1. Dependently-Typed Programming in Coq
   - Subset Types
   - Inductive Families
   - Examples & Exercises

2. A bit about models
   - The groupoid model
   - Homotopy Type Theory
Correctness by construction

Example: bounds checking.

▶ Partial function

**Definition** `nth_impossible : ∀ A, nat → list A → A.`
Abort.

▶ A total version in Coq

**Check** `nth : ∀ A, nat → list A → ∀ default : A, A.`

+ lemmas: ∀ l, n < length l, nth l n default = nth (map id l) n default

▶ The total function, with types depending on values:

**Example** `nth : ∀ A (l : list A), {n : nat | n < length l} → A.`

*Exercise.*

Defined.
Subset types are $\Sigma$-types:

$$\{ \ x : A \ | \ P \ \} \leftrightarrow \text{sigma } A \ (\text{fun } x : A \Rightarrow P)$$

- **Constructor:**

  \[\text{Check } (\text{exist : } \forall \ A \ (P : A \rightarrow \text{Prop}), \forall \ x : A, \ P \ x \rightarrow \{ \ x : A \ | \ P \ x \ \}).\]

- **Projections:**

  \[\text{Check } (\text{proj1}_\text{sig} : \forall \ A \ P, \ \{x : A \ | \ P \ x\} \rightarrow A).\]
  \[\text{Check } (\text{proj2}_\text{sig} : \forall \ A \ P \ (p : \{x : A \ | \ P \ x\}), \ P \ (\text{proj1}_\text{sig} \ p)).\]

- **Binding notation:**

  \[\text{Check } (\text{proj2}_\text{sig} : \forall \ A \ P \ (x : A \ | \ P \ x), \ P \ (\text{proj1}_\text{sig} \ x)).\]
Strong specifications

Program Definition euclid_type :=
\[ \forall (x: \text{nat}) (y: \text{nat} \mid 0 < y), \]
\[ \{ (q, r) : \text{nat} \times \text{nat} \mid x = q \times y + r \} \].
▷ A tool to program with subset types.
▷ Coq’s type system + the rules:

\[
\vdash t : \{ x : A \mid P \}

\]

\[
\text{------------------- (Sub-Proj)}
\]

\[
\vdash t : A
\]

\[
\vdash t : A \vdash P : \text{Prop}

\]

\[
\text{------------------- (Sub-Proof)}
\]

\[
\vdash t : \{ x : A \mid P \}
\]

Note Sub-Proof does *not* require a proof term.

▷ An elaboration to Coq terms with holes for the missing proof terms.
▶ An elaboration to Coq terms with holes for the missing proof terms.

⊢ \( t : \{ x : A \mid P \} \)
------------------- (App)
⊢ proj1_sig \( t : A \)

⊢ \( t : A \quad x : A \mid P : \text{Prop} \mid \vdash ?p : P \quad t \)
--------------------------------------------------------------------------------------------------------------- (App)
⊢ exist \( t \ ?p : \{x : A \mid P\} \)

▶ This inserts projections and coercions everywhere needed.
▶ The translation is correct for a proof-irrelevant Coq only (Sozeau’08, PhD)
Require Import Arith Omega.

Program Fixpoint euclid (x : nat) 
  (y : nat | 0 < y) (* Binder for subsets *)
  { wf lt x } : (* Wellfounded definition *)
  { (q, r) : nat × nat | x = q × y + r } (* Rich type *) :=
  _ (* Hint: use lt_dec *).

Next Obligation.

  Exercise.

  Defined.
Program Fixpoint euclid (x : nat)
    (y : nat | 0 < y) (* Binder for subsets *)
    { wf lt x } : (* Wellfounded definition *)
    { (q, r) : nat × nat | x = q × y + r } (* Rich type *) :=
        if lt_dec x y then (0, x)
        else
            let 'pair q r := euclid (x - y) y in
            (S q, r).

Next Obligation. omega. Qed.
Next Obligation.
    destruct euclid. simpl in *. subst x0.
    omega.
Defined.

