Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Formalização da correção do algoritmo Radix Sort utilizando PVS Specification and Verification System

Brasília, 17 de junho de 2019

Lista de Figuras

Capítulo 1

Introdução

A formalização de teoremas é de extrema relevância na área da computação. Provar um teorema garante que o que se está querendo provar é válido sob alguma perspectiva e pode ser utilizado como base para resolução de outros problemas. Entretanto, provar formalmente teoremas utilizando regras disponíveis em algum tipo de lógica pode se tornar uma tarefa extremamente complexa para ser realizada manualmente. Para auxiliar tal atividade, existem *softwares* que auxiliam na resolução de provas de maneira mecanizada, através da aplicação de heurísticas e das regras de uma lógica específica, dado que provar um teorema através de um algoritmo é um problema indecidível.

Na disciplina de Lógica Computacional 1 do curso de Ciência da Computação da Universidade de Brasília, estudam-se majoritariamente dois sistemas para resolução de provas formais: Dedução Natural e o Cálculo de Sequentes (ou Cálculo de Gentzen)[1]. Tais sistemas estabelecem regras que podem ser aplicadas sobre estruturas representativas e permitem provar os chamados **Sequentes**, que introduzem inferências ou informações à respeito de algo. Na disciplina é introduzido ainda o uso da ferramenta **PVS Specification and Verification System**, *software* que utiliza o sistema lógico de Cálculo de Sequentes para auxiliar na resolução de provas.

Como método de aplicação do conteúdo ministrado na disciplina, foi estabelecido como projeto prático a utilização do **PVS** para formalizar propriedades de uma especificação do algoritmo de ordenação *Radix Sort* aplicado ao conjunto $\mathbb{N}$ dos naturais. Tal especificação se baseia na utilização do algoritmo de ordenação estável *Merge Sort* (este já provado) em passos intermediários.

Este trabalho está organizado da maneira a seguir. Na próxima seção será especificado melhor o problema e o método de solução utilizado. Após, serão detalhadas as soluções apresentadas para a resolução do problema. Então, será exposta a formalização final da prova. Finalmente, serão indicadas as conclusões deste projeto.

Capítulo 2

Explicação do problema e do método de solução

A utilização de algoritmos de ordenação é plenamente presente em soluções da computação. Ordenar conjuntos é um problema fundamental e que, por vezes, pode necessitar de soluções rápidas, tendo sempre a necessidade de garantia da correção de sua aplicação sobre um conjunto de dados.

Um algoritmo amplamente conhecido e utilizado em diversas soluções é o *Radix Sort*. Aplicações de *Radix Sort* podem ser implementadas quando é conhecida alguma informação sobre a estrutura das chaves a serem ordenadas. A ideia do algoritmo citado, quando aplicado sobre um conjunto **N** de números naturais $\mathbb{N}$, consiste em ordenar os mesmos a partir da observação das colunas dos números (unidades, dezenas, centenas...), ordenando-as utilizando outro algoritmo de ordenação em passos intermediários.

A ordenação utilizada nos passos intermediários é feita utilizando o algoritmo estável *Merge Sort*. Pode-se dizer que um algoritmo de ordenação é **Estável** quando este não altera a posição relativa de dois elementos iguais durante o processo de ordenação. A prova de tal algoritmo foi formalizada em projetos de períodos anteriores, o que permite a utilização do mesmo para completar a formalização da correção do *Radix Sort*.

Foram solicitadas as formalizações de 3 propriedades específicas da aplicação de *Radix Sort* utilizada neste projeto:

1. **radix-sort-d-sort**: esta propriedade afirma que, se uma lista **N** de naturais $\mathbb{N}$ está ordenada até o seu *d-ésimo* dígito, a aplicação de *Radix Sort* entre os dígitos **d** e **k** arbitrário resultará na ordenação da lista até o dígito **k+1**.

2. **radixsort-permutes**: esta propriedade está relacionada ao fato de a função *Radix Sort* preservar os argumentos das listas utilizadas como argumento, avaliando as permutações dos elementos em relação à lista original.

