Capítulo 4 – Lógica Proposicional no Lean

4.1. Expressões para proposições e provas

- Em sua essência, Lean é um verificador de tipos. Isso significa que podemos escrever expressões e pedir ao sistema que verifique se eles estão bem formadas e também pedir ao sistema que nos diga o tipo de objeto que eles denotam.

```
variables A B C : Prop

#check A ∧ ¬ B → C
```

- Nesse exemplo, declaramos três variáveis variando sobre proposições e pedimos a Lean para verificar a expressão $A \land \neg B \to C$. A saída do comando #check é $A \land \neg B \to C$: Prop, que afirma que $A \land \neg B \to C$ é do tipo Prop. Em Lean, toda expressão bem formada tem um tipo.

- Além de declarar variáveis, se P é qualquer expressão do tipo Prop, podemos declarar a hipótese de que P é verdadeiro.

```
variables A B : Prop
variable  h : A ∧ ¬ B

#check h
```

- Qualquer proposição pode ser vista como um tipo. Uma hipótese, ou premissa, é apenas uma variável desse tipo. Construir provas é uma questão de escrever as expressões do tipo de escrita**(?)**. Por exemplo, se P é qualquer expressão do tipo A ∧ B, então and.left P é uma expressão do tipo A e and.right P é uma expressão do tipo B. Em outras palavras, se P é uma prova de A ∧ B, and.left P é um nome para a prova que você obtém aplicando a regra de eliminação da esquerda para o and.

$$\begin{array}{c} \vdots \\ P \\ \vdots \\ \dfrac{A \wedge B}{A} \end{array}$$

- De forma análoga, and.right P é a prova de B. Continuando esse exemplo, podemos escrever:

```
variables A B : Prop
variable h : A ∧ ¬ B

#check and.left h
#check and.right h
```

- Que nos representará as seguintes provas:

$$\dfrac{\overline{A \wedge \neg B}^{\,H}}{A} \qquad \dfrac{\overline{A \wedge \neg B}^{\,H}}{\neg B}$$

- Ao representarmos nossas provas dedução natural através do LEAN, não teremos nenhuma hipótese "livre", sem "label". Todas as nossas hipóteses precisarão ser declaradas para serem efetivamente utilizadas.

- Se h1 for uma prova de A e h2 é uma prova de B, então and.intro h1 h2 é uma prova de A ∧ B.

```
variables A B : Prop
variable h : A ∧ ¬ B

#check and.intro (and.right h) (and.left h)
```

```
test.lean C:\lean330test\bin 1
[Lean] ⟨h.right, h.left⟩ : ¬B ∧ A (4, 1)
```

- A regra de eliminação da implicação é simples: se P_1 for uma prova de $A \to B$ e P_2 é uma prova de A, então P_1 P_2 é uma prova de B. Observe que nem precisamos nomear a regra: você apenas escreve P1 seguido de P2, como se estivéssemos aplicando o primeiro ao segundo. Se P1 e P2 são expressões compostas, colocamos parênteses ao seu redor para deixar claro onde cada uma expressão começa e onde a outra termina.

```
variables A B C D : Prop

variable h1 : A → (B → C)
variable h2 : D → A
variable h3 : D
variable h4 : B

#check h2 h3
#check h1 (h2 h3)
#check (h1 (h2 h3)) h4
```

```
[Lean] h2 h3 : A (8, 1)
[Lean] h1 (h2 h3) : B → C (9, 1)
[Lean] h1 (h2 h3) h4 : C (10, 1)
```

- Lean adota a convenção de que as **aplicações** se associam **à esquerda**, de modo que uma expressão h1 h2 h3 seja interpretada como (h1 h2) h3. As **implicações** se associam **à direita**, de modo que A → B → C seja interpretado como A → (B → C).

```
variables A B C D : Prop

variable h1 : A → B → C
variable h2 : D → A
variable h3 : D
variable h4 : B

#check h2 h3
#check h1 (h2 h3)
#check h1 (h2 h3) h4
```

```
[Lean] h2 h3 : A (8, 1)
[Lean] h1 (h2 h3) : B → C (9, 1)
[Lean] h1 (h2 h3) h4 : C (10, 1)
```

