Universidade de Brasília

Instituto de Ciências Exatas

Departamento de Ciência da Computação

Lógica Computacional 1

Relatório Radix Sort

Diego Antônio Barbosa Cardoso	16/0005116
Diego Vaz Fernandes	16/0117925
Gabriel dos Santos Martins	15/0126298

21 de Novembro de 2018

1 Introdução

Atualmente algoritmos são comumente usados na solução de problemas do nosso dia a dia, sendo então crucial termos certeza de que esse algoritmo está correto e funciona como deveria. Sendo assim, após a elaboração de um algoritmo, é importante mostrar que o mesmo está funcionando corretamente. Existem várias formas de se testar um algoritmo, como por exemplo testes unitários de software ou testes de integração mas é praticamente impossível testar todas as possibilidades e variações de um software e testes convencionais não garantem uma certeza absoluta sobre o funcionamento do software testado. Então em sistemas críticos que são sistemas que precisam funcionar perfeitamente pois um erro neles podem causar um problema muito grande como a morte de pessoas e outras tragédias temos que ter uma maneira melhor de garantir esse funcionamento e para isso vamos pras provas formais que nos permitem assegurar muitas propriedades do software que queremos testar, sendo a prova formar a técnica que foi utilizada neste projeto, utilizando a linguagem do assistente de demonstração PVS (*Specification and Verification System*) que é um software que auxilia na construção de provas formais.

2 Contextualização do Problema

O Radix sort é um algoritmo de ordenação rápida e estável que pode ser usado para ordenar chaves únicas.Cada chave é uma cadeia de caracteres ou cadeia de números, e o radix sort ordena estas chaves em qualquer ordem relacionada com a lexicografia. Um algoritmo é dito estável se a posição relativa de dois elementos iguais permanece inalterada durante o processo de ordenação.

Existem duas classificações para o Radix Sort: LSD (*Least significant digit – Dígito menos significativo*) e MSD (*Most significant digit – Dígito mais significativo*).

Como já dito anteriormente o Radix Sort ordena chaves em uma ordem qualquer utilizando como critério a lexicografia e para tal ordenação é preciso de um algortimo auxiliar pra realizar a ordenação propriamente dita. como queriamos ter um algortimo estável escolhemos o Merge Sort para isso. Ele que tem seu famoso processo de dividir para conquistar. Sua ideia principal consiste na Divisão de um problema em vários partes menores e resolver essas partes menores dividindo-as novamente em mais partes menos repetindo esse processo até a divisão não ser mais possivél sendo esse um processo dito recursivo. Então no nosso problema o papel do merge sorte é ordenar as partes das listas que forem enviadas pra ele seguindo esse processo descrito acima

3 Explicação das Soluçoes

3.1 Questão 1

Na questão 1, tinhamos que provar que a "conjecture"mostrada na figura 1 era verdeira.

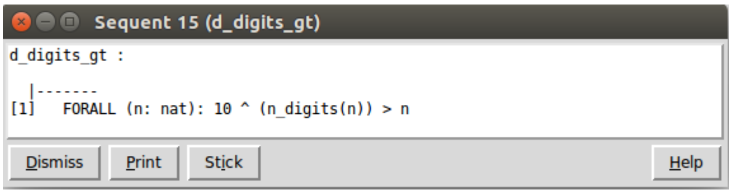

Figura 1: Conjecture da questão 1.

Começamos a prova utilizando **(measure-induct+ "n" ("n"))**, como foi pedido na questão, pois se tratava de indução forte. O measure-induct nos deu a hipótese mostrada na 2.

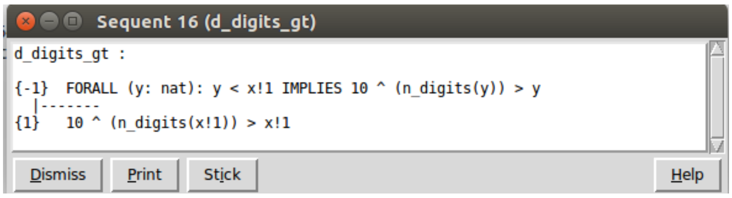

Figura 2: Hipótese fornecida.

