Formalização da Correção e Complexidade do Algoritmo de Ordenação por Inserção

Lucas de Sena

Silvio Castanheira Oddone

1. Introdução

A proposta deste projeto é formalizar, para o algoritmo de ordenação por inserção, a sua correção e sua complexidade temporal no pior caso. A formalização e a prova foram realizadas no assistente de provas Coq.

O algoritmo de ordenação por inserção (*insertion sort*) foi definido em listas (em vez de sobre vetores) e é implementado de forma recursiva (em vez de forma iterativa). Como listas são definidas indutivamente em Coq, uma abordagem recursiva é a forma mais natural de se manipular listas.

$$ordena(l) := \begin{cases} nil, & se\ \ l = nil \\ insere(h,\ ordena(tl)), & se\ \ l = h::tl \end{cases}$$

Figura 1: Definição recursiva do algoritmo de ordenação por inserção

Da forma que definimos, o algoritmo aplicado em uma lista não vazia retorna a inserção do primeiro elemento (*cabeça*) sobre a versão ordenada do resto (*cauda*), em uma posição que mantenha a lista resultante ordenada. O algoritmo depende portanto da definição de uma função auxiliar que recebe um elemento e uma lista ordenada.

$$insere(x,\ l) := \begin{cases} x::nil, & se\ \ l = nil \\ x::l, & se\ \ l = h::tl\ \ e\ \ x \le h \\ h::insere(x,\ tl), & se\ \ l = h::tl\ \ e\ \ x > h \end{cases}$$

Figura 2: Definição recursiva da função auxiliar de inserção

2. Da Correção

A formalização da correção do algoritmo de ordenação por inserção requer enunciar e provar o teorema que estabelece que, para qualquer lista dada como entrada, a lista resultante será ordenado, e será uma permutação da lista original.

$$\forall\ l{:}lista,\ \ ordenado(l') \ \wedge\ \ permutado(l, l'),\ \ onde\ \ l' := ordena(l).$$

Figura 3: Teorema da correção da ordenação por inserção

Temos então dois ramos da prova, cada um enunciado por um lema auxiliar.

$$\dfrac{\dfrac{\dfrac{}{\forall\ \text{l} \vdash \text{ordenado(ordena(l))}}\ (?) \qquad \dfrac{}{\forall\ \text{l} \vdash \text{permutação(l, ordena(l))}}\ (?)}{\forall\ \text{l} \vdash \text{ordenado(ordena(l))} \wedge \text{permutação(l, ordena(l))}}\ (\wedge_{elim})}{\vdash \text{correto(ordena)}}\ (\wedge_{elim})$$

Figura 4: Prova da correção da ordenação por inserção

2.1. Prova de que o resultado é ordenado

Antes de enunciar o lema que garante a ordenação, precisamos definir o predicado *ser ordenado*. Esse predicado é definido indutivamente com três construtores.

$$\dfrac{}{\vdash \text{ordenado(nil)}}\,S_0 \qquad \dfrac{}{\text{x} \vdash \text{ordenado(x::nil)}}\,S_1 \qquad \dfrac{\text{x y} \vdash \text{x} \leq \text{y} \qquad \text{y l} \vdash \text{ordenado(y::l)}}{\text{x y l} \vdash \text{ordenado(x::y::l)}}\,S_n$$

Figura 5: Regras de inferência do predicado “ser ordenado”

O lema que enuncia que o resultado de uma ordenação é ordenado é bem simples, assim como sua prova, feita por indução na lista dada como entrada.

$$\dfrac{\dfrac{\dfrac{}{\vdash \text{ordenado(nil)}}\,(S_0)}{\vdash \text{ordenado(ordena(nil))}}\,(\text{def}) \qquad \dfrac{\dfrac{}{\text{ordenado(ordena(tl))} \vdash \text{ordenado(insere(h, ordena(tl)))}}\,(*)}{\text{ordenado(ordena(tl))} \vdash \text{ordenado(ordena(h::tl))}}\,(\text{def})}{\vdash \text{ordenado(ordena(l))}}\,(\text{ind l})$$

