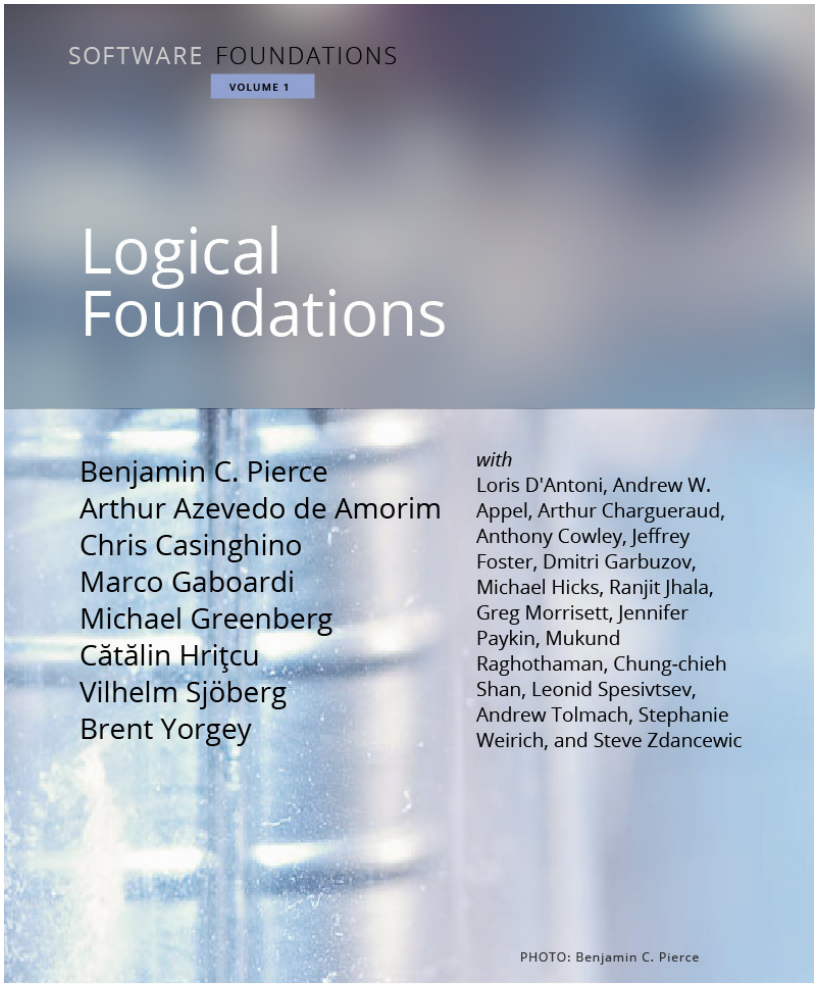
Introdução ao Assistente de Provas Lean

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

O que é Lean?

- Lean é uma linguagem funcional e um assistente de provas baseado em Teoria de Tipos Dependentes.
- Criado em 2013, por Leonardo de Moura, enquanto trabalhava na Microsoft Research.
- Atualmente, é mantida pela organização sem fins lucrativos Lean FRO (Focused Research Organization).

Programação Funcional

- Paradigma de programação declarativo em que a computação é baseada principalmente na aplicação e composição de funções.

C++ (Imperativo)

```
int sum(vector<int> &v) {
    int s=0;
    for(int i=0;i<v.size();i++)
        s += v[i];
    return s;
}
```

Haskell (Funcional)

```
sum [] = 0
sum v  = (head v) + (sum (tail v))
```

Quicksort Imperativo vs Funcional

C++ (Imperativo)

```
void qsort(vector<int> &v,
     int l, int r) {
    if (l >= r) return;
    int p = v[l];
    int i = l + 1, j = r;
    while (i <= j) {
      while (i <= r & v[i] < p)
        i++;
      while (j >= l & v[j] > p)
        j--;
      if (i <= j)
        swap(v[i++], v[j--]);
    }
    swap(v[l], v[j]);
    qsort(v, l, j - 1);
    qsort(v, j + 1, r);
}
```