Extraction Inline proj1_sig projT2 projT1.
Recursive Extraction euclid.
Inductive Families

Inductive vect (A : Set) : nat → Set :=
  | vnil : vect A 0
  | vcons (a : A) (n : nat) : vect A n → vect A (S n).

▶ Indexed
▶ Recursive
▶ Terms and types carry more information

Example x : vect bool 3 :=
  vcons _ true 2 (vcons _ true 1 (vcons _ false 0 (vnil _))).
Arguments vnil \( \{A\} \).

Arguments vcons \( \{A\} a \{n\} \nu \).

Fixpoint vect_map \( \{A B : \text{Set}\} \{n\} (f : A \to B) (\nu : \text{vect} A n) \) :
\( \text{vect} B n := \)

\[
\begin{align*}
\text{match } \nu \text{ with } \\
\quad | \text{vnil} & \Rightarrow \text{vnil} \\
\quad | \text{vcons } a n \nu' & \Rightarrow \oplus\text{vcons} B (f a) n (\text{vect}_\text{map} f \nu') \\
\end{align*}
\]
end.
Arguments \( vnil \) \( \{ A \} \).

Arguments \( vcons \) \( \{ A \} \ a \ \{ n \} \ v \).

Fixpoint \( \text{vect}_\map \) \( \{ A \ B : \text{Set} \} \ \{ n \} \ (f : A \to B) \ (v : \text{vect} \ A \ n) \) :
\( \text{vect} \ B \ n := \)

\[
\text{match } v \text{ with }
| \ vnil \Rightarrow vnil \\
| \ vcons \ a \ n \ v' \Rightarrow \mathbb{vcons} \ B \ (f \ a) \ n \ (\text{vect}_\map \ f \ v')
\emph{end.}
\]

Fixpoint \( \text{vect}_\map' \) \( \{ A \ B : \text{Set} \} \ \{ n \} \ (f : A \to B) \ (v : \text{vect} \ A \ n) \) :
\( \text{vect} \ B \ n := \)

\[
\text{match } v \text{ in } \text{vect}_\ n \text{ return } \text{vect} \ B \ n \text{ with }
| \ vnil \Rightarrow vnil \\
| \ vcons \ a \ n \ v' \Rightarrow \text{vcons} \ (f \ a) \ (\text{vect}_\map' \ f \ v')
\emph{end.}
\]
Refining the input type

Definition vect_hd \{ A : \text{Set} \} \{ n \} (v : \text{vect} A (S n)) : A :=
  \text{match} \ v \ \text{in} \ \text{vect} - n
  \text{return} \ \text{match} \ n \ \text{with} \ 0 \Rightarrow \text{unit} \ | \ S \ n \Rightarrow \ A \ \text{end}
  \text{with}
  | \ \text{vnil} \Rightarrow \text{tt}
  | \ \text{vcons} \ a \ n \ v' \Rightarrow \ a
\text{end}.

Definition vect_tl \{ A : \text{Set} \} \{ n \} (v : \text{vect} A (S n)) : \text{vect} A n.

Exercise.
Defined.
Definition `concat_vect {A : Set} {m n}`

\[(v : vect A m) (w : vect A n) : vect A (m + n).\]


- Non-linear case: harder to program, needs *explicit* equality manipulations in CoQ.

Example diagonal `{A : Set} {m}`:

`vect (vect A m) m → vect A m`.

- Certified Programming with Dependent Types (Chlipala, MIT Press) goes into the many tricks needed to program with these types in CoQ.
It’s all the same

- Inductive families vs subset types.
- Structure vs property.

**Definition** `ilist` \( \{ A : \text{Set} \} (n : \text{nat}) := \{ l : \text{list } A \mid \text{length } l = n \}. \)

**Record** `Iso (A B : \text{Type}) := \{ f : A \to B; g : B \to A; \}

\[ \text{fog} : \forall x, f(g(x)) = x; \]

\[ \text{gof} : \forall x, g(f(x)) = x \}. \]

**Program Definition** `vect_ilist` \( \{ A : \text{Set} \} (n : \text{nat}) : \)

\[ \text{Iso (vect } A \ n) (@\text{ilist } A \ n) := \{ f := _; g := _ \}. \]

Next Obligation. **Exercise.** Qed.

