Recall from lecture 5

- The \textit{Prop} sort and extraction;
- The accessibility predicate and mathematical induction.
- A constructive variant of Markov’s principle
Today

1. Mathematical Structures
   - Record Types
   - Exercise

2. Elaboration: Coercions, Unification and Implicit Arguments
   - Coercions
   - Exercise

3. Type Classes and Modular Developments
   - Modules vs Type Classes in Coq
   - Type Classes in theory
   - Type Classes in practice
   - Variations on type classes
   - Additional exercises

Goal: familiarize yourself with records and generic formalization techniques along with the elaboration features of Coq.
Record Types

Record \( R \ (A : \text{Type}) : s := \)
\{ \( f_1 : \text{tau}_1; \ldots; f_n : \text{tau}_n \) \}.

Inductive \( R \ (A : \text{Type}) : s := \)
\text{Build}_R (f_1 : \text{tau}_1) \ldots (f_n : \text{tau}_n) : R \ A.

No indices, no recursion

Projection is pattern-matching

Definition \( fi \ (A : \text{Type}) \ (r : R \ A) : \text{tau}_i[fi \ r/fi] := \)
match \( r \) with \text{Build}_R t_1 \ldots t_n => t_i \end{.
Records II

- **Definitional Equality:**
  \[ fi \ (\text{Build}_\text{rec} \ t_1 .. \ t_n) \rightarrow_{\beta\delta\eta} t \]

- **No \(\eta\) law:**
  \(\text{Build}_\text{R} \ (f_1 \ t) .. (f_n \ t) \equiv_{\beta\delta\eta} t\) **is not valid** (unless using primitive projections).

- Only derivable for Leibniz equality.

An example declaration:

```coq
definitions
Record positive := { pos_nat : nat;
  pos_proof : 0 < pos_nat }.
```

Coq defines:

```coq
definitions
pos_nat : positive \rightarrow nat
pos_proof : \forall (p : positive), 0 < pos_nat p
```
Warm-up demo:

- Define the dependent pair type (a.k.a. sigma types for $A : \text{Type}$ and $P : A \to \text{Type}$) as a record.
- Prove the $\eta$ rule (surjective pairing) for it.
- Define $\mathbb{Q}$ as a record of two integers.
- Define the equivalence of two rationals as a proposition.
- Define addition of two rationals and show that it respects the equivalence (on pairwise equivalent arguments, it gives equivalent results).
Mathematical Structures

Defining composite structures as first-class objects:

Record monoid := { carrier : Type; unit : carrier;
   op : carrier -> carrier -> carrier;
   left_unit : forall x, op x unit = x; ... }.

Generic programming for any monoid:

Definition monsquare (m : monoid)
   (x : carrier m) : carrier m := op m x x.

Type and value dependency is essential to make this typecheck.
Mathematical Structures Exercise (20 min)

- Define the full interface of a monoid given partially above.
- Define the \((0, +)\) and \((1, \ast)\) monoids on \(\mathbb{N}\). You can use \texttt{lia} to solve the goals for laws. (\texttt{Require Import Lia first})
- Define a generic exponentiation operation \(a^n\) where \(n : \mathbb{N}\) and \(a\) is in a monoid.
- Show that the exponentiation of the unit is always equal to the unit.
Elaboration

Type theory is **verbose**: use an elaboration from source syntax ("expressions") to core terms.

- User-syntax: \([3 :: 1]\)
- Low-level terms:
  \[@(\text{cons}, [\text{nat}; @(@(@(@(@(@(@(@(@(@('S', [(@('S', [(@('S', [('O']))))))))))))))))))\]

To tame this:

- **Notations**: abbreviations of combined terms and a complex parser. User-extensible, e.g.:
  
  **Notation** ""\(\sum\) x .. y , P" := \((\sigma \text{ma} (\text{fun} x \Rightarrow .. \,(\sigma \text{ma} \text{(fun} y \Rightarrow P)) ..))\)

- **Coercions**: implicit casting of one type into another
- **Terms with holes** (aka existential variables) and unification
Coercions

Record monoid := { carrier : Type;
                 unit : carrier; op : carrier -> carrier -> carrier }.

Generic programming for any monoid:
Coercion carrier : monoid --> Sortclass.

Compare: without the carrier coercion:

    Definition monsquare (m : monoid)
        (x : carrier m) : carrier m := op m x x.

With the carrier coercion:

    Definition monsquare (m : monoid)
        (x : m) : m := op m x x.

