Capítulo 9 – Lógica de Primeira Ordem no Lean

9.1 – Funções, predicados e relações

- Nos últimos dois capítulos, vimos um pouco sore a lógica de primeira ordem. Agora, vamos utilizar o LEAN para nos auxiliar a construir nossas provas, pois o mesmo possui a lógica de primeira ordem incorporada a si.
- Assim, tudo que pode ser escrito e provado em lógica simbólica, pode ser escrito e comprovado em LEAN.
- LEAN foi feito com base na Teoria dos Tipos, onde cada variável representa elementos de algum tipo, que, de certa forma, é equivalente a noção de "Universo de Interesse", visto na lógica simbólica.

- Por exemplo, suponha que queremos uma certa linguagem de lógica de primeira ordem que tem:
- Um símbolo de constantes, c;
- Um símbolo de função unária, f;
- Um símbolo de função binária, g;
- Um símbolo de relação unária, P;
- Um símbolo de relação binária, R;
- Com base nisso, podemos declarar um novo tipo U e representar os símbolos da seguinte maneira:

```
constant U : Type

constant c : U
constant f : U → U
constant g : U → U → U
constant P : U → Prop
constant R : U → U → Prop
```

- Depois disso, podemos utilizá-los em outras operações:

```
variables x y : U

#check c
#check f c
#check g x y
#check g x (f c)

#check P (g x (f c))
#check R x y
```

- Como vimos anteriormente, #check nos diz qual é o tipo da expressão que passamos para ele. As 4 primeiras tem tipo U, enquanto as duas últimas tem tipo Prop.

- Algumas observações:

- Uma função unária é representada como um objeto do tipo U → U. Escreveríamos f x para denotar a aplicação de f a x.
- Uma função binária é representada como um objeto do tipo U → U → U. Escreveríamos g x y para denotar a aplicação de g a x e y.
- Um relação unária (ou predicado) é representada como um objeto do tipo U → Prop. Escrever P x denota que o predicado P holds of x(?).
- Uma relação binária é representada como um objeto do tipo U → U → Prop. Escrever R x y denota que R holds of x and y(?).

- Da mesma maneira que definimos constantes, poderíamos ter definido variáveis:

```
variable U : Type

variable c : U
variable f : U → U
variable g : U → U → U
variable P : U → Prop
variable R : U → U → Prop

variables x y : U

#check c
#check f c
#check g x y
#check g x (f c)

#check P (g x (f c))
#check R x y
```

- Apesar dos exemplos funcionarem da mesma maneira, “constant” e “variable” funcionam de maneira bastante diferente.
- O comando “constant” declara um novo objeto, axiomaticamente, e o adiciona à lista de objetos que o Lean conhece.
- Em contrapartida, o comando “variable”, quando executado pela primeira vez, não cria nada.
- Em vez disso, sempre que inserirmos uma expressão usando o identificador correspondente, ela deverá criar uma variável temporária do tipo correspondente.

- Vamos modelar (e testar) a linguagem da aritmética utilizando o tipo nat, que representa os números naturais:

```
constant mul : ℕ → ℕ → ℕ
constant add : ℕ → ℕ → ℕ
constant square : ℕ → ℕ
constant even : ℕ → Prop
constant odd : ℕ → Prop
constant prime : ℕ → Prop
constant divides : ℕ → ℕ → Prop
constant lt : ℕ → ℕ → Prop
constant zero : ℕ
constant one : ℕ
variables w x y z : ℕ

#check mul x y
#check add x y
#check square x
#check even x
```

- Como visto anteriormente, podemos usar os conectivos lógicos para montarmos expressões dentro do nosso universo:

```
constant square : ℕ → ℕ
constant prime : ℕ → Prop
constant even : ℕ → Prop

variables w x y z : ℕ

#check even (x + y + z) ∧ prime ((x + 1) * y * y)
#check ¬ (square (x + y * z) = w) ∨ x + y < z
#check x < y ∧ even x ∧ even y → x + 1 < y
```

- O LEAN nos ajuda a distinguir entre termos e fórmulas. Se digitamos #check x + y + 1, somos informados de que ela tem o tipo ℕ, isto é, que denota um número natural. Se escrevemos #check even(x + y + 1), nos é dito que ela possui o tipo Prop.