- A regra de introdução da implicação é mais complexa, porque geralmente cancelamos uma hipótese para demonstrá-la. Em termos de expressões Lean, a regra se traduz da seguinte maneira. Supondo que A e B são do tipo Prop, e, supondo que h seja a premissa de que A é válido , P é prova de B, possivelmente envolvendo h. Então, a expressão **assume h: A, P** é uma prova de A → B. Por exemplo, podemos construir uma prova de A → A ∧ A da seguinte maneira:

```
variable A : Prop

#check (assume h : A, and.intro h h)
```

```
[Lean] λ (h : A), ⟨h, h⟩ : A → A ∧ A (3, 1)
```

- Já não temos que declarar A como premissa. A palavra *assume* torna a premissa local à expressão entre parênteses, e após a hipótese ser feita, podemos nos referir a h. Dado o pressuposto h: A, and.intro h h é uma prova de A ∧ A, e assim a expressão assume h: A, and.intro h h é uma prova de A → A ∧ A.

- Acima, provamos ¬ B ∧ A a partir da premissa A ∧ ¬ B. Podemos, em vez disso, obter uma prova de A ∧ ¬ B → ¬ B ∧ A da seguinte maneira:

```
variables A B : Prop

#check (assume h : A ∧ ¬ B, and.intro (and.right h) (and.left h))
```

```
[Lean] λ (h : A ∧ ¬B), (h.right, h.left) : A ∧ ¬B → ¬B ∧ A (3, 1)
```

- Tudo que fizemos foi colocar h: A ∧ B em um escopo local.
- Antes:

```
variables A B : Prop
variable h : A ∧ ¬ B

#check and.intro (and.right h) (and.left h)
```

```
test.lean C:\lean330test\bin 1
[Lean] (h.right, h.left) : ¬B ∧ A (4, 1)
```

4.2. Outros comandos do LEAN

- O primeiro comando a ser apresentado é o example. Este comando diz ao Lean que estamos prestes a provar um teorema ou, em geral, anotar uma expressão de um dado tipo. Após isso, ele deve ser seguido pela prova ou pela própria expressão.

```
variables A B : Prop

example : A ∧ ¬ B → ¬ B ∧ A :=
assume h : A ∧ ¬ B,
and.intro (and.right h) (and.left h)
```

- Lean verifica se a expressão após o := tem o tipo certo. Se assim for, ela é uma prova válida. Caso contrário, ele é incorreta e o LEAN levanta uma mensagem de erro.

```
1   variables A B : Prop
2
3   example : A ∧ ¬ B → ¬ B ∧ A :=
4   assume h : A ∧ B,
5   and.intro (and.right h) (and.left h)
```

```
[Lean] type mismatch at application h.right term h has type A ∧ B but is expected to have type ?m_1 ∧ ¬B (5, 12)
```

- Como o comando nos diz qual expressão (ou tipo) esperamos ter(neste caso, a proposição que está sendo provada), as vezes pode ser mais conveniente omitir informações. Por exemplo, não precisamos explicitar o tipo da suposição:

```
variables A B : Prop

example : A ∧ ¬ B → ¬ B ∧ A :=
assume h,
and.intro (and.right h) (and.left h)
```

- Podemos ir também na direção contrária e explicitar mais ainda as informações, com o comando ***show***. "**show A, from P**" significa que provamos A a partir de P. O LEAN, ao verificar estar expressão, confirma se P é realmente uma prova de A, antes de continuar analisando o restante da expressão. Assim, o exemplo anterior pode ser reescrito da seguinte maneira:

```
variables A B : Prop

example : A ∧ ¬ B → ¬ B ∧ A :=
assume h : A ∧ ¬ B,
show ¬ B ∧ A, from and.intro (and.right h) (and.left h)
```