The relationship can be made explicit, categorically or using a universe of datatypes, see the literature on Ornaments (McBride, Ghani, Dagand, ...)
More examples

- Matrices, any bounded datastructure

Definition \textit{square\_matrix} \{A\} n := vect (vect A n) n.

- Regular expressions indexed by their semantics

Require Import String.

Inductive regexp : (string \rightarrow Prop) \rightarrow Type :=
| empty : regexp (fun s \Rightarrow s = ""%string)
| or x y (a : regexp x) (b : regexp y):
  regexp (fun s \Rightarrow x s \lor y s).

Definition matches \times (e : regexp x) (s : string) : bool.

Lemma regexp\_interp \times (e : regexp x) (s : string):
  matches \times e s = true \rightarrow \times s.
Proof. ..... Defined.
Finite sets ($\text{fin } n$ has $n$ elements)

**Inductive** $\text{fin} : \text{nat} \rightarrow \text{Set} :=$

| fin0 $n : \text{fin} (\text{S } n)$ |
| finS $n (f : \text{fin } n) : \text{fin} (\text{S } n)$. |

**Definition** $\text{lookup} \{A : \text{Set}\} \{n\} (v : \text{vect } A \ n) (f : \text{fin } n) : A$.

**Exercise.**

Defined.
Inductive ty := nat_ty | arrow (t u : ty).
Definition ctx := vect ty.

Inductive expr {n} (Γ : ctx n) : ty → Set :=
  | var (f : fin n) : expr Γ (lookup Γ f)
  | app {tau tau’}
      (f : expr Γ (arrow tau tau’)) (u : expr Γ tau)
    : expr Γ tau’
  | lam {tau tau’}
      (t : @expr (S n) (vcons tau Γ) tau’)
    : expr Γ (arrow tau tau’).
And definitional interpreters

```ml
Fixpoint interp_type (ty : ty) : Set :=
  match ty with
  | nat_ty ⇒ nat
  | arrow tau tau' ⇒ interp_type tau → interp_type tau'
end.

Definition interp n (Γ : ctx n) (ty : ty) (x : expr Γ ty) :
  interp_type ty.
  ☐

Exercise.

Defined.

- No ill-typed terms, by construction.
```
Some history

Many flavors of inductive families and dependent pattern-matching.

- DML (Xi): ML + integer indexed types (presburger arithmetic)
- Agda (Norell), Epigram (McBride): have the K rule that allows working with non-linear cases and a higher level construction (the Equations plugin (Sozeau) does the same for Coq).
- Haskell, OCaml GADTs: indices can be types only, not arbitrary terms.
- CoqMT (Strub): Coq Todulo Theories, extends conversion to arbitrary decidable theories, including presburger arithmetic. No equality manipulations necessary!

And many others: ATS (Xi), Beluga (Pientka), Ωmega (Sheard), Trellys (Weirich), . . .
Some bibliography

On dependent pattern-matching and inductive families:

➤ Inductive types in the system Coq, Paulin, TLCA’93.
➤ Eliminating dependent pattern-matching, Goguen, McBride and McKinna, LNCS, 2006. McBride’s papers include a large number of examples.
➤ Program: in Coq’s reference manual and Programming Finger Trees in Coq, Sozeau, ICFP’07
1. Dependently-Typed Programming in Coq
   - Subset Types
   - Inductive Families
   - Examples & Exercises

2. A bit about models
   - The groupoid model
   - Homotopy Type Theory
A model of (Martin-Löf) Type Theory can be constructed in groupoids. The idea stems from the structure of propositional equality:

Inductive eq \{A : Type\} (a : A) : A → Type :=
  eq_refl : eq a a.

Definition eq_sym \{A\} (a b : A) : eq a b → eq b a.
Proof. destruct 1; reflexivity. Defined.

Definition eq_trans \{A\} \{a b c : A\} : eq a b → eq b c → eq a c.
Proof. intros H H'. destruct H; apply H'. Defined.
A groupoid is just a type with a notion of “equivalence”, that is an equivalence relation. Clearly \textit{eq} is one.

