

Universidade de Brasília - UnB
Departamento de Ciência da Computação

Formalização do Insertion Sort

Relatório de Projeto de Lógica Computacional 1

Autora:

Tarsila Marques de Oliveira Alves – 242012029

Brasília - DF
Novembro de 2025

Sumário

1	Introdução	2
2	Objetivos	2
2.1	Objetivo Geral	2
2.2	Objetivos Específicos	2
3	Implementação do Algoritmo e Especificação	2
3.1	Bibliotecas e Dependências	2
3.2	Definição do Algoritmo	3
3.2.1	Função Auxiliar (<code>insert</code>)	3
3.2.2	Função Principal (<code>insertion_sort</code>)	3
3.3	Especificação das Propriedades (Lemas e Teorema)	3
3.3.1	Definição dos Lemas Auxiliares	3
3.3.2	Definição do Teorema Principal	4
4	Estratégia de Prova	4
4.1	Decomposição do Problema	4
4.2	Plano de Execução	5
5	Formalização das Provas	5
5.1	Lema 1: Preservação da Ordenação	5
5.2	Lema 2: Propriedade de Permutação	6
5.3	Teorema Principal: Correção Total	6
6	Considerações sobre o Desenvolvimento	7
6.1	Adaptação Teórica e Ferramental	7
6.2	Estratégia de Prova e Obstáculos Técnicos	7
6.3	Desafios na Documentação	8
6.4	Conclusão do Desenvolvimento	8
7	Conclusão	8

1 Introdução

Este relatório documenta a formalização e verificação do algoritmo de ordenação *Insertion Sort* (Ordenação por Inserção), utilizando o assistente de provas Rocq (Coq) [1], que permite escrever algoritmos funcionais e provar teoremas sobre eles no mesmo ambiente.

2 Objetivos

2.1 Objetivo Geral

O objetivo principal deste projeto é aplicar técnicas de Lógica Computacional e Verificação Formal para garantir a corretude do algoritmo *Insertion Sort*. Diferentemente de abordagens empíricas baseadas em testes unitários, busca-se aqui uma prova matemática irrefutável de que o algoritmo funciona para qualquer lista de entrada de números naturais, utilizando o assistente de provas Rocq (Coq).

2.2 Objetivos Específicos

1. **Implementação Funcional:** Codificar o algoritmo *Insertion Sort* e sua função auxiliar de inserção (`insert`) respeitando o paradigma funcional e a recursão estrutural exigidos pelo Coq.
2. **Especificação Formal:** Definir matematicamente o que constitui a “correção” de um algoritmo de ordenação. No contexto deste trabalho, a correção é definida pela conjunção de duas propriedades:
 - A lista de saída deve estar ordenada (**Sorted**).
 - A lista de saída deve ser uma permutação da lista de entrada (**Permutation**).

3 Implementação do Algoritmo e Especificação

A implementação foi realizada utilizando a nativa do assistente de provas Rocq (Coq). O código fonte (`insertion_sort.v`) está estruturado em três partes lógicas: importação de bibliotecas, definição do algoritmo e especificação das propriedades (Lemas e Teoremas).

Abaixo, os detalhes cada componente do código desenvolvido.

3.1 Bibliotecas e Dependências

O projeto inicia importando bibliotecas fundamentais do Coq que fornecem as estruturas de dados e definições necessárias para números naturais, listas, permutações e ordenação.

```
1 Require Import Arith List.  
2 Require Import Permutation.  
3 Require Import Sorted.  
4 Import ListNotations.
```

3.2 Definição do Algoritmo

O algoritmo é composto por duas funções recursivas. A primeira é a função auxiliar de inserção.

3.2.1 Função Auxiliar (insert)

Esta função é responsável por inserir um número natural x em uma lista l , assumindo que l já está ordenada.

Listing 1: Implementação da função insert

```
1 Fixpoint insert (x:nat) (l:list nat) :=  
2   match l with  
3   | [] => [x]  
4   | h::tl => if (x <=? h)  
5               then x :: l  
6               else h :: (insert x tl)  
7   end.
```

Explicação: O código utiliza *pattern matching*. Se a lista for vazia, cria-se uma lista unitária. Se houver elementos, compara-se x com a cabeça h . Se x for menor ou igual, ele assume a primeira posição; caso contrário, a função chama a si mesma recursivamente para encontrar a posição correta na cauda tl .

3.2.2 Função Principal (insertion_sort)

A função principal orquestra a ordenação percorrendo a lista de entrada.

