软件理论基础与实践

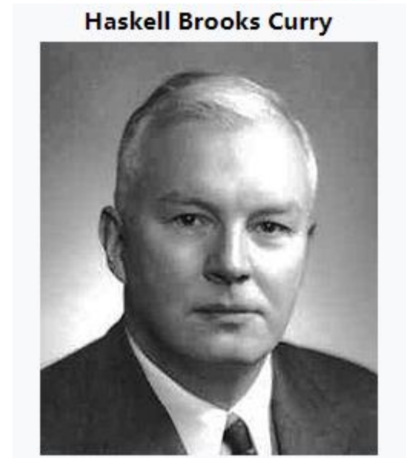
ProofObjects: The Curry-Howard Correspondence

熊英飞
北京大学

动机

- 很多策略做多种看似不同的事情
  - destruct
    - 分解归纳定义
    - 分解逻辑与
    - 分解逻辑或
    - 证明前提有False的定理
    - 分离存在量词
- 一些书写定理的符号和函数定义完全相同
  - 如：->，forall
- 背后的原因是什么？

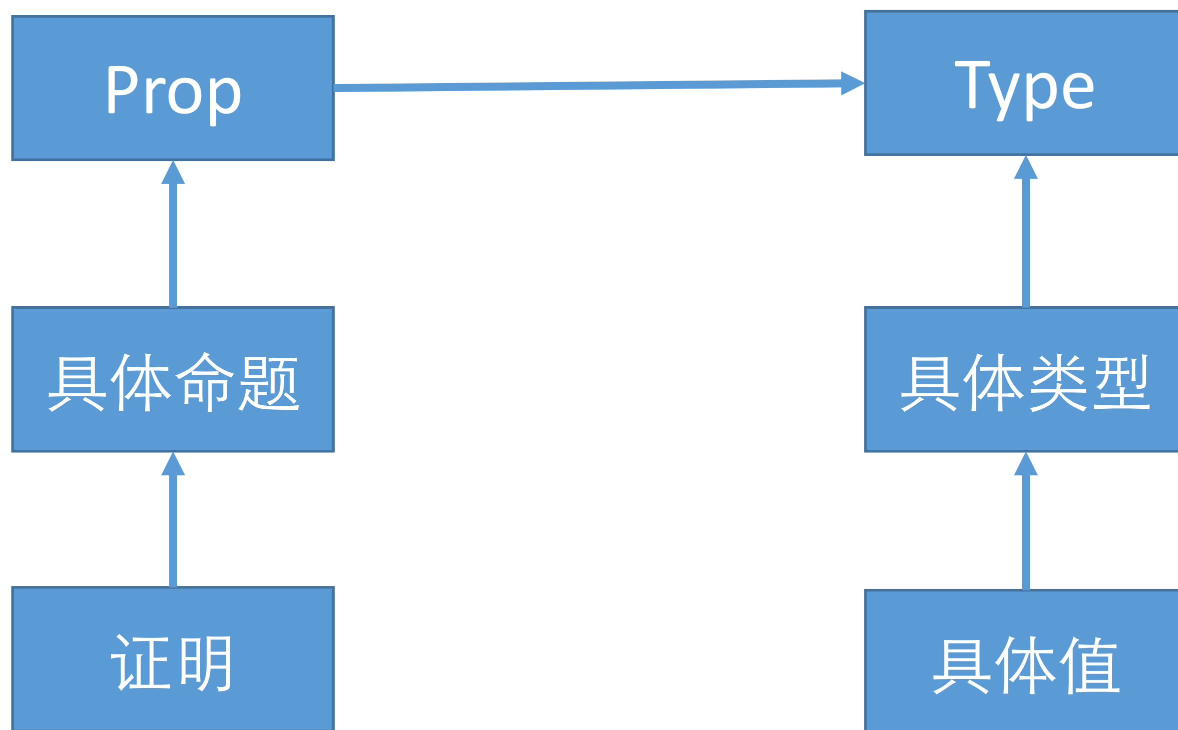
Curry-Howard Correspondence



- 由Haskell Brooks Curry和William Alvin Howard在1934-1969年间三个主要发现构成
- 命题 ⇔ 类型
  - A->B ⇔ A->B
- 证明 ⇔ 值

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
Hilbert-style deduction system	type system for combinatory logic
natural deduction	type system for lambda calculus

Curry-Howard Correspondence in Coq



箭头表示“S的类型是T”