Haskell (Funcional)

```
qsort [] = []
qsort (p:xs) =
    qsort [x | x <- xs, x <= p]
    ++ [p] ++
    qsort [x | x <- xs, x > p]
```

Programação Funcional

- Paradigma de programação declarativo em que a computação é baseada principalmente na aplicação e composição de funções.
- Principais características:
  - Imutabilidade de dados – estados não são modificados.
  - Recursão em vez de laços imperativos (*e.g.*, `for` e `while`).
  - Funções puras (mesmo input $\rightarrow$ mesmo output, sem efeitos colaterais).
  - Funções de ordem superior (recebem e retornam funções).

O que são Assistentes de Prova?

- Ferramentas desenvolvidas para auxiliar a condução de provas formais de teoremas escritos sobre definições do usuário.
- Têm sido amplamente adotados por matemáticos, como Terence Tao e Kevin Buzzard.
- Principais usos de Assistentes de Prova:
  - Conduzir provas de novos teoremos e revisar antigos.
  - Ferramenta de estudos para disciplinas como Introdução à Algebra, Matemática Discreta, Teoria da Computação, ...
  - Verificação de programas (escritos manualmente ou por IA).
  - Geração de código em outras linguagens.
- Exemplos são: Lean, Rocq (antigo Coq), Agda, Isabelle, ...

