Relatório Final do Projeto PVS para Formalização de Propriedades do Algorítmo Radix Sort

Grupo 2

21 de Novembro de 2018

Resumo

Relatório final de projeto desenvolvido na discplina Lógica Computacional 1 do curso de Ciência da Computação da Universidade de Brasília sob orientação do Professor Doutor Flávio Leonardo Cavalcanti de Moura. O projeto teve como intuito utilizar o assistente de provas PVS para formalizar propriedades presentes em implementação do algorítmo de ordenação de dados Radix Sort. Este trabalho foi desenvolvido pelo Grupo 2 que tem como integrantes: Bruno Cordeiro Mendes, Matheus Luiz Freitas Bastos e Fillype Alves do Nascimento.

1 Introdução

O intuito deste projeto é formalizar, através do auxílio do sistema de verificação e especificação PVS, a correção de algumas propriedades presentes em uma implementação do algoritmo de ordenação Radix Sort.

1.1 Contextualização

Podemos ordenar conjuntos de números usando diferentes métodos que os ordenem da maneira desejada. Para o caso utilizado no projeto, por exemplo, desejamos ordenar uma lista de números inteiros. Existem vários algorítmos de ordenação que são capazes de ordenar essa lista (em ordem crescente, decrescente...). Iremos fazer uso de propriedades presentes em implementação do algorítmo Radix Sort.

O Radix Sort ordena, neste caso, as listas de inteiros comparando os dígitos presentes nos inteiros a serem ordenados, começando pelo dígito menos significativo até o mais significativo. Isso garante que a lista seja ordenada em passos intermediários que não desordenem o que foi anteriormente ordenado, como poderia acontecer caso se começasse a ordenar pelo dígito mais significativo de cada número.

Para a realização desse projeto foram fornecidos pelo professor arquivos com algumas propriedades desse algorítmo já provadas e propriedades a provar. Nisto consistiu o projeto: provar as Conjecturas (assim chamaremos as propriedades a serem provadas) e, se necessário, fazer uso dos Lemas (assim chamaremos lemas/propriedades já provadas).

Em um dos arquivos fornecidos, encontram-se os algoritmos para as Conjecturas d-digits-gt, merge-permutes e merge-sort-d-sorts. Devemos provar as três Conjecturas citadads do o algoritmo. Segue uma breve descrição das mesmas:

- **d-digits-gt:** esta conjectura afirma que todo natural n é menor que 10 elevado à quantidade de dígitos de n; a questão 1 do projeto é provar essa conjectura.
- **merge-permutes:** esta propriedade afirma que dadas duas listas de naturais *l1* e *l2*, e um natural *d*, a concatenação de *l1* e *l2* é uma permutação da lista resultante da ordenação de *l1* e *l2* segundo o *d-ésimo* dígito dos elementos daquelas listas; a questão 2 do projeto é provar essa conjectura.
-**merge-sort-d-sorts:** esta conjectura afirma que se uma lista l é ordenada por um dígito d, então essa lista também é ordenada pelo merge-sort dessa mesma lista com um digito d, comparada com o digito seguinte a d(d + 1); a questão 3 do projeto é provar essa conjectura.

2 Explicação das soluções

Considerando a descrição das questões mencionadas acima, explicaremos as soluções desenvolvidas, mostrando a abordagem tomada e relações dos comandos PVS utilizados e suas relações com o cálculo de sequentes.

2.1 Questão 1

A primeira questão, como descrito anteriormente, afirma que 10 elevado ao número de digitos de um natural n, é maior que esse número n. Para prosseguirmos na prova utilizamos, como orientado, o princípio de **indução forte**. A hipótese de indução leva em conta a função *n-digits* que conta a quantidade de dígitos do natural n. O segundo passo feito para se fazer a prova foi expandir a função n-digits pois, ao expandir a função, conseguimos chegar a uma visão mais abrangente do sequente envolvido na prova. Este caso mais geral criado consiste de um *if* e um *else* para a função n-digits, que nos diz que se a entrada da função for menor do que 10 entao o numero de digitos é igual a 1, senão vai ser igual a chamada recursiva da função para a entrada dividida por 10(ndiv(x!1, 10)). Como a expansão de n-digits está no expoente do número 10, entao aplicamos o comando **lift-if** para distribuir esse 10 dentro da expansão de n-digits. Após esses comandos foi necessário somente dar o comando **prop** para ramificar a árvore de provas entre os casos gerados pelo *if* e *else*

. O primeiro dos casos, trata de quando "x!1"for menor que 10, que como consequente temos o retorno de 10^1. Para deixar o antecedente mais parecido com o consequente e resolver a primeira prova expandimos primeiramente a exponenciação no consequente utilizando o comando **expand**. A exponenciação tem em sua definição o uso da outra função "expt", e o resulta da expansão da exponenciação gerou "expt(10, 1) $>$ x!1". Depois disso expandimos a função expt duas vezes para gerar o resultado de 10^1, ou seja, no final desse ramo tinhamos no consequente "10 $>$ x!1", e como o antecedente estava igual ao consequente, conseguimos provar o primeiro ramo, e para passar para o segundo ramo utilizamos o comando "assert"que terminou a nossa primeira prova e passou para a segunda.