Listing 2: Implementação da função insertion_sort

```
1 Fixpoint insertion_sort (l: list nat) :=  
2   match l with  
3   | [] => []  
4   | h::tl => insert h (insertion_sort tl)  
5   end.
```

Explicação: A recursão percorre a lista até o fim. No retorno da recursão (quando a pilha desempilha), ela insere o elemento atual (h) na cauda que já foi ordenada pela chamada recursiva (`insertion_sort tl`).

3.3 Especificação das Propriedades (Lemas e Teorema)

Além do algoritmo, o código fonte contém a especificação formal das propriedades que o algoritmo deve satisfazer. Estas definições preparam o terreno para as provas.

3.3.1 Definição dos Lemas Auxiliares

Para aplicar a estratégia de divisão e conquista, definimos dois lemas que descrevem o comportamento esperado da função `insert`.

Listing 3: Definição dos Lemas Auxiliares

```
1 Lemma insert_sorted: forall x l,  
2   Sorted le l -> Sorted le (insert x l).
```

```

3
4 Lemma insert_perm: forall x l,
5   Permutation (insert x l) (x :: l).

```

Explicação:

- **insert_sorted**: Especifica que a inserção preserva a ordem. Lê-se: "Para todo x e l , se l é ordenada, então $\text{insert } x \ l$ também é ordenada".
- **insert_perm**: Especifica que a inserção não altera os dados. Lê-se: " $\text{insert } x \ l$ é uma permutação da lista formada por x adicionado a l ".

3.3.2 Definição do Teorema Principal

Por fim, o código define o objetivo final do projeto, que une as propriedades de ordenação e permutação na função principal.

Listing 4: Teorema de Correção

```

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

```

Explicação: Este teorema afirma a corretude total do algoritmo: para qualquer lista l , o resultado da ordenação satisfaz simultaneamente o predicado **Sorted** (está ordenado) e **Permutation** (contém os mesmos elementos da entrada).

4 Estratégia de Prova

Para este projeto, adotou-se a estratégia de **Divisão e Conquista**, aliada ao princípio da **Indução Estrutural**.

Abaixo os detalhes dos pilares desta estratégia.

4.1 Decomposição do Problema

A especificação de correção total do *Insertion Sort* exige a satisfação simultânea de duas propriedades:

1. **Ordenação (Sorted)**: Garantir que a relação de ordem \leq se aplica a todos os elementos adjacentes da lista resultante.
2. **Integridade (Permutation)**: Garantir que nenhum elemento foi adicionado, removido ou duplicado indevidamente durante o processo.

Tentar provar essas propriedades diretamente sobre a função principal (**insertion_sort**) seria inviável, pois ela depende inteiramente do comportamento da função auxiliar (**insert**). Se não provarmos primeiro que a inserção funciona, não podemos provar que a ordenação funciona.

4.2 Plano de Execução

Com base na decomposição acima, o plano de prova foi estruturado em duas etapas sequenciais:

1. **Verificação da Função Auxiliar:** O primeiro passo consiste em isolar a função `insert` e provar que ela respeita as invariantes necessárias. Para isso é necessário dois lemas específicos:
 - Um lema para garantir que inserir um elemento em uma lista ordenada preserva a ordem.
 - Um lema para garantir que a inserção é, por definição, uma operação de permutação.
2. **Indução no Teorema Principal:** Uma vez garantida a correção da função auxiliar, utilizamos **Indução Estrutural** sobre a lista de entrada da função principal.
 - No **Caso Base**, provamos a correção trivial para a lista vazia.
 - No **Passo Indutivo**, assumimos que a cauda da lista já foi ordenada corretamente (Hipótese de Indução) e utilizamos os lemas auxiliares provados na etapa anterior para justificar que a inserção da cabeça na cauda ordenada produz o resultado correto.

5 Formalização das Provas

Nesta seção, apresenta-se a demonstração formal dos lemas e do teorema principal. A argumentação segue o raciocínio indutivo utilizado na verificação mecânica, traduzido aqui para a notação matemática padrão.

5.1 Lema 1: Preservação da Ordenação

Objetivo: Demonstrar que a inserção de um elemento x em uma lista ordenada l preserva a propriedade de ordenação.

Enunciado Formal:

$$\forall x \in \mathbb{N}, \forall l \in \text{List}(\mathbb{N}), \text{Sorted}(l) \implies \text{Sorted}(\text{insert}(x, l))$$

Demonstração. A prova é realizada por **indução estrutural** sobre a lista l .

