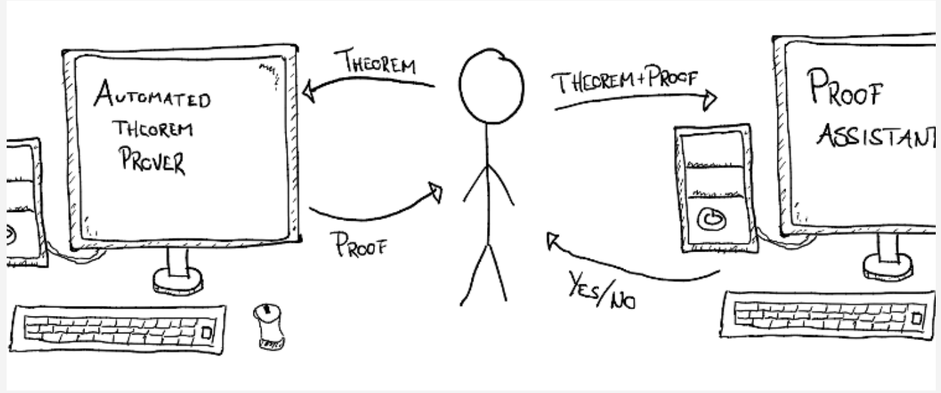
BIENAL DE MATEMÁTICA
2024

06/2024



PROVA INTERATIVA DE TEOREMAS

Éricles Lima

PRESENTED BY
your name

Introdução

Sobre o que vamos falar:

- Programas;
- Provas;
- Programas que são provas;
- Matemática;
- Computação.

Introdução

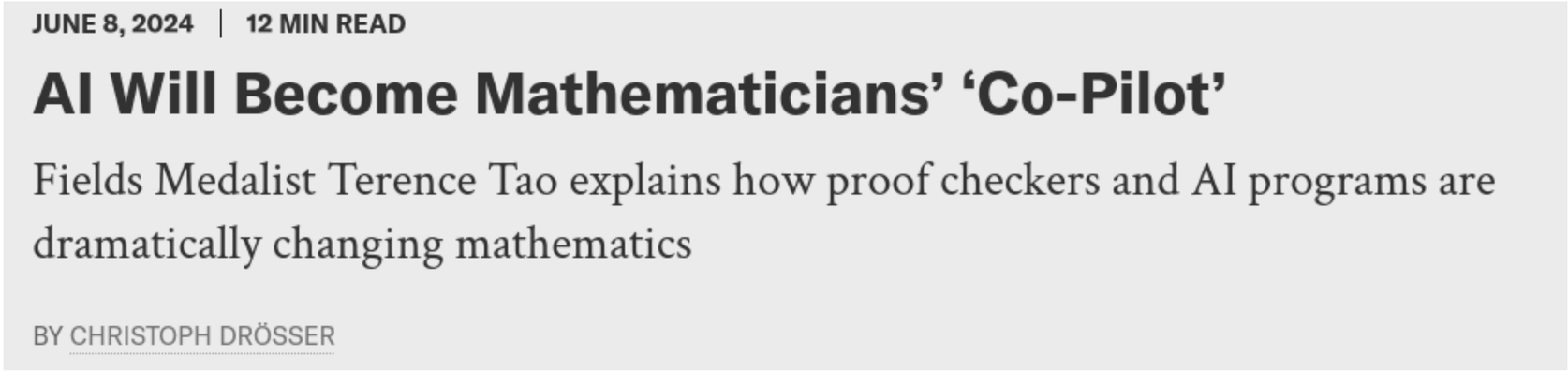
(o que) A ideia é

- aplicar técnicas de programação para provar asserções matemáticas,e
- aplicar técnicas matemáticas para provar asserções sobre programas.

(como) Para isso, vamos usar

- correspondência de Curry-Howard (aka. "isomorfismo de Curry-Howard"): um programa é secretamente uma prova;
- A linguagem de programação Lean 4.

Mas, antes... Por quê?



JUNE 8, 2024 | 12 MIN READ

AI Will Become Mathematicians' 'Co-Pilot'

Fields Medalist Terence Tao explains how proof checkers and AI programs are dramatically changing mathematics

BY CHRISTOPH DRÖSSER

Scientific American, sobre Lean 4

Mas, antes... Por quê?

Problema:

- Esse teorema em que estou trabalhando é muito complicado - é difícil ter 100% de certeza de que a prova está correta.

Solução:

- Faça o teorema como um programa automaticamente checável.

Mas, antes... Por quê?

Problema:

- É difícil colaborar em matemática - Se você não conhece e confia na pessoa, precisa checar o trabalho dela linha a linha.

