Coq

Outline

- Curry-Howard Isomorphism
- Calculus of Inductive Constructions
- Theorem Provers and Meta Languages
- Coq
- Further Resources

- On the one hand, Coq can seem “way out there”
- On the other hand, Coq can seem like a natural unification of every class topic
  - Theorem proving, type systems, lambda calculus, dependent types, polymorphism, truth vs. provability, small-step opsem and normal forms, etc.

Curry-Howard Isomorphism

- There is a direct equivalence between computer programs and mathematical proofs
- The (Intuitionistic) Natural Deduction Proof System can be directly interpreted as the Typed Lambda Calculus [Howard, 1969]

- "A proof is a program, and the formula it proves is the type for the program."
- How?

Curry-Howard Correspondences

Logic	Programming
Implication	Function Type
Conjunction	Product Type
Disjunction	Sum Type
True Formula	Unit Type
False Formula	Bottom Type
Hypotheses	Free Variables
Implication Elimination	Application
Implication Introduction	Abstraction
Universal Quantification	Generalized Product Type (Π)
Existential Quantification	Generalized Sum Type (Σ)
Natural Deduction	Type System for Lambda Calculus

One Example

- Consider: $P \rightarrow (Q \rightarrow P)$
- It is an axiom (tautology) in logic

P	Q	$Q \rightarrow P$	$P \rightarrow (Q \rightarrow P)$
T	T	T	T
T	F	T	T
F	T	F	T
F	F	T	T

One Example

- Consider: $P \rightarrow (Q \rightarrow P)$
- It is an axiom (tautology) in logic

P	Q	Q → P	P → (Q → P)
T	T	T	T
T	F	T	T
F	T	F	T
F	F	T	T

- So a program exists with type $\sigma \rightarrow (\tau \rightarrow \sigma)$

$$\lambda x{:}\sigma.\ \lambda y{:}\tau.\ x$$

Curry-Howard Correspondence 2

Hilbert-style intuitionistic implicational logic	Simply typed combinatory logic
$\dfrac{\alpha \in \Gamma}{\Gamma \vdash \alpha}$ Assum	$\dfrac{x : \alpha \in \Gamma}{\Gamma \vdash x : \alpha}$
$\dfrac{}{\Gamma \vdash \alpha \to (\beta \to \alpha)}$ Ax_K	$\dfrac{\boxed{\lambda x{:}\sigma.\ \lambda y{:}\tau.\ x}}{\Gamma \vdash K : \alpha \to (\beta \to \alpha)}$
$\dfrac{}{\Gamma \vdash (\alpha \to (\beta \to \gamma)) \to ((\alpha \to \beta) \to (\alpha \to \gamma))}$ Ax_S	$\dfrac{\boxed{\lambda x{:}\tau \to \tau' \to \tau''.\ \lambda y{:}\tau \to \tau'.\ \lambda z{:}\tau.\ xz(yz)}}{\Gamma \vdash S : (\alpha \to (\beta \to \gamma)) \to ((\alpha \to \beta) \to (\alpha \to \gamma))}$
$\dfrac{\Gamma \vdash \alpha \to \beta \qquad \Gamma \vdash \alpha}{\Gamma \vdash \beta}$ Modus Ponens	$\dfrac{\Gamma \vdash E_1 : \alpha \to \beta \qquad \Gamma \vdash E_2 : \alpha}{\Gamma \vdash E_1\ E_2 : \beta}$

Constructive Logic

- In Constructive (or Intuitionistic) Logic a statement is only true if there is a constructive proof for it
- Competing Philosophies:
  - Formalism. A statement is either true or false regardless of whether we have evidence.
    Thus P || !P (excluded middle).
    Thus !!P → P (double negation elim). [Hilbert]
  - Intuitionism. A statement is only true if there is a proof for it. [LEJ Brouwer]

Intuition Was Radical

- "At issue in the sometimes bitter disputes was the relation of mathematics to logic, as well as fundamental questions of methodology, such as how quantifiers were to be construed, to what extent, if at all, nonconstructive methods were justified, and whether there were important connections to be made between syntactic and semantic notions."
  - Dawson's biography of Godel