- Algumas observações sobre a sintaxe:

- Funções são aplicadas sem utilizar parenteses. Ao invés de escrever square(x) ou add(x,y), simplesmente escrevemos square x ou add x y. O mesmo vale para relações e predicados.
- A notação add $\mathbb{N} \to \mathbb{N} \to \mathbb{N}$ denota que a adição recebe dois argumentos pertencentes aos naturais e retorna outro número natural.
- Similarmente, divides : $\mathbb{N} \to \mathbb{N} \to$ Prop indica que divides é uma relação binária, recebendo dois números naturais e retornando uma proposição (que pode ser verdadeiro ou falsa). Divides x y indica que x é divisível por y.

- No capítulo 7, comentamos sobre a "many-sorted logic", onde podemos ter mais de um "Universo de Interesse". Podemos voltar ao exemplo de linhas e pontos e modelá-lo da seguinte maneira:

- Após ter definido que p,q, ... Representariam pontos e que L, R, M, ... Representariam linhas, chegamos a este predicado:

$$\forall p, q, L, M\ (on(p, L) \wedge on(q, L) \wedge on(p, M) \wedge on(q, M) \rightarrow L = M).$$

- Agora, em LEAN:

```
variables Point Line : Type
variable  lies_on : Point → Line → Prop

#check ∀ (p q : Point) (L M : Line),
p ≠ q → lies_on p L → lies_on q L → lies_on p M → lies_on q M → L = M
```

- Não precisaríamos declamar quais eram os tipos de p, q, L e M, pois, a partir da relação lies_on, LEAN consegue determinar qual o tipo de cada variável. Assim:

```
#check ∀ p q L M, p ≠ q → lies_on p L → lies_on q L →
lies_on p M → lies_on q M → L = M
```

9.2 – Utilizando o Quantificador Universal

- Vamos ver agora algumas maneiras de como podemos usar o quantificador universal:

```
constant prime : ℕ → Prop
constant even : ℕ → Prop
constant odd : ℕ → Prop

#check ∀ x, (even x ∨ odd x) ∧ ¬ (even x ∧ odd x)
#check ∀ x, even x ↔ 2 ∣ x
#check ∀ x, even x → even (x^2)
#check ∀ x, even x ↔ odd (x + 1)
#check ∀ x, prime x ∧ x > 2 → odd x
#check ∀ x y z, x ∣ y → y ∣ z → x ∣ z
```

- Lembre-se que LEAN requer uma vírgula após o quantificador universal. Lembre-se também que ele também assumirá o maior escopo possível.

- Para montarmos uma prova sobre o quantificador universal, seguiremos este padrão:

```
variable U : Type
variable P : U → Prop

example : ∀ y, P y :=
assume x,
show P x, from sorry

example : ∀ x, P x :=
assume y,
show P y, from sorry
```

- Entenda o assume como: “Dado um valor arbitrário x pertencente a U”.
- Como podemos renomear variáveis ligadas e de ligação, as duas provas são equivalentes.

- O que acabamos de ver constitui a regra de introdução do quantificador existencial. Ela é bastante parecida com a regra de introdução da implicação, vista no capítulo 4.
- A regra de eliminação terá a seguinte cara:

```
variable U : Type
variable P : U → Prop
variable h : ∀ x, P x
variable a : U

example : P a :=
show P a, from h a
```

- Novamente, a prova se parece bastante com a regra de eliminação da implicação. P a é obtido através da aplicação de h a a.

- Aqui está um exemplo de como usá-los:

```
variable U : Type
variables A B : U → Prop

example (h1 : ∀ x, A x → B x) (h2 : ∀ x, A x) : ∀ x, B x :=
assume y,
have h3 : A y, from h2 y,
have h4 : A y → B y, from h1 y,
show B y, from h4 h3
```

- A aplicação de h1 a y produz uma prova de A y → B y (h3), que então aplicamos a h2 y (h4), que é uma prova de A y(h4 h3). O resultado é a conclusão que estávamos atrás.