- **Caso Base** ($l = \text{nil}$): A função retorna a lista unitária $[x]$. Como qualquer lista contendo apenas um elemento é trivialmente ordenada, a propriedade é satisfeita ($\text{Sorted}([x])$ é verdadeiro).
- **Passo Indutivo** ($l = h :: t$): Assumimos como Hipótese de Indução (H.I.) que a propriedade vale para a cauda t :

$$\text{Sorted}(t) \implies \text{Sorted}(\text{insert}(x, t))$$

Queremos provar que $\text{Sorted}(\text{insert}(x, h :: t))$ é verdadeiro, dado que $\text{Sorted}(h :: t)$. O algoritmo compara x com a cabeça h :

1. **Se $x \leq h$:** O algoritmo retorna $x :: h :: t$. Sabemos que $x \leq h$ e que $h :: t$ já era ordenada (logo $h \leq \text{head}(t)$). Portanto, a sequência x, h, \dots mantém a ordem não decrescente.
2. **Se $x > h$:** O algoritmo retorna $h :: \text{insert}(x, t)$. Pela H.I., sabemos que a cauda gerada $\text{insert}(x, t)$ é ordenada. Resta garantir que a cabeça h é menor ou igual ao primeiro elemento dessa nova cauda. Como h era menor que todos os elementos da cauda original (t) e também temos $h < x$ (pela condição do caso), conclui-se que h é um limite inferior válido para toda a nova lista.

□

5.2 Lema 2: Propriedade de Permutação

Objetivo: Demonstrar que a lista resultante da inserção contém exatamente os mesmos elementos da lista original acrescida do novo valor.

Enunciado Formal:

$$\forall x \in \mathbb{N}, \forall l \in \text{List}(\mathbb{N}), \text{insert}(x, l) \equiv_{\text{perm}} (x :: l)$$

Demonstração. Procede-se por **indução estrutural** em l .

- **Caso Base** ($l = \text{nil}$): $\text{insert}(x, []) = [x]$, que é trivialmente uma permutação de $x :: []$.
- **Passo Indutivo** ($l = a :: l'$): Queremos provar que $\text{insert}(x, a :: l') \equiv_{\text{perm}} x :: a :: l'$.
 1. **Se $x \leq a$:** O algoritmo retorna $x :: a :: l'$. Pela reflexividade, a lista é uma permutação dela mesma.
 2. **Se $x > a$:** O algoritmo retorna $a :: \text{insert}(x, l')$. Pela H.I., sabemos que $\text{insert}(x, l') \equiv_{\text{perm}} x :: l'$. Logo, temos que provar que $a :: (x :: l') \equiv_{\text{perm}} x :: a :: l'$. Isso é garantido pela propriedade de *swap* (troca de elementos adjacentes) da relação de permutação.

□

5.3 Teorema Principal: Correção Total

Objetivo: Unificar os lemas anteriores para provar a correção do algoritmo completo.

Enunciado Formal:

$$\forall l \in \text{List}(\mathbb{N}), \text{Sorted}(\text{insertion_sort}(l)) \wedge \text{insertion_sort}(l) \equiv_{\text{perm}} l$$

Demonstração. A prova é realizada por **indução na lista** l .

- **Caso Base** ($l = \text{nil}$): O algoritmo retorna nil . A lista vazia é ordenada e é uma permutação de si mesma.

- **Passo Indutivo** ($l = h :: t$): Assumimos a Hipótese de Indução (H.I.) de que a chamada recursiva para a cauda está correta:

$$\text{Sorted}(\text{insertion_sort}(t)) \wedge \text{insertion_sort}(t) \equiv_{\text{perm}} t$$

O passo do algoritmo é computar $\text{insert}(h, \text{insertion_sort}(t))$. Precisamos provar as duas propriedades para este resultado:

1. **Ordenação:** Pela H.I., sabemos que a lista $L_{\text{cauda}} = \text{insertion_sort}(t)$ é ordenada. Aplicando o **Lema 1** com o elemento h e a lista L_{cauda} , concluímos que $\text{insert}(h, L_{\text{cauda}})$ resulta em uma lista ordenada.
2. **Permutação:** Pelo **Lema 2**, sabemos que $\text{insert}(h, L_{\text{cauda}}) \equiv_{\text{perm}} h :: L_{\text{cauda}}$. Pela H.I., sabemos que $L_{\text{cauda}} \equiv_{\text{perm}} t$. Por transitividade, se substituirmos L_{cauda} por t , temos que o resultado final é uma permutação de $h :: t$.

□

6 Considerações sobre o Desenvolvimento

O desenvolvimento deste projeto representou um desafio multifacetado, exigindo não apenas a aplicação de conceitos lógicos, mas também uma adaptação profunda a novos paradigmas de programação e ferramentas de documentação. Abaixo, detalha-se a trajetória de aprendizado e as decisões técnicas adotadas.