3. **radixsort-sorts**: esta propriedade consiste, finalmente, na afirmação de que a aplicação de *Radix Sort* sobre uma lista **N** de naturais $\mathbb{N}$ gera uma lista ordenada.

Para formalizar a prova de cada uma das propriedades citadas, foi utilizado o **PVS Specification and Verification System** como assistente de provas, se atentando para a

estrutura das fórmulas utilizadas bem como à correta aplicação das regras relacionadas ao Cálculo de Sequentes. Através de comandos específicos, o **PVS** permite aplicar diversas regras disponíveis no Cálculo de Gentzen a fim de gerar provas formais de teoremas.

A aplicação das regras permite formalizar provas segundo a lógica Clássica, construindo uma árvore de prova que começa com a proposição disponibilizada e termina com axiomas proposicionais em seus galhos. Tal árvore de prova é construída pelo **PVS** e pode ser visualizada através da execução do comando `xpr`.

Capítulo 3

Explicação das soluções

3.1 Propriedade 1: *radix-sort-d-sort*

A primeira conjectura diz que se ordenamos os primeiros d-dígitos com *radixsort* até k-dígitos sendo d <= k então o próximo dígito estará ordenado também , ou seja, k+1-dígitos estará ordenada. Sendo a conjectura definida desta forma em *pvs*:

$\forall(l : list[nat], k : nat, d : nat | d <= k) :$
is_sorted_ud?(l,d) $\implies$ *is_sorted_ud?(radixsort(l,k,d), k+1)*
Agora começando a prova com quantificador universal a direita podemos usar o comando *(skeep)* que em Cálculo de Gentzen é equivalente ao $(R\forall)$.

Sendo assim a árvore geradora fica assim após o comando:

$$\frac{\forall A \qquad \textit{is\_sorted\_ud?(l, x!2)}}{\textit{is\_sorted\_ud?(radixsort(l,x!1, x!2), x!1+1)}}$$

Então expandimos o *radixsort* e vemos os dois casos bases do algoritmo que utiliza o algoritmo *mergesort* como algoritmo auxiliar. O primeiro caso se o d-dígito ja for igual ao k-digito então aplica-se *mergersort* se fazer a regra do *Corte* e aplicando o caso de x!1 = x!2 então chegamos em :

$$\frac{\forall A \qquad x!1 = x!2}{\textit{is\_sorted\_ud?(mergesort(l,x!1), 1 + x!1}}$$

Como já foi provado como lema que o *merge_sort_d_sorts* utilizaremos ele. E novamente utilizaremos a regra do *Corte* aplicando o lema e fazendo as devidas instanciações , chegamos a esta conclusão :

$$\frac{\forall A \qquad \textit{is\_sorted\_ud?(l,x!1)} \implies \textit{is\_sorted\_ud?(mergesort(l,x!1) x!1 + 1)}}{\textit{is\_sorted\_ud?(mergesort(l,x!1) x!1 + 1)}}$$

Com a implicação na esquerda faremos $(L \rightarrow)$ que tem como equivalente em *pvs* o comando *(split)* dividindo em dois subcasos onde meu consequente da implicação vai para direita e meu antecedente para esquerda.

Olhando para o meu consequente do lema aplicado em cima podemos ver intuitivamente que folha da esquerda irá fechar. E por causa de existir um axioma com $is\_sorted\_ud?(l, x!2)$ e $x!1 = x!2$ então subárvore da direita irá fechar.

Agora fechado o lado esquerdo da prova vamos para seu segundo caso e já sabendo que x!1 = x!2 provado anteriormente faremos a instaciação do $\forall A$ chegando a esta conclusão sendo A = $x!1-(1+x!2) < x!1-x!2$ e B = $is\_sorted\_ud?(mergesort(l, x!2)x!2+1)$.

$$\frac{A \implies B \implies \textit{is\_sorted\_ud?(radixsort(mergesort(l,x!2),x!1, 1 + x!2), x!1+1)} \qquad \text{C}}{\textit{is\_sorted\_ud?(radixsort(mergesort(l,x!2),x!1, 1 + x!2), x!1+1)}}$$