- Podemos usar o show até nas expressões menores and.right h e and.left h, da seguinte maneira:

```
variables A B : Prop

example : A ∧ ¬ B → ¬ B ∧ A :=
assume h : A ∧ ¬ B,
show ¬ B ∧ A, from and.intro
  (show ¬ B, from and.right h)
  (show A, from and.left h)
```

- Embora o comando show não seja necessário para a construção de uma prova, temos ótimos motivos para usá-lo.
- Primeiro, e talvez o mais importante, ele torna as provas mais fáceis de ler e entender.
- Em segundo lugar, torna as provas mais fáceis de escrever: Por explicitar qual tipo iríamos provar, ao cometer um erro, é mais fácil para Lean indicar qual expressão foi tipada errada e fornecer uma mensagem de erro significativa.
- Finalmente, provar informações com o show geralmente permite que omitamos informações em outros lugares, uma vez que o LEAN pode inferir essa informação daquilo que tivemos intenção de declarar. Mas, como nosso intuito por agora é clareza e explicitar informações, não iremos nos preocupar com esta parte.

- Ao invés de declarar as variáveis e as premissas de antemão, tornando as globais, podemos apresentá-las como "argumentos" para o example, seguido de dois pontos. Quando terminarmos de provar o example, não será mais possível utilizá-las:

```
example (A B : Prop) : A ∧ ¬ B → ¬ B ∧ A :=
assume h : A ∧ ¬ B,
show ¬ B ∧ A, from and.intro (and.right h) (and.left h)
```

- Há mais dois "truques" que podem ajudá-lo a escrever provas em Lean: O **Sorry** e o **Placeholder**.
- Sorry é um termo do LEAN que serve como prova para qualquer coisa. Mesmo que temporário, ele é bastante útil: Se o LEAN aceitou a nossa prova até agora, então estamos montando tudo corretamente. Depois disso, basta substituir cada sorry por uma prova real.

```
1   variables A B : Prop
2
3   example : A ∧ ¬ B → ¬ B ∧ A :=
4   assume h, sorry
5
6   example : A ∧ ¬ B → ¬ B ∧ A :=
7   assume h, and.intro sorry sorry
8
9   example : A ∧ ¬ B → ¬ B ∧ A :=
10  assume h, and.intro (and.right h) sorry
11
12  example : A ∧ ¬ B → ¬ B ∧ A :=
13  assume h, and.intro (and.right h) (and.left h)
```

```
⚠ [Lean] declaration '[anonymous]' uses sorry (3, 1)
⚠ [Lean] declaration '[anonymous]' uses sorry (6, 1)
⚠ [Lean] declaration '[anonymous]' uses sorry (9, 1)
```

- Placeholder é representado por um underscore _.
- Ao utilizá-lo em uma expressão, estamos pedindo ao sistema que tente preencher aquela parte da prova com algum valor.
- Isso não significa pedir ao LEAN para construir a prova de um teorema automaticamente; Em vez disso, ele irá inferir um valor a partir das informações que estão disponíveis. Se você usar um sublinhado onde uma prova deveria estar, LEAN nos dará uma mensagem de erro que indica o que está faltando.

- Substituindo todo os sorrys da prova anterior:

```
1  variables A B : Prop
2
3  example : A ∧ ¬ B → ¬ B ∧ A :=
4  assume h, _
5
6  example : A ∧ ¬ B → ¬ B ∧ A :=
7  assume h, and.intro _ _
8
9  example : A ∧ ¬ B → ¬ B ∧ A :=
10 assume h, and.intro (and.right h) _
11
12 example : A ∧ ¬ B → ¬ B ∧ A :=
13 assume h, and.intro (and.right h) (and.left h)
14
```

```
[Lean] don't know how to synthesize placeholder context: A B : Prop, h : A ∧ ¬B ⊢ ¬B ∧ A (4, 11)
[Lean] don't know how to synthesize placeholder context: A B : Prop, h : A ∧ ¬B ⊢ ¬B (7, 21)
[Lean] don't know how to synthesize placeholder context: A B : Prop, h : A ∧ ¬B ⊢ A (7, 23)
[Lean] don't know how to synthesize placeholder context: A B : Prop, h : A ∧ ¬B ⊢ A (10, 35)
```

