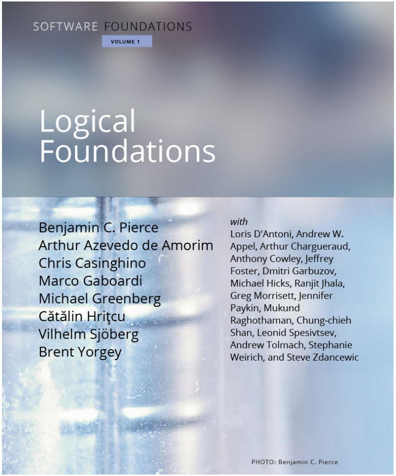
Introdução ao Assistente de Provas Lean

André Luiz Feijó dos Santos

Semana da Informática
Universidade Federal de Viçosa

Agosto de 2025

Vários dos exemplos desta apresentação foram baseados, principlamente, nos contidos em “Logical Foundations”, de Benjamin C. Pierce.



https://github.com/andrefeijosantos

Conteúdo

1. Provas em Lean
2. Simplificação
3. Reescrita
4. Análise de Casos
5. Indução
6. Observações e Conclusão
7. Referências

Lean como Assistente de Provas

- Como conduzir provas em Lean?
  1. Declarar (ou importar) as definições sobre as quais se deseja escrever um teorema.
  2. Introduzir o teorema que se deseja provar. Lean irá definir a validade deste teorema como seu **objetivo inicial**.
  3. Aplicar **táticas de prova** para alterar (de maneira segura) o objetivo de prova até se chegue em uma meta da qual Lean consiga verificar sua validade sozinho (*e.g.*, A = A... ou coisas mais complexas).

Táticas de Prova

Seja n e m números naturais. Imagine que desejamos provar:

Objetivo: $(n = 1) \to (m + n > 0)$

- Geralmente, uma prova como essa começa assumindo a Hipótese $(n = 1)$ como verdadeira.

Táticas de Prova

Seja n e m números naturais. Imagine que desejamos provar:

Hipótese: $n = 1$

Objetivo: $n + m > 0$

- Geralmente, uma prova como essa começa assumindo a Hipótese ($n = 1$) como verdadeira.
- Note que o objetivo de prova foi alterado para um mais simples de ser provado.

Provas por Simplificação

- A própria definição dos tipos sobre os quais estamos conduzindo provas pode ser suficiente para concluir o objetivo.

```
def andb (b1 b2 :  MyBool) :=
     match b1 with
     | False => False
     | True => b2

notation p " ∧ " q := andb p q
```

- Suponha que queremos provar

$$\forall b \in \texttt{MyBool},\ \texttt{False} \land \texttt{b = False}.$$

- Neste caso, a própria definição de `and` prova esse teorema.

Provas por Simplificação

- O teorema anterior pode ser escrito em Lean da seguinte forma:

```
theorem false_and_b_eq_false :
    ∀(b : MyBool), False ∧ b = False
```

Provas por Simplificação

- ```
  theorem false_and_b_eq_false :
      ∀(b : MyBool), False ∧ b = False :=
  by
  ```

  Objetivo (goal): Provar $\forall$(b : MyBool), False $\wedge$ b $=$ False

Tática `intros`

- A primeira etapa para provar esse teorema é usar a tática `intros`.
- Neste contexto, a tática `intros` introduz a variável correspondente no contexto como um objeto arbitrário.
- Informalmente, é equivalente a dizer “Assuma que b é um booleano qualquer”.
  ```

Provas por Simplificação

- ```
  theorem false_and_b_eq_false :
      ∀(b : MyBool), False ∧ b = False :=
  by intros b
  ```

  Objetivo (goal): Provar False $\wedge$ b $=$ False
  ```

Provas por Simplificação

-
```
theorem false_and_b_eq_false :
    ∀(b : MyBool), False ∧ b = False :=
by intros b
```

Objetivo (goal): Provar False ∧ b = False

Tática `simp`

- `simp` vem de "*simplify*" (simplificar).
- Com essa tática, são aplicadas reescritas baseadas em definições registradas na base do Lean. O objetivo é substituir o objetivo atual por algo "mais simples".
- A tática ainda faz a checagem da igualdade após a simplificação, o que pode levar à conclusão da prova.

Provas por Simplificação

```
theorem false_and_b_eq_false :
    ∀(b : MyBool), False ∧ b = False :=
by intros b
   simp [andb]
```

Objetivo (goal): Nenhum objetivo restante. Prova concluída!

