A Correção do Algoritmo de Ordenação por Inserção

Vitor Camilo Lemoss - 202037720
Gustavo Barbosa De Almeida - 202037589

[1]Dep. Ciência da Computação – Universidade de Brasília (UnB)
1 de dezembro de 2025

Resumo. *Este relatório trata-se do detalhamento para o desenvolvimento do trabalho da disciplina de Lógica Computacional 1. O problema computacional escolhido foi o da proposta 1, que consiste em provar a correção do algoritmo de ordenação por inserção (Insertion Sort) utilizando o assistente de provas Rocq. O relatório aborda a formalização do algoritmo, a definição de uma lista ordenada e o conceito de permutação. A demonstração de correção foi separada em duas propriedades centrais: garantir que o algoritmo realmente gera uma lista ordenada e que a lista produzida é uma permutação da lista inicial.*

1. Introdução

Este relatório apresenta o desenvolvimento do projeto final da disciplina de Lógica Computacional 1. A proposta escolhida foi a de número 1, cuja tarefa consiste em formalizar, no assistente de provas Rocq, o algoritmo de ordenação por inserção (Insertion Sort) e demonstrar sua correção.

A escolha dessa proposta ocorreu porque ela pareceu, inicialmente, a mais direta, sobretudo considerando a dificuldade de traduzir propriedades lógicas para o Rocq. O trabalho concentrou-se em demonstrar duas propriedades fundamentais do algoritmo: (i) que a lista produzida pela ordenação está corretamente ordenada e (ii) que essa lista mantém exatamente os mesmos elementos da lista original, ou seja, é uma permutação dela.

O objetivo final do projeto é demonstrar o teorema $\mathrm{insertion\_sort\_correct}$, que afirma que, para qualquer lista l, vale que $\mathrm{ord}(\mathrm{insertion\_sort}(l))$ e $\mathrm{Permutation}(l,\ \mathrm{insertion\_sort}(l))$.

Este documento detalha as estratégias adotadas, os desafios encontrados e os conhecimentos adquiridos ao longo da realização do projeto.

2. Ferramentas Utilizadas

As principais ferramentas utilizadas foram:

- **Neovim** com o plugin **Coqtail**
- **Coq e coq-lsp** instalados via **OPAM**

3. Desenvolvimento e Metodologia

3.1. Definição das Funções Principais

Começamos o trabalho fazendo uma análise da função insert, que constitui a parte central do algoritmo. Ela é responsável por inserir um elemento em uma lista previamente

ordenada, garantindo que a ordenação seja preservada após a inserção. Sua definição é inteiramente recursiva. Compreender o funcionamento de insert foi essencial para formalizar corretamente o comportamento do insertion-sort, já que este utiliza repetidamente essa função para ordenar a lista como um todo. O insertion-sort percorre a lista de forma recursiva, aplicando insert para construir gradualmente o resultado ordenado.

Além disso, foi definido o predicado ord, que descreve formalmente o que significa uma lista estar ordenada. Essa definição foi indispensável para expressar de maneira precisa a propriedade que desejamos demonstrar sobre o algoritmo.

```
Fixpoint insert (x:nat) l :=
  match l with
  | [] => [x]
  | h :: tl =>
      if (x <=? h)
      then x :: l
      else h :: (insert x tl)
  end.
```

Insere um elemento em uma lista ordenada, mantendo a ordenação

```
Fixpoint insertion_sort (l : list nat) :=
  match l with
  | [] => []
  | h :: tl => insert h (insertion_sort tl)
  end.
```

Percorre a lista recursivamente, inserindo cada elemento em uma sublista ordenada.

O objetivo desta proposta é mostrar que o algoritmo insertion-sort l retorna uma permutação ordenada da lista l:

```
Theorem insertion_sort_correct : forall l,
  Sorted le (insertion_sort l) /\
  Permutation (insertion_sort l) l.
```

3.2. Lema `insert_perm`

Ao provar o lema `insert_perm`, foi possivel entender melhor como a função `insert` preserva os elementos da lista. No caso em que `x` é menor ou igual a `h`, a permutação é imediata, pois o elemento é apenas colocado no início. Já quando `x` é maior, a prova depende da chamada recursiva: uso a hipótese de indução para mostrar que `insert x tl` é permutação de `x :: tl`, e depois aplico `perm_skip` e `perm_swap` para ajustar a posição de `h` e `x`. Esse processo deixou mais claro como estruturar provas usando transitividade, especialmente quando a função reorganiza elementos sem alterar o conteúdo total da lista.