Figura 6: Prova de que a ordenação por inserção ordena

O ramo esquerdo da prova (a base indutiva) é simples e basta aplicar a definição seguida da primeira regra de inferência de ordenação. O lado direito da prova (o passo indutivo) é mais complexo. Para este, definimos um lema auxiliar que afirma que a função *insere* preserva a propriedade de *ser ordenado*.

$$\forall\, l{:}lista,\ \ \forall\, x{:}\mathbb{N},\ \ ordenado(l) \longrightarrow ordenado(insere(h, l))$$

Figura 7: Lema da preservação da ordenação pela inserção

De início, tentamos provar esse lema por indução na lista *l*. Essa estratégia é possível, porém gera uma prova longa com ramos repetidos. Enquanto estavamos seguindo esta estratégia, notamos que a definição do predicado *ser ordenado*, assim como a definição de lista, é indutiva. Poderíamos então provar por indução na hipótese de que a lista dada como entrada é ordenada (e não por indução na lista em si). E foi exatamente isso que fizemos. A prova mostrou-se ser bem mais fácil.

2.2. Prova de que o resultado é permutação

Antes de enunciar o lema que garante que a ordenação de uma lista resulta na permutação desta lista, precisamos definir o predicado *ser permutação de*. Esse predicado é uma função simples definida sobre uma função auxiliar que computa o número de ocorrências de um elemento na lista.

O predicado diz basicamente que uma lista l_1 é a permutação de uma lista l_2 se, para cada elemento *n*, o número de ocorrências de *n* em l_1 é igual ao número de ocorrências de *n* em l_2.

$$permutação(l_1, l_2) := \forall\, n{:}\mathbb{N},\ \ ocorrências(n,\ l_1) = ocorrências(n,\ l_2).$$

Figura 8: Definição de permutação

A função auxiliar é definida recursivamente sobre a estrutura da lista.

$$ocorrências(n,\ l) := \begin{cases} 0, & se\ l = nil \\ ocorrências(n,\ tl) + 1, & se\ l = h :: tl\ e\ n = h \\ ocorrências(n,\ tl), & se\ l = h :: tl\ e\ n \neq h \end{cases}$$

Figura 9: Definição recursiva da função auxiliar de número de ocorrências

O lema que enuncia que o resultado da ordenação de uma lista é a permutação da lista original é feita sobre indução sobre a lista de entrada. O caso base é trivial. O caso indutivo requer simplificar a definição de permutação na definição de número de ocorrências. O caso indutivo é subdividido adicionalmente em dois casos:

- Quando o elemento a ser inserido em uma lista é diferente do elemento cujo número de ocorrências está sendo contado.
- Quando o elemento a ser inserido em uma lista é igual ao elemento cujo número de ocorrências está sendo contado.

Esses dois subcasos são provados em dois lemas auxiliares.

$$\forall\, l{:}lista,\ \forall\, n, a{:}\mathbb{N},\ n \neq a \longrightarrow ocorrências(n,\ insere(a,\ l)) = ocorrências(n,\ l).$$
$$\forall\, l{:}lista,\ \forall\, n{:}\mathbb{N},\ ocorrências(n,\ insere(n,\ l)) = ocorrências(n,\ l) + 1.$$

Figura 10: Lemas para número de ocorrências de um dado elemento

As provas de ambos os lemas é feita por indução na lista dada como entrada. No caso do lema em que o elemento contado é o mesmo a ser inserido, penso que fizemos uma prova convoluta, ramificada em vários subcasos. Talvez poderíamos fazer uma prova mais simples e elegante, porém não conseguimos identificar como.

3. Do Custo Temporal

Não há como *extrair* o custo temporal de um algoritmo a partir de sua definição (pelo menos acho que não há como). Então, para falar sobre o custo de tempo de um algorítmo, é preciso definir uma função que calcule tal custo dado uma entrada (ou o tamanho de uma entrada).

O cálculo de custo de um algorítmo é um tanto arbitrário, pois depende das condições de máquina e implementação, além de que diferentes trechos de código é executado em tempos diferentes. Neste projeto, definimos o custo como o número de comparações feitas.