6.1 Adaptação Teórica e Ferramental

O primeiro obstáculo encontrado foi a transposição da lógica de dedução natural, praticada em sala de aula em pequenos teoremas, para a verificação de um algoritmo completo de ordenação. Foi necessário alterar a perspectiva usual de programação: deixar de encarar o algoritmo apenas como uma sequência de instruções imperativas e passar a visualizá-lo como um teorema matemático a ser provado.

O contato com o assistente de provas Rocq (Coq) impôs uma curva de aprendizado acentuada. Acostumado a linguagens de programação convencionais, precisei me adaptar à sintaxe da linguagem do RCOQ. Para mitigar essa dificuldade, realizei inicialmente a prova de teoremas menores e mais simples, permitindo a familiarização com a lógica das táticas e o funcionamento do ambiente de prova antes de atacar o problema principal.

6.2 Estratégia de Prova e Obstáculos Técnicos

A análise do código disponibilizado no repositório revelou que o algoritmo precisava ser complementado por uma estrutura de prova robusta. A minha percepção inicial de que o código "não estava completo", ele não tinha as garantias formais intermediárias.

Para solucionar isso, adotei a estratégia de **Divisão e Conquista**. Ao invés de tentar provar a correção do `insertion_sort` de forma única, decompos o problema. Essa decisão não foi arbitrária, mas sim uma resposta direta a três dificuldades técnicas que identifiquei:

- **O Problema da Indução:** Percebi que se eu tentasse provar o teorema principal diretamente, chegaria a um estado impossível. No passo indutivo, o Coq não

consegue deduzir automaticamente que a função `insert` preserva a ordem apenas expandindo a definição. Foi necessário, portanto, isolar essa propriedade no lema `insert_sorted`.

- **Gerenciamento de Hipóteses Condicionais:** Um grande desafio foi lidar com os condicionais (`if`) dentro das provas. Aprendi que, ao usar a tática `destruct` nas comparações, é crucial usar o modificador `eqn:Heq`. Sem isso, eu perdia a informação lógica de que $x \leq h$ (ou $x > h$), tornando impossível provar a ordenação nos casos específicos.

Dessa forma, a estratégia final consistiu em focar primeiro nas propriedades da função `insert`, isolando a complexidade da inserção da complexidade da recursão principal.

6.3 Desafios na Documentação

A etapa de documentação apresentou desafios técnicos paralelos à prova formal. O uso de \LaTeX para a confecção do relatório era inédito.

Inicialmente, tentei configurar um ambiente de compilação local utilizando `Makefiles` para gerenciar a geração do documento. No entanto, devido a incompatibilidades de ambiente e complexidade de configuração, optei por uma solução mais eficiente: a migração para a plataforma online Overleaf. Essa decisão permitiu focar no aprendizado da sintaxe \LaTeX e na qualidade da escrita do relatório, abstraindo os problemas de configuração de software.

6.4 Conclusão do Desenvolvimento

Apesar das dificuldades iniciais com as ferramentas e a mudança de paradigma, o projeto foi concluído com êxito. A combinação do estudo prévio de casos simples com a estratégia de divisão e conquista provou-se eficaz para superar a complexidade da verificação formal.

Esta experiência evidenciou que a formalização, oferece um nível de garantia de qualidade de software inalcançável por testes convencionais.

7 Conclusão

Este trabalho apresentou a formalização completa do algoritmo *Insertion Sort*. Através da definição de lemas auxiliares e do uso de indução estrutural, foi possível substituir as admissões (`Admitted`) do código original por provas rigorosas (`Qed`).

Demonstrou-se matematicamente que a implementação funcional satisfaz a especificação de correção total, garantindo integridade dos dados e ordenação para qualquer entrada. O uso do assistente Rocq permitiu verificar mecanicamente a validade de cada passo lógico, eliminando a possibilidade de erros humanos na dedução.

O projeto cumpriu todos os objetivos propostos, entregando um artefato de software verificado e documentado.

Referências

- [1] The Coq Development Team. *The Coq Proof Assistant Reference Manual*. Disponível em: <https://coq.inria.fr>. Acesso em: 26 nov. 2025.
- [2] Coq Standard Library. *Library Coq.Lists.List*. Disponível em: <https://coq.inria.fr/library/Coq.Lists.List.html>. Acesso em: 26 nov. 2025.
- [3] Moura, Flávio L. C. de. *Projeto de Lógica Computacional 1: Especificação do Projeto*. Universidade de Brasília, 2025.