3.3. Lema `insert_sorted`

Ao provar o lema `insert_sorted`, mostramos como a função `insert` preserva a ordenação da lista. Quando `x` é menor ou igual ao primeiro elemento `h`, a prova é direta,

pois x simplesmente entra antes de h. Já no caso em que x é maior, a chamada recursiva garante, pela hipótese de indução, que insert x tl continua ordenada. O ponto essencial é verificar que h permanece menor ou igual ao novo primeiro elemento da lista resultante, o que depende da forma como x é inserido. Essa análise dos casos reforçou como a estrutura da ordem é mantida mesmo quando a função decide recursivamente onde inserir o novo elemento.

3.4. Lema `insertion_sort_perm`

Ao provar o lema insertion_sort_perm, ficou claro como o algoritmo insertion_sort mantém exatamente os mesmos elementos da lista original. No caso base, a prova é imediata. No caso indutivo, a chave é observar que insert produz uma permutação (insert_perm) e, em seguida, usar a hipótese de indução para garantir que insertion_sort tl também é permutação de tl. A transitividade de permutação conecta esses passos, mostrando que inserir o elemento h na lista já ordenada preserva o conjunto de elementos.“‘

3.5. Lema `insertion_sort_sorted`

Ao provar o lema insertion_sort_perm, ficou claro como o algoritmo insertion_sort mantém exatamente os mesmos elementos da lista original. No caso base, a prova é imediata. No caso indutivo, a chave é observar que insert produz uma permutação (insert_perm) e, em seguida, usar a hipótese de indução para garantir que insertion_sort tl também é permutação de tl. A transitividade de permutação conecta esses passos, mostrando que inserir o elemento h na lista já ordenada preserva o conjunto de elementos.“‘

Dificuldades e Aprendizados

Ao desenvolver este trabalho, foi encontrado um equilíbrio entre fatores que facilitaram o processo e outros que tornaram o andamento mais desafiador. Um dos pontos que mais ajudou foi contar com o apoio de um amigo que já havia sido monitor da disciplina. As explicações dele tornaram vários passos das provas mais claros, especialmente na interpretação das táticas do Coq e na organização das hipóteses durante as demonstrações.

Outra vantagem foi já ter cursado a disciplina de Linguagens de Programação, onde o contato com programação funcional ajudou bastante a compreender o funcionamento recursivo das definições e do estilo declarativo das provas, o que reduziu parte da curva de aprendizado.

Por outro lado, o final do semestre acabou trazendo bastante dificuldade. Com outras disciplinas exigindo atenção simultaneamente, organizar o tempo para estudar, revisar as provas e resolver os erros do Coq se tornou mais cansativo e demorado. Em vários momentos, o cansaço e a pressão das demais matérias acabaram tornando o progresso mais lento.

Mesmo assim, o processo trouxe aprendizado importante. Além de melhorar minha compreensão sobre ordenação e provas formais, também reforçou a importância de apoio, planejamento e paciência em períodos mais intensos do semestre.

Conclusão

O desenvolvimento deste projeto permitiu consolidar de forma prática diversos conceitos fundamentais da disciplina, especialmente no que diz respeito à formalização de algoritmos e à construção de provas no Coq. Demonstrar a correção do `insertion_sort` — tanto no aspecto de preservação dos elementos quanto na garantia de ordenação — tornou mais clara a relação entre definição funcional e raciocínio lógico, mostrando como propriedades simples se combinam para formar argumentos mais robustos.

Apesar dos desafios enfrentados ao longo do processo, o trabalho proporcionou um aprendizado significativo sobre organização de provas, uso de indução e entendimento profundo da estrutura do algoritmo. Além disso, reforçou a importância de apoio teórico, prático e colaborativo, mostrando que a formalização é um esforço que se beneficia tanto de técnica quanto de troca de conhecimento.

No conjunto, o projeto contribuiu para o desenvolvimento de uma visão mais madura sobre provas formais, fortalecendo habilidades que serão úteis em disciplinas e projetos futuros na área de lógica e verificação de programas.

Referências

[1] M. Ayala-Rincón and F. L. C. de Moura. *Applied Logic for Computer Scientists - Computational Deduction and Formal Proofs*. UTCS. Springer, 2017.

[2] Eric Kidd. Proving sorted lists correct using the coq proof assistant. `https://www.randomhacks.net/2015/07/19/proving-sorted-lists-correct-using-coq-proof-assistent/`, 2015. Accessed: 2025-12-06.

[3] Flavio Moura. *Projeto De Lógica Computacional 1 (2025/2)*. Flavio de Moura, 2025.

[4] Benjamin C. Pierce et al. Sort: Insertion sort. `https://softwarefoundations.cis.upenn.edu/vfa-current/Sort.html`, 2025. Accessed: 2025-12-06.

[5] Siraben. Proving insertion sort correct easily in coq. `https://gist.github.com/siraben/3fedfc2c5a242136a9bc6725064e9b7d`, 2022. Accessed: 2025-12-06.

[5] [2] [4] [3] [1]