Utilizamos um *(expand "n_ digits" 1)* para expandir a função *n_ digits* e nos dar mais opções para trabalhar. Depois do expand, ele trocou o n_digits pela declaração da função. Como tínhamos um *10^ IF*, jogamos o 10 para dentro do IF, para podermos trabalhar somente com o *IF ELSE* utilizando o *(lift-if)* que nos deu um bloco *IF THEN ELSE*. Como tínhamos esse bloco, utilizamos um *(prop)* para quebrar *IF THEN ELSE* em todas as opções possíveis nos dando duas folhas.

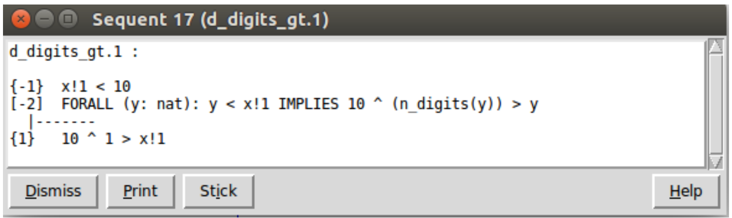

Figura 3: Folha 1 da esquerda formada após o comando "prop".

Do lado esquerdo, Figura 3, conseguimos enxergar claramente que a expressão -1 e a expressão 1 são equivalentes. Pórem, o pvs não tem a mesma capacidade de percepção. Por isso tivemos de expandir o "^", para obter uma função que o pvs consiga entender melhor. Após, o expand no ^, ele nos retornou outra função equivalente ao ^: a função *expt*. Usando o *expand no expt* chegamos na situação mostrada na figura 4 .

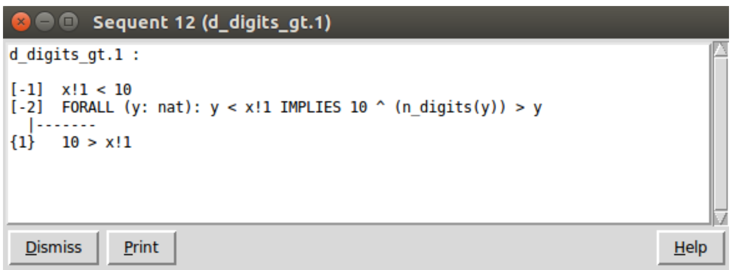

Figura 4: Raiz da folha 1.

Tínhamos duas expressões, uma de cada lado, que diziam a mesma coisa. A expressão -1 e a expressão 1. Porém, o pvs ainda não tinha reconhecido que essa duas expressões estavam afirmando a mesma coisa. Então usamos o comando *Assert* para conseguir fechar a folha.

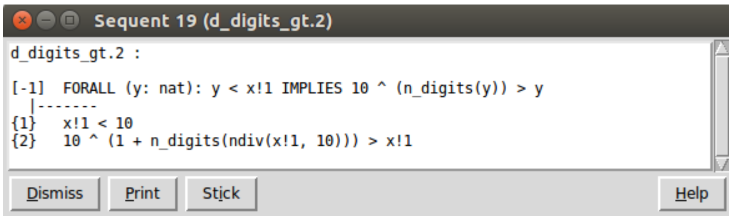

Figura 5: Folha da direita formada após o comando "prop".

Já no lado direito, Figura 5, tínhamos um *ndiv(x!1,10)* e em cima não, por isso usamos o *(inst -1 "ndiv(x!1,10)")* para tentar fazer aparecer um *ndiv* em cima também. Como tínhamos uma implicação *(IMPLIES)*, utilizamos o comando *(split)* para quebrarmos essa implicação em duas folhas, como mostrado na figura ??.