Intuitionism Was Radical 2

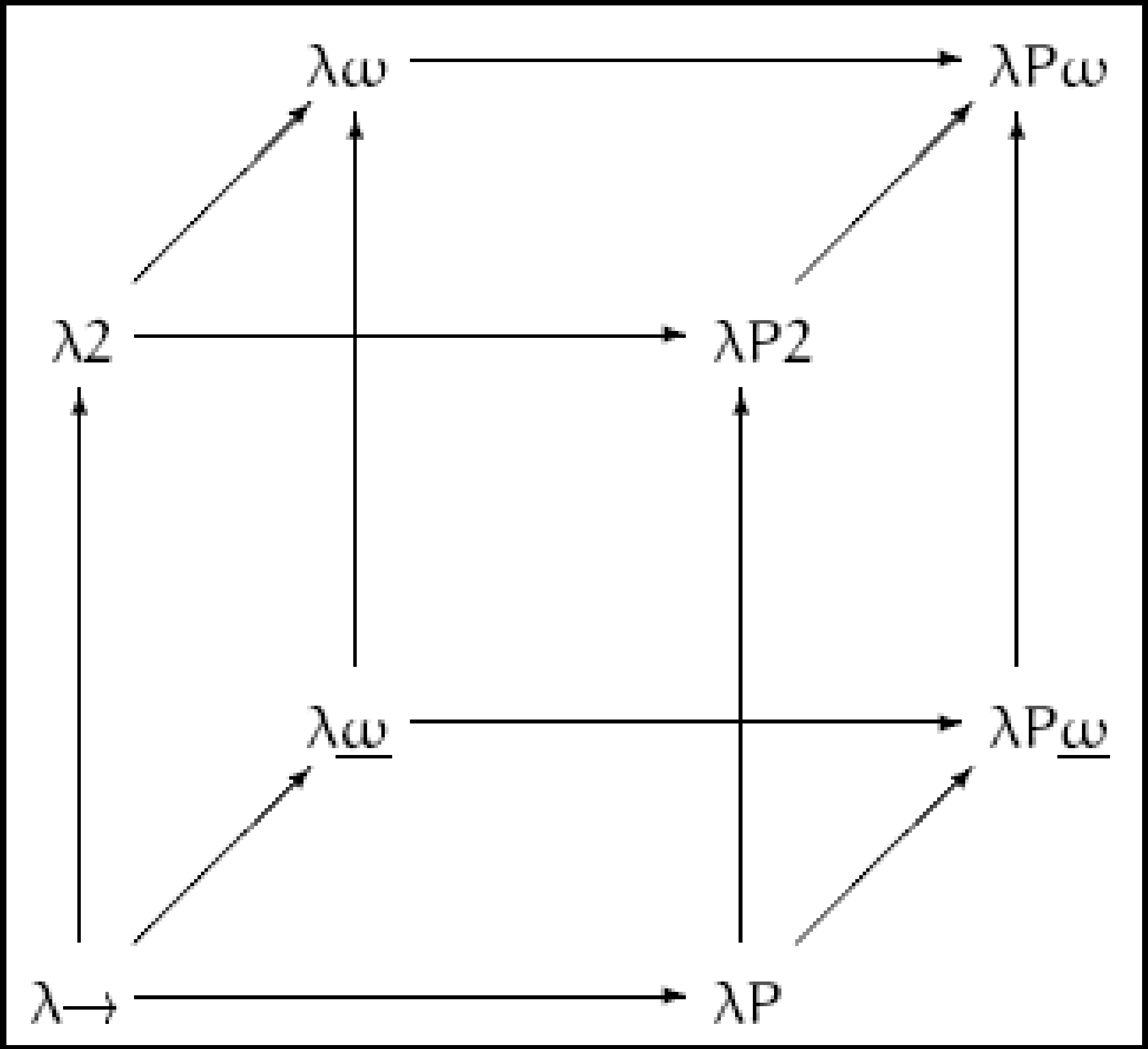
- "Taking the principle of excluded middle [P or not P] from the mathematician would be the same, say, as proscribing the telescope to the astronomer or to the boxer the use of his fists. To prohibit existence statements and the principle of excluded middle is tantamount to relinquishing the science of mathematics altogether."
  - Hilbert

Constructive Implications

- In a constructive logic, you do ***not*** have
  - Excluded Middle: P || !P
  - Double Negation Elimination: !!P → P
- However, you ***do*** have the existence property
  - The existence property or witness property is satisfied by a theory if, whenever a sentence (∃x)A(x) is a theorem, where A(x) has no other free variables, then there is some term *t* such that the theory proves A(*t*)

Lambda Cube

[Barendregt, 1991]