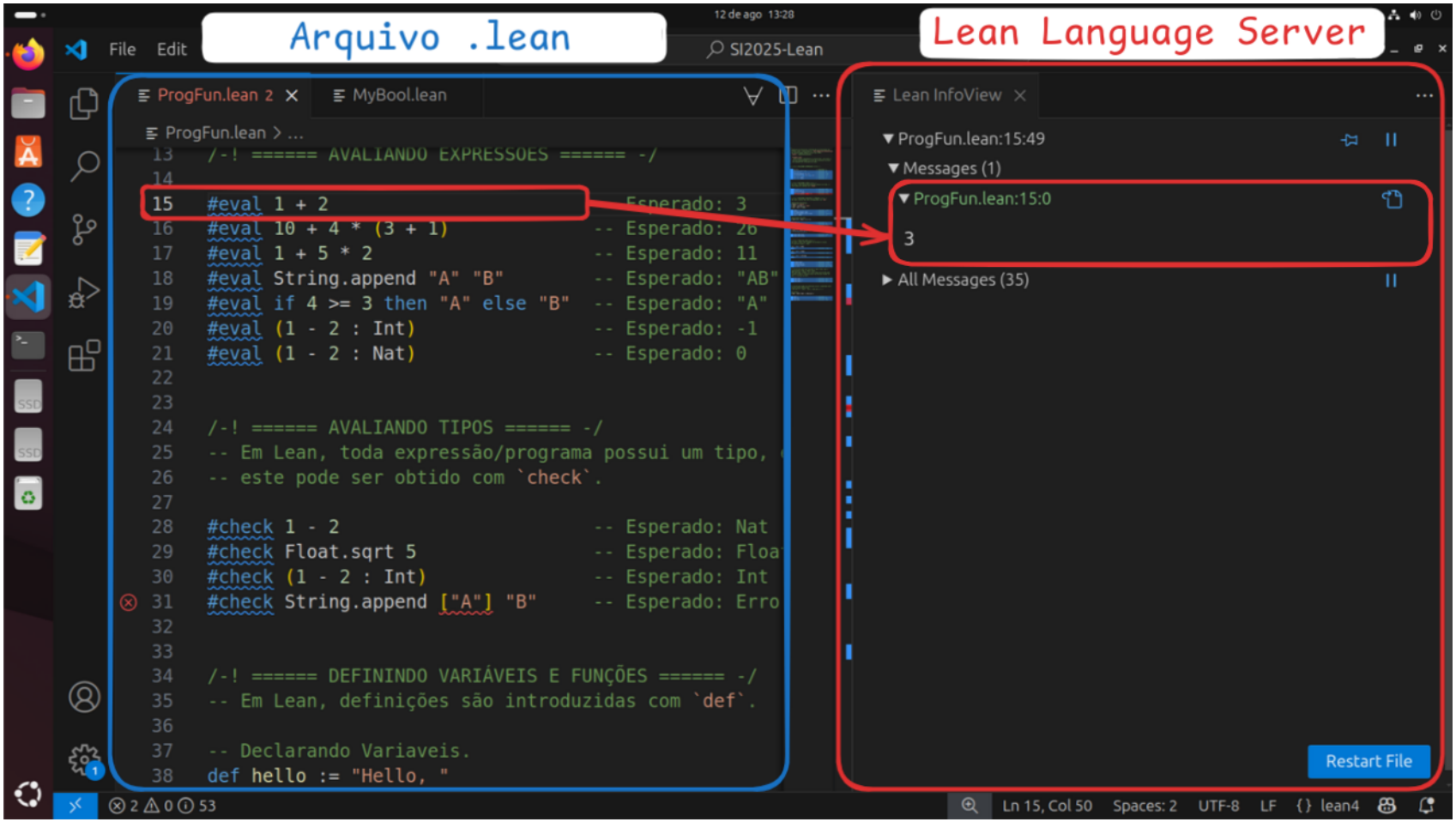
Primeiros Passos em Lean

- Lean pode ser instalado em Windows, Linux e Mac. É recomendado pela própria Lean FRO que seja utilizado no Visual Studio Code.
- Além disso, também possui uma versão online em `https://live.lean-lang.org/`.
- Quando o VS Code identifica um arquivo `.lean` (ou quanto o Lean Web é aberto), inicializa-se o **Lean Language Server**.

O Lean Language Server

- Implementa o Language Server Protocol.
- Integração do compilador com o editor de texto, que oferece *feedback* **enquanto** o programa é escrito.
- Mostra informações de tipo, erros, avaliação de expressões e estados intermediários de prova.

O Lean Language Server



Avaliando Expressões

Em Lean, podemos avaliar expressões utilizando `#eval`.

```
#eval 1 + 2                    -- Esperado:  3
#eval 10 + 4 * (3 + 1)         -- Esperado:  26
#eval 1 + 5 * 2                -- Esperado:  11
```

Avaliando Expressões

Em Lean, podemos avaliar expressões utilizando `#eval`.

```
#eval 1 + 2                                  -- Esperado:  3
#eval 10 + 4 * (3 + 1)                       -- Esperado:  26
#eval 1 + 5 * 2                              -- Esperado:  11
#eval String.append "A" "B"                  -- Esperado:  "AB"
#eval if 4 >= 3 then "A" else "B"  -- Esperado:  "A"
```

Avaliando Expressões

Em Lean, podemos avaliar expressões utilizando `#eval`.

```
#eval 1 + 2                                  -- Esperado:  3
#eval 10 + 4 * (3 + 1)                       -- Esperado:  26
#eval 1 + 5 * 2                              -- Esperado:  11
#eval String.append "A" "B"                  -- Esperado:  "AB"
#eval if 4 >= 3 then "A" else "B"            -- Esperado:  "A"
#eval (1 - 2 :  Int)                         -- Esperado:  -1
#eval (1 - 2 :  Nat)                         -- Esperado:  0
```

Avaliando Tipos

Toda expressão possui um tipo, o qual pode ser obtido com `#check`.

```
#check 1 - 2                          -- Esperado:  Nat
#check Float.sqrt 5                   -- Esperado:  Float
#check (1 - 2 :  Int)                 -- Esperado:  Int [1]
#check String.append ["A"] "B"        -- Esperado:  Erro
```

[1] Sem realizar a operação de subtração.