4.3. Construíndo Provas de Dedução Natural

- Nesta seção, descrevemos uma tradução mecânica a partir de provas de dedução natural, dando uma tradução para cada regra de dedução natural.
- Antes de começarmos, temos mais uma definição a ser dada:
- Para delimitar o escopo de variáveis ou premissas introduzidas com o comando variables, coloque-os em um bloco que comece com a palavra **section** e termine com o a palavra **end**.

4.3.1. Implicação

- Nós já explicamos que a introdução da implicação é implementada com assume, e a eliminação da implicação é escrita como uma aplicação.

```
variables A B : Prop

example : A → B :=
assume h : A,
show B, from sorry

section
  variable h1 : A → B
  variable h2 : A

  example : B := h1 h2
end
```

4.3.2. Conjunção

- Nós já vimos que a introdução da conjunção é implementada com and.intro, e as regras de eliminação são and.left e and.right.

```
variables A B : Prop
section
  variables (h1 : A) (h2 : B)

  example : A ∧ B := and.intro h1 h
end

section
  variable h : A ∧ B

  example : A := and.left h
  example : B := and.right h
end
```

4.3.3. Disjunção

- As regras de introdução da disjunlão são fornecidas por or.inl e or.inr.

```
section
  variable h : A

  example : A ∨ B := or.inl h
end

section
  variable h : B

  example : A ∨ B := or.inr h
end
```

- A regra de eliminação é um pouco mais complexa. Para provar C de A v B, você precisa de três argumentos: uma prova h de A v B, uma prova de C a partir A e uma prova de C a partir de B.

```
variables A B C : Prop
section
  variable h : A ∨ B

  example : C :=
  or.elim h
    (assume h1 : A,
      show C, from sorry)
    (assume h1 : B,
      show C, from sorry)
end
```

4.3.4 - Negação

- Internamente, a negação ¬ A é definida por A → false, que significa chegar a uma contradição a partir de A. As regras de negação são, portanto, semelhantes às regras de implicação. Para provar ¬ A, assumimos A e derivamos uma contradição.

```
section
  example : ¬ A :=
  assume h : A,
  show false, from sorry
end
```

- Se você provou a negação ¬ A, você pode obter uma contradição aplicando-a uma prova de A.

```
section
  variable h1 : ¬ A
  variable h2 : A

  example : false := h1 h2
end
```

4.3.5. Verdade e falsidade

- Ex-falso quodlibet é chamado de false.elim:

```
section
  variable h : false

  example : A := false.elim h
end
```

- Não tem muito a se dizer sobre True, já que ele sempre é trivialmente verdade:

```
example : true := trivial
```

4.3.6. Se e somente se: Dupla-Implicação

- A regra de introdução para “se e somente se" é iff.intro.

```
example : A ↔ B :=
iff.intro
  (assume h : A,
    show B, from sorry)
  (assume h : B,
    show A, from sorry)
```

- As regras de eliminação são iff.elim_left e iff.elim_right:

```
section
  variable h1 : A ↔ B
  variable h2 : A

  example : B := iff.elim_left h1 h2
end

section
  variable h1 : A ↔ B
  variable h2 : B

  example : A := iff.elim_right h1 h2
end
```

- O Lean reconhece a abreviação iff.mp para iff.and_elim_left, onde "mp" significa "modus ponens". Da mesma forma, você pode usar iff.mpr, para "modus ponens reverse", em vez de iff.and_elim_right.