Solução:

- Libere o teorema como um programa automaticamente checável.

Mas, antes... Por quê?

Problema:

- Estou confiante de que o resultado é correto, mas tem um passo ou outro que estão faltando.

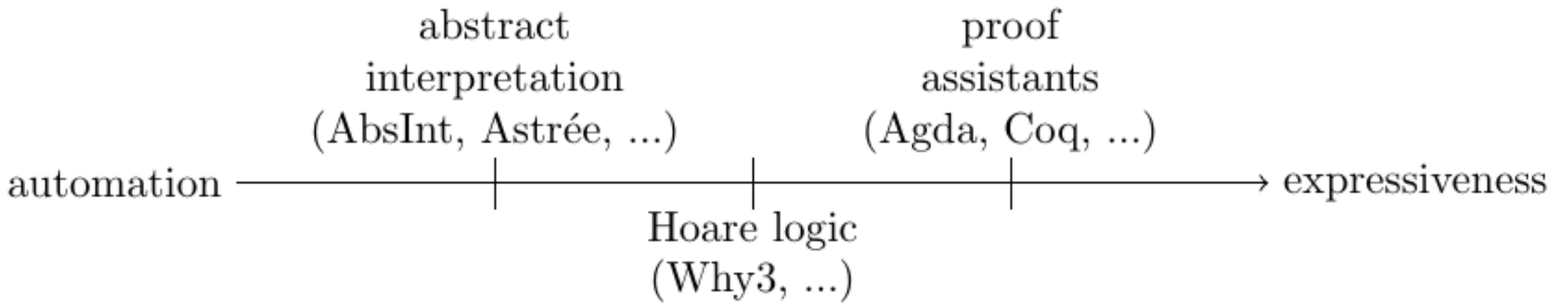
Solução:

- Faça o teorema em um provador de teoremas interativo e veja as dicas do programa - saiba exatamente quais passos faltam, o que precisa para preenhê-los, e tenha dicas sobre o que fazer.

Ok. Mas, por onde começar?

Foram dessenvolvidos diversos assistentes de provas para esses fins, como Coq, Agda e Lean.

Essa tecnologia é parte de uma família maior, de métodos formais, com vários graus de automação e expressividade. No geral, quão mais expressivo o sistema for, menos automatizado - MAS isso está mudando (mais sobre isso em slides a seguir)



Program=Proof, de Samuel Mimram

Clientes Satisfeitos

I now do my mathematics with a proof assistant and do not have to worry all the time about mistakes in my arguments or about how to convince others that my arguments are correct.

Vladmir Voevodsky, fields medalist

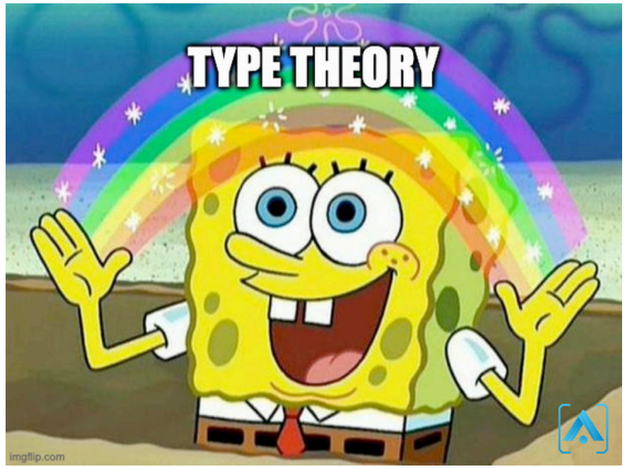
"This experiment has changed drastically my impression of how capable [proof assistants] are," Scholze said. "I now think it's sensible in principle to formalize whatever you want in Lean. There's no real obstruction."

Peter Scholze, fields medalist

Teoria dos tipos

Lean é baseado em **Tipos Dependentes**.

Mais especificamente, em uma versão de tipos dependentes conhecida como **Cálculo de Construções**, com uma hierarquia de "uniersos" e **tipos indutivos** (calma, você vai entender o que isso quer dizer).



Tipos Dependentes

Tipos dependentes expandem a teoria dos tipos mais simples.

Com **tipos dependentes,** tipos se tornam cidadãos de primeira classe.
Isso quer dizer que se pode operar sobre tipos da mesma forma que operaria com outros objetos quaisquer.