O segundo caso trata o caso em que x!1 é maior que 10, e como o consequente utiliza da função ndiv, nosso primeiro passo foi instanciar um valor para o antecedente gerado com a indução forte no começo da prova, e o valor escolhido para a instaciação foi "ndiv(x!1, 10)". Com a instanciação foram gerados os casos em que

10 elevado ao número de dígitos de "x!1"dividido por 10 é maior que "x!1"dividido por 10; e o caso em que no consequente temos que x!1 dividido por 10 é menor que x!1.

Para o primeiro caso do segundo ramo tentamos novamente deixar o antecedente parecido com o consequente. Para isso expandimos a exponenciação no consequente e aplicamos o **assert** para o PVS excluir os casos que já tinham sido provados. Depois novamente aplicamos expansão em "expt"e depois novamente na exponenciação, só que dessa vez no antecedente. Para gerar mais argumentos para a prova aplicamos o comando **(typepred "ndiv(x!1, 10))"** e o comando **(typepred "rem(x!1)(10))"**. Como tinhamos argumentos o suficiente para provar o ramo, usamos **name-replace** para substituir as ocorrências de "expt(10, n-digits(10, n-digits(ndiv(x!1, 10)))"por "xdiv". Com os argumentos organizados aplicamos o comando **assert** para o PVS concluir a prova e seguir para o último ramo.

O último ramo trata do caso em que sabemos que "ndiv(x!1, 10) $<$ 10", como essa informação já é tratada como um consequente, não tínhamos nenhum antecedente para terminar a prova; por isso aplicamos novamente **typepred "ndiv(x!1,10)** para expandir a função com base na definição de tipos e gerar argumentos para terminar a prova. Com isso aplicamos novamente o **assert** e terminamos a resolução da prova da questão 1.

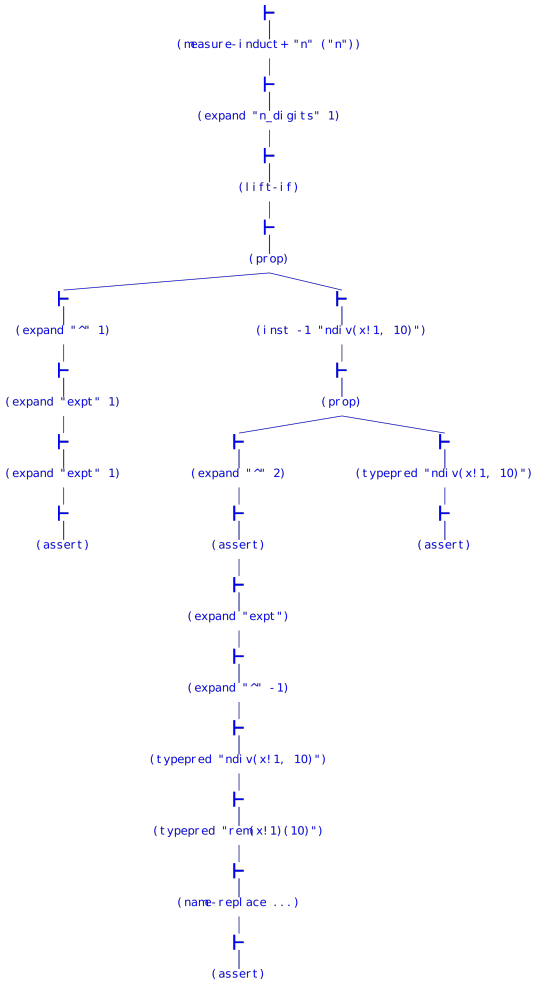

Figura 1: Árvore de prova da propriedade da questão 1

2.2 Questão 2

A segunda questão consiste em provar que o merge de duas listas preserva os itens originais das listas anteriores, a prova começa com a aplicação da indução forte, pois no começo só temos um consequente do que queremos provar. Aplicar a indução forte, além de ser uma boa estratégia, nos dá uma hipótese a mais para iniciar a prova.