Provas por Simplificação

- Note que, junto ao `simp`, deixamos claro qual definição queremos usar para simplificar nosso objetivo.

```
theorem false_and_b_eq_false :
    ∀(b : MyBool), False ∧ b = False :=
by intros b
   simp [andb]
```

Provas por Simplificação

- Isso pode ser evitado se reescrevermos a definição de `andb` com a *flag*:

```
@[simp]
def andb (b1 b2 : MyBool) :=
     match b1 with
     | False => False
     | True => b2
```

- Isso nos permite usar a tática como:

```
theorem false_and_b_eq_false :
    ∀(b : MyBool), False ∧ b = False :=
by intros b
   simp
```

Provas por Simplificação

- Isso pode ser evitado se reescrevermos a definição de `andb` com a *flag*:

```
@[simp]
def andb (b1 b2 : MyBool) :=
     match b1 with
     | False => False
     | True => b2
```

- Isso nos permite usar a tática como:

```
theorem false_and_b_eq_false :
    ∀(b : MyBool), False ∧ b = False :=
by intros b
   simp
```

...Por que é assim?

A Tática rfl

- Outra tática interessante é a `rfl`, cujo nome vem de "reflexivity" (reflexividade).
- A tática `rfl`, assim como a `simp`, tem a capacidade de finalizar uma prova, desde que ambos os lados de uma igualdade sejam – por definição – iguais. Por exemplo:

```
theorem false_and_b_eq_false :
    ∀(b : MyBool), False ∧ b = False :=
by intros b
   rfl
```

A Tática `rfl`

- Outra tática interessante é a `rfl`, cujo nome vem de "reflexivity" (reflexividade).
- A tática `rfl`, assim como a `simp`, tem a capacidade de finalizar uma prova, desde que ambos os lados de uma igualdade sejam – por definição – iguais.
- Ainda que essas táticas **pareçam** similares, existe uma diferença fundamental:
  - `rfl` apenas computa a definição de algo (*e.g.*, $\wedge$) e checa a igualdade;
  - `simp` aplica todas as regras marcadas com `@[simp]` e, caso encontre uma igualdade por definção, aplica internamente reflexividade.

Provas por Reescrita (*Rewriting*)

- Vamos voltar ao teorema:

$$\forall n, m \in \texttt{MyNat}, \quad (n = m) \to (n + n = m + m).$$

- Em vez de fazermos uma declaração universal, sobre quaisquer n e m do tipo `MyNat`, estamos tratando de um caso mais específico, em que $n = m$.
- Sendo assim, não podemos apenas simplificar o teorema e identificar reflexividade, nós precisamos **reescrevê-lo** considerando nossa hipótese.

Provas por Reescrita (*Rewriting*)

```
theorem plus_id :
    ∀(n m :  MyNat), (n = m) → ((n + n) = (m + m)) :=
by intros n m
```

Objetivo (goal): Provar $(n = m) \to ((n + n) = (m + m))$

Provas por Reescrita (*Rewriting*)

```
theorem plus_id :
    ∀(n m :  MyNat), (n = m) → ((n + n) = (m + m)) :=
by intros n m
   intros H
```

Objetivo (goal): Provar $(n + n) = (m + m)$
Hipótese H: $n = m$

Tática `intros`

- A tática `intros` também pode ser utilizada para introduzir hipóteses.

Provas por Reescrita (*Rewriting*)

```
theorem plus_id :
    ∀(n m :  MyNat), (n = m) → ((n + n) = (m + m)) :=
by intros n m
   intros H
   rw [H]
```

Objetivo (goal): Nenhum objetivo restante. Prova concluída!
Hipótese H: $n = m$

Tática `rw`

- `rw` vem de "*rewrite*" (reescrever).
- Reescreve um teorema aplicando uma premissa. Lean é capaz de finalizar a prova com a tática `rw`.

Provas por Análise de Casos

- Imagine que temos que provar um teorema sobre alguma enumeração (*e.g.*, `MyBool`).
- Podemos testar se o teorema vale para cada constructo separadamente.
- Se o teorema é válido para todos os constructos, então está provado para qualquer variável daquele tipo.

Provas por Análise de Casos

- Vamos provar o seguinte teorema:

```
theorem andb_comm :
    ∀(b1 b2 :  MyBool), b1 ∧ b2 = b2 ∧ b1
```

Goal: $\forall(b1\ b2 : \text{MyBool}), b1 \wedge b2 = b2 \wedge b1$

Provas por Análise de Casos

- ```
  theorem andb_comm :
      ∀(b1 b2 : MyBool), b1 ∧ b2 = b2 ∧ b1 :=
  by intros b1 b2
  ```

  Goal: b1 $\wedge$ b2 $=$ b2 $\wedge$ b1

Tática cases

- Usamos `cases x` para abrir uma análise de casos em `x`.
- Cria um *sub goal* (sub objetivo) para cada possível constructo para a variável `x`.
- Para selecionar o caso que desejamos provar, basta utilizar a tática `case` (constructo).
  ```

Provas por Análise de Casos