- No capítulo anterior, vimos a seguinte prova em dedução natural:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\forall x\ A(x)}\,1}{A(y)} \quad \dfrac{\dfrac{}{\forall x\ B(x)}\,2}{B(y)}}{A(y) \wedge B(y)}}{\forall y\ (A(y) \wedge B(y))}}{\forall x\ B(x) \to \forall y\ (A(y) \wedge B(y))}\,2}{\forall x\ A(x) \to (\forall x\ B(x) \to \forall y\ (A(y) \wedge B(y)))}\,1$$

- Aqui está ela, em LEAN:

```
variable U : Type
variables A B : U → Prop

example : (∀ x, A x) → (∀ x, B x) → (∀ x, A x ∧ B x) :=
assume hA : ∀ x, A x,
assume hB : ∀ x, B x,
assume y,
have Ay : A y, from hA y,
have By : B y, from hB y,
show A y ∧ B y, from and.intro Ay By
```

9.3. Utilizando o Quantificador Existencial

- Vamos começar falando sobre a regra de introdução do quantificador existencial. Devemos utilizar o comando exists.intro, que requer dois argumentos: um termo, e uma prova de que este termo satisfaz a propriedade requerida para se introduzir o quantificador existencial.

```
variable U : Type
variable P : U → Prop

example (y : U) (h : P y) : ∃ x, P x :=
exists.intro y h
```

- A prova de eliminação é dada por exists.elim. Se conhecemos ∃x, P x e estamos tentando provar Q, basta assumirmos que temos uma variável arbitrária y que satisfaz P y.

```
variable U : Type
variable P : U → Prop
variable Q : Prop

example (h1 : ∃ x, P x) (h2 : ∀ x, P x → Q) : Q :=
exists.elim h1
  (assume (y : U) (h : P y),
    have h3 : P y → Q, from h2 y,
    show Q, from h3 h)
```

- Um exemplo que utilizamos tanto a regra de introdução quanto a de eliminação:

```
variable U : Type
variables A B : U → Prop

example : (∃ x, A x ∧ B x) → ∃ x, A x :=
assume h1 : ∃ x, A x ∧ B x,
exists.elim h1
  (assume y (h2 : A y ∧ B y),
    have h3 : A y, from and.left h2,
    show ∃ x, A x, from exists.intro y h3)
```

- No último capítulo, vimos esta prova:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\forall x\ (A(x) \to \neg B(x))}\,1}{A(x) \to \neg B(x)} \quad \dfrac{\dfrac{}{A(x) \wedge B(x)}\,3}{A(x)}}{\neg B(x)} \quad \dfrac{\dfrac{}{A(x) \wedge B(x)}\,3}{B(x)}}{\bot}\,3}{\exists x\ (A(x) \wedge B(x))}\,2}{\bot}}{\neg\exists x\ (A(x) \wedge B(x))}\,2}{\forall x\ (A(x) \to \neg B(x)) \to \neg\exists x\ (A(x) \wedge B(x))}\,1$$

- Vamos reproduzí-la em LEAN:

```
variable U : Type
variables A B : U → Prop

example : (∀ x, A x → ¬ B x) → ¬ ∃ x, A x ∧ B x :=
assume h1 : ∀ x, A x → ¬ B x,
assume h2 : ∃ x, A x ∧ B x,
exists.elim h2 $
assume x (h3 : A x ∧ B x),
have h4 : A x, from and.left h3,
have h5 : B x, from and.right h3,
have h6 : ¬ B x, from h1 x h4,
show false, from h6 h5
```

- Fizemos a eliminação do quantificador existencial para introduzir A x ∧ B x para, assim, poder fazer a introdução da negação e concluir a prova.
- Observe também que utilizamos o $ ao invés de ( ).
- Vamos apresentar agora um exemplo mais peculiar sobre a eliminação do ∃.

```
variable U : Type
variable u : U
variable P : Prop

example : (∃x : U, P) ↔ P :=
iff.intro
  (assume h1 : ∃x, P,
    exists.elim h1 $
    assume x (h2 : P),
    h2)
  (assume h1 : P,
    exists.intro u h1)
```

- Não podemos construir esta prova sem declarar a variável u de tipo U, mesmo que u não seja utilizado diretamente na prova.
- Mais a frente, veremos que a semântica da lógica de primeira ordem pressupõe um universo não vazio. Entretanto, no LEAN, podemos ter tipos vazios. Assim, para montarmos nossa prova, precisamos povoar o universo.