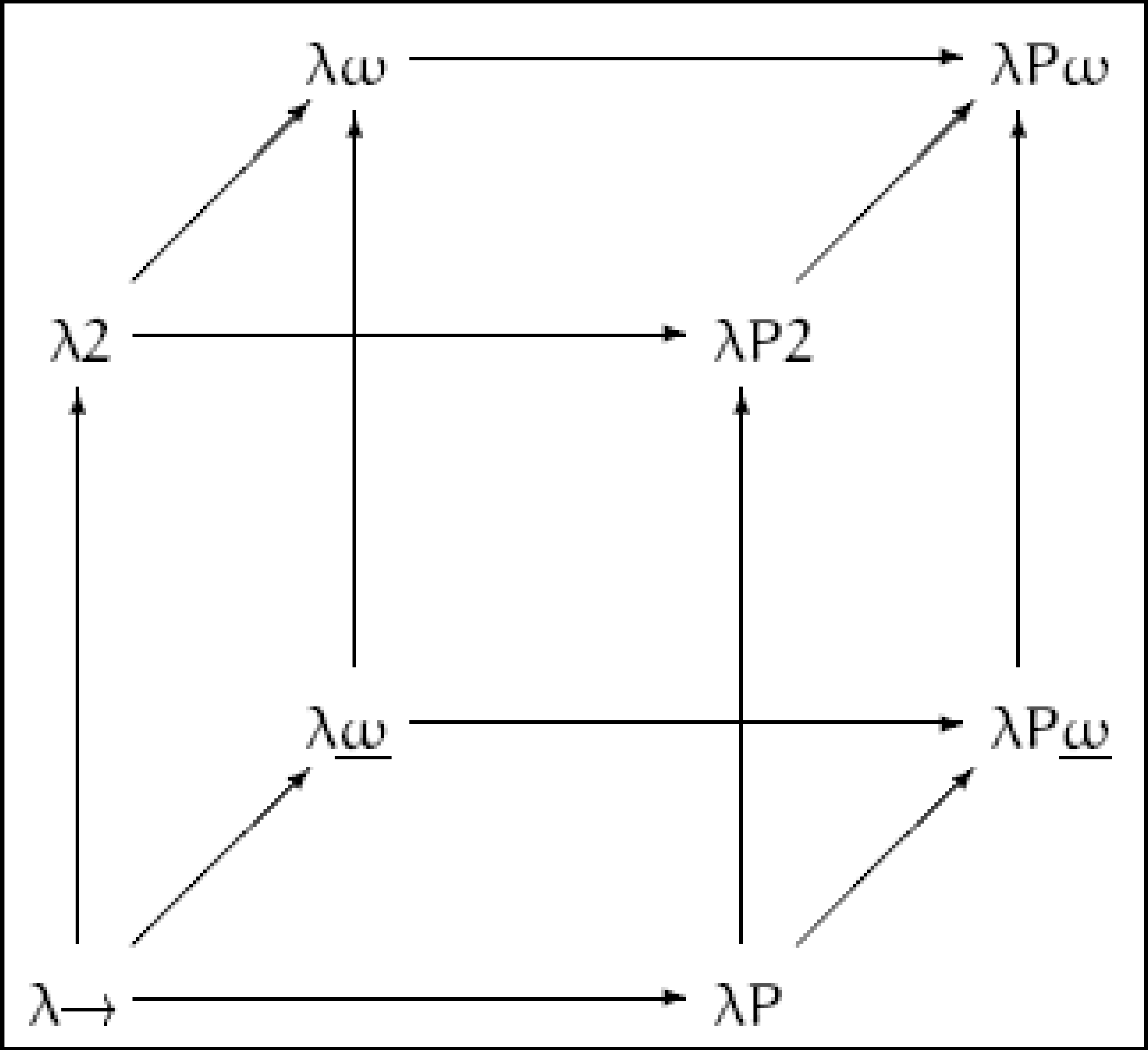
Add Polymorphism →

Add Type Operators →

Add Dependent Types →

Lambda Cube

[Barendregt, 1991]



Calculus of Inductive Constructions

System F

Add Polymorphism →

Add Type Operators →

Lambda Calculus

Add Dependent Types →

Calculus of Inductive Constructions

- It is a Type Theory and Programming Language (higher-order typed lambda calculus)
- Also a Foundation for Mathematics
- It is Strongly Normalizing
  - Every sequence of rewrites terminates with a normal form
  - That is, every program terminates
  - Not provable inside the the system itself [Godel]

Aside: Theorem Provers and ML

- "Historically, ML was conceived to develop proof tactics in the LCF theorem prover (whose language, *pplambda*, a combination of the first-order predicate calculus and the simply-typed polymorphic lambda calculus, had ML as its metalanguage)."