Fazendo novamente o comando de *(split)* chegamos em tres casos. O caso mais esquerda da árvore é intuitivamente resolvido já que possuimos a mesma propriedade na direita e na esquerda do sequente sendo está propriedade $is\_sorted\_ud?(radixsort(mergesort(l, x!2), x!1, x!2), x!1+1)$. O terceiro caso ou também o caso mais a direita da sub-árvore também é trivialmente resolvido pois temos $x!1-(x!2+1) < x!1-x!2$ do lado direito do sequente sendo está equação verdadeira chegamos há um axioma.O caso que não é intuitivo e o segundo caso onde obtemos $is\_sorted\_ud?(l, x!2)$ do lado esquerdo do sequente então aplicamos a o *Corte* com o lema *merge_sort_d_sorts* e fazendo a instanciação do lema conseguimos a implicação *is_sorted_ud?(l,x!2)* $\implies$ *is_sorted_ud?(mergesort(l,x!2) x!2 + 1)* com o comando *split* percebemos intuitivamente que fecharemos os dois lados das subárvores . No primeiro caso teremos:

$$\frac{\text{A} \qquad \textit{is\_sorted\_ud?(mergesort(l,x!2) x!2 + 1)}}{\textit{is\_sorted\_ud?(mergesort(l,x!2) x!2 + 1)}}$$

Observe temos a mesma propriedade dos dois lados do sequente então chegamos em um Axioma. Agora no segundo caso teremos:

$$\frac{\textit{is\_sorted\_ud?(l,x!2)} \qquad \text{B}}{\textit{is\_sorted\_ud?(l,x!2)}}$$

Fechando assim o segundo caso do *(split)* e o segundo caso do *radixsort* e terminando a prova.

3.2 Propriedade 2: *radixsort-permutes*

A segunda propriedade consiste em mostrar que uma ordenação de uma lista qualquer ordenada por *Radix Sort* consiste em uma permutação dos elementos da lista original. A propriedade é apresentada afirmando que: para toda lista l de naturais e inteiros **k** e **d**, ***d* <= *k***, então, a lista gerada ao aplicar *Radix Sort* entre os dígitos k e d corresponde a uma permutação da lista l original.