Como há dois algoritmos sendo implementados neste projeto (o algoritmo de inserção e o de ordenação por inserção), definimos duas funções de custo para cada um deles. Uma função de custo calcula o custo absoluto dada uma entrada; a outra função de custo calcula o custo no pior caso dado o tamanho de uma entrada. Note que apenas a segunda definição será usada para fins de análise de crescimento assintótico.

Algoritmo	Função de custo por entrada	Função de custo por tamanho
Inserção	$Tinsere(x,\ l)$	$Tinsere_w(n)$
Ordenação	$Tordena(x,\ l)$	$Tordena_w(n)$

Tabela 1: Funções de cálculo de custo temporal.

A formalização da complexidade temporal no pior caso é feita em dois teoremas para cada algorítmo (o de inserção e o de ordenação); totalizando quatro teoremas.

- Um teorema diz que o custo temporal no pior caso é de fato o custo temporal no pior caso.
- O outro teorema estabelece como o custo no pior caso cresce assintoticamente.

3.1. Prova de que pior caso é de fato pior caso

Os teoremas de que o custo temporal no pior caso são de fato o pior caso são formalizados comparando a função de custo geral em uma dada entrada com a função de custo no pior caso no tamanho dessa entrada.

Para o algoritmo de inserção, o teorema é o seguinte. A prova é por indução na estrutura da lista e é bastante simples. O caso base é trivial. O passo indutivo requer prova por caso, uma simples manipulação aritmética e a aplicação do sucessor em ambos os lados da inequação para obter a hipótese indutiva. A tática `lia` se mostrará uma mão na roda a partir daqui.

$$\forall\, l{:}lista,\ \ \forall\, x{:}\mathbb{N},\ \ Tinsere(x,\ l)\ \leq\ Tinsere_w(tamanho(l)).$$

Figura 11: Teorema que garante o custo de inserção no pior tempo

Para o algoritmo de ordenação, o teorema é o seguinte. A prova também é por indução com um caso base trivial. O passo indutivo requer a aplicação de certos lemas aritméticos simples providos pela biblioteca *Arith* do *Coq* e alguns lemas que definimos manualmente (de que a ordenação não altera o tamanho da lista, e de que a inserção incrementa o tamanho da lista). Após as manipulações aritméticas necessárias, a prova é trivial.

$$\forall\, l{:}lista,\ \ Tordena(\,l)\ \leq\ Tordena_w(tamanho(l)).$$

Figura 12: Teorema que garante o custo de ordenação no pior tempo

3.2. Prova da análise assintótica do crescimento

Os teoremas de análise de crescimento assintótico do custo temporal no pior caso é formalizado estabelecendo uma cota superior para tal custo.

Antes de enunciar os teoremas, precisamos definir a função assintótica *Big O*. A formalização que usamos tem as seguintes diferenças em relação às definições mais comumente usadas nos livros-texto:

- Em vez de definir *O* como um conjunto de funções ($f \in O(g)$), o definimos como uma relação entre duas funções ($O(f,g)$). Isso é necessário pois é muito mais fácil trabalhar com proposições em Coq do que com conjuntos.
- Em vez de passar a função a ser analizada como primeiro argumento ($O(f,g)$), a passamos como segundo argumento ($O(g,f)$). Isso é necessário pois o argumento variante é o *f*, não o *g*. Assim, podemos definir crescimento linear e crescimento quadrático fixando o primeiro argumento (*g*) e *curryficando* a função *O*.

$$\begin{aligned} O(g,f) &:= \exists c, n_0{:}\mathbb{N}^+,\ \ \forall n{:}\mathbb{N}^+,\ \ n \geq n_0\ \longrightarrow\ f(n)\ \leq\ c\cdot g(n). \\ Olinear(f) &:= O((\lambda x\ \longmapsto\ x), f). \\ Oquadratico(f) &:= O((\lambda x\ \longmapsto\ x^2), f). \end{aligned}$$

Figura 13: Definições da função Big-O e derivados

O crescimento do algoritmo de inserção é linear. A prova é bastante simples e não requer indução. Eliminamos o existencial dando a evidência de que existem c e n_0 ambos iguais a 1. Em algum momento da prova (identificada abaixo por um asterisco), substituímos a função recursiva $Tinsere_w$ por uma função geradora não-recursiva equivalente. Essa função geradora é a própria função identidade, com a qual $Tinsere_w$ é comparada. A prova dessa equivalência é feita por indução em um lema auxiliar que é tão trivial que é feita numa única linha no Coq.

