Recap

Simple inductive types (datatypes):

\[
\begin{align*}
\text{Inductive } & \text{ nat : Type := O : nat | S : nat -> nat.} \\
\text{Inductive } & \text{ bool := true | false.} \\
\text{Inductive } & \text{ list (A:Type) : Type :=} \\
& \quad \text{nil | cons (hd:A) (tl:list A).} \\
\text{Inductive } & \text{ tree (A:Type) :=} \\
& \quad \text{leaf | node (_,A) (_,nat->tree A).}
\end{align*}
\]

Smallest type closed by introduction rules (constructors)

Parameters: cons : forall A:Type, A -> list A -> list A

Coq prelude: cons 0 nil : list nat
Recap: Elimination rules

Generated elimination scheme (not primitive):

```
nat_rect
  :  forall P:nat->Type,
      P O -> (forall n, P n -> P (S n)) ->
      forall n, P n.
  := fun P h0 hS => fix F n :=
      match n return P n with
      | O => h0
      | S k => hS k (F k)
  end
```

Eliminator of recursive type =
dependent pattern-matching + guarded fixpoint
Recap: Logical connectives

Logical connectives and their non-dependent elimination schemes:

\textbf{Inductive} \ True \ : \ Prop \ := \ I. \\
\quad \text{True_rect} \ : \ \forall \ P:Type, \ P \ \rightarrow \ True \ \rightarrow \ P.

\textbf{Inductive} \ False \ : \ Prop \ := \ . \\
\quad \text{False_rect} \ : \ \forall \ P:Type, \ False \ \rightarrow \ P

\textbf{Inductive} \ and \ (A \ B:Prop) \ : \ Prop \ := \\
\quad \text{conj} \ (\_:\!A) \ (\_:\!B). \\
\quad \text{and_rect} \ : \ \forall \ (A \ B:Prop) \ (P:Type), \ (A\rightarrow B\rightarrow P) \rightarrow \ A/\!\!\backslash B \ \rightarrow \ P

\textbf{Inductive} \ or \ (A \ B:Prop) \ : \ Prop \ := \\
\quad \text{or_introl} \ (\_:\!A) \ | \ \text{or_intror} \ (\_:\!B). \\
\quad \text{or_ind} \ : \ \forall \ (A \ B \ P:Prop), \ (A\rightarrow P) \ \rightarrow \ (B\rightarrow P) \ \rightarrow \ P.
Overview

1. Inductive types
   - Equality
   - Arithmetic
   - Vectors

2. Theory of Inductive types
   - Strict Positivity
   - Dependent pattern-matching
   - Guarded fixpoint
   - The guard condition
Equality as an inductive family

Inductive eq (A:Type) (x:A) : A -> Prop :=
| eq_refl : eq A x x.

Elimination:

- eq_rect : \(\forall A \ x \ (P:A\rightarrow Type), \ P \ x \rightarrow \forall y, x=y \rightarrow P \ y\)

- Dependent version (generated by Scheme):
  \(\forall A \ x \ (P:\forall z, x=z \rightarrow Type), \ P \ x \ eq\_refl \rightarrow \forall y (e:x=y) \rightarrow P \ y \ e\)
Equational theory of \texttt{nat}

Dependent elimination needed to prove minimality:

\begin{verbatim}
match n return n=0 \/ \exists m, n=S m with
| 0 => inl eq_refl : (0=0 \/ \exists m, 0=S m)
| S k => inr (ex_intro k eq_refl)
    : (S k = 0 \/ \exists m, S k = S m)
end
\end{verbatim}
Equational theory of \texttt{nat}

Injectivity of constructors:

\begin{verbatim}
Definition pred (n:nat) :=
    match n with O => O | S k => k end.

f_equal pred : S n = S m -> n = m
\end{verbatim}

\textbf{Tactic} \texttt{injection H:}

- applies this construction on hyp \texttt{H: C t_1..t_n = C u_1..u_n}
- derives proofs of \( t_1 = u_1 \ldots t_n = u_n \)
Equational theory of \texttt{nat}