Logo apos a indução forte foi feita a regra **L**$\forall$ do cálculo de sequentes para remover o quantificador universal e então instânciar os parâmetros na função merge e append. Ápos a etapa descrita anteriormente seguimos para a expansão da função merge, a expansão consiste em aplicar a definição de função, porém ao expandir são gerado cerca de quatro casos para se provar, esses casos serão descritos separadamente nos próximos paragráfos.

No primeiro caso nós temos como hipótese que o primeiro argumento da função append (que foi instânciado anteriormente) é nulo, porém essa hipótese pode ser provada de maneira simples, pois ela pode ser resolvida apenas expandindo a função permutations (função que está presente no consequente e possui duas funções append como parâmetros e que possuem os mesmo argumentos), e então teremos uma igualdade no consequente que, de fato, é verdadeira, resolvendo então a prova.

Parecido com o primeiro caso, temos neste segundo na hipótese que o segundo argumento da função append vai ser nulo, mas como na solução anterior essa informação pode ser obtida de maneira simples e ápos expandirmos a função permutations (que é a mesma do caso anterior) chegamos em uma igualdade no consequente que, de fato, é verdadeira.

No terceiro caso tinhamos como hipótese que um determinado dígito do primeiro argumento de append pode ser menor ou igual ao segundo dígito do segundo argumento, além de ter toda a hipótese gerada pela indução forte. Vale pontuar também que o consequente permutations agora recebe como primeiro argumento a função append que possui duas listas como argumento que vamos chamar de 'x!1' e 'x!2', que no segundo temos um construtor de uma lista que recebe a cabeça de "x!1"e como a cauda a aplicação da função merge na cauda de "x!1", com "x!2", tendo como base um dígito "d". Para provar o terceiro caso utilizamos do comando **grind** que aplicou vários comandos como **skolem, inst, e if-lifting** repetidas vezes para provar o ramo.

Para o quarto caso temos a hipótese de quando o d-ésimo dígito da cabeça da lista "x!1"for maior que o d-ésimo dígito da cabeça de "x!2". Para começar a prova do quarto ramo instanciamos o dígito "d"no antecedente para trabalharmos em um caso específico da aplicação da função permutations. Em seguida expandimos a função permutations tanto na hipótese como no consequente, com o intuito de criar outros dois ramos para serem provados. Como a expansão de permutations resultou em um caso geral de um dígito "x", aplicamos o comando (**skeep 2**) para instanciar o valor de "x"na função occurences. A função occurences, no consequente obtido, diz que o número de ocorrências de "x"na junção das listas x!1 e x!2 é igual o número de ocorrências de "x"no resultado da função merge de uma lista "x!1"com uma lista "x!2"que tem como primeiro elemento da lista a cabeça de "x!2". Utilizando o comando (**split**) para separar a hipótese do "IMPLIES"gerado pela indução forte. O primeiro caso gerado depois do split nos diz na hipótese que para todo "x"pertencente aos naturais o número de ocorrências de "x"na junção de "x!1"com a calda de "x!2"é igual ao número de ocorrências de "x"na aplicação

da função merge(por um dígito "d") de "x!1"com a calda de "x!2"e o primeiro passo para começar a prova foi instanciar o valor de "x"na hipótese para um caso específico. Depois utilizamos o LEMMA "occurences-of-app"que diz que o número de ocorrências de "x"na junção de uma lista "l1"com uma lista "l2"é igual o número de ocorrências de "x"em "l1"somado com o número de ocorrências de "x"em "l2". Copiamos o LEMMA para utilizarmos ele duas vezes e instanciamos no primeiro LEMMA as variáveis "x!1", "cdr(x!2)"e "x"; enquanto que na copia do LEMMA instanciamos "x!1", "x!2"e "x". Para terminar a prova desse ramo aplicamos **grind** que faz com que o PVS tente terminar a prova utilizando a aplicação de vários comandos de forma recursiva. O segundo caso do quarto ramo não possuia nenhuma hipótese somente antecedentes, dessa forma, aplicamos novamento o comando **grind** que terminou o segundo caso do quarto ramo, e assim terminou a prova da questão 2.

