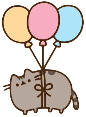
Proof Assistant - A 猫咪 Approach

喵喵

2022.3



自我介绍

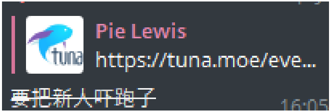
喵喵： 猫猫

Previously...

```
定理证明是一种形式化验证方法，它的目的是通过计算机程序来进行数学定理的证明。对于不同的公理系统，定理证明工具（Proof As
Assistant 入门方式：基于常见工程编程语言的类型系统，类比逻辑系统，在 <del>不加严格证明的情况下</del> 直觉引导下，尝

以下为喵喵的附言：

    喵喵，我们的 Proof Assistant 可以证明任何一个定理，但是我们不能证明任何一个定理是正确的或者错误的。
    喵喵，我们的 Proof Assistant 可以证明任何一个定理，但是我们不能证明任何一个定理是正确的或者错误的。
    喵喵，我们的 Proof Assistant 可以证明任何一个定理，但是我们不能证明任何一个定理是正确的或者错误的。
    喵喵，我们的 Proof Assistant 可以证明任何一个定理，但是我们不能证明任何一个定理是正确的或者错误的。
    喵喵，我们的 Proof Assistant 可以证明任何一个定理，但是我们不能证明任何一个定理是正确的或者错误的。
    喵喵，我们的 Proof Assistant 可以证明任何一个定理，但是我们不能证明任何一个定理是正确的或者错误的。
```

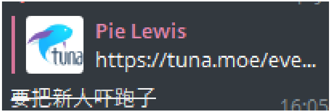
Previously...

```
定理证明是一种形式化验证方法，它的目的是通过计算机程序来进行数学定理的证明。对于不同的公理系统，定理证明工具（Proof As
Assistant 入门方式：基于常见工程编程语言的类型系统，类比逻辑系统，在 <del>不加严格证明的情况下</del> 直觉引导下，尝

以下为喵喵的附言：

    喵喵，我们的 Proof Assistant 可以证明任何一个定理，但是我们不能证明任何一个定理是正确的或者错误的。
    喵喵，我们的 Proof Assistant 可以证明任何一个定理，但是我们不能证明任何一个定理是正确的或者错误的。
    喵喵，我们的 Proof Assistant 可以证明任何一个定理，但是我们不能证明任何一个定理是正确的或者错误的。
    喵喵，我们的 Proof Assistant 可以证明任何一个定理，但是我们不能证明任何一个定理是正确的或者错误的。
    喵喵，我们的 Proof Assistant 可以证明任何一个定理，但是我们不能证明任何一个定理是正确的或者错误的。
    喵喵，我们的 Proof Assistant 可以证明任何一个定理，但是我们不能证明任何一个定理是正确的或者错误的。
```



Previously...

```
定理证明是一种形式化验证方法，它的目的是通过计算机程序来进行数学定理的证明。对于不同的公理系统，定理证明工具（Proof As
Assistant 入门方式：基于常见工程编程语言的类型系统，类比逻辑系统，在 <del>不加严格证明的情况下</del> 直觉引导下，尝

以下为喵喵的附言：

    喵喵，我们的 Proof Assistant 可以证明任何一个定理，但是我们不能证明任何一个定理是正确的或者错误的。
    喵喵，我们的 Proof Assistant 可以证明任何一个定理，但是我们不能证明任何一个定理是正确的或者错误的。
    喵喵，我们的 Proof Assistant 可以证明任何一个定理，但是我们不能证明任何一个定理是正确的或者错误的。
    喵喵，我们的 Proof Assistant 可以证明任何一个定理，但是我们不能证明任何一个定理是正确的或者错误的。
    喵喵，我们的 Proof Assistant 可以证明任何一个定理，但是我们不能证明任何一个定理是正确的或者错误的。
    喵喵，我们的 Proof Assistant 可以证明任何一个定理，但是我们不能证明任何一个定理是正确的或者错误的。
```



https://tuna.moe/event/2020/coq-proof/

Motivation

Motivation

图: 喵喵 embedded in $S^1 \times D^2$

Motivation (Cont.)