Em particular, isso quer dizer que cada tipo tem também o seu tipo, **Type**.

```
#check Nat                    -- Type
#check Bool                   -- Type
#check Nat → Bool             -- Type
#check Nat × Bool             -- Type
#check Nat → Nat              -- ...
#check Nat × Nat → Nat
```

Tipos Dependentes

Isso quer dizer que você pode declarar novas constantes do tipo Type:

```
def α : Type := Nat
def β : Type := Bool
def F : Type → Type := List
def G : Type → Type → Type := Prod
```

```
#check G α      -- Type → Type
#check G α β    -- Type
#check G α Nat  -- Type
```

Tipos Dependentes

Tipos dependentes são bem comuns em Lean.
Por exemplo, se **α** é do tipo **Type**, "**List α**" é do tipo **Type**.

Dado que todo elemento tem um tipo, é natural se perguntar: qual é o tipo de **Type**?

A resposta é: **Type** tem tipo **Type 1**, que tem tipo **Type 2,** que tem tipo...
Enfim, Lean tem uma hierarquia contável de tipos ~ a lá Russel

```
#check Type      -- Type 1
#check Type 1    -- Type 2
#check Type 2    -- Type 3
#check Type 3    -- Type 4
#check Type 4    -- Type 5
```

Tipos Dependentes

Mais em geral, um tipo é dito “*dependente*” se sua definição depende de um valor. O exemplo canônico é o de um vetor em $\underline{R^n}$ - ou, mais em geral, o tipo **Vector a n** - o vetor com **n** elementos de tipo **a** (eg., R).

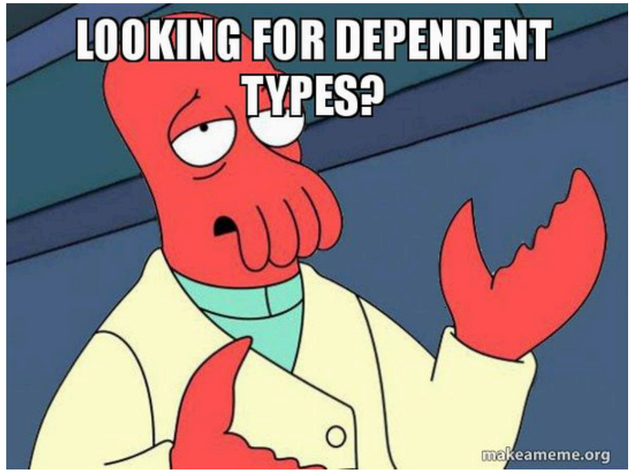
Esse tipo depende de dois parâmetros - **a** e **n**.

Tipos Dependentes

Tem dois tipos dependentes especialmente importantes:

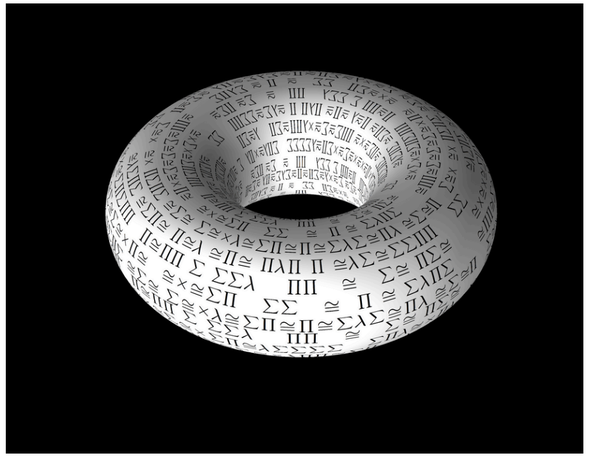
- o de produto generalizado (tipo Π)
- o de soma generalizada (tipo Σ)

obs.: “produto generalizado” e “soma generalizada” não são padrões, e ocasionalmente outros termos são usados no lugar. O “produto” e “soma” vêm de teoria de categoria.



Tipos Dependentes

Tipo de Produto generalizado (tipo Π)



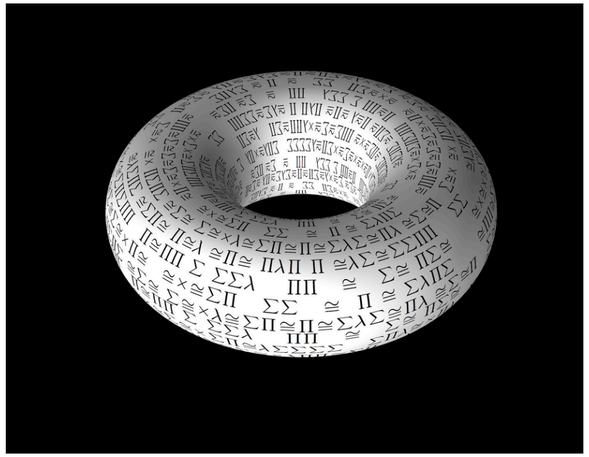
- tipo de funções dependentemente tipadas

Exemplo: o tipo **(a : α) → β a**, onde **α : Type** e **β : α → Type**
(ie., **α** é um tipo, **β** é uma função que para cada elemento a do tipo **α** retorna um novo tipo (**β a**)

Quando o valor de **β** depende de **a : α**, então **(a : α) → β** denota o tipo de *“funções dependentes”*.
*Q*uando o valor de **β** *não* depende de **a : α**, então **(a : α) → β** é o mesmo que o tipo (de função) **α → β**.