```
theorem andb_comm :
    ∀(b1 b2 : MyBool), b1 ∧ b2 = b2 ∧ b1 :=
by intros b1 b2
   cases b1
```

Sub Goal 1: True $\wedge$ b2 $=$ b2 $\wedge$ True
Sub Goal 2: False $\wedge$ b2 $=$ b2 $\wedge$ False

Provas por Análise de Casos

```
theorem andb_comm :
    ∀(b1 b2 : MyBool), b1 ∧ b2 = b2 ∧ b1 :=
by intros b1 b2
    cases b1
    case True =>
```

Sub Goal 1: True ∧ b2 = b2 ∧ True

Provas por Análise de Casos

```
theorem andb_comm :
    ∀(b1 b2 : MyBool), b1 ∧ b2 = b2 ∧ b1 :=
by intros b1 b2
    cases b1
    case True =>
     cases b2 =>
```

Sub Goal 1.1: True ∧ True = True ∧ True
Sub Goal 1.2: True ∧ False = False ∧ True

Provas por Análise de Casos

- ```
  theorem andb_comm :
      ∀(b1 b2 : MyBool), b1 ∧ b2 = b2 ∧ b1 :=
  by intros b1 b2
      cases b1
      case True =>
       cases b2 =>
       case True =>
  ```

  Sub Goal 1.1: True ∧ True = True ∧ True
  ```

Provas por Análise de Casos

- ```
  theorem andb_comm :
      ∀(b1 b2 : MyBool), b1 ∧ b2 = b2 ∧ b1 :=
  by intros b1 b2
      cases b1
      case True =>
       cases b2 =>
       case True => rfl
  ```

  Sub Goal 1.1: Nenhum sub objetivo restante. Sub Prova concluída!
  ```

Provas por Análise de Casos

- ```
  theorem andb_comm :
      ∀(b1 b2 : MyBool), b1 ∧ b2 = b2 ∧ b1 :=
  by intros b1 b2
      cases b1
      case True =>
       cases b2 =>
       case True => rfl
       case False => rfl
      case False =>
       cases b2 =>
       case True => rfl
       case False => rfl
  ```

Goal: Nenhum objetivo restante. Prova concluída!
  ```

De Volta ao Primeiro Teorema

- Nossa primeira prova foi que

$$\forall b \in \texttt{MyBool}, \quad \texttt{False} \land \texttt{b = False}$$

- Como poderíamos provar o seguinte teorema?

$$\forall b \in \texttt{MyBool}, \quad \texttt{b} \land \texttt{False = False}$$

De Volta ao Primeiro Teorema

Vamos começar tentando uma prova por simplificação.

```
theorem b_and_false_eq_false :
  ∀(b :  MyBool), b ∧ False = False :=
```

De Volta ao Primeiro Teorema

Vamos começar tentando uma prova por simplificação.

```
theorem b_and_false_eq_false :
  ∀(b : MyBool), b ∧ False = False :=
by intros b
```

De Volta ao Primeiro Teorema

Vamos começar tentando uma prova por simplificação.

```
theorem b_and_false_eq_false :
  ∀(b :  MyBool), b ∧ False = False :=
by intros b
   simp [andb] ×
```

De Volta ao Primeiro Teorema

E uma prova por casos?

```
theorem b_and_false_eq_false :
  ∀(b : MyBool), b ∧ False = False :=
by intros b
   cases b
   case True => rfl
   case False => rfl
```

De Volta ao Primeiro Teorema

Mas mais simples que isso: reaproveitar resultados já obtidos!

```
theorem b_and_false_eq_false :
  ∀(b : MyBool), b ∧ False = False :=
by intros b
   rw [andb_comm]          -- b ∧ False = False ∧ b
   rfl
```

Provando $\forall n, n + 0 = n$ por Casos

- Vamos tentar agora provar o seguinte teorema por casos:

```
theorem plus_n_O :
  ∀(n : MyNat),  n + O = n :=
by intros n
   cases n
```

Provando $\forall n, n + 0 = n$ por Casos

- Vamos tentar agora provar o seguinte teorema por casos:

```
theorem plus_n_0 :
  ∀(n : MyNat),  n + 0 = n :=
by intros n
   cases n
   case 0 => rfl
```

Sub Goal 1: Provar `0 + 0 = 0`

Provando $\forall n, n + 0 = n$ por Casos

- Vamos tentar agora provar o seguinte teorema por casos:

```
theorem plus_n_0 :
   ∀(n : MyNat),   n + 0 = n :=
by intros n
   cases n
   case 0 => rfl
   case S n' =>
```

Sub Goal 2: Provar `S n' + 0 = S n'`

Provas por Indução

- Para provar teoremas interessantes sobre naturais ou outros tipos definidos de maneira indutiva (como listas e árvores), geralmente vamos precisar de uma abordagem mais poderosa: a **indução**.
- Para mostrar que P vale para todos os naturais:
  - Mostramos que um caso base P(O) vale;
  - Mostramos que, para qualquer n, se P(n) vale, então P($S\ n$) também vale;
  - Finalmente, concluímos que P(n) vale para todo n.