`https://github.com/caotic123/PomPom-Language`

Motivation (Cont.)

https://github.com/caotic123/PomPom-Language

能否在 1000 行内使用 Rust 实现一个 Proof Assistant?

Motivation (Cont.)

https://github.com/caotic123/PomPom-Language

能否在 1000 行内使用 Rust 实现一个 Proof Assistant?

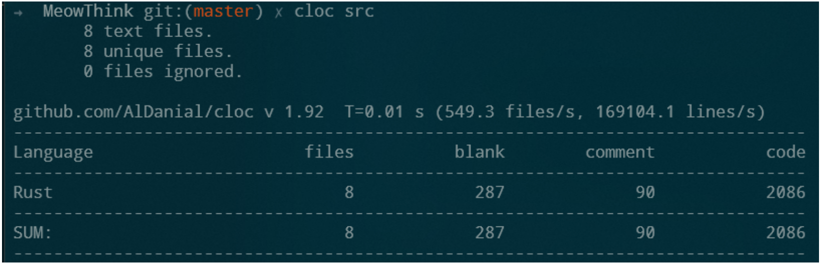


图: "2000 LOC can (sort of)"

https://github.com/CircuitCoder/MeowThink

第二部分

命题逻辑

直觉逻辑 - Why

直觉逻辑 - Why

中国民间数学家定理 (CFMT)

任意大于 2 的偶数，都可以表示成两个质数的和。

直觉逻辑 - Why

中国民间数学家定理 (CFMT)

任意大于 2 的偶数，都可以表示成两个质数的和。

”CFMT 和 ZFC 系统独立”

中国民间数学家定理 (CFMT)

任意大于 2 的偶数，都可以表示成两个质数的和。

”CFMT 和 ZFC 系统独立”

中国民间数学家定理 (CFMT)

任意大于 2 的偶数，都可以表示成两个质数的和。

"CFMT 和 ZFC 系统独立"

"CFMT 是哪个公理的结果？"

直觉逻辑 - Why

中国民间数学家定理 (CFMT)

任意大于 2 的偶数，都可以表示成两个质数的和。

"CFMT 和 ZFC 系统独立"

"CFMT 是哪个公理的结果？"

直觉逻辑 - How

经典逻辑：真值表是本质的。

枚举 P, Q 取值可以证明 $\overline{P \wedge Q} \to \overline{P} \vee \overline{Q}$

直觉逻辑 - How

经典逻辑：真值表是本质的。

枚举 P, Q 取值可以证明 $\overline{P \wedge Q} \to \overline{P} \vee \overline{Q}$

直觉逻辑：推理，也就是**蕴含** $(\to)$ 是本质的。

如果已知 P, 已知 $P \to Q$, 可以证明 Q

直觉逻辑 - How

经典逻辑：真值表是本质的。

枚举 P, Q 取值可以证明 $\overline{P \wedge Q} \to \overline{P} \vee \overline{Q}$

直觉逻辑：推理，也就是**蕴含** ($\to$) 是本质的。

如果已知 P, 已知 $P \to Q$, 可以证明 Q

$$\neg A := A \to \bot$$

$$\bot \to B$$

直觉逻辑 - How

经典逻辑：真值表是本质的。

$$\text{枚举 P, Q 取值可以证明 } \overline{P \wedge Q} \to \overline{P} \vee \overline{Q}$$