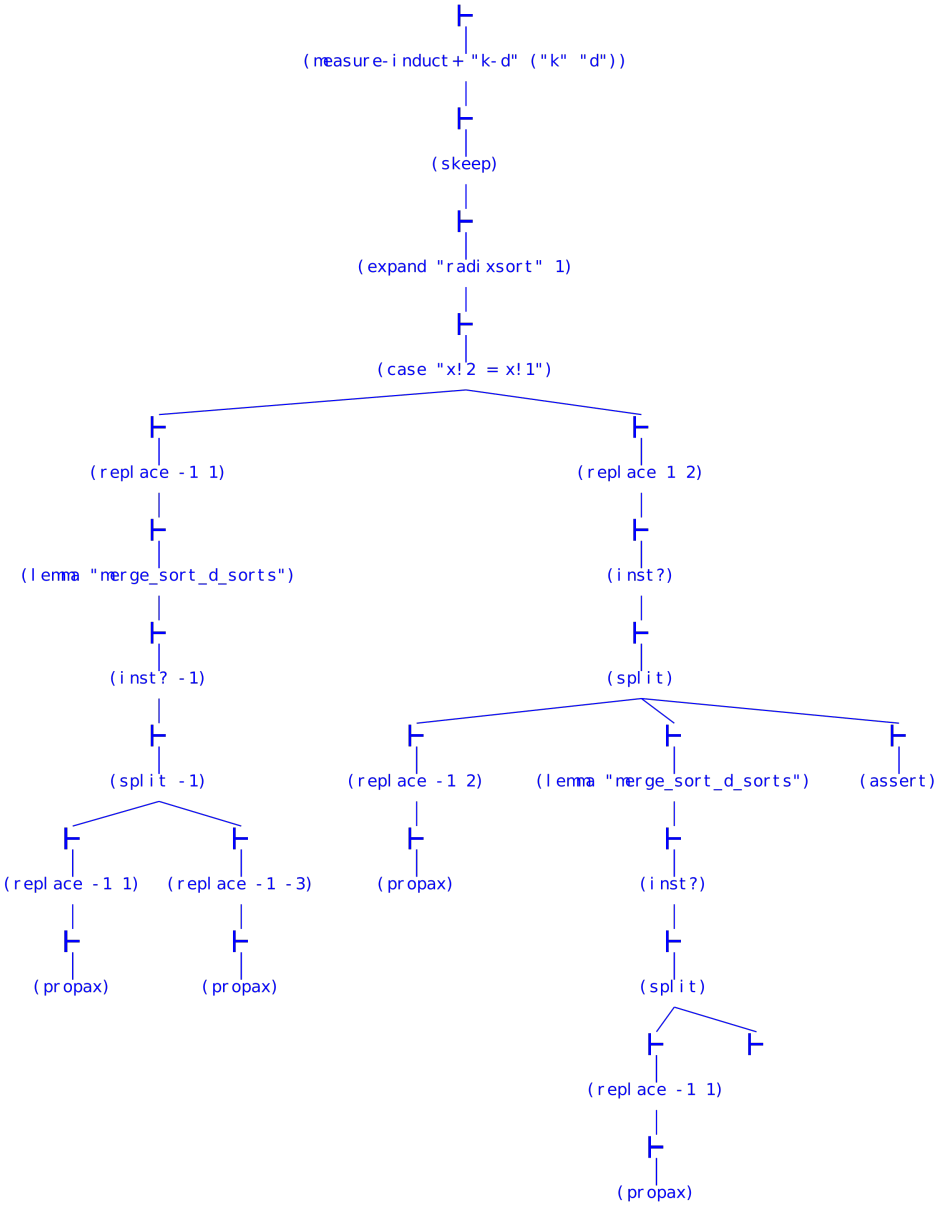

Figura 3.1: Árvore de prova da propriedade *radix-sort-d-sort*

Após aplicar o ***princípio da indução forte*** sobre ***k-d*** e aplicar o comando *skeep* que corresponde à regra $(R\forall)$, podemos enxergar um caso no qual a regra do corte *cut* pode ajudar, através da aplicação do comando *case*. Com isso, podemos aplicar a definição para o caso específico em que ***x!1 = x!2*** e dividir nossa prova em 2 ramos.

No ramo à esquerda, temos a hipótese de que ***x!1 = x!2*** no antecedente e temos um caso particular onde podemos aplicar a regra *replace*. A partir daí, podemos aproveitar a definição do lemma $merge_sort_permutesparatrazerparaoantecedenteahipótesequetemosnoconsequentee$

