao retirar FORALL, instanciamos 'x' com a função utilizando **inst -1 "x"** e finalizamos com utilizando **grind**. No caso da árvore 3.1.2, não conseguimos prosseguir com sua prova.

Para a árvore 3.2, utilizando **inst -2 "x!1" "x!2"** e **inst -2 "d"**, realizamos uma sequência de instanciações, retirando múltiplos FORALL. Usando **assert**, simplificamos os casos e, por meio de **split** dividimos a implicação em dois subcasos.

Na árvore 3.2.1, expandimos a definição de 'permutations' usando **expand "permutations"** e instanciamos x por meio de **inst -1 "x"**. Então, reescrevemos '$occurrences\ (append(x!1, x!2))(x)$' como '$occurrences(x!1)(x) + occurrences(x!2)(x)$' por meio da função **rewrite "occurrences_of_app"** e criamos dois subcasos em que 'x!1 = x' ou não.

A árvore 3.2.1.1 é finalizada de forma trivial, visto que -1 e [-1] são iguais. Enquanto que a árvore 3.2.1.2 não foi completada.

Retornando ao subcaso 3.2.2, apresentamos a definição de 'occurrences_of_app' nas hipóteses por meio de **lemma "occurrences_of_app"**. Em seguida, retiramos o FORALL utilizando **inst -1 "x!1" "x!2" "x"** e escondemos expressões que não serão necessárias usando **hide (-2 3 4 1)**. Por fim, dividimos a árvore em duas subárvores, considerando que 'car(x!1) = x' ou não.

A árvore 3.2.2.1 é finalizada por meio de um **assert**. Enquanto que a árvore 3.2.2.2 não foi completada.

Para o quarto caso (Árvore 4), retiramos dois FORALL por meio de **inst -1 "x!1" "x!2"** e **inst -1 "d"**. Utilizamos **assert** para simplificarmos os subcasos, dividindo-os por meio de **split**.

O subcaso 4.1 se inicia com a apresentando a definição de 'permutations' por meio de **expand "permutations"**, retiramos o FORALL usando **inst -1 "x"**, importamos a definição de 'occurrences_of_app' para as hipóteses por meio de **lemma "occurrences_of_app"**, realizamos outra instanciação **inst -1 "x!1" "x!2" "x"** para retirar outro FORALL. Por fim, subdividimos a árvore em dois casos usando **(case "cons(car(x!2), merge(x!1, cdr(x!2), d)) = append(x!1, x!2)")**.

A árvore 4.1.1 é finalizada com um assert. Enquanto que, na árvore 4.1.2, escondemos expressões desnecessárias por meio de **hide (4 5 2 1)**, aplicamos a definição de 'occurrences' no terceiro caso da expressão [1] usando **expand "occurrences"1 3** e separamos os casos condicionais utilizando as funções **lift-if** e **prop**.

Finalizamos ambas as subárvores 4.1.2.1 e 4.1.2.2 por meio de **grind**.

Retornando para a árvore 4.2, apresentamos a definição de 'occurrences_of_app' nas hipóteses por meio de **lemma "occurrences_of_app"**, retiramos o FORALL utili-

zando a instanciação **inst -1 "x!1" "x!2" "x"** e separamos em dois subcasos por meio de **case "append(x!1, x!2) = cons(car(x!2), merge(x!1, cdr(x!2), d))"**.

O primeiro subcaso 4.2.1 é finalizado por meio de um assert. Enquanto que a árvore 4.2.2 não foi finalizada.

3 Formalização

3.1 Primeira Questão

A árvore 1 foi fechada por um grind, esse justificado pelo fato das expressões [-1] '$x!1 < 10$' e [1] '$10^1 > x!1$' serem essencialmente iguais, assim fechando o galho da árvore por meio do axioma do Cálculo de Gentzen.

A árvore 2.1 pode fechada por um assert, visto que as afirmações [-1] '$expt10 > ndiv(rem(10)(x!1) + 10 * ndiv(x!1, 10), 10)$' e [1] '$10 * expt10 > rem(10)(x!1) + 10 * ndiv(x!1, 10)$' são iguais, caso dividamos [1] por 10 ou multiplicamos [-1] por 10. Dessa forma, o galho é fechado, novamente, pelo axioma do Cálculo de Gentzen.

Já na árvore 2.2, consta no antecedente que '$x!1 = 10 * ndiv(x!1, 10) + rem(10)(x!1)$'. Porém, isso é equivalente a '$x!1 = x!1 + rem(10)(x!1)$'. Ou seja, que o resto da divisão de 10 por $x!1$ é igual a zero. Logo, $x!1$ precisa ser um número, menor que 10 cuja divisão seja inteira. Isso faz com $x!1$ seja 1. Logo, os consequentes [1] e [2] sejam verdadeiros, uma vez que $ndiv(1, 10)$ é menor que 1 e 10 e '$10\hat{}(1 + n\_digits(0)) > 1$', satisfazendo o ramo e o finalizando.

3.2 Segunda Questão

Por meio de '*rewrite occurrences_of_app*', podemos fechar as árvores 1.1 e 1.2 de forma similar, de tal forma que, no caso da 1.1, as expressões [-1] e [1] se tornarão iguais, tal qual 1.2, em que as afirmações [-1] e [2] serão iguais.

A árvore 2.1 considera o caso em que [1]'$x!2$' é nulo, assim, encaixando com a equação [-2], fechando a árvore.

A árvore 2.2 é resolvida por meio da expressão [-1] a qual é função '*null?*', que questiona se '$x!2$' é nula, entretanto, o consequente [1] afirma que '$x!2$' é nula, o que fecha a árvore.

Na árvore 3.1.1, utilizando a expressão [-1], onde '$occurrences(append(x!1, x!2))(x) = occurrences(merge(x!1, x!2, d))(x)$', conseguimos enxergar que a expressão [-2] '$occurrences(append(cdr(x!1), x!2))(x) = occurrences(cdr(x!1))(x) + occurrences(x!2)(x)$' e a expressão [1] '$occurrences(cdr(x!1))(x) + occurrences(x!2)(x) = occurrences(merge(cdr(x!1), x!2, d))(x)$' eram idênticas, cuja única diferença era o método de junção das listas utilizado. Como essa igualdade era garantida pela [-1], utilizamos **grind** para finalizar o ramo.

No ramo 3.2.2.1, o lado direito e o lado esquerdo, respectivamente das expressões [-2] e [-1] eram iguais a '$occurrences(x!1)(x) + occurrences(x!2)(x)$'. Este ramo, bem como os outros finalizados, não conseguimos justificar de maneira certeira.

Este problema não foi solucionado por completo.

4 Conclusão

Por fim, apesar de não conseguirmos solucionar todas as questões, foi possível aprender e aplicar o Cálculo de Gentzen de forma natural e intuitiva em alguns casos.

Também nos foi permitido entender o básico dos comandos PVS, linguagem essa que, a princípio, nos apresentou uma série de problemas, os quais foram superados, em sua maioria, ao longo do desenvolvimento do projeto.

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] NASA PVS Library — Langley Formal Methods Program — Disponível em: `https://shemesh.larc.nasa.gov/fm/ftp/larc/PVS-library/`