直觉逻辑：推理，也就是**蕴含** $(\to)$ 是本质的。

$$\text{如果已知 } P\text{, 已知 } P \to Q\text{, 可以证明 } Q$$

$$\neg A := A \to \bot$$

$$\bot \to B$$

在直觉逻辑中，**排中律**和**双重否定去除**不自然成立。

Curry-Howard 对应

$A \to B$

Curry-Howard 对应

$$A \to B$$

证明方法：在已知 A 的情况下，尝试证明 B。

Curry-Howard 对应

$$A \to B$$

证明方法：在已知 A 的情况下，尝试证明 B。

使用方法：如果拿到了 A 的证明，那么就证出来了 B。

Curry-Howard 对应

$$A \to B$$

证明方法：在已知 A 的情况下，尝试证明 B。

使用方法：如果拿到了 A 的证明，那么就证出来了 B。

```
impl Fn(A) -> B
```

Curry-Howard 对应

$$A \to B$$

证明方法：在已知 A 的情况下，尝试证明 B。

使用方法：如果拿到了 A 的证明，那么就证出来了 B。

```
impl Fn(A) -> B
```

实现方法：在有 A 的情况下，尝试给出类型为 B 的东西。

Curry-Howard 对应

$A \to B$

证明方法：在已知 A 的情况下，尝试证明 B。

使用方法：如果拿到了 A 的证明，那么就证出来了 B。

```
impl Fn(A) -> B
```

实现方法：在有 A 的情况下，尝试给出类型为 B 的东西。

使用方法：喂一个 A 类型的东西，吐出一个 B 类型的东西。

Curry-Howard 对应

$$A \to B$$

证明方法：在已知 A 的情况下，尝试证明 B。

使用方法：如果拿到了 A 的证明，那么就证出来了 B。

```
impl Fn(A) -> B
```

实现方法：在有 A 的情况下，尝试给出类型为 B 的东西。

使用方法：喂一个 A 类型的东西，吐出一个 B 类型的东西。

直觉逻辑的证明和程序存在一一对应的关系

We can prove stuff!

```
fn id<A>(a: A) -> A;

fn concat<A, B, C>(
  first: impl Fn(A) -> B,
  second: impl Fn(B) -> C
) -> impl Fn(A) -> C;
```

We can prove stuff!

```
fn id<A>(a: A) -> A;

fn concat<A, B, C>(
  first: impl Fn(A) -> B,
  second: impl Fn(B) -> C
) -> impl Fn(A) -> C;
```

$$A \to A$$

$$(A \to B) \to (B \to C) \to (A \to C)$$

We can prove stuff!

```
fn id<A>(a: A) -> A;

fn concat<A, B, C>(
  first: A -> B,
  second: B -> C
) -> (A -> C);
```

$$A \to A$$

$$(A \to B) \to (B \to C) \to (A \to C)$$

$\wedge, \vee, \top, \bot$

$\wedge, \vee, \top, \bot$

```
(A, B)
```

$\wedge, \vee, \top, \bot$

```
(A, B)
// Case study = match
enum Either<A, B> {
  Left(A),
  Right(B),
}
```

$\wedge, \vee, \top, \bot$

```
(A, B)
// Case study = match
enum Either<A, B> {
  Left(A),
  Right(B),
}
struct Trivial; // Or ()
```

```
(A, B)
// Case study = match
enum Either<A, B> {
  Left(A),
  Right(B),
}
struct Trivial; // Or ()
enum False {}; // Or !
```

We can prove (other) stuff

```
fn and_to_or<a, b>(and: (a, b)) -> Either<a, b> {
  Either::left(and.0)
}
```

$$a \land b \to a \lor b$$

We can prove (other) stuff

```
fn and_to_or<A, B>(and: A × B) -> (A + B) {
  Left(and.first)
}
```

$$a \wedge b \rightarrow a \vee b$$

We can prove (other) stuff!

LEM Irrefutable

$$\neg\neg(P \vee \neg P)$$

We can prove (other) stuff!

LEM Irrefutable

$$\neg\neg(P \vee \neg P)$$

```
fn lem_irrefutable<P>(
  lem_false: (P + (P -> !)) -> !
) -> ! {
  let p_to_false: P -> !
    = |p: P| lem_false(Left(p));
  lem_false(Right(p_to_false))
}
```

Wait a minute...