Discrimination of constructors:

\begin{verbatim}
Definition P (n:nat) :=
  match n return Prop with O => True | S k => False end.

match (e:0=1) in _=y return P y with
  | eq_refl => I : P 0 (* P 0 = True *)
end : P 1 (* P 1 = False *)
\end{verbatim}

Tactic \texttt{discriminate}:

- \textbf{solves goals of the form} \( C \ t_1 \ldots t_n \not= D \ u_1 \ldots u_k \)
- \texttt{discriminate} \( H \) \textbf{solves the goal when} \( H : C \ t_1 \ldots t_n = D \ u_1 \ldots u_k \)
Vectors (Lists with size)

Inductive type with parameters and index:

\[
\text{Inductive vect (A:Type) : nat -> Type :=}
| \text{niln : vect A O}
| \text{consn :}
\quad A \to \forall n:nat, \text{vect A n -> vect A (S n)}.
\]

which defines

- a family of types-predicates:
  \[\Gamma \vdash \text{vect : Type \to nat \to Type}\]
- a set of introduction rules for the types in this family

\[
\frac{\Gamma \vdash A : \text{Type}}{
\Gamma \vdash \text{niln}_A : \text{vect A O}}
\]

\[
\frac{\Gamma \vdash A : \text{Type} \quad \Gamma \vdash a : A \quad \Gamma \vdash n : nat \quad \Gamma \vdash l : \text{vect A n}}{
\Gamma \vdash \text{consn}_A a n l : \text{vect A (S n)}}
\]
Vectors : elimination

- an elimination rule (pattern-matching operator with a result depending on the object which is eliminated)

\[
\Gamma \vdash v : vect A n \quad \Gamma, p : nat, x : vect A p \vdash C(p, x) : s \\
\Gamma \vdash t_1 : C(O, \text{nil}_n A) \\
\Gamma, a : A, n : nat, l : vect A n \vdash t_2 : C(S n, \text{cons}_n A a n l) \\
\Gamma \vdash \left( \text{match } v \text{ as } x \text{ in } vect _ A p \text{ return } C(p, x) \text{ with } \right) \\
\left( \begin{array}{l}
\text{niln } \Rightarrow t_1 \\
\text{consn } a n l \Rightarrow t_2 \\
\end{array} \right) : C(n, v)
\]

- and the obvious reduction rules ($\iota$-reduction)
Well-formed inductive definitions
Constructors of the inductive definition $I$ have type:

$$
\kappa : \forall (z_1 : C_1) \ldots (z_k : C_k). I\ a_1 \ldots a_n
$$

where $C_i$ can feature instances of $I$.
Question: can these instances be arbitrary?
Constructors of the inductive definition \( I \) have type:

\[
\kappa : \forall (z_1 : C_1) \ldots (z_k : C_k). I \, a_1 \ldots a_n
\]

where \( C_i \) can feature instances of \( I \).

Question: can these instances be arbitrary? No!

Example:

```lean
Inductive lambda : Type :=
| Lam : (lambda -> lambda) -> lambda
```

Constructors of the inductive definition $I$ have type:

$$\kappa : \forall (z_1 : C_1) \ldots (z_k : C_k). I a_1 \ldots a_n$$

where $C_i$ can feature instances of $I$.

Question: can these instances be arbitrary? No!