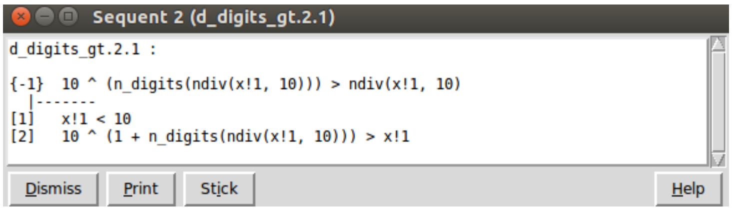

Figura 6: Sub-árvore da esquerda.

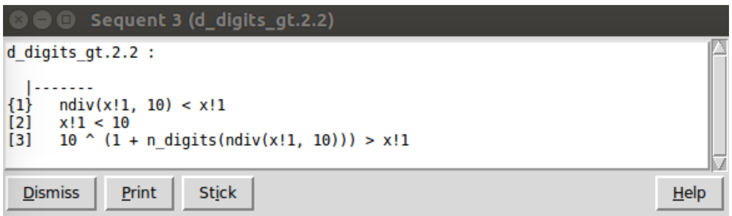

Figura 7: Sub-árvore da direita.

A primeira folha, a da esquerda, Figura 6, tínhamos coisas parecidas em cima e em baixo, a estratégia foi utilizar o comando *(expand "^" 2)*, para expandir o ^ do 2 O *(expand)* nos deu um *IF THEN ELSE* que carrega consigo a definição matemática da exponenciação. A primeira condição é sempre positiva, pois para qualquer numero que for passado para essa função, ela retornará a quantidade de digitos que possuia o numero passado. E isso nunca poderá ser um número negativo. Demos um *(assert)* para provar o IF, que é sempre verdade, e como podemos ver, ele entrou somente no caso onde era verdade, ou seja, provamos que era uma tautologia. Ficamos somente com a parte verdadeira do *IF THEN ELSE*. Após isso demos um *(expand)* no *"expt"*. Em seguida demos outro *(expand)*, mas agora na nossa hipótese, para ficarmos com *"expt"* em cima e embaixo. Utilizamos o *(typepred)* para buscar hipóteses que já utilizamos anteriormente, no caso foi e buscamos a *(typepred "ndiv(x!1, 10)")*. Já com as hipóteses utilizadas anteriormente, demos um novo *(expand "<=" -2)* que nos retornou um *OR do <=*. Utilizamos o *(split)* para fazer a eliminação do OR dividindo-o em 2 folhas. Na folha da esquerda, utilizamos o comando *name-replace* para atribuir a uma variável o valor da expressão que tínhamos, para conseguirmos ver melhor, e assim, fazer com que o pvs também enxergasse melhor.

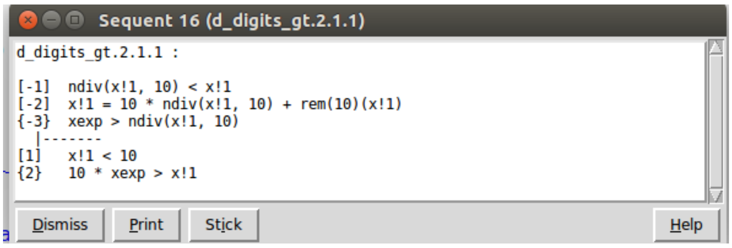

Figura 8: Sequente após a substituição

Com isso, temos uma igualdade na hipotese -2. Substituimos essa igualdade nas outras expressões e percebemos que a expressão -3 e a expressão 2 eram equivalentes, porém mais uma vez o pvs não conseguia exergar isso. Tentamos muitas manerias de provar isso, porém nenhuma foi satisfatória. Então demos um **assert** e conseguimos fechar a árvore.

Já na outra folha que foi gerada a partir do slip, tinha a mesma situação que na folha anterior, então fizemos o mesmo processo.