- Compare: SQL for Database Queries

Aside: Theorem Provers and ML 2

- “In ML the various parts of the object language---terms, declarations, proofs and rules---are data types. By defining a formal metalanguage we have made concrete the structure and elements of the object language. We can then write ML programs that manipulate objects of the object language. Thus, for example, we can write a program to return the subterms of a term or one that substitutes a term for a free variable in a term. More importantly, we can write ML functions which search for or transform proofs. We can then use such automated proof techniques and theorem-proving heuristics, tactics, while writing proofs.
- A tactic is a function written in ML which partially automates the process of theorem proving […].”

Coq

- **Coq** is a dependently typed functional programming language based on the calculus of constructions
- Associated with an interactive theorem prover
- Influential author: Thierry Coquand
  - "CoC" → "Coq" (French for rooster)
- 2013 ACM Software System Award
- Associated with the CompCert project

Coq's Magic Power

- Recall the existence property: whenever a sentence $(\exists x)A(x)$ is a theorem, where $A(x)$ has no other free variables, then there is some term *t* such that the theory proves $A(t)$
- So if you can prove “There exists *x* such that *x* is a function that sorts a list of numbers” in Coq
- Then Coq will produce a program *x* doing so
  - Coq will write the source code to “sort” for you!

This Merits Repeating

- Because Coq is constructive and because proofs are related to programs …

  … if you can prove something in Coq, you get the corresponding program for free!
- "An interesting additional feature of Coq is that it can automatically extract executable programs from specifications, as either Objective Caml or Haskell source code."

Coq Example: Naturals

- "Proof development in Coq is done through a language of tactics that allows a user-guided proof process. […] the curious user can check that tactics build lambda-terms."
- Coq "data type":

```
Inductive nat : Set :=
| O : nat
| S : nat -> nat.
```

Coq Example: Lists

- Naturals:

```
Inductive nat : Set :=
| 0 : nat
| S : nat -> nat.
```

- Lists with element type *A*:

```
Inductive list (A:Type) : Type :=
| nil : list A
| cons : A -> list A -> list A.
```

Coq Example: Function

- Addition of Naturals:

```
Fixpoint plus (n m:nat) {struct n} : nat :=
  match n with
  | O => m
  | S p => S (plus p m)
  end
where "p + m" := (plus p m).
```

Coq Example: Function

- Addition of Naturals:

Which structure are we inducting on? Recall: strongly normalizing!

```
Fixpoint plus (n m:nat) {struct n} : nat :=
  match n with
  | O => m
  | S p => S (p + m)
  end
where "p + m" := (plus p m).
```

Coq Example: Proof that “length” is correct

```
Inductive seq : nat -> Set :=
| niln : seq 0
| consn : forall n : nat, nat -> seq n -> seq (S n).
```

Each sequence is a list that also stores its own length!

```
Fixpoint length (n : nat) (s : seq n) {struct s} : nat :=
  match s with
  | niln => 0
  | consn i _ s' => S (length i s')
  end.
```

What if I try to recompute the length recursively? will I get the same answer as the “stored” length?

Coq Example: A Theorem

```
Theorem length_corr :
  forall (n : nat) (s : seq n),
    length n s = n.
```

- Recall: Coq is an interactive theorem prover!

```
Proof.
```

- To prove "forall *n*", we say "introduce an arbitrary *n* about which we know nothing"

```
    intros n s.
```

Coq Example: A Proof

```
forall (n : nat) (s : seq n), length n s = n.
Proof.
  Intros n s.
```

- Now we decide to reason by [structural] induction on *s*. It has two cases, *niln* and *consn*, so we have two subgoals.

```
      induction s.
```

Coq Example: A Proof

```
forall (n : nat) (s : seq n), length n s = n.
Proof.
  Intros n s.
    induction s.
```

- We are in the case where *s* is *niln*. We simply substitute that into the body of *length* …

```
Fixpoint length (n : nat) (s : seq n) {struct s} : nat :=
  match s with  | niln => 0  | consn i _ s' => S (length i s')
  end.
```

- … and get *length 0 niln = 0*.

```
          simpl.
```

Coq Example: A Proof

```
forall (n : nat) (s : seq n), length n s = n.
Proof.
  Intros n s.
    induction s.
      simpl.
```