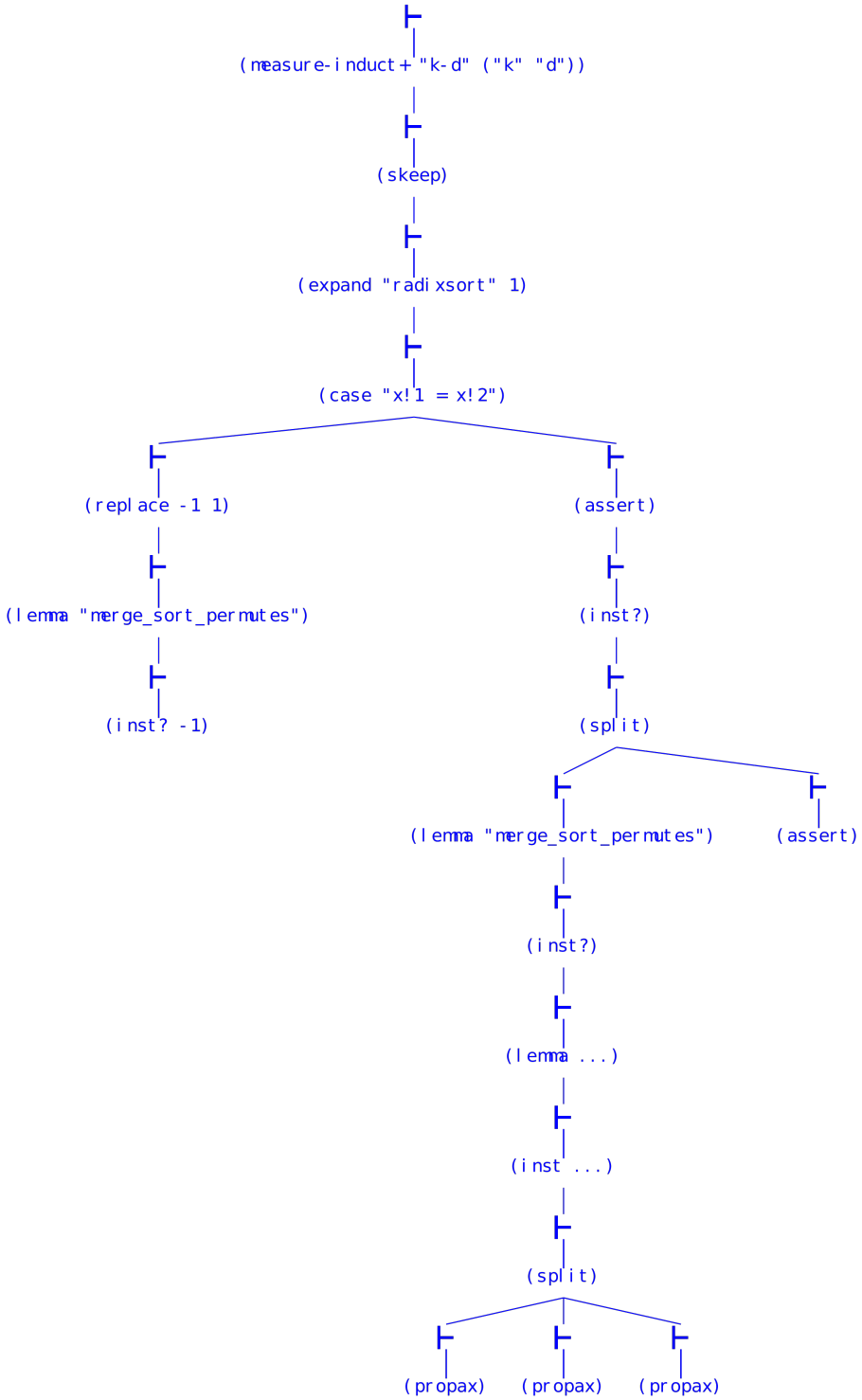

Figura 3.2: Árvore de prova da propriedade *radixsort-permutes*

3.3 Propriedade 3: *radixsort-sorts*

A terceira propriedade consiste em afirmar que, dado uma lista qualquer, podemos ordenar seus elementos utilizando o algoritmo radixsort. Dessa forma, é muito intuitivo o uso de

algumas propriedades que já foram provadas anteriormente, como radix-sort-d-sort.

Portanto, temos o sequente: $\longrightarrow \forall l.is\_sorted?(radixsort(l))$*, sendo* l *uma lista de naturais.*

A partir disso, a estratégia para a prova sugerida pelo professor foi começar pelo comando skeep, que instância o quantificador universal que temos a direita do sequente. O próximo passo tomado foi expandir a definição de radixsort, dado que na definição o algoritmo é dividido em dois subcasos, o que dividira o problema em dois casos menos complexos ao utilizar o comando prop, que separa fórmulas proposicionais em seus casos.

Assim, a árvore gerada após aplicar os comandos seria:

$$\frac{length(l) <= 0 \longrightarrow is\_sorted?(l) \qquad \longrightarrow length(l) <= 0, is\_sorted?(radixsort(l, max\_digits(l), 0))}{\longrightarrow \forall l.is\_sorted?(radixsort(l))}$$

É possível perceber que o lado esquerdo da árvore tem prova simples, dado que ele afirma que, dado uma lista com 0 ou menos elementos, essa lista está ordenada, e uma das propriedades da lista vazia é estar ordenada. Dessa forma, focaremos no lado direito da prova.