Example:

```plaintext
Inductive lambda : Type :=
| Lam : (lambda -> lambda) -> lambda
```

We define:

```plaintext
Definition app (x y:lambda) := match x with (Lam f) => f y end.
Definition Delta := Lam (fun x => app x x).
Definition Omega := app Delta Delta.
```

and the evaluation of $\Omega$ loops.
Things can even be worse:

```plaintext
Inductive lambda : Type :=
| Lam : (lambda -> lambda) -> lambda
```

Now define:

```plaintext
Fixpoint lambda_to_nat (t : lambda) : nat :=
  match t with Lam f -> S (lambda_to_nat (f t)) end.
```
Necessity of restrictions

Things can even be worse:

```latex
Inductive lambda : Type :=
| Lam : (lambda -> lambda) -> lambda
```

Now define:

```latex
Fixpoint lambda_to_nat (t : lambda) : nat :=
  match t with Lam f -> S (lambda_to_nat (f t)) end.
```

What happens with `(lambda_to_nat (Lam (fun x => x)))`?
The way out: (strict) positivity condition

- An inductive type is defined as the smallest type generated by a set $(\kappa_i)_{1 \leq i \leq n}$ of constructors.
- We can see it as $\mu X, \bigoplus_{1 \leq i \leq n} \kappa_i(X)$ (with $\mu$ a fixpoint operator on types).
  Eg: $\mathbb{N} = \mu X. \ 1 + X$ and so $\mathbb{N} = 1 + \mathbb{N}$
- The existence of this smallest type can be proved at the impredicative level when the operator \( \lambda X, \bigoplus_{1 \leq i \leq n} \kappa_i(X) \) is monotonic.
  \( \mu X : \prod (X \to A) \to A \) has a fixpoint...
- In order both to ensure monotonicity and to avoid paradox (predicativity of Type), Coq enforces a strict positivity condition: \( X \) should never appear on the left of an arrow in the type of its constructors.
The way out: (strict) positivity condition

More precisely, if the type (a.k.a. arity) of a constructor is:

\[ c : C_1 \rightarrow \ldots \rightarrow C_k \rightarrow I \ a_1 \ldots a_k \]

it is well-formed when:

- \( I \ a_1 \ldots a_k \) is well-formed w.r.t. the uniformity of parametric arguments and typing constraints;
- \( I \) does not appear in any of the \( a_1, \ldots, a_k \);
- Each \( C_i \) should either not refer to \( I \) or be of the form:

\[ C'_{1} \rightarrow \ldots \rightarrow C'_m \rightarrow I \ b_1 \ldots b_k \]

well typed and with no other occurrence of \( I \).

And the rule generalizes as such to dependent products (instead of arrow). Said otherwise:

\[ c : (\forall \Gamma_i, C_i) \rightarrow I \ a_1 \ldots a_k \text{ where } I \notin FV(\Gamma_i) \text{ and } \]

\[ C_i = \begin{cases} I \ b_1 \ldots b_k & \text{where } I \notin FV(b_1 \ldots b_k) \\ T & \text{otherwise} \end{cases} \]
There are more constraints, that will be explained later:

1. **predicativity/impredicativity**
   An inductive is predicative when all constructor argument types live in a sort not bigger than the declared sort for the inductive.

2. **restriction on eliminations**
   when the predicativity condition is not satisfied
Girard’s paradox:

- **Type : Type**
- Generalizes to $X : \text{Type}$ with an embedding $\text{Type} \rightarrow X$

**Inductive** $e \ (A : s_1) : s_2 := C \ (_ : A)$.

- $C : A \rightarrow e(A)$
- pattern-matching: $e(A) \rightarrow A$
- reduction: $C$ and pattern-matching are inverses

If $s_2 : s_1$, the paradox applies...

Conclusion: *inductive definitions must be predicative, otherwise eliminations must be restricted* (see Paulin’s Habilitation thesis)
Dependent pattern-matching

**Inductive** \( I \ (p: \text{Par}) : A \rightarrow s := \)

\[ | \ k \ (x_1:C_1) \ldots (x_n:C_n) : I \ p \ u \]

\[ | \ldots \]

**match** \( t \text{ as } h \text{ in } I \_ \ a \text{ return } P(a,h) \text{ with} \)

\[ | \ k \ x_1 \ldots x_n => e \]

\[ \ldots \]

**end**

**Typing conditions:**

\[ \vdash t : I \ q \ b \]

\[ a : A[q/p], h : I \ q \ a \vdash P : s' \]

\[ x_1 : C_1[q/p], \ldots, x_n : C_n[q/p] \vdash e : P(u[q/p], k \ q \ x_1 \ldots x_n) \]

Then the match has type \( P(b, t) \)
Tactics for case analysis

- `case t` is the most primitive. It:
  - generates a (proof) term of the form `match t with ...`;
  - guesses the return type from the goal (under the line);
  - does not introduce/name the arguments of the constructor by default, but there is a syntax for choosing names.