9.4 - Provas de igualdade e cálculo

- No LEAN, as propriedades reflexiva, simétrica e transitiva são listadas da seguinte maneira:

```
variable A : Type

variables x y z : A
variable P : A → Prop

example : x = x :=
show x = x, from eq.refl x

example : y = x :=
have h : x = y, from sorry,
show y = x, from eq.symm h

example : x = z :=
have h1 : x = y, from sorry,
have h2 : y = z, from sorry,
show x = z, from eq.trans h1 h2

example : P y :=
have h1 : x = y, from sorry,
have h2 : P x, from sorry,
show P y, from eq.subst h1 h2
```

- A reflexão assume x como argumento, pois não precisamos de uma hipótese para fazer esta inferência. Todas as outras regras usam premissas como argumentos.
- Aqui está um exemplo onde usamos algumas destas relações:

```
variables (A : Type) (x y z : A)

example : y = x → y = z → x = z :=
assume h1 : y = x,
assume h2 : y = z,
have h3 : x = y, from eq.symm h1,
show x = z, from eq.trans h3 h2
```

- Uma outra maneira de se executar este cálculo é através do comando rewrite.

- Digitar *begin* e *end* em uma prova coloca o Lean no "modo tático", o que significa que ele “espera” receber uma lista de instruções. O comando rewrite então muda o objetivo.

```
example : y = x → y = z → x = z :=
assume h1 : y = x,
assume h2 : y = z,
show x = z,
  begin
    rewrite ←h1,
    apply h2
  end
```

- O primeiro comando muda o objetivo x = z para y = z; o sinal ← diz ao Lean para usar a equação na direção inversa. Depois disso, podemos concluir o objetivo aplicando h2.
- Uma alternativa é reescrever a meta usando h1 e h2, o que reduz a meta para x = x.

```
example : y = x → y = z → x = z :=
assume h1 : y = x,
assume h2 : y = z,
show x = z,
  begin
    rw ←h1,
    rw h2
  end
```

- Ao reduzirmos a prova para uma única tática, podemos usar o "show... By ..." ao invés do begin e end:

```
example : y = x → y = z → x = z :=
assume h1 : y = x,
assume h2 : y = z,
show x = z, by rw [←h1, h2]
```

- Na hora de construírmos provas matemáticas comuns, muita das vezes é normal fazermos contas da seguinte maneira:

$$\begin{aligned} t_1 &= t_2 \\ \ldots &= t_3 \\ \ldots &= t_4 \\ \ldots &= t_5. \end{aligned}$$

- LEAN nos permite modelar provas de cálculo da mesma maneira. Sempre que precisarmos provar uma equação, podemos utilizar o comando *calc*, seguido de uma sequência de igualdades e justicativas. Por exemplo:

- calc

  e1 = e2 : justification 1

  ... = e3 : justification 2

  ... = e4 : justification 3

  ... = e5 : justification 4

- Aqui está o exemplo anterior, utilizando o calc:

```
example : y = x → y = z → x = z :=
assume h1 : y = x,
assume h2 : y = z,
calc
    x = y : eq.symm h1
  ... = z : h2
```

- Aqui estão algumas das identidades básicas dos inteiros, que poderão ser usadas para facilitar nossos cálculos:

```
variables x y z : int

example : x + 0 = x :=
add_zero x

example : 0 + x = x :=
zero_add x
```

```
example : (x + y) + z = x + (y + z) :=
add_assoc x y z

example : x + y = y + x :=
add_comm x y

example : (x * y) * z = x * (y * z) :=
mul_assoc x y z

example : x * y = y * x :=
mul_comm x y

example : x * (y + z) = x * y + x * z :=
left_distrib x y z

example : (x + y) * z = x * z + y * z :=
right_distrib x y z
```

- Usando estes axiomas, aqui está o mesmo cálculo, novamente, feito como um teorema sobre os inteiros:

```
example (x y z : int) : (x + y) + z = (x + z) + y :=
calc
   (x + y) + z = x + (y + z) : add_assoc x y z
           ... = x + (z + y) : eq.subst (add_comm y z) rfl
           ... = (x + z) + y : eq.symm (add_assoc x z y)
```

- Poderíamos usar também o rewrite:

```
example (x y z : int) : (x + y) + z = (x + z) + y :=
calc
  (x + y) + z = x + (y + z) : by rw add_assoc
          ... = x + (z + y) : by rw [add_comm y z]
          ... = (x + z) + y : by rw add_assoc
```

- Ainda outra prova, sobre a multiplicação:

```
variables a b d c : int

example : (a + b) * (c + d) = a * c + b * c + a * d + b * d :=
calc
  (a + b) * (c + d) = (a + b) * c + (a + b) * d : by rw left_distrib
    ... = (a * c + b * c) + (a + b) * d         : by rw right_distrib
    ... = (a * c + b * c) + (a * d + b * d)     : by rw right_distrib
    ... = a * c + b * c + a * d + b * d         : by rw ←add_assoc
```