Wait a minute...

```rust
fn bad<T>(t: T) -> ! {
  loop { /* Meow */ }
}

fn worse<T>(t: T) -> ! {
  // Meow meow
  worse(t)
}
```

Wait a minute...

```rust
fn bad<T>(t: T) -> ! {
  loop { /* Meow */ }
}

fn worse<T>(t: T) -> ! {
  // Meow meow
  worse(t)
}
```

函数

在数学中为两不为空集的集合间的一种对应关系：
输入值集合中的每项元素皆能对应**唯一一项**输出值集合中的元素。

Totality checker

递归必须结束。

Totality checker

递归必须结束。

检查循环是否终止比较麻烦。

Totality checker

递归必须结束。

检查循环是否终止比较麻烦。
循环可以用递归替代 - 不允许循环。

第三部分

谓词逻辑

一阶逻辑？

我们如何表示谓词?

命题 Even

n 是自然数, $Even(n) \mapsto$ n 是个偶数

一阶逻辑？

我们如何表示谓词?

命题 Even

n 是自然数, $Even(n) \mapsto$ n 是个偶数

$Even:$ 自然数 $\rightarrow$ 命题

一阶逻辑？

我们如何表示谓词?

命题 Even

$$n \text{ 是自然数}, Even(n) \mapsto n \text{ 是个偶数}$$

$$Even : \text{自然数} \to \text{命题}$$

```
P: Nat -> Type
```

Data -> Type??

Data -> Type??

Type -> Type: 泛型 (Generic)

Data -> Type??

`Type -> Type`: 泛型 (Generic)
`Constant -> Type`: Const Generic

Data -> Type??

Type -> Type: 泛型 (Generic)
Constant -> Type: Const Generic
Data -> Type: Dependent Typing

Dependent Typing

允许:

Dependent Typing

允许:

- Dependent Function: `(a: A) -> P(a)`

Dependent Typing

允许:

- Dependent Function: `(a: A) -> P(a)`
- Dependent Pair: `(a: A) × P(a)`

Dependent Typing

允许:

- Dependent Function: `(a: A) -> P(a)`
- Dependent Pair: `(a: A) × P(a)`

```
let dep_fun: (a: A) -> P(a);
dep_fun(some_a): P(some_a);

let dep_pair: (a: A) × P(a);
dep_pair.second: P(dep_pair.first);
```

Dependent Typing

允许:

- Dependent Function: `(a: A) -> P(a)`
- Dependent Pair: `(a: A) × P(a)`

```
let dep_fun: (a: A) -> P(a);
dep_fun(some_a): P(some_a);

let dep_pair: (a: A) × P(a);
dep_pair.second: P(dep_pair.first);
```

量词: $\forall, \exists$

类型表示

```
fn id<A>(a: A) -> A;
```

类型表示

```
fn id<A>(a: A) -> A;
```

$\Downarrow$

```
id<A>: A -> A
```

类型表示

```
fn id<A>(a: A) -> A;
```

$\Downarrow$

```
id<A>: A -> A
```

$\Downarrow$

```
id: (A: Type) -> A -> A
```

类型表示

```
fn id<A>(a: A) -> A;
```

$\Downarrow$

```
id<A>: A -> A
```

$\Downarrow$

```
id: (A: Type) -> A -> A
```

$\Downarrow$

```
fn id(A: Type, a: A) -> A;
```

We can prove (more) stuff!

Axiom of Choice...?

如果对于一组集合 S(i) 中的每一个，都可以选出来一个元素 s 满足 P(i, s)，那么存在一个选择函数，从每个 S(i) 中选出一个，使得每个选择的结果都满足谓词 P

We can prove (more) stuff!

Axiom of Choice...?

如果对于一组集合 S(i) 中的每一个，都可以选出来一个元素 s 满足 P(i, s)，那么存在一个选择函数，从每个 S(i) 中选出一个，使得每个选择的结果都满足谓词 P

$$
\begin{aligned}
&\forall i \in I, \exists s \in S(i), P(i, s) \\
&\rightarrow \\
&\exists f : ((i : I) \rightarrow S(i)) \\
&\forall i \in I, P(i, f(i))
\end{aligned}
$$

We can prove (more) stuff!

```
fn not_exactly_choice(
  Index: Type,
  Set: Index -> Type,
  Pred: Index -> Set -> Type,

  all_has_elem: (
    (i: Index)
    -> ((s: Set(i)) × Pred(i, s))
  )
) -> (
  (f: (i: Index) -> Set(i))
  × ((any: Index) -> Pred(any, f(any)))
);
```