4.3.7. Reductio ad absurdum (prova por absurdo)

- A regra é chamada de by_contradiction. Ela tem apenas um argumento, que é uma prova do falso a partir de ¬ A. Para usar a regra, temos que usar o argumento **open classical** antes da prova. Devemos utilizar isso antes de escrevermos nossa prova:

```
section
  open classical

  example : A :=
  by_contradiction
    (assume h : ¬ A,
      show false, from sorry)
end
```

4.3.8. Exemplos

- No ultimo capítulo, nós construímos a prova A→C a partir de A→B e B→C.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{A}\,1 \qquad A \to B}{B} \qquad B \to C}{C}}{A \to C}\,1$$

- Podemos modelá-la da seguinte maneira no LEAN:

```
variables A B C : Prop

variable h1 : A → B
variable h2 : B → C

example : A → C :=
assume h : A,
show C, from h2 (h1 h)
```

- Outro exemplo que trabalhamos foi o seguinte:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{A \to (B \to C)}\,{}^{2} \qquad \dfrac{\dfrac{}{A \wedge B}\,{}^{1}}{A}}{B \to C} \qquad \dfrac{\dfrac{}{A \wedge B}\,{}^{1}}{B}}{C}}{A \wedge B \to C}\,{}^{1}}{(A \to (B \to C)) \to (A \wedge B \to C)}\,{}^{2}$$

- E aqui está a sua representação no LEAN:

```
example (A B C : Prop) : (A → (B → C)) → (A ∧ B → C) :=
assume h1 : A → (B → C),
assume h2 : A ∧ B,
show C, from h1 (and.left h2) (and.right h2)
```

$$\dfrac{\dfrac{\dfrac{\dfrac{}{A \wedge (B \vee C)}{}^{2}}{B \vee C} \qquad \dfrac{\dfrac{\dfrac{\dfrac{}{A \wedge (B \vee C)}{}^{2}}{A} \quad \dfrac{}{B}{}^{1}}{A \wedge B}}{(A \wedge B) \vee (A \wedge C)} \qquad \dfrac{\dfrac{\dfrac{\dfrac{}{A \wedge (B \vee C)}{}^{2}}{A} \quad \dfrac{}{C}{}^{1}}{A \wedge C}}{(A \wedge B) \vee (A \wedge C)}}{(A \wedge B) \vee (A \wedge C)}{}^{1}}{(A \wedge (B \vee C)) \rightarrow ((A \wedge B) \vee (A \wedge C))}{}^{2}$$

- Aqui está sua representação no LEAN:

```
example (A B C : Prop) : A ∧ (B ∨ C) → (A ∧ B) ∨ (A ∧ C) :=
assume h1 : A ∧ (B ∨ C),
or.elim (and.right h1)
  (assume h2 : B,
    show (A ∧ B) ∨ (A ∧ C),
      from or.inl (and.intro (and.left h1) h2))
  (assume h2 : C,
    show (A ∧ B) ∨ (A ∧ C),
      from or.inr (and.intro (and.left h1) h2))
```

4.4. Forward Reasoning

- Lean dá suporte a Forward Reasoning, permitindo que você escreva provas usando o comando have:

```
variables A B C : Prop

variable h1 : A → B
variable h2 : B → C

example : A → C :=
assume h : A,
have h3 : B, from h1 h,
show C, from h2 h3
```

- O commando have h : A, from P verifica se A é provado a partir de P e, em seguida, rotula toda a prova P como h. Assim, a última linha da prova anterior pode ser considerada como abreviação de h2 (h1 h), uma vez que h3 abreviou h1 h. Tais ajudam bastante, especialmente quando a prova P é muito longa.
- Há várias vantagens em usar have. Por um lado, torna a prova mais legível; Ele nos poupa de repetirmos a mesma prova várias vezes: h3 pode ser usado repetidamente após a introdução; Finalmente, ele torna mais fácil a construção e depuração da prova: declarar B como o que iremos provar torna mais fácil para Lean mostrar uma mensagem de erro mais clara quando não montarmos nossa prova corretamente.