Do lado direito, temos que, sabendo que o tamanho da lista é maior que zero, provar que $radixsort(l, max\_digits(l), 0)$ *ordena uma lista por completo. Sabendo que a função max_digits retorna o número de digitos do valor que tem mais dígitos da lista, o que o comando quer dizer é que, ao aplicar radixsort nas colunas de digitos 0 até a última da lista l, obteremos uma lista ordenada.*

Com isso, como conhecendo a propriedade radix-sort-d-sort, percebe-se que o sequente é muito similar, ao utilizar a propriedado, podemos instância-la de forma a ser útil, dado que ela afirma que a partir de uma lista ordenada até o d-ésimo dígito, podemos ordenar a partir dele até o k-ésimo usando radixsort. Assim, precisamos apenas mostrar que a partir de uma lista ordenada a partir do 0-ésimo dígito, ou seja, com nenhum dígito ordenado, ordenaremos até o max_digit(l), ou seja, até o último dígito usando o radixsort.

Após importar a propriedade 1 e instância-la corretamente, dividiremos a árvore de provas novamente, os novos sequentes serão:

$$\frac{A \longrightarrow B \qquad \longrightarrow is\_sorted\_ud?(l, 0)}{\longrightarrow B}$$

Sendo:

$A = is\_sorted\_ud?(radixsort(l, max\_digits(l), 0), max\_digits(l) + 1)$

$B = length(l) <= 0, is\_sorted?(radixsort(l, max\_digits(l), 0))$

A partir da árvore, é possível notar que o sequente $\longrightarrow is\_sorted\_ud?(l, 0)$ *tem prova simples, dado que afirma que não é preciso de nenhuma hipótese para provar que qualquer*

lista está ordenada até seu 0-ésimo dígito, sendo que os dígitos são contados a partir de 1.

Portanto, focaremos na prova do sequente $A \longrightarrow B$. *Nessa prova, basta apenas mostrar que existe equivalência entre as definições de is_sorted_ud? e is_sorted? no caso em questão. Para que isso seja possível, após aplicar alguns comandos, é necessário provar alguns subcasos. Sendo esses:*

$$\frac{k <= length(l, max\_digits(l), 0) - 2}{k + 1 <= length(l, max\_digits(l), 0)}$$

O que efetivamente é uma igualdade, dado que o predicado length tem como retorno um natural.

$$\frac{}{0 <= nth(radixsort(l, max\_digits(l), 0), k)}$$

O que também é de fato verdade, já que o retorno de radixsort é uma lista de naturais, k-ésimo elemento dessa lista é um natural, portanto, maior ou igual a 0 por definição.

$$\frac{rem(A)(B) <= rem(A)(C)}{B <= C}$$

Sendo:

$A = 10^(1 + max\_digits(l))$

$B = nth(radixsort(l, max\_digits(l), 0), k)$

$C = nth(radixsort(l, max\_digits(l), 0), k + 1)$

O que é fácil de provar usando o lemma "rem_mod2", entretanto, o uso desse lemma faz com que seja necessário provar $B < A$ *e* $C < A$ *para os mesmos A, B e C definidos acima. E, ao provar esses dois últimos subcasos, a prova de radixsort-sorts é completa.*

Contudo, para que isso seja possível, foi necessário criar um outro lemma, o max-digit-gt, que afirma que $\forall k.10^(max\_digits(l)+1) < nth(l, k)$, *que tem uma prova bastante intuitiva, dado que 10 elevado ao número de digitos do elemento com mais digitos de uma lista é, por definição, maior que o próprio número. Uma forma simples de se notar isso é pensar no número 999, o maior que possui 3 dígitos e, 10 elevado a 3 é 1000 que por sua vez é maior que 999.*