Coq中的类型定义

```
Inductive bool : Type :=
  | true
  | false.
```

Coq中的命题定义

```
Inductive bool : Type :=
  | true
  | false.
```

```
Inductive True : Prop :=
  | I : True.
```

定义类型的值

```
Inductive bool : Type :=
  | true
  | false.
```

```
Coq < Compute true.
     = true
     : bool
```

定义命题的证明

```
Inductive bool : Type :=
  | true
  | false.
```

```
Coq < Compute true.
     = true
     : bool
```

```
Inductive True : Prop :=
  | I : True.
```

```
Coq < Compute I.
     = I
     : True
```

Tactic: 生成证明的命令

```
Lemma True_is_true : True.
Proof.
   apply I.
   Show Proof.
   (* I *)
Qed.
```

```
Print True_is_true.
(* True_is_true = I : True *)
```

```
Definition True_is_true := I.
```

定理=命题+证明

Tactic: 生成程序的命令

```
Definition FalseTerm : bool.
apply false.
Defined.
```

```
Print FalseTerm.
(* FalseTerm = false : bool *)
```

```
Definition FalseTerm := false.
```

Definition是Lemma, Theorem, Example等的别名。
Defined是Qed的别名。

带参数的归纳类型定义

```
Inductive list (X:Type) : Type :=
  | nil
  | cons (x : X) (l : list X).
```

```
Inductive nnlist : bool -> Type :=
| nnnil : nnlist false
| nncons {b:bool} (x : nat) (l : nnlist b) : nnlist true.

Definition fst(l:nnlist true) :=
  match l with
  | nncons x l => x
  end.
Fail Compute (fst nnnil).
(* The term "nnnil" has type "nnlist false" while it is
expected to have type "nnlist true". *)
```

依赖类型，参数表示列表非空

复习：首行冒号左边的参数叫做parameter，用于所有的constructor
首行冒号右边的参数叫做index或者annotation，可由具体的constructor填充

带参数的归纳命题定义

```
Inductive ev : nat -> Prop :=
| ev_0 : ev 0
| ev_SS (n : nat) (H : ev n) : ev (S (S n)).

Check ev_0 : ev 0.
Check ev_SS : forall n, ev n -> ev (S (S n)).
```

```
Theorem ev_4 : ev 4.
Proof.
  apply ev_SS. Show Proof.
  (* (ev_SS 2 ?Goal) *)
  apply ev_SS. Show Proof.
  (* (ev_SS 2 (ev_SS 0 ?Goal)) *)
  apply ev_0. Show Proof.
  (* (ev_SS 2 (ev_SS 0 ev_0)) *)
Qed.
```

```
Definition ev_4 := ev_SS 2 (ev_SS 0 ev_0).
```

apply: 生成对应函数调用，将输入参数标记为?Goal
?Goal的类型为当前的证明目标

回顾常见逻辑连接符和相关策略

量词、蕴含、多态和函数

```
Theorem ev_plus4 : forall n, ev n -> ev (4 + n).
Proof.
  intros n H.
  (* (fun (n : nat) (H : ev n) => ?Goal) *)
  apply ev_SS.
  apply ev_SS.
  apply H.
Qed.
```

```
Definition ev_plus4' : forall n, ev n -> ev (4 + n) :=
  fun (n : nat) => fun (H : ev n) =>
    ev_SS (S (S n)) (ev_SS n H).
```

intros: 生成函数声明
?Goal的上下文中能访问到的参数就是目前可用的假设

generalize dependent

- 如果采用generalize dependent会发生什么？

```
Lemma commutativity_test: forall n m, n+m=m+n.
Proof.
  intros.
  (* (fun n m : nat => ?Goal) *)
  generalize dependent n.
  (* (fun n m : nat => ?Goal n) *)
```

Coq证明构造过程会不断精化?Goal，但不会删除已经生成的代码

等价和自反性

```
Inductive eq {X:Type} (x:X) : X -> Prop :=
| eq_refl : eq x x.

Notation "x = y" := (eq x y)
                       (at level 70, no associativity)
                     : type_scope.
```

```
Lemma four: 2 + 2 = 1 + 3.
Proof.
  reflexivity.
Qed.
```

```
Definition four' : 2 + 2 = 1 + 3 :=
  eq_refl 4.
```

reflexivity: 等价于apply eq_refl

用destruct代替rewrite

```
Theorem eq_add :
  forall (n1 n2 : nat), n1 = n2 -> (S n1) = (S n2).
Proof.
  intros n1 n2 Heq.
  (* n1, n2: nat
     Heq: n1 = n2
     =============
     S n1 = S n2 *)
  destruct Heq.
  (* n1: nat
     =============
     S n1 = S n1 *)
  apply eq_refl.
Qed.
```