相等

相等

图: `m ==(T) n`

相等

$$\forall n : Nat, \forall m : Nat, (n \equiv m) \to (2n \equiv 2m)$$

相等

$$\forall n : Nat, \forall m : Nat, (n \equiv m) \to (2n \equiv 2m)$$

```
(n: Nat) -> (m: Nat)
 -> (n ==(Nat) m)
 -> (double(n) ==(Nat) double(m))
```

相等的行为

```
==(_): (T: Type) -> (lhs: T) -> (rhs: T) -> Type
```

相等的行为

```
==(_): (T: Type) -> (lhs: T) -> (rhs: T) -> Type
```

- 自反性
  ```
  refl: (T: Type) -> (x: T) -> (x ==(T) x)
  ```

相等的行为

```
==(_): (T: Type) -> (lhs: T) -> (rhs: T) -> Type
```

- 自反性
  ```
  refl: (T: Type) -> (x: T) -> (x ==(T) x)
  ```
- 类型转换
  ```
  cast: (T: Type) -> (U: Type) -> (T ==(Type) U) ->
  T -> U
  ```

相等的行为

```
==(_): (T: Type) -> (lhs: T) -> (rhs: T) -> Type
```

- 自反性
  ```
  refl: (T: Type) -> (x: T) -> (x ==(T) x)
  ```
- 类型转换
  ```
  cast: (T: Type) -> (U: Type) -> (T ==(Type) U) ->
  T -> U
  ```
- 映射后不变
  ```
  ap: (f: T -> U) -> (a ==(T) b) -> (f(a) ==(U)
  f(b))
  ```

对称性 / 传递性在哪里？

对称性 / 传递性在哪里？

```
fn eq_sym(
  T: Type,
  a: T, b: T,
  eq: a ==(T) b,
) -> b ==(T) a {
```

对称性 / 传递性在哪里？

```
fn eq_sym(
  T: Type,
  a: T, b: T,
  eq: a ==(T) b,
) -> b ==(T) a {
  let start: (a ==(T) a) = refl(T, a);
```

对称性 / 传递性在哪里？

```
fn eq_sym(
  T: Type,
  a: T, b: T,
  eq: a ==(T) b,
) -> b ==(T) a {
  let start: (a ==(T) a) = refl(T, a);
  let ty_eq: (a ==(T) a) ==(Type) (b ==(T) a)
    = ap(|x: T| { x ==(T) a}, eq);
```

对称性 / 传递性在哪里？

```
fn eq_sym(
  T: Type,
  a: T, b: T,
  eq: a ==(T) b,
) -> b ==(T) a {
  let start: (a ==(T) a) = refl(T, a);
  let ty_eq: (a ==(T) a) ==(Type) (b ==(T) a)
    = ap(|x: T| { x ==(T) a}, eq);
  cast(/* ... */, ty_eq, start)
}
```

We can prove (even more) stuff

```
enum C3 {
  Zero, One, Two,
}
// Define mul: C3 -> C3 -> C3

fn C3_order_3(
  input: C3
) -> mul(mul(input, input), input) ==(C3) C3::Zero;
```

第四部分

归纳

归纳定义 / 证明

回忆自然数的定义:

- 0 是自然数。
- 如果 n 是自然数，那么 n 的后继是自然数。

归纳定义 / 证明

回忆自然数的定义：

- 0 是自然数。
- 如果 n 是自然数，那么 n 的后继是自然数。

```
enum Nat {
  Zero,
  Succ(Nat),
}
```

使用 Nat 的函数

```
fn add(a: Nat, b: Nat) -> Nat {
  match a {
    Nat::Zero => b,
    Nat::Succ(prev) => Nat::Succ(add(prev, b)),
  }
}
```

使用 Nat 的函数

```
fn add(a: Nat, b: Nat) -> Nat {
  match a {
    Nat::Zero => b,
    Nat::Succ(prev) => Nat::Succ(add(prev, b)),
  }
}
```