3.2 Questão 2

Começamos a segunda questão de uma maneira bastante parecida com a questão 1 então utilizamos o *(measure-induct+ "length(l1)+length(l2)"("l1l2")* pois novamente precisamos de indução forte para realizar essa prova. Utilizamos (skeep) para remover o forall que se encontrava na parte de baixo e então demos um *(expand "merge"1)* para termos mais opções para trabalhar na prova. Depois disso utilizamos *(lift-if)* e *(prop)* sendo o primeiro para propagarmos para dentro do if os termos que ficaram fora e prop para quebrar if nos casos possíveis. Com esse prop nossa árvore se abriu em 4 ramos onde os dois primeiros foram triviais pois ao utilizarmos *(expand "permutations")* conseguimos o chegar ao *(propax)*os outros 2 ramos não são nada triviais como os primeiros aqui já citados. Basicamente eles eram bem parecidos onde no primeiro tínhamos um forall (onde dentro tinha um *merg(cdr(x1) x2) (x)*) e o segundo a mesma coisa.

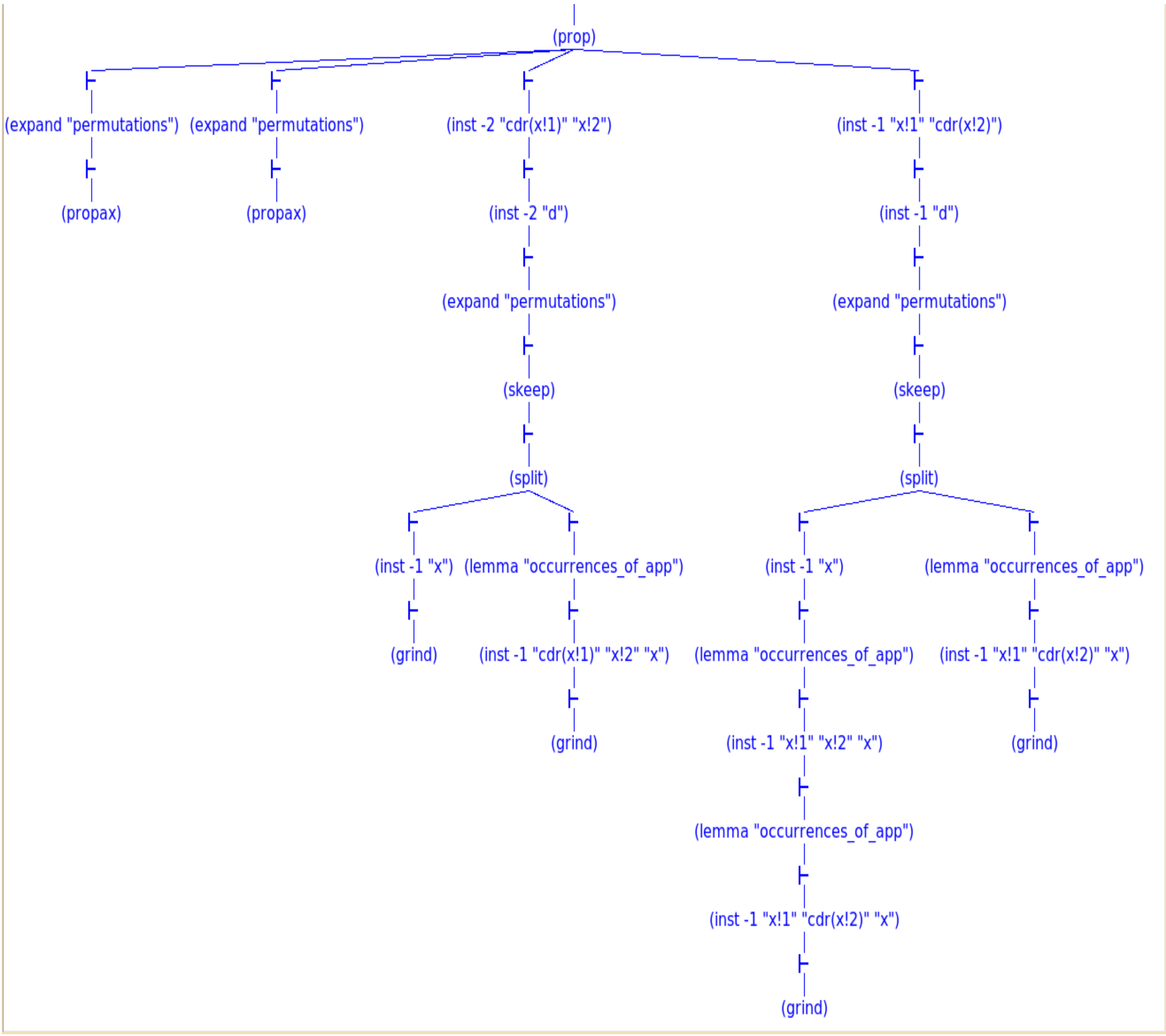

Figura 9

Então nos dois ramos tivemos ideias bem parecidas pois acima tínhamos um forall e para removê lo precisamos usar um inst então usamos insts de modo que em cima e embaixo ficassem iguais: *(inst -2 "cdr(x!1)x!2") (inst -1 "x!1cdr(x!2)x")*.
Então tentamos nos aproximar o máximo possível porém como já citado anteriormente como o grupo não tinha pleno domínio sobre os comando do PVS não conseguimos deixar os dois exatamente iguais então apelamos para o grind pra fechar a prova.

Outra coisa importante de se falar é que tentamos essa prova a priori por outro caminho sem dar expand no merge no início e por conta disso entramos em um loop infinito

4 Especificação do problema e explicação do método de solução

O primeiro problema consistia em provar a conjectura *d_digits_gt*, utilizando indução forte e também foi utilizado algumas propriedades de resto da divisão, e a divisão de dois

números inteiros. *rem(a)(b)* e *ndiv(a,b)*