逻辑与：定义

```
Inductive and (P Q : Prop) : Prop :=
| conj : P -> Q -> and P Q.

Arguments conj [P] [Q].

Notation "P /\ Q" := (and P Q) : type_scope.
```

```
Inductive prod (X Y : Type) : Type :=
| pair (x : X) (y : Y).

Arguments pair {X} {Y} _ _.

Notation "( x , y )" := (pair x y).

Notation "X * Y" := (prod X Y) : type_scope.
```

逻辑与：split

```
Lemma and_intro' : forall A B : Prop, A -> B -> A /\ B.
Proof.
  intros A B HA HB. split. Show Proof.
  (* (fun (A B : Prop) (HA : A) (HB : B) => conj ?Goal ?Goal0) *)
  - apply HA.
  - apply HB.
  Show Proof.
  (* (fun (A B : Prop) (HA : A) (HB : B) => conj HA HB) *)
Qed.
```

split: 如果目标只有一个constructor，生成该constructor，并将参数类型定义为目标

逻辑与：split

```
Lemma and_intro' : forall A B : Prop, A -> B -> A /\ B.
Proof.
  intros A B HA HB. apply conj. Show Proof.
  (* (fun (A B : Prop) (HA : A) (HB : B) => conj ?Goal ?Goal0) *)
  - apply HA.
  - apply HB.
  Show Proof.
  (* (fun (A B : Prop) (HA : A) (HB : B) => conj HA HB) *)
Qed.
```

split等价于apply constructor

逻辑与：split

```
Lemma truth : True.
Proof.
  split.
Qed.
```

split也不一定会产生新的目标

逻辑与： destruct

```
Theorem proj1' : forall P Q,
    P /\ Q -> P.
Proof.
  intros P Q HPQ. Show Proof.
  (* (fun (P Q : Prop) (HPQ : P /\ Q) => ?Goal) *)
  destruct HPQ as [HP HQ]. Show Proof.
  (* (fun (P Q : Prop) (HPQ : P /\ Q) => match HPQ with
                                      | conj HP HQ => ?Goal
                                        end) *)
  apply HP.
Qed.
```

destruct: 根据参数的归纳定义生成match

逻辑与: destruct

destruct有可能形成多个分支

```
Definition somefun: nat->bool.
intros H.
destruct H.
- apply true.
- apply false.
Defined.
```

以上代码定义了什么?

```
Definition iszero :=
fun H : nat => match H with
               | 0 => true
               | S _ => false
               end.
```

False

```
Inductive False : Prop := .
```

没有Constructor所以永远构造不出False的证明

```
Definition false_implies_zero_eq_one : False -> 0 = 1.
Proof.
  intros.
  destruct H.
Qed.
```

```
Definition false_implies_zero_eq_one : False -> 0 = 1 :=
  fun contra => match contra with end.
```

没有分支的match表达式具有任意类型

逻辑或：定义

```
Inductive or (P Q : Prop) : Prop :=
| or_introl : P -> or P Q
| or_intror : Q -> or P Q.

Arguments or_introl [P] [Q].
Arguments or_intror [P] [Q].

Notation "P \/ Q" := (or P Q) : type_scope.
```

逻辑或： left, right

```
Theorem inj_l' : forall (P Q : Prop), P -> P \/ Q.
Proof.
  intros P Q HP. left. apply HP.
Qed.
```

```
Definition inj_l : forall (P Q : Prop), P -> P \/ Q :=
  fun P Q HP => or_introl HP.
```

left： 等价于apply or_introl
right： 等价于apply or_intror

存在量词：定义

```
Inductive ex {A : Type} (P : A -> Prop) : Prop :=
| ex_intro : forall x : A, P x -> ex P.

Notation "'exists' x , p" :=
  (ex (fun x => p))
    (at level 200, right associativity) : type_scope.
```