递归 = 归纳定义

使用 Nat 的函数

```
fn add_zero(a: Nat)
  -> (add(a, Nat::Zero) == a) {
    match a {
      Nat::Zero => refl(Nat, Nat::Zero),
      Nat::Succ(prev) => ap(Nat::Succ, add_zero(prev)),
    }
  }
```

使用 Nat 的函数

```
fn add_zero(a: Nat)
  -> (add(a, Nat::Zero) == a) {
    match a {
      Nat::Zero => refl(Nat, Nat::Zero),
      Nat::Succ(prev) => ap(Nat::Succ, add_zero(prev)),
    }
  }
```

递归 = 归纳证明

第五部分

公理

公理

" 存在一个类型是... 的项"

公理

" 存在一个类型是… 的项"
不一定会导致不可计算性。

排中律

- `lem: (P + !P)`
- `dne: !!P -> P`
- `peirce: ((P -> Q) -> P) -> P`
- `implies: (P -> Q) -> (Q + !P)`
- `de_morgan: !(!P × !Q) -> P + Q`

外延性

$$\{2x|x \in \mathbb{Z}\} =?\{2x + 2|x \in \mathbb{Z}\}$$

外延性

$\{2x|x \in \mathbb{Z}\} =? \{2x+2|x \in \mathbb{Z}\}$
简单算法 $=?$ 快速幂

外延性

$\{2x|x \in \mathbb{Z}\} =? \{2x+2|x \in \mathbb{Z}\}$
简单算法 $=?$ 快速幂

- 命题外延性: `(P -> Q) × (Q -> P) -> P = Q`

外延性

$\{2x|x \in \mathbb{Z}\} =?\{2x + 2|x \in \mathbb{Z}\}$
简单算法 $=?$ 快速幂

- 命题外延性: `(P -> Q) × (Q -> P) -> P = Q`
- 函数外延性: `(a -> f(a) == g(a)) -> f == g`

外延性

$\{2x|x \in \mathbb{Z}\} =? \{2x+2|x \in \mathbb{Z}\}$
简单算法 $=?$ 快速幂

- 命题外延性: `(P -> Q) × (Q -> P) -> P = Q`
- 函数外延性: `(a -> f(a) == g(a)) -> f == g`
- Univalence(类型外延性): $(A \simeq B) \simeq (A = B)$

第六部分

碎碎念

What is Type

What is Type

- `Type: Type`

What is `Type`

- `Type: Type`
  "The set of all sets"

What is Type

- `Type: Type`
  "The set of all sets"
  非直谓性 (Impredicativity) & `Girard's paradox`
- `Type` 是特殊的
  "The **class** of all sets"

What is Type

- `Type: Type`
  "The set of all sets"
  非直谓性 (Impredicativity) & `Girard's paradox`
- `Type` 是特殊的
  "The **class** of all sets"
  "`List<Type>`"

What is Type

- `Type: Type`
  "The set of all sets"
  非直谓性 (Impredicativity) & `Girard's paradox`
- `Type` 是特殊的
  "The **class** of all sets"
  "`List<Type>`"
- `Type 0` $\in$ `Type 1` $\in$ `Type 2` ...

Axiom K & HoTT

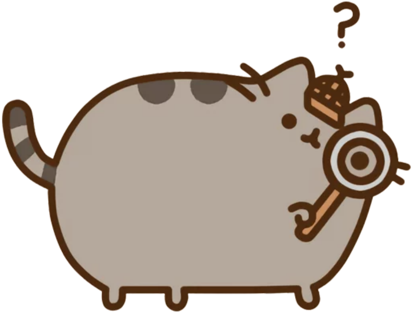
```
_ ==(a ==(T) b) _
```

```
_ ==(a ==(T) b) _
```

图: HoTT

类型 = 拓扑空间，相等 = 路径，ap = 函数都是连续的。

That's All!



Question time!

https://github.com/CircuitCoder/MeowThink
https://github.com/CircuitCoder/LocalNeko