$$
\begin{array}{cl}
\hline
n : \mathbb{N}^+;\ n \geq 1 \vdash n \leq n & \text{(trivial)} \\
\hline
n : \mathbb{N}^+;\ n \geq 1 \vdash Tinsere_w(n) \leq n & (^*) \\
\hline
n : \mathbb{N}^+;\ n \geq 1 \vdash Tinsere_w(n) \leq (\lambda x \mapsto x)(n) & \text{(simpl)} \\
\hline
\vdash \exists c, n_0 : \mathbb{N}^+,\ \forall n : \mathbb{N}^+,\ n \geq n_0 \longrightarrow Tinsere_w(n) \leq c \cdot (\lambda x \mapsto x)(n) & \text{(existe c=1, } n_0\text{=1)} \\
\hline
\vdash Olinear(Tinsere_w) & \text{(def)}
\end{array}
$$

Figura 14: Prova do crescimento assintótico linear da inserção

O crescimento do algoritmo de ordenação é quadrático. A prova é similar à do crescimento da inserção, feita eliminando o existencial, etc. Em algum momento, substituímos a função recursiva *Tordena*$_w$ por uma função geradora não-recursiva equivalente. Essa função geradora é $(\lambda x \mapsto x^2/2\text{-}x/2)$. A prova dessa equivalência é feita em um lema auxiliar que, diferente do anterior, não é nada trivial. Provar essa equivalência foi a parte do projeto que deu mais trabalho, e requer mais detalhes (veja abaixo).

$$
\begin{array}{cl}
\hline
n : \mathbb{N}^+;\ n \geq 1 \vdash -n \leq n^2 & \text{(trivial)} \\
\hline
n : \mathbb{N}^+;\ n \geq 1 \vdash n^2 - n \leq 2n^2 & \text{(simpl)} \\
\hline
n : \mathbb{N}^+;\ n \geq 1 \vdash n^2/2 - n/2 \leq n^2 & \text{(simpl)} \\
\hline
n : \mathbb{N}^+;\ n \geq 1 \vdash Tordena_w(n) \leq n^2 & (^*) \\
\hline
n : \mathbb{N}^+;\ n \geq 1 \vdash Tordena_w(n) \leq (\lambda x \mapsto x^2)(n) & \text{(simpl)} \\
\hline
\vdash \exists c, n_0 : \mathbb{N}^+,\ \forall n : \mathbb{N}^+,\ n \geq n_0 \longrightarrow Tordena_w(n) \leq c \cdot (\lambda x \mapsto x^2)(n) & \text{(existe c=1, } n_0\text{=1)} \\
\hline
\vdash Oquadratico(Tordena_w) & \text{(def)}
\end{array}
$$

Figura 15: Prova do crescimento assintótico linear da inserção

Como dito anteriormente, provar que $(\lambda x \mapsto x^2/2\text{-}x/2)$ é a função geradora da recorrência *Tordena*$_w$ foi a parte que deu mais trabalho. A prova desse lema requer muita manipulação aritmética, de forma que uma simples aplicação da tática `lia` não é o bastante. Tivemos de procurar por lemas na biblioteca do Coq e aplicá-los; mas certos lemas não existiam. Depois de muito tempo de busca, definimos alguns lemas auxiliares (esses sim prováveis pela tática `lia`) dentro da própria prova (usando a tática `cut`) e os aplicamos quando necessários.

4. Conclusão

As provas foram relativamente simples; necessitando apenas entender quando (e como) quebrar a prova em casos, e quando (e em quê) aplicar indução.

A parte mais trabalhosa de todo o projeto foi a manipulação aritmética. Igualdades que são triviais quando provadas em papel precisaram ser manipuladas por lemas auxiliares que podem ou não estar na biblioteca do *Coq*. Ainda que a biblioteca *Lia* forneça uma tática capaz de provar certas relações aritméticas, essa tática não é mágica e nem sempre sabe como quebrar a prova em subprovas mais triviais.

Após apanhar para o LaTeX e perder, o relatório foi escrito em Troff.