Tipos Dependentes

Tipo de Soma generalizada (tipo Σ)



Análogo ao exemplo anterior, mas para produtos:

- **(a : α) × β a** - generaliza a noção de produto certesiano

Também podem ser escritos como **Σ a : α, β a**.

A notação ⟨**a, b**⟩ cria um produto dependente (no vscode, use “\<” e “\>”)

Hierarquia de tipos

Tipos dependentes são bem comuns em Lean.
Por exemplo, se **α** é do tipo **Type**, "**List α**" é do tipo **Type**.

Dado que todo elemento tem um tipo, é natural se perguntar: qual é o tipo de **Type**?

A resposta é: **Type** tem tipo **Type 1**, que tem tipo **Type 2,** que tem tipo...
Enfim, Lean tem uma hierarquia contável de tipos ~ a lá Russel

```
#check Type     -- Type 1
#check Type 1   -- Type 2
#check Type 2   -- Type 3
#check Type 3   -- Type 4
#check Type 4   -- Type 5
```

Operações polimórficas

Algumas funções precisam ser polimorficas: o tipo **List a** deve fazer sentido independente de em qual o universo o tipo **a** vive.
Isso explica a sua assinatura:

```
#check List -- Type u_1 → Type u_1
```

Lean oferece o comando “universe” para explicitar as “variáveis de universo” (como são chamadas):

Operações polimórficas

Lean oferece o comando “universe” para explicitar as “variáveis de universo” (como são chamadas):

```
universe u
def F (α : Type u) : Type u := Prod α α
#check F     -- Type u_1 → Type u_1
```

O uso do comando “universe” pode ser evitado, se usar a sintaxe “F.{u}”:

```
def G.{u} (α : Type u) : Type u := Prod α α
#check G     -- Type u_1 → Type u_1
```

Funções (lambda)

(assim como em boa parte das linguagens no paradigmas de programação funcional) funções em Lean podem ser escritas como expressões lambda:

```
λ (x : Nat) => x + 5
```

Que é (definicionalmente) o mesmo que escrever:

```
fun (x : Nat) => x + 5
```

Muito do ferramental teórico e prático usado vem do cálculo lambda - em particular, a noção de *equivalências-α* (mais a seguir).

Propositions as types

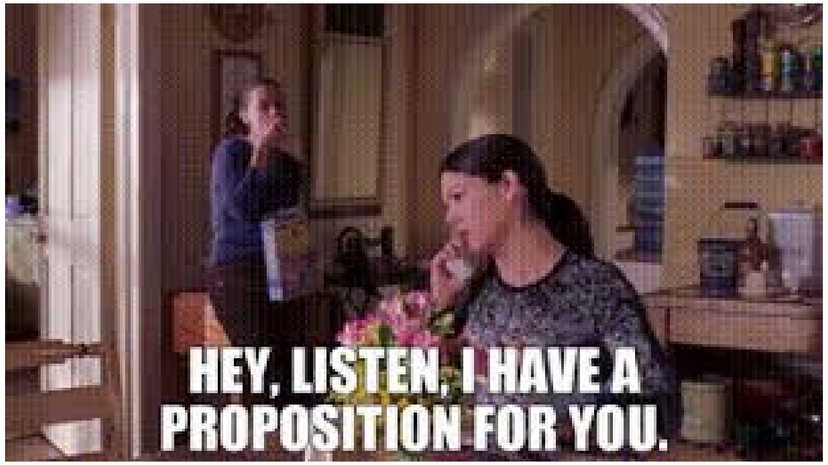
Encurtando a história:

- conectivos lógicos ($\wedge$, $\vee$, $\rightarrow$, etc.), assim como quantificadores ($\exists$ e $\forall$) são definidos por meio de **regras de introdução** e **regras de eliminação**;
- é de verificação direta essas regras de introdução e eliminação estão em correspondência direta com operações realizadas em tipos;

Propositions as types

Para tanto, fazemos:

- ***Prop***, o tipo para proposições (ie., *“p: Prop”* significa “p é do tipo Prop”, que significa “p é uma proposição”);
- Se **p: Prop**, p é também interpretado como um tipo - o tipo das provas de p. Isto é:
  - **t: p** é interpretado como a asserção de que “t é uma prova de p”

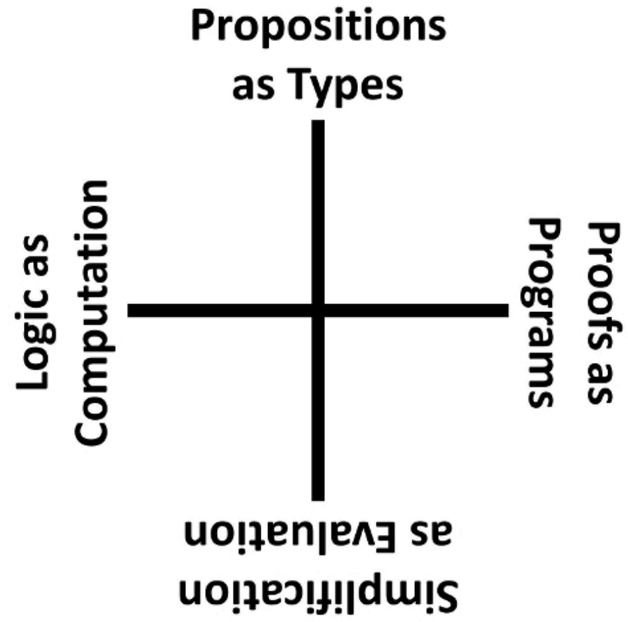


Propositions as types

Assim, os construtos usuais da linguagem de programação são usados como operadores lógicos para a construção de provas.

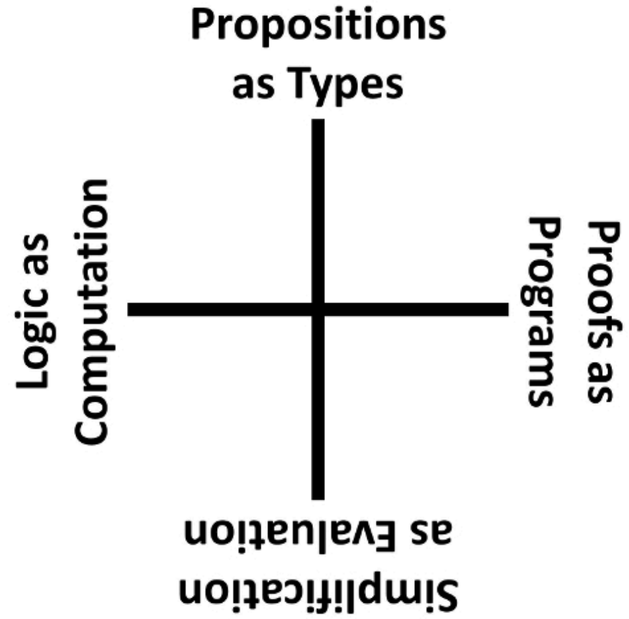
Por exemplo, o “→” de implicação é o mesmo “→” de “função”:

- dizer que “a implica b”, a→b, se reduz a *“uma função que, dada uma prova de a constrói uma prova de b”, a→b*.



Propositions as types

O fato de que tem uma correspondência exata entre as regras para implicação e as regras para abstração e aplicação de funções é uma instância da *“correspondência de Curry-Howard”*.



Propositions as types

- Para expressar formalmente uma afirmação matemática, precisamos exibir um termo **p: Prop** (um termo p, do tipo Prop);
- Para *provar* essa afirmação, precisamos exibir um terpo **t: p** (um termo t, do tipo p).
- A tarefa do Lean é ajudar a construir **t**, e a nos assegurar de que ele tem o tipo correto.

Propositions as types

Para introduzir um novo teorema, usamos o *“theorem”*:

```
variable {p : Prop}
variable {q : Prop}

theorem t1 : p → q → p := fun hp : p => fun hq : q => hp
```

Essa palavra-chave *“theorem”* na verdade se comporta da mesma forma que a *“def”* - provar o teorema *“p → q → p”* é o mesmo que definir um elemento desse tipo

Propositions as types

Lean vem com todos os conectivos lógicos definidos - ie., com suas operações de introdução e eliminação.

Contudo, sob o paradigma de “proposições-como-tipos”, todos esses conectivos são construtivos (“construtivo” como em “lógica construtiva”).