- Na última seção, consideramos a seguinte prova:

```
example (A B C : Prop) : (A → (B → C)) → (A ∧ B → C) :=
assume h1 : A → (B → C),
assume h2 : A ∧ B,
show C, from h1 (and.left h2) (and.right h2)
```

- Utilizando o have, ela pode ser reescrita das seguintes maneiras:

```
example (A B C : Prop) : (A → (B → C)) → (A ∧ B → C) :=
assume h1 : A → (B → C),
assume h2 : A ∧ B,
have h3 : A, from and.left h2,
have h4 : B, from and.right h2,
show C, from h1 h3 h4
```

```
example (A B C : Prop) : (A → (B → C)) → (A ∧ B → C) :=
assume h1 : A → (B → C),
assume h2 : A ∧ B,
have h3 : A, from and.left h2,
have h4 : B, from and.right h2,
have h5 : B → C, from h1 h3,
show C, from h5 h4
```

- Adicionar mais informações nem sempre torna uma prova mais legível; quando as expressões individuais são pequenas e fáceis de entender, escrevê-las com muitos detalhes pode ser mais bagunçado do que escrever uma prova menor.
- Compare estas versões da mesma prova de $A \wedge B \to B \wedge A$:

```
example (A B : Prop) : A ∧ B → B ∧ A :=
assume h1 : A ∧ B,
have h2 : A, from and.left h1,
have h3 : B, from and.right h1,
show B ∧ A, from and.intro h3 h2
```

```
example (A B : Prop) : A ∧ B → B ∧ A :=
assume h1 : A ∧ B,
show B ∧ A, from
  and.intro
    (show B, from and.right h1)
    (show A, from and.left h1)
```

```
example (A B : Prop) : A ∧ B → B ∧ A :=
λ h, and.intro (and.right h) (and.left h)
```

- A única diferença entre estas três provas é a legibilidade, pois para o LEAN, elas significam a mesma coisa.

- Utilizar o commando have para explicitar qual disjunção usaremos, ajuda bastante na hora de fazer a or.elim:

```
example (A B C : Prop) : C :=
have h : A ∨ B, from sorry,
show C, from or.elim h
  (assume h1 : A,
    show C, from sorry)
  (assume h2 : B,
    show C, from sorry)
```

4.5. Definições e Teoremas

- O LEAN nos permite nomear definições e teoremas para serem utilizadas mais tarde.

```
def triple_and (A B C : Prop) : Prop :=
A ∧ (B ∧ C)
variables D E F G : Prop
#check triple_and (D ∨ E) (¬ F → G) (¬ D)
```

```
[Lean] triple_and (D ∨ E) (¬F → G) (¬D) : Prop (4, 1)
```

- Da mesma maneira que tratávamos o example, não importa se declaramos as variáveis antes de fazermos a definição ou dentro da mesma.

- Um das ferramentas mais importantes do LEAN é o comando theorem, que nos permite provar e nomear teoremas para serem utilizados mais tarde.
- Uma vez definidos, Podemos utilizá-los livremente:

```
theorem and_commute (A B : Prop) : A ∧ B → B ∧ A :=
assume h, and.intro (and.right h) (and.left h)

variables C D E : Prop
variable h1 : C ∧ ¬ D
variable h2 : ¬ D ∧ C → E

#check and_commute C (¬D)

example : E := h2 (and_commute C (¬ D) h1)
```

```
[Lean] and_commute C (¬D) : C ∧ ¬D → ¬D ∧ C (8, 1)
```

- Nem sempre precisamos passar os argumentos C e ¬D explicitamente, pois estes podem se extraídos a partir de h1. Para fazer isso, basta escrevermos nosso teorema da seguinte maneira:

```
theorem and_commute {A B : Prop} : A ∧ B → B ∧ A :=
assume h, and.intro (and.right h) (and.left h)
```

- As chaves indicam que os argumentos A e B estão implícitos, ou seja, LEAN deve extraí-los do contexto quando o teorema for utilizado. Assim, nosso ultimo exemplo pode ser escrito da seguinte maneira:

```
variables C D E : Prop
variable h1 : C ∧ ¬ D
variable h2 : ¬ D ∧ C → E

example : E := h2 (and_commute h1)
```