- The `case_eq` variant modifies the guessing of the return type so that equalities are generated.

- The `destruct` variant modifies the guessing of the return type so that it generalizes the hypotheses depending on `t`. The `destruct t eqn:H` variant allows to keep an equality `H` as well between `t` and each pattern.
The fixpoint operator (reduction)

Fixpoint expression with dependent result

\[(\text{fix } f \ (x : A) : B(x) := t(f, x))\]

- Typing

\[
\begin{align*}
& f : (\forall (x : A), B(x)), \ x : A \vdash t : B(x) \\
\therefore & (\text{fix } f \ (x : A) : B(x) := t(f, x)) : \forall (x : A), B(x)
\end{align*}
\]
Fixpoint operator : well-foundness

Requirement of the Calculus of Inductive Constructions :

- the argument of the fixpoint has type an inductive definition
- recursive calls are on arguments which are structurally smaller

Example of recursor on natural numbers

\[
\begin{align*}
\lambda P &: \text{nat} \to \text{s}, \\
\lambda H_O &: P(O), \\
\lambda H_S &: \forall m : \text{nat}, P(m) \to P(S m), \\
\text{fix } f (n : \text{nat}) &: P(n) := \\
& \text{match } n \text{ as } y \text{ return } P(y) \text{ with} \\
& O \Rightarrow H_O | S m \Rightarrow H_S m (f m) \\
& \text{end}
\end{align*}
\]

is correct with respect to CCI : recursive call on \( m \) which is structurally smaller than \( n \) in the inductive \text{nat}. 
Fixpoint operator : typing rules

\[\Gamma \vdash \text{fix } f (x : I) : C := t : \forall x : I, C\]

the main rules for \(\text{fix}_{f}^{\rho} x \) are:

\[
\begin{align*}
\text{match } z u_1 \ldots u_n \text{ return } A \text{ with } (c_i \bar{x}_i \Rightarrow t_i)_i \text{ end}_{f}^{\rho} x \\
t \neq (z \bar{u}) \text{ for } z \in \rho \cup \{x\} \quad t_{f}^{\rho} x \quad A_{f}^{\rho} x \quad (t_{i}^{\rho}_{f} x \text{ for } i)\\
y \in \rho \quad f (y \ u_1 \ldots u_n)_{f}^{\rho} x \\
f \not\in FV(t) \quad t_{f}^{\rho} x
\end{align*}
\]

+ contextual rules …
Remarks on the criteria

- It covers simply the schema of primitive recursive definitions and proofs by induction which have recursive calls on all subterms.

\[
\lambda P : \text{list } A \to s,
\lambda f_1 : P \text{ nil},
\lambda f_2 : \forall (a : A)(l : \text{list } A), P l \to P (\text{cons } a l),
\text{fix } \text{Rec } (x : \text{list } A) : P x :=
\quad \text{match } x \text{ return } P x \text{ with}
\quad \text{nil } \Rightarrow f_1 | (\text{cons } a l) \Rightarrow f_2 a l (\text{Rec } l)
\text{end}
\]

- has type

\[
\forall P : \text{list } A \to s,
P \text{ nil}, \to
(\forall (a : A)(l : \text{list } A), P l \to P (\text{cons } a l)) \to
\forall (x : \text{list } A), P x
\]
Remarks on the criteria