Na segunda questão, provamos utilizando indução forte a conjectura merge_permutes com o auxilio do lema: *merge_sort_permute*. No inicio, tivemos um problema de toda folha chegar no mesmo lugar, pois acabamos em loop infinito. Depois disso decidimos ir por outro caminho, começando a prova com utilizando *skeep* para remover os *FORALL* e dando um *(expand "merge"1)* o que nos tirou do loop.

5 Descrição da formalização

A formalização da prova do algoritmo *Radimerge_sort_permutex Sort* se deu a partir das três questões propostas. Na questão 1, consistia em demonstrar a conjectura ***d_digits_gt***, que dizia que para qualquer **natural n**, *10 ^ n_digits(n) > n*
A questão 2 temos a utilização do merge que preserva os elementos das listas dadas como argumento, onde temos a conjectura *merge_permutes*:

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

Foi utilizada também uma função merge que recebe duas listas de naturais como argumento e um natural d, e retorna um merge dessas listas.
A questão 3 é referente a estabilização do merge_sort.

6 Conclusões

Ao fim das resoluções das questões 1 e 2 do projeto , tivemos uma importante experiência de como que é feita uma prova formal de um algoritmo. suas dificuldades sua importância seus resultados e como deve ser utilizada. Para esse projeto utilizamos o PVS este que se mostrou uma ferramenta bastante poderosa na elaboração de provas formais de um algoritmo. Uma das principais dificuldades foi o grupo não estar totalmente familiarizado com a sintaxe e com os códigos em LISP, e a lém disso a interface e usabilidade do pvs não ser muito boa nem intuitiva Mas mesmo com todas essas adversidades foi possível compreender a importância de uma ferramenta que auxilie em provas formais de algoritmos. Pois através dela conseguimos assegurar que uma prova está correta e que não estamos utilizando uma regra ou propriedade de maneira leviana através das regras do cálculo de Gentzen e utilizando indução estrutural , foi possível possível a prova das questões já vistas neste documento.

7 Lista de referências

M.Ayala-Rincon and F. L. C. de Moura. Applied Logic for Computer Scientists - Computational Deduction and Formal Proofs. Undergraduate Topics in Computer Science. Springer, 2017.