A lógica clássica usual adiciona o princípio do terceiro excluído (“*excluded middle*”, “*em*”). Para fazer uso disso, é preciso usar *“open Classical”*:

```
open Classical

variable (p : Prop)
#check em p -- em p : p ∨ ¬p
```

Propositions as types

Exemplo: eliminação de negação dupla (lógica clássica):

```
open Classical

example {p : Prop} (h : ¬¬p) : p :=
  Or.elim (em p)
    (fun hp : p => hp)
    (fun hnp : ¬p => absurd hnp h)
```

Propositions as types

Quantificadores

Encurtando a história:

- Lembra do produto dependente (ou função dependente)?

  O ∀ se comporta exatamente igual:

  ∀ x : α, p é uma notação diferente para **(x : α) → p**

Propositions as types

Quantificadores

Encurtando a história:

- Já o existencial, “∃”, se comporta de forma similar ao tipo Σ - neste caso, entretanto, são implementadas de forma ligeiramente diferente, por questões de tipagem de Prop que não vêm ao caso agora.
- o “∃” vem com suas regras de introdução de eliminação (Exists.elim, Exists.intro)

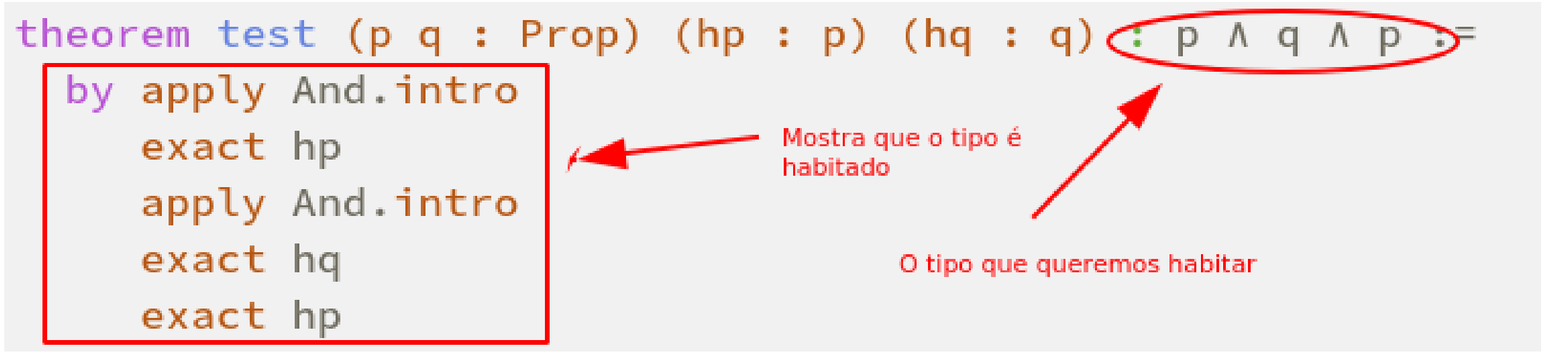
Propositions as types

Curry–Howard correspondence is pretty epic. Read about intuitionistic type theory

Logic side	Programming side
universal quantification	generalised product type (Π type)
existential quantification	generalised sum type (Σ type)
implication	function type
conjunction	product type
disjunction	sum type
true formula	unit type
false formula	bottom type

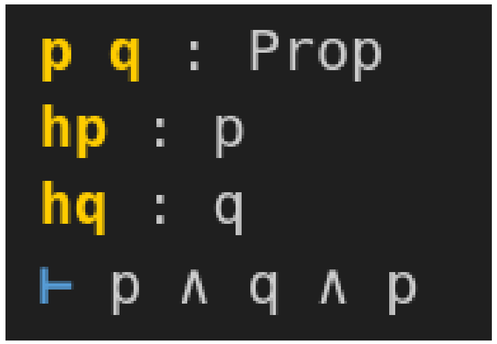
Tactics and tacticals

Um teorema em Lean (e em muitos outros ITPs) é basicamente um programa para habitar um tipo - mais especificamente, o tipo da proposição em questão.