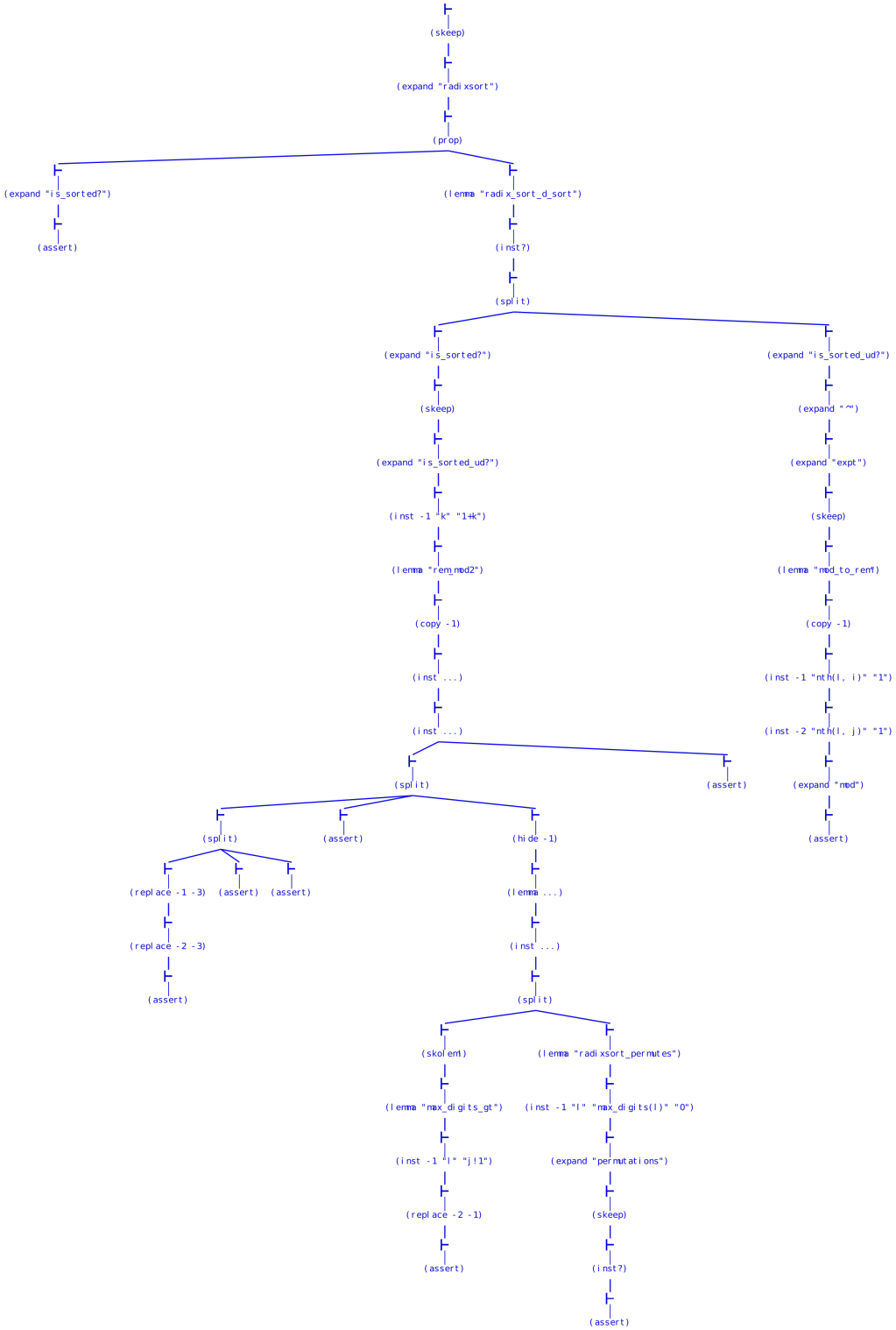

Figura 3.3: Árvore de prova da propriedade *radixsort-sorts*

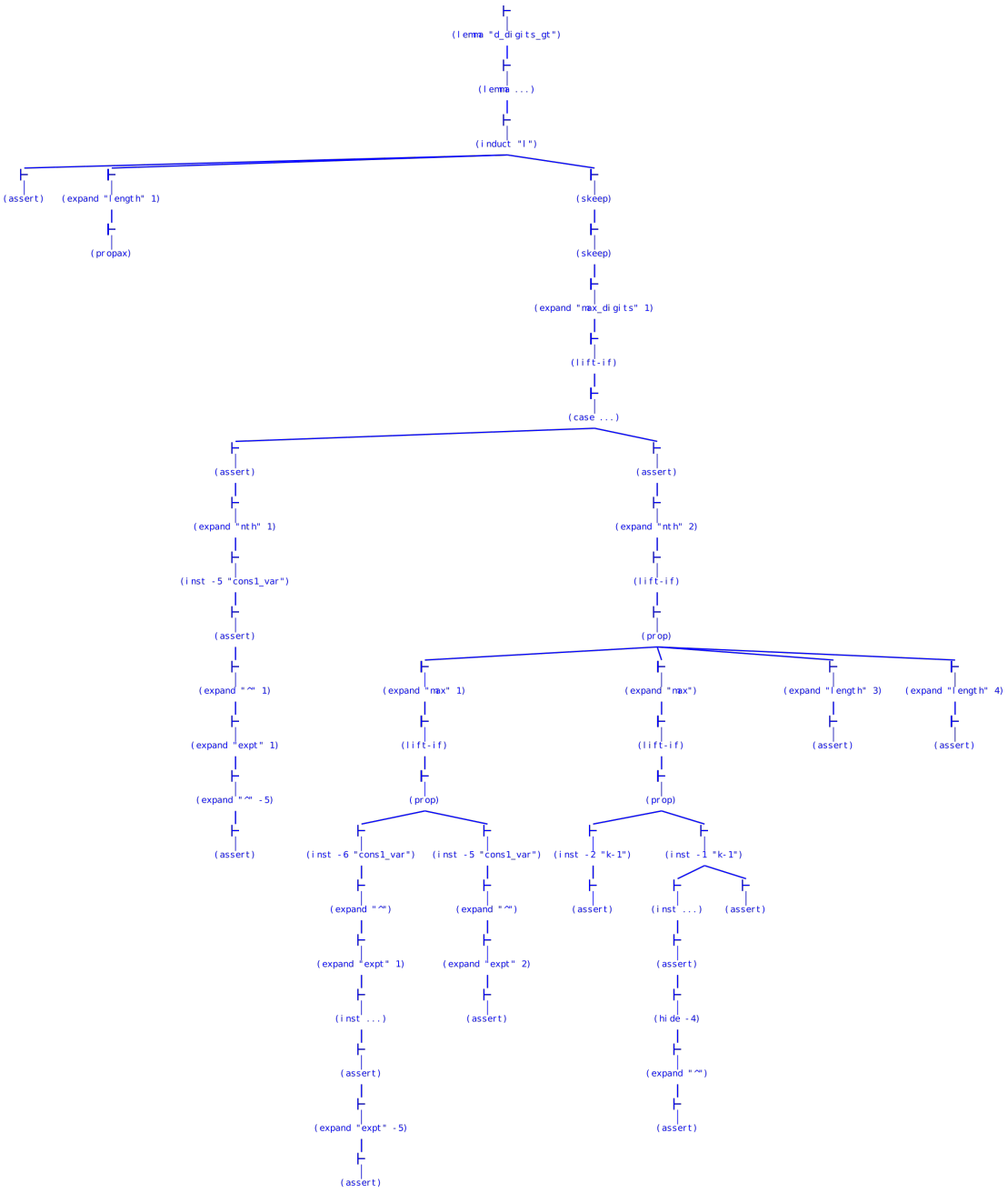

Figura 3.4: Árvore de prova do lema *max-digits-gt*

Capítulo 4

Conclusões

Depois de abordamos todos os capítulos anteriores onde revemos os motivos de usar provadores mecânico para provas formais e a importância da utilização do ***PVS Specification and Verification System*** *e como o algoritmo Radix Sort funciona, sua prova de corretude e suas ligações de sua prova formal no* ***PVS Specification and Verification System*** *com Calculo de Gentzen[1]. Concluímos que a utilização de provadores mecânicos dão créditos as nossas provas pois com eles todos os passos não considerados nas provas escritas passam a ser avaliados deixando assim o sistema de correção de provas consistente afim de produzir resultados condizentes. Porém com grande detalhamento do problema em provas formais pode acabar dificultando problemas até então de simples solução para um problema complexo.*

Referências

Referências

[1] Ayala-Rincón, Mauricio e Flávio LC De Moura: Applied Logic for Computer Scientists: Computational Deduction and Formal Proofs. *Springer, 2017. 1, 12*