Definindo de Variáveis e Funções

- Em *Lean*, definições são introduzidas com `def`.

```
-- Declarando variaveis
def hello := "Hello, "
def lean :  String := "Lean"
def ano :  Nat := 2025

-- Declarando funções
def sum (n m :  Nat) := n + m
def mask (n :  Nat) (b :  Bool) :  Nat :=
    if b then n else 0
```

Tipos em Lean

- Em Lean, tipos são **cidadãos de primeira classe**.
- O tipo dos tipos é `Type`.

```
def α : Type := Nat
def β : Type := Bool
def γ : Type := String
```

Tipos em Lean

- Em Lean, tipos são **cidadãos de primeira classe**.
- O tipo dos tipos é `Type`.

```
def α : Type := Nat
def β : Type := Bool
def γ : Type := String
```

- Qual é o tipo de `Type`?

Tipos em Lean

- Em Lean, tipos são **cidadãos de primeira classe**.
- O tipo dos tipos é `Type`.

```
def α : Type := Nat
def β : Type := Bool
def γ : Type := String
```

- Qual é o tipo de `Type`? `Type 1`.

Tipos em Lean

- Em Lean, tipos são **cidadãos de primeira classe**.
- O tipo dos tipos é `Type`.

```
def α : Type := Nat
def β : Type := Bool
def γ : Type := String
```

- Qual é o tipo de `Type`? `Type 1`.
- E o de `Type 1`?

Tipos em Lean

- Em Lean, tipos são **cidadãos de primeira classe**.
- O tipo dos tipos é `Type`.

```
def α : Type := Nat
def β : Type := Bool
def γ : Type := String
```

- Qual é o tipo de `Type`? `Type 1`.
- E o de `Type 1`? `Type 2`.

Tipos em Lean

- Em Lean, tipos são **cidadãos de primeira classe**.
- O tipo dos tipos é `Type`.

```
def α : Type := Nat
def β : Type := Bool
def γ : Type := String
```

- Qual é o tipo de `Type`? `Type 1`.
- E o de `Type 1`? `Type 2`.
- E o de `Type 2`? ...

Tipos em Lean

- Em Lean, tipos são **cidadãos de primeira classe**.
- O tipo dos tipos é `Type`.

```
def α : Type := Nat
def β : Type := Bool
def γ : Type := String
```

- Qual é o tipo de `Type`? `Type 1`.
- E o de `Type 1`? `Type 2`.
- E o de `Type 2`? ...

- `Type` é um apelido para `Type 0`.

Tipos em Lean

- Se α e β são tipos, então:
  - $\alpha \to \beta$ denota uma função de α em β.
  - $\alpha \times \beta$ denota um par que consiste em um elemento do tipo α e outro do tipo β.

```
#check Float.sqrt    -- Esperado:  Float → Float
#check Float.sqrt 4  -- Esperado:  Float
#check String.length -- Esperado:  String → Nat
#check (1, 0)        -- Esperado:  Nat × Nat
```

Tipos em Lean

- A função `Nat.mul` tem tipo `Nat → Nat → Nat`, que é equivalente a `Nat → (Nat → Nat)`.
- `Nat.mul` pode, então, ser vista como uma função que recebe um `Nat` e retorna outra função que recebe um `Nat`, e retorna um `Nat`.

```
#check Nat.mul      -- Esperado:  Nat → Nat→ Nat
#check Nat.mul 3 2  -- Esperado:  Nat
#check Nat.mul 5    -- Esperado:  Nat → Nat
```

Tipos em Lean

- A função `Nat.mul` tem tipo `Nat → Nat → Nat`, que é equivalente a `Nat → (Nat → Nat)`.
- `Nat.mul` pode, então, ser vista como uma função que recebe um `Nat` e retorna outra função que recebe um `Nat`, e retorna um `Nat`.

```
#check Nat.mul      -- Esperado:  Nat → Nat→ Nat
#check Nat.mul 3 2  -- Esperado:  Nat
#check Nat.mul 5    -- Esperado:  Nat → Nat

 -- Ou seja, a função dbl pode ser definida assim
def dbl (n :  Nat) :  Nat := Nat.mul 2 n
```