Provas por Indução

- Finalmente, podemos provar:

```
theorem plus_n_0 :
   ∀(n : MyNat),  n + 0 = n :=
```

Provas por Indução

- Finalmente, podemos provar:

```
theorem plus_n_0 :
   ∀(n : MyNat),  n + 0 = n :=
```

Goal: Provar $\forall (n : \texttt{MyNat})$, `n + 0 = n`

Provas por Indução

- Finalmente, podemos provar:

```
theorem plus_n_0 :
  ∀(n : MyNat),  n + 0 = n :=
by intros n
```

Goal: Provar `n + 0 = n`

Tática `induction`

- Abre casos correspondentes aos constructos de um tipo indutivo.
- Permite assumirmos uma **hipótese indutiva** nos casos de constructos recursivos.

Provas por Indução

- Finalmente, podemos provar:

```
theorem plus_n_O :
   ∀(n : MyNat),  n + O = n :=
by intros n
  induction n with
```

Goal: Provar `n + O = n`

Provas por Indução

- Finalmente, podemos provar:

```
theorem plus_n_0 :
   ∀(n : MyNat),   n + 0 = n :=
by intros n
  induction n with
  | 0 =>
```

Sub Goal 1: Provar `0 + 0 = 0`

Provas por Indução

- Finalmente, podemos provar:

```
theorem plus_n_0 :
  ∀(n : MyNat),  n + 0 = n :=
by intros n
  induction n with
  | 0 => rfl
```

Sub Goal 1: Prova de Sub Objetivo Concluída!

Provas por Indução

- Finalmente, podemos provar:

```
theorem plus_n_0 :
  ∀(n : MyNat),  n + 0 = n :=
by intros n
  induction n with
  | 0 => rfl
  | S n' IHn' =>
```

Sub Goal 2: Provar `S n' + 0 = S n'`
Hipótese (IHn'): `n' + 0 = n'`

Provas por Indução

- Finalmente, podemos provar:

```
theorem plus_n_0 :
   ∀(n : MyNat),  n + 0 = n :=
by intros n
  induction n with
  | 0 => rfl
  | S n' IHn' => simp [plus]
```

Sub Goal 2: Provar `n' + 0 = n'`
Hipótese (IHn'): `n' + 0 = n'`

Provas por Indução

- Finalmente, podemos provar:

```
theorem plus_n_O :
   ∀(n : MyNat),  n + O = n :=
by intros n
  induction n with
  | O => rfl
  | S n' IHn' => simp [plus]
                 rw [IHn']
```

Sub Goal 2: Prova de Sub Objetivo Concluída!
Hipótese (IHn'): `n' + O = n'`

Observações Finais

- Durante o minicurso, criamos tipos indutivos como `MyBool` e `MyNat`. Porém, Lean possui suas próprias definições de `Bool` e `Nat`.
- Também há um número considerável de teoremas já provados em lean. O que faz provas com esses tipos muito mais fáceis:

```
theorem prova_lean_ex1 :
   ∀(n : Nat), 0 + n = 0 :=
by intros n
   simp
```

Observações Finais

- Durante o minicurso, criamos tipos indutivos como `MyBool` e `MyNat`. Porém, Lean possui suas próprias definições de `Bool` e `Nat`.
- Também há um número considerável de teoremas já provados em lean. O que faz provas com esses tipos muito mais fáceis:

```
theorem prova_lean_ex2 :
   ∀(n : Nat), n + 0 = 0 :=
by intros n
   simp
```

Conclusão

- Nesta segunda parte do minicurso, vimos:
  - Como conduzir provas em Lean;
  - Estratégias de prova (simplificação, *rewritting*, análise de casos, indução;
  - Táticas `simp`, `rfl`, `rw`, `cases` e `induction`.

Lean é um assistente de provas extremamente poderoso e tem sido cada vez mais usado pela comunidade de computação e matemática.

Referências

[1] GRAHAM HUTTON. **Functional Programming in Haskell**. Disponível em: https://www.youtube.com/watch?v=qThX0aoW9YI&list=PLF1Z-APd9zK7usPMx3LGMZEHrECUGodd3. Acesso em: 12 ago. 2025.

[2] **Lean Programming Language**. Disponível em: <https://lean-lang.org/documentation/>. Acesso em: 12 ago. 2025.

[3] PIERCE, Benjamin C. et al. **Logical Foundations**. University of Pennsylvania. Disponível em: https://softwarefoundations.cis.upenn.edu/lf-current. Acesso em: 12 ago. 2025.

Obrigado!

Dúvidas?

andre.santos1@ufv.br