- Now we have to prove the equality between *length n s* and *n*. But currently *length 0 niln = 0*, so we just have to prove *0 = 0*.

```
        trivial.
```

Coq Example: A Proof

```
forall (n : nat) (s : seq n), length n s = n.
Proof. Intros n s.
    induction s.
      simpl. Trivial. (* base case *)
```

- Now the inductive case where *s = consn n e s*. We again simply substitute in the body of length …

```
Fixpoint length (n : nat) (s : seq n) {struct s} : nat :=
  match s with  | niln => 0  | consn i _ s' => S (length i s')
```

… but we also have an inductive hypothesis for any smaller sequence *s*'.

```
        simpl.
```

Coq Example: A Proof

```
forall (n : nat) (s : seq n), length n s = n.
Proof. Intros n s.
    induction s.
      simpl. Trivial. (* base case *)
      simpl.
```

- The inductive hypothesis has type *length n s = n* (for smaller sequences). We apply it!

```
rewrite Ihs.
```

- This rewrites *length i s'* into *n*

```
match s with  | niln => 0  | consn i _ s' => S (length i s')
```

Coq Example: A Proof

```
forall (n : nat) (s : seq n), length n s = n.
Proof. Intros n s.
    induction s.
      simpl. Trivial. (* base case *)
      simpl. rewrite IHs.
```

- Now the goal is $S\ n = S\ n$, which is trivial.

```
Trivial. (* inductive step *)
```

- And now both sub-cases are handled, so we close off the inductive case analysis and forall-introductions:

```
Qed.
```

That Interactive Session Generates A Machine-Checkable Proof

- Coq is an interactive theorem prover. Here's the proof:

```
length_corr =
  fun (n : nat) (s : seq n) =>
    seq_ind (fun (n0 : nat) (s0 : seq n0) => length n0 s0 = n0)
      (refl_equal 0)
      (fun (n0 _ : nat) (s0 : seq n0) (IHs : length n0 s0 = n0) =>
        eq_ind_r
          (fun n2 : nat => S n2 = S n0)
          (refl_equal (S n0)) IHs) n s
: forall (n : nat) (s : seq n), length n s = n
```

Generating OCaml

- Consider:

  forall b:nat, b > 0 -> forall a:nat, diveucl a b

- where *diveucl* is a [dependent] type (i.e., a specification) for the pair of the quotient and the modulo
- That is, we are saying "there exists a function that takes all naturals *a* and *b* with *b>0* and returns the euclidean division of them"
  - Once we prove that theorem, Coq will generate a correct OCaml implementation for us!

```
type nat =| O | S of nat
type sumbool =| Left | Right
(** val sub : nat -> nat -> nat **)
let rec sub n m =
  match n with
  | O -> n
  | S k -> (match m with
            | O -> n
            | S l -> sub k l)
(** val le_lt_dec : nat -> nat -> sumbool **)
let rec le_lt_dec n m =
  match n with
  | O -> Left
  | S n0 -> (match m with
             | O -> Right
             | S m0 -> le_lt_dec n0 m0)
```

```
(** val le_gt_dec : nat -> nat -> sumbool **)
let le_gt_dec =
  le_lt_dec

type diveucl =
| Divex of nat * nat

(** val eucl_dev : nat -> nat -> diveucl **)
let rec eucl_dev n m =
  let s = le_gt_dec n m in
  (match s with
   | Left ->
     let d = let y = sub m n in eucl_dev n y in
     let Divex (q, r) = d in Divex ((S q), r)
   | Right -> Divex (O, m))
```

Rosetta Stone

```
euclid(m,  n):
  r = m;
  q = 0;
  while (r >= n):
    r = r - n;
    q = q + 1;
  return (q,r);
```

```
let rec eucl_dev n m =
  let s = le_gt_dec n m in
  (match s with
   | Left ->  (* Left means >= is true *)
     let d =
       let y = sub m n in
       eucl_dev n y in
     let Divex (q, r) = d in
     Divex ((S q), r)
   | Right -> Divex (O, m))
  (* Right: >= is false *)
```

Further Resources

- Certified Programming with Dependent Types
  - Adam Chlipala

  http://adam.chlipala.net/cpdt/
- “A traditional hardcopy version is available from MIT Press, who have graciously agreed to allow distribution of free versions online indefinitely, minus the benefits of the Press' copy editing!”
- Outside of France, Adam is our leading Coq wizard …