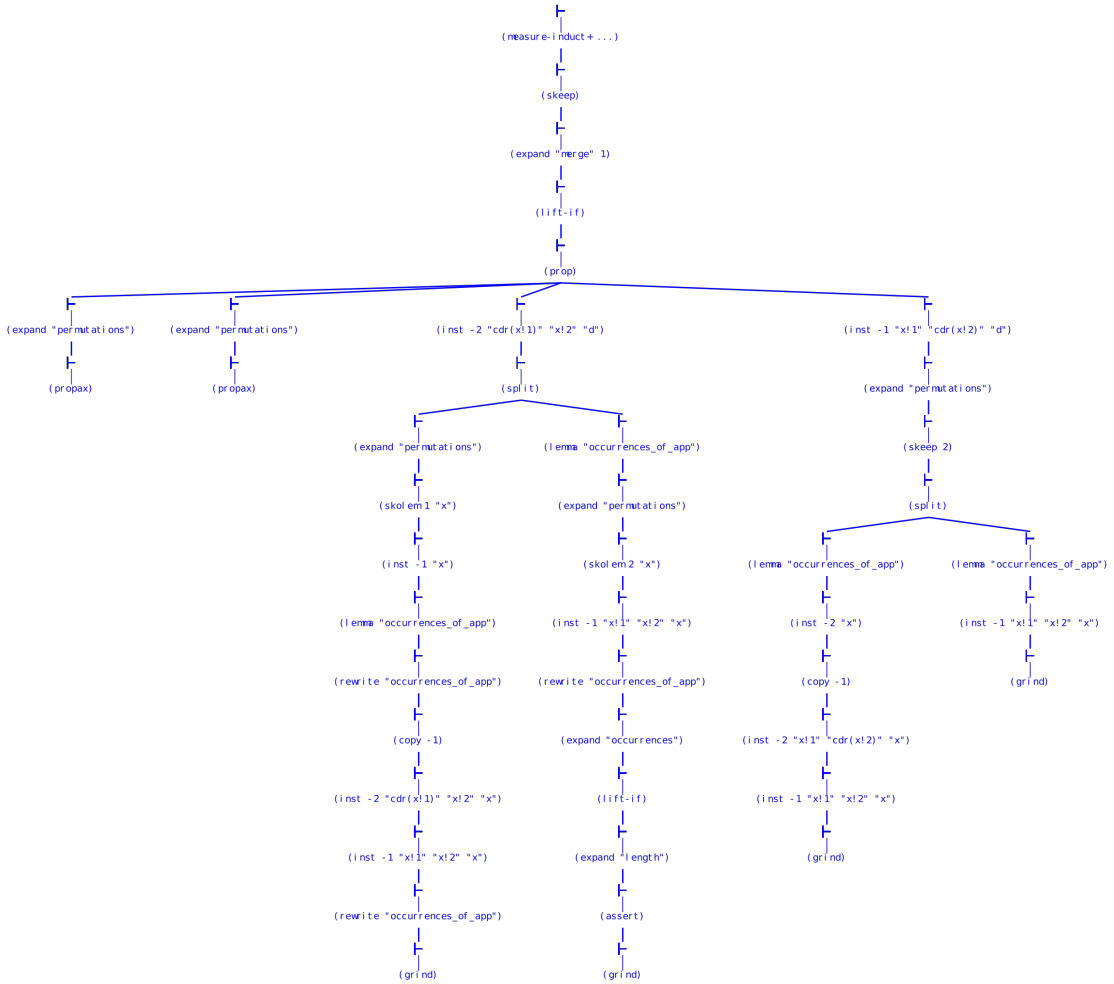

Figura 2: Árvore de prova da propriedade da questão 2

2.3 Questão 3

A tericera questão consistia na resolução final do problema, que era garantir que toda a lista está ordenada usando a estratégia do radix sort.

O grupo não teve sucesso na prova da questão, pois encontramos dificultades na

utilização dos lemas que serviriam de auxilio na prova, não sabiamos como encaixar ele na prova e como eles poderiam nos ajudar.

3 Conclusões

Durante a elaboração das soluções nós tivemos a oportunidade de aprender mais sobre o cálculo de sequentes e suas aplicações em diferentes provas, mas o grande diferencial foi ver a aplicação prática do Cálculo de Gentzen, que é provar a corretude de algoritmos que podem ser critícos em projetos maiores e de grande investimento.

Foi observado pelo grupo também que o PVS pode trazer algumas dificuldades para a prova de um algoritmo grande, não por conta de sua implementação, pois ele cumpre sua proposta muito bem, mas por contra do fato de ser necessário saber a sintaxe do PVS para provar algoritmos usando a plataforma, felizmente nesse projeto não foi necessário escrever algoritmos, apenas prova-lós, que ainda sim traz uma curva grande de aprendizado, pois muitas vezes ficamos presos em partes em que sabiámos a solução, mas não como expressar para o PVS.

Referências

[1] M. Ayala-Rincon and F.L.C. de Moura. *Applied Logic for Computer Scientists - computational deduction and formal proofs* -. Notas de aula, 2018.