Tipos em Lean

- A função `Nat.mul` tem tipo `Nat → Nat → Nat`, que é equivalente a `Nat → (Nat → Nat)`.
- `Nat.mul` pode, então, ser vista como uma função que recebe um `Nat` e retorna outra função que recebe um `Nat`, e retorna um `Nat`.

```
#check Nat.mul      -- Esperado:  Nat → Nat→ Nat
#check Nat.mul 3 2  -- Esperado:  Nat
#check Nat.mul 5    -- Esperado:  Nat → Nat

 -- Ou assim
def dbl :  Nat → Nat := Nat.mul 2
```

Enumerações

- Vamos definir o tipo `MyBool`, cujos membros são `True` e `False`.

```
-- Definição do tipo MyBool.
inductive MyBool where
   | True :  MyBool
   | False :  MyBool

#check MyBool.True [2]
#check MyBool.False
#check MyBool
```

[2] É possível omitir o "`MyBool.`" do constructo `True` utilizando `open`, porém – como também há `True` definido pelo Lean – em alguns casos será necessário para evitar ambiguidade. Nessa apresentação – por simplificação – sempre será omitido.

Casamento de Padrões (*Pattern Matching*)

- Verificar a presença de um determinado padrão em uma dada entrada.
- No caso do tipo `MyBool`, isto é consideravelmente simples: ou é `True` ou é `False`.

```
-- Definição do operador lógico AND.
def andb (b1 b2 : MyBool) : MyBool :=
  match b1 with
    | False => False
    | True => b2

#eval andb True True      -- Esperado: True
#eval andb False True     -- Esperado: False
```

Declarando Notações

- É possível declarar notações para aplicações de funções utilizando `notation`.

```
-- Notação para o operador lógico AND.
notation:60 x " ∧ " y => andb x y

-- Notação para o operador lógico OR.
notation:60 x " ∨ " y => orb x y

-- Notação para o operador lógico NOT.
notation:60 "¬"x => negb x
```

Constructos com Argumentos

- É também possível definir enumerações cujos constructos recebem argumentos.

```
inductive Bit where
   | B0 :  Bit
   | B1 :  Bit

inductive Bytes where
   | Bits (B0 B1 B2 B3 B4 B5 B6 B7 :  Bit)

 -- True caso o byte esteja zerado.  Senão, False.
def all_zero (b :  Byte) :  MyBool :=
   match b with
   | Bits B0 B0 B0 B0 B0 B0 B0 B0 => MyBool.True
   | Bits _ _ _ _ _ _ _ _ => MyBool.False
```

Tipos Indutivos

- Todos os tipos que definimos até agora (`MyBool`, `Byte`) são finitos. O tipo dos naturais – por outro lado – é infinito.
- Existem diversas maneiras de representar números naturais (base decimal, binária, hexadecimal, ...).
- Vamos definir os números naturais usando dois constructos: O, representando o 0, e S, representando o sucessor de um natural (n+1).

```
-- Definição do tipo MyNat.
inductive MyNat where
    | O : MyNat
    | S (n :  MyNat) :  MyNat
```

- 0 é `O`, 1 é `S O`, 2 é `S (S O)`, 3 é `S (S (S O))`, ...

Casamento de padrões - MyNat

```
-- Soma de naturais.
def plus (n m : MyNat) : MyNat :=
  match n with
  | 0 => m
  | S n' => S (plus n' m)

-- Subtração de naturais.
def minus (n m : MyNat) : MyNat :=
  match n, m with
  | 0, _ => 0
  | _, 0 => n
  | S n', S m' => minus n' m'
```

Conclusão

Nessa primeira parte do minicurso vimos:

- O que é Lean;
- Avaliação de expressões e tipos;
- Tipos em Lean;
- Enumerações, Tipos Indutivos e Casamento de Padrões.

Agora que estamos familiarizados com **Linguagem Funcional** Lean, vamos conhecer o **Assistente de Provas** Lean.

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.