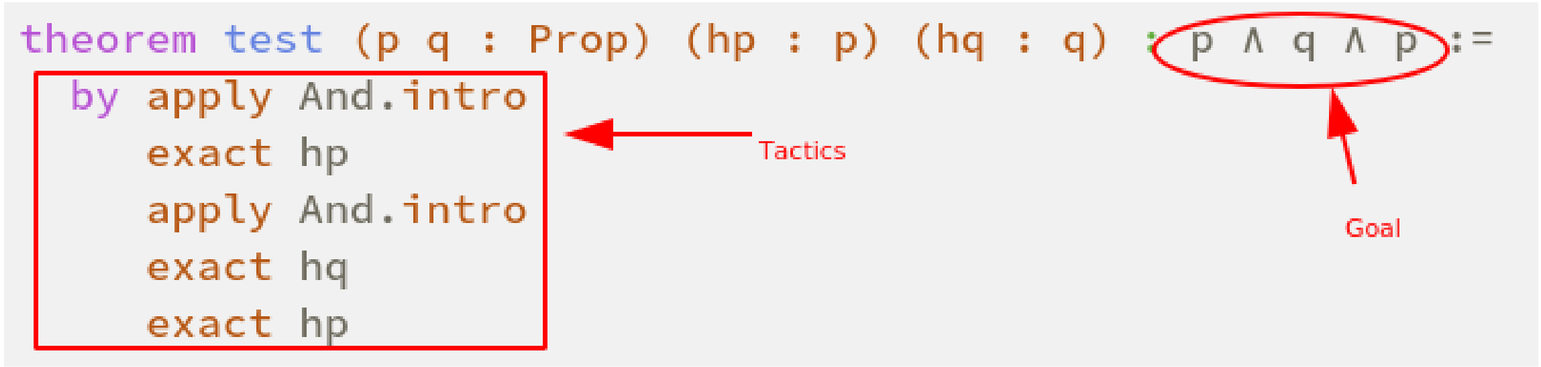
Tactics and tacticals

Trabalhando de forma interativa, isto é, escrevendo primeiro a asserção do teorema, trabalhamos com “*goals*”, ie., objetivos.
O *goal* inicial é dado pelo tipo do teorema (que é a proposição que queremos provar, na interpretação de *“propositions as types”*) no nosso exemplo,



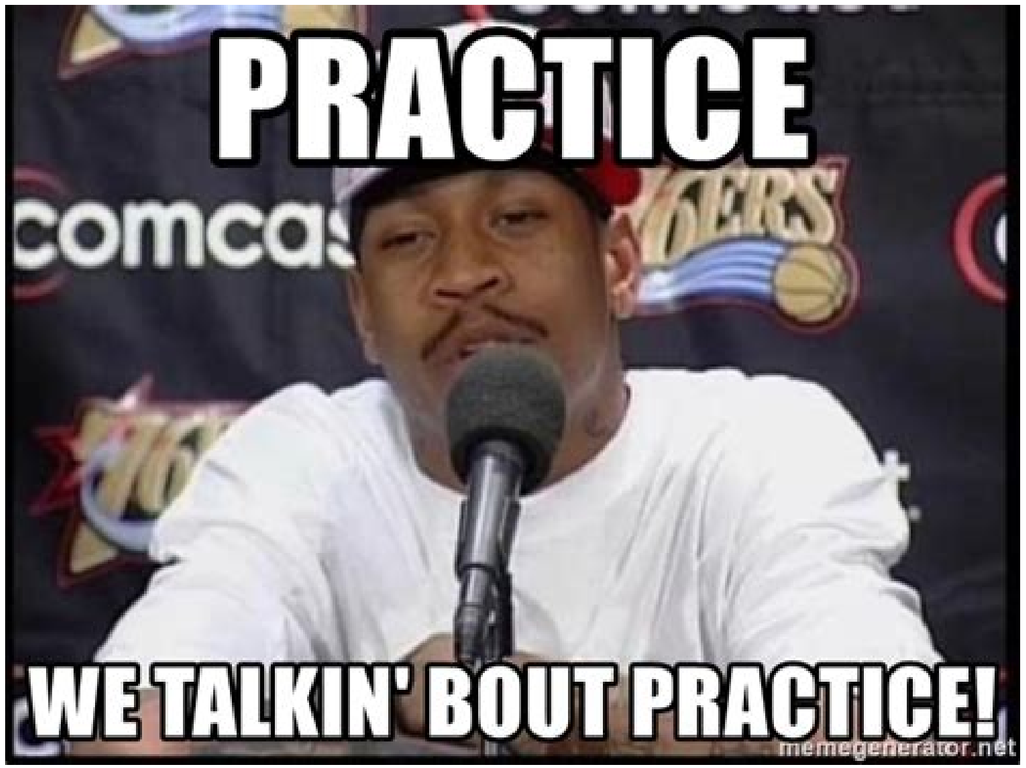
Tactics and tacticals

Podemos escrever uma prova na forma de *termos,* que vão construir um habitante para esse tipo;
alternativamente, o Lean (e outros ITPs) oferecem as (assim chamadas) ***“tactics”***.
Tactics são programas que manipulam o *goal* - o reescrevem, simplificam - e oferecem uma parcela de automatização na prova de teoremas.
A entrada no “modo tático” é indicada pelo termo-chave ***“by”***:

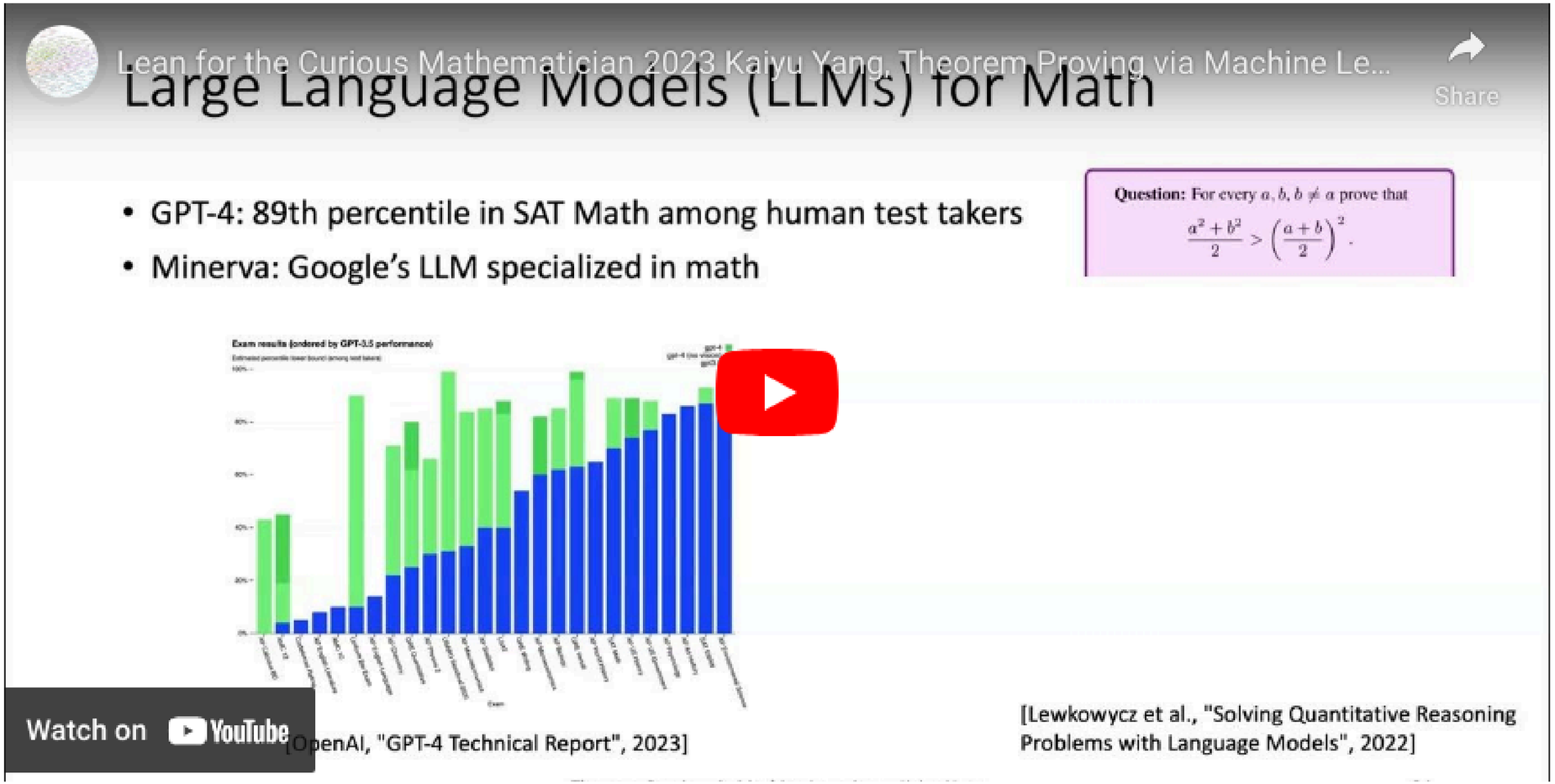


Prática

Entre no link: Mathematics in Lean



Próximos passos



Referências

Theorem proving in Lean 4 (intro to language)

The mechanics of proof (for early undergrads)

Mathematics in Lean (intro to math usage, with pointers)

Functional Programming in Lean (more in depth into the language)

Lean Manual

The Hitchhiker's Guide to Logical Verification (textbook for an msc level course)

Program=Proof (thourough material on the theory (up to HoTT). Uses Coq/Ocaml/Agda)

Natural number game (for fun)