- Each type gets interpreted as itself plus its equality.
- Each term is invariant under equality so transports that information (it’s a groupoid functor).
- This forms a categorical model: \( \Gamma \vdash t : T \Rightarrow [t] : [\Gamma] \to [T] \).
Equality is non-trivial

Notice how we can define also:

Definition eq_trans’ \{A\} \{a b c : A\} :
  eq a b → eq b c → eq a c.

Proof. intros H H'; destruct H, H'; apply eq_refl. Defined.

eq_trans and eq_trans’ are not definitionally equal, but again propositionally!

Lemma eq_trans_trans’ \{A\} (a b c : A) (H : eq a b) (H' : eq b c) :
  eq (eq_trans H H') (eq_trans’ H H').

Proof.
  Fail eq_refl.
  destruct H, H'. simpl.
  apply eq_refl.
Qed.
This begs the question, are all equality proofs propositionally equal? Which reduces to:

**Definition** \( \text{UIP} \) \( \{ A \} \ \{ a : A \} := \forall p : \text{eq} \ a \ a, \ p = \text{eq}\_\text{refl} \ a. \)

- The groupoid model shows that this is an *independent* principle, from the usual MLTT and Coq, i.e. you can’t prove it for all \( A \).

- However it is provable for (first-order) datatypes, e.g. nat, bool etc... Hedberg (98) showed any type with *decidable* equality has \( \text{UIP} \).
  - OTT/Epigram is a theory where all datatypes have \( \text{UIP} \), even without decidability.
  - Its natural interpretation is in *setoids*, the degenerate case of *groupoids* with \( \text{UIP} \).
Why is UIP independent?

- Recall a groupoid is just a type with an equivalence relation.
- Find a type and an equivalence relation (interpreting some type of type theory) with multiple different proofs of equality.

Canonical example: Isomorphisms of types are not unique. E.g. proofs of $\text{Iso bool bool}$. (Exercise: built the two inhabitants of that type). So types with iso proofs form a groupoid and type theory can be interpreted this way, contradicting UIP. Qed.
Why is UIP independent?

- Recall a groupoid is just a type with an equivalence relation.
- Find a type and an equivalence relation (interpreting some type of type theory) with multiple different proofs of equality.
- Canonical example: Isomorphisms of types are not unique. E.g. proofs of \textit{Iso bool bool}. (Exercise: built the two inhabitants of that type).
- So types with iso proofs form a groupoid and type theory can be interpreted this way, contradicting \textit{UIP}. Qed.
Comes V. Voevodsky (and A. Warren and S. Awodey and ...), looking at the groupoid model.
Comes V. Voevodsky (and A. Warren and S. Awodey and ...), looking at the groupoid model.

- They figure how to build a model in (weak) \( \text{infinity} \)-groupoids. i.e. the rich structure of equalities is infinite (proofs of proofs of equalities are related by proofs of proofs of proofs of equality etc... all the way up (\text{without} an end).)

To Homotopy Type Theory
Comes V. Voevodsky (and A. Warren and S. Awodey and ...), looking at the groupoid model.

- They figure how to build a model in (weak) infinity-groupoids. i.e. the rich structure of equalities is infinite (proofs of proofs of equalities are related by proofs of proofs of proofs of equality etc... all the way up (without an end).)

- The model validates a strong extensionality principle, the univalence principle.
From univalence follows:

- Proof-irrelevance: all proof of the same logical statements are equal
- Propositional extensionality: all logical statements are equal if biequivalent
- Functional extensionality: all functions are equal if pointwise equal
- Invariance under isomorphism: all terms can be transported through isomorphisms of types.

This goes beyond making type theory a good language for classical mathematics, it’s a new *foundation*. See the HoTT book.
Current state of the art:

- We don’t have a computational interpretation of the principle (i.e. it’s an inert axiom now, and the models are non-constructive)
- A cubical set model was introduced recently solving (part of) this issue (by Coquand et al).
- We get new datatypes called Higher Inductive Types allowing to form quotients internally, don’t know how to give them a syntax and elimination rules.