Possibility of recursive call on deep subterms

Fixpoint mod2 (n:nat) : nat :=
    match n with
    O => O | S O => S O
    | S (S x) => mod2 x
    end

Possibility of recursive call on terms build by case analysis if each branch is a strict subterm (actual rule very complex!):

Definition pred (n:nat) : n<>0->nat:=
    match n return n<>0->nat with
    S p => (fun (h:S p<>0) => p)
    | O => (fun (h:0<>0) =>
            match h (refl_equal 0) return nat with end)
    end

Fixpoint F (n:nat) : C :=
    match iszero n with
    (left H (*H:n=0*)) => ...
    | (right H (*H:n<>0*)) => F (pred n H)
    end
Remarks on the criteria

Note: only the recursive arguments with the same type are considered recursive (otherwise paradox related to impredicativity)

Definition ID : Prop := forall (A:Prop), A → A.
Definition id : ID := fun A x => x.
Inductive Singl : Prop := c (_:ID). (* non recursive *)
Fixpoint f (x : Singl) : bool :=
    match x with (c a) => f (a Singl x) end.

\[ f (c id) \xrightarrow{\lambda} f (id Singl (c id)) \xrightarrow{\beta} f (c id) \]
Tactics for induction

\texttt{fix } \langle n \rangle, \textbf{where } \langle n \rangle \textbf{ is a numeral is the most primitive. It:}

- generates a (proof) term of the form:

\[
\begin{align*}
\text{fun } g_1 \ g_2 & \Rightarrow \text{fix } f \ h_1 \ h_2 \ t \ h_3 \ \{ \text{struct } t \} := \ ?F \ h_1 \ h_2 \ t
\end{align*}
\]

where:

- \(g_1, g_2\) are the objects in the context (above the line);
- \(h_1, h_2, t, h_3\) are the objects quantified in the goal (under the line);
- \(?F\) can call \(f\) (= recursive calls);
- the termination of \(f\) is should eventually be guaranteed by structural recursion on \(t\);

\texttt{Qed} checks the well-formedness, which was not guaranteed so far: error messages come late and may be difficult to interpret.
Tactics for induction

```
elim t applies an induction scheme, i.e. a lemma of the form:
forall P : T -> Type, .... -> forall t' : T, P t'
```

- It guesses argument $P$ from the goal (under the line),
  abstracting all the occurrences of $t$.
- It guesses the elimination scheme to be used ($T_{\text{ind}},$
  $T_{\text{rect}},...$) from the sort of the goal and the type of $t$.
- The `elim t using S` variant allows to provide a custom
  elimination scheme (or lemma!) $S$, with the same
  unification heuristic.
- The `induction t` tactic guesses argument $P$ taking into
  account the possible hypotheses depending on $t$ present
  in the context (above the line). Plus it can introduce and
  name things automatically.

Remark: the `rewrite` tactic does a similar guessing job...
We would expect the usual expansion rule for fixpoints:

$$\text{fix } f \ (x:A): B \ x := t(f,x) \ e \rightarrow t(\text{fix } f \ (x:A): B \ x := t(f,x), \ e)$$
Fixpoint expansion

We would expect the usual expansion rule for fixpoints:

\[(\text{fix } f \ (x:A): B \ x \ := \ t(f,x))e\]
\[\rightarrow t(\text{fix } f \ (x:A): B \ x \ := \ t(f,x), \ e)\]

... but this leads to infinite unfolding (SN broken)
We would expect the usual expansion rule for fixpoints:

\[
(fix \ f \ (x:A): B \ x := t(f,x))e \\
\rightarrow t(fix \ f \ (x:A): B \ x := t(f,x), e)
\]

... but this leads to infinite unfolding (SN broken)

Solution: allow this reduction only when \( e \) is a constructor

Beware:

- Guard condition ensures consistency (meaningful definition)
- Expansion restriction imposes a strategy
Next week...

Advanced features of inductive types

- Prop vs Type
- Impredicative inductive definitions