P: 给定一个值，构造一个命题（即存在量词的body）
P x: 对于某个具体x值的证明

存在量词：exists策略

```
Theorem some_nat_is_even : exists n, ev n.
Proof.
  exists 0.
  apply ev_0.
Qed.
```

```
Definition some_nat_is_even' : exists n, ev n :=
  ex_intro ev 0 ev_0.
```

exists：根据当前目标构造ex_intro

存在量词：exists策略

```
Theorem some_nat_is_even : exists n, ev n.
Proof.
  apply ex_intro with (x:=0).
  apply ev_0.
Qed.
```

```
Definition some_nat_is_even' : exists n, ev n :=
  ex_intro ev 0 ev_0.
```

exists n等价于apply ex_intro with (x:=n)
apply with：为apply应用过程中所需要的类型参数提供值

逻辑非:定义

复习

```
Definition not (P:Prop) := P -> False.

Notation "~ x" := (not x) : type_scope.
```

小结

- 除了forall和->，其他逻辑运算符都是inductive或普通definition
- 当inductive defintion出现在前提中的时候
  - 采用destruct分解
- 当inductive definition出现在结论中的时候
  - 采用split, left, right, exists分解
  - 或者统一采用apply xxx_constructor分解
  - 记不住构造函数的名字也记不住策略名字怎么办?

Constructor策略

- constructor调用第一个类型符合的构造函数

```
Lemma and_intro' : forall A B :
Prop, A -> B -> A /\ B.
Proof.
  intros A B HA HB.
  constructor. (* 代替split *)
  - apply HA.
  - apply HB.
Qed.
```

Constructor策略

- constructor i调用第i个构造函数

```
Lemma or_intro_r : forall A B :
Prop, B -> A \/ B.
Proof.
  intros A B HB.
  constructor 2. (* 代替right *)
  apply HB.
Qed.
```

Constructor策略

- constructor i with给forall后面的参数赋值

```
Lemma four_is_Even' : Even 4.
Proof.
  unfold Even.
  constructor 1 with (x:=2).
  (* 代替exists 2. *)
  reflexivity.
Qed.
```

在内部实现中，split, exists, left, right都是constructor的变体

回顾其他常见策略

作用到假设的策略

```
Theorem ev_plus4 : forall n, ev n -> ev (4 + n).
Proof.
  intros n H.
  (* (fun (n : nat) (H : ev n) => ?Goal) *)
  apply ev_SS in H.
 (* (fun (n : nat) (H : ev n) =>
     let H0 : ev (S (S n)) := ev_SS n H in ?Goal@{H:=H0}) *)
  apply ev_SS.
  apply H.
Qed.
```

用let引入新生成的值，同时用@标记生成之后的替换

simpl和unfold

```
Theorem ev_plus4 : forall n, ev (4 + n) -> ev n.
Proof.
  intros n H.
  (* H: ev (4 + n)
   * (fun (n : nat) (H : ev (4 + n)) => ?Goal) *)
  simpl in H.
  (*  ev (S (S (S (S n))))
   * (fun (n : nat) (H : ev (4 + n)) => ?Goal) *)
```

不会对生成的proof object起任何作用
即使假设/目标会发生变化

assert

```
Theorem ev_plus4''''' : forall n, ev n -> ev (4 + n).
Proof.
  intros n H.
  assert (H0: forall n, ev n -> ev (2 + n)).
  (* (fun (n : nat) (H : ev n) =>
     let H0 : forall n0 : nat, ev n0 -> ev (2 + n0)
     := ?Goal in ?Goal0) *)
  { apply ev_SS. }
  apply H0 in H. apply H0 in H. apply H.
Qed.
```

用let引入新生成的定理，同时将证明部分标记为?Goal

injection

```
Theorem S_injective' : forall (n m : nat),
  S n = S m -> n = m.
Proof.
  intros n m H.
  injection H as Hnm.
  apply Hnm.
Qed.
```

```
Definition S_injective'' : forall (n m : nat),
  S n = S m -> n = m :=
  fun (n m : nat) (H : S n = S m) =>
    f_equal (fun e : nat => match e with
      | 0 => n
      | S n0 => n0
    end) H.
```

injection：生成从复杂结构到简单结构的函数并调用f_equal

复习：f_equal

```
Theorem f_equal :
  forall (A B : Type) (f: A -> B) (x y: A),
  x = y -> f x = f y.
Proof. intros A B f x y eq.
       rewrite eq.  reflexivity.  Qed.
```

剩余策略

- Symmetry, f_equal, transitiviy等都是应用已经证明好的定理
- induction、discriminate和rewrite生成的证明对象在下一章介绍
  - 涉及结构归纳法定理

作业

- 完成ProofObject中standard非optional的9道习题
  - 请使用最新英文版教材