- The term m is coerced to a sort (its carrier) in the types.
- Closer to informal mathematical notation
Coercions

- Coercions are not displayed (by default, see `Set Printing Coercions`)
- The actual core term is the same as before
- Coherence of coercions (only one path up to conversion between two types) and decidability ensured by a syntactic check.
- Uses: types embedded in records, injections (i.e. from N to Z), modeling a subtyping relation...

```
Coercion Z_of_nat : nat >-> Z.
```
Exercise

Mathematical Structures and Coercions: exercise (10min)

- Define the coercion from \texttt{nat} to \texttt{bool}, sending 0 to \texttt{false} and everything else to the \texttt{true} constructor.
- Define the converse coercion from \texttt{bool} to \texttt{nat}, giving value 1 to \texttt{true}.
- Show that \texttt{bool} \(\rightarrow\) \texttt{nat} \(\rightarrow\) \texttt{bool} is the identity (pointwise).
- Which one makes more sense as a coercion?
Exercise

Unification

Add a new term former and environment to the calculus: existential variables / metavariables.

\[(X : \Delta \vdash \tau) \in \Sigma\]
\[
\Sigma; \Gamma \vdash \sigma : \Delta
\]
\[
\Sigma; \Gamma \vdash ?X[\sigma] : \tau[\sigma]
\]

- Existential variables are shared among a global state \(\Sigma\) that declares their types and optionally, their instance.
- \(\sigma\) is a substitution for the variables of the existential.

Example:

Goal forall (x : nat), exists y, x < y.
Proof.

eexists.
(* x : nat
  ______________
  x < ?y[x] *)
Unification

Existential variables can be instantiated by well-typed instances:

\[
\frac{(X : \Delta \vdash \tau) \in \Sigma \\
\Sigma ; \Delta \vdash t : \tau}{\Sigma [X := t : \Delta \vdash \tau] ; \Gamma \vdash}
\]

Example:

Goal \( \forall (x : \text{nat}), \exists y, x < y \).

Proof.

\begin{verbatim}
eexists.
instantiate (1 := S x).
(* x : \text{nat}
   ___________
   x < S x *)
\end{verbatim}

BEWARE: the instance is typed in the context of the existential, NOT the current goal.
Once we have existential variables we can formalize unification:

\[ \Sigma; \Gamma \vdash t \simeq u \leadsto \Sigma' \]

Informally: match-up terms \( t \) and \( u \), assigning holes in one term with the corresponding subterm of the other one, producing an instantiation in \( \Sigma' \).

Using first-order heuristics:

\[ f ?x \simeq f y \leadsto ?x := y \]

Up to reduction:

\[ 1 + ?x \simeq 4 \leadsto ?x := 3 \]

Not up to equations:

\[ ?P \land Q \land R \simeq R \land Q \land S \not\leadsto \]
Unification II

- Is a congruence
- Includes $\beta\delta\iota\zeta\mu\nu$-conversion: makes it powerful but also unpredictable sometimes
- Includes instantiation:

$$\Sigma; \Gamma \vdash ?X[\sigma] \simeq t \leadsto \Sigma[X := t[\sigma^{-1}]]$$

(under some conditions, notable an occurs-check).

- Is undecidable outside 1st order and pattern fragments ("simple" unification problems). Huet’78: The Undecidability of Unification in Third Order Logic.
Unification for tactics

Unification is at the basis of most tactics. Let’s look at `apply`:

- Assume goal $\Gamma \vdash G$, and lemma $l : \forall x : A, C x \to B x$.
- `apply l` builds $l (?a[id_\Gamma] : A)(?c : C ?a[id_\Gamma]) : B ?a[id_\Gamma]$
- Tries to unify $B ?a[id_\Gamma]$ with $G$.
- If it succeeds, filling $?a$ with some term $t$, it generates a goal $C ?a[id_\Gamma] \equiv C t[id_\Gamma] \equiv C t$.

Example:

```latex
Goal forall (x : nat), exists y, x < y.
Proof.
  eexists. (* x : nat |- x < ?y *)
  eapply (leS (* ?z < S ?z *)).
    (* Unifies ?z with x and ?y with S ?z = S x *)
Qed.
```
Unification for implicits

Implicit arguments:
At each application of a constant, turn the argument into an existential variable to be resolved.

Example:

cons : \forall \{A\}, A \to \text{list} A \to \text{list} A
nil : \forall \{A\}, \text{list} A

Source: cons 1 nil : list ?X
**Unification for implicants**

**Implicit arguments:**
At each application of a constant, turn the argument into an existential variable to be resolved.

**Example:**

\[
\text{cons} : \forall \{A\}, \ A \rightarrow \text{list} \ A \rightarrow \text{list} \ A \\
\text{nil} : \forall \{A\}, \\text{list} \ A
\]

- Source: \text{cons} 1 \text{nil} : \text{list} \ ?X
- Desugar: \text{cons} \ (S \ O) \text{nil} : \text{list} \ ?X
Unification for implicits

**Implicit arguments:**
At each application of a constant, turn the argument into an existential variable to be resolved.

**Example:**

```plaintext
cons : \{A\}, A \rightarrow \text{list } A \rightarrow \text{list } A
nil : \{A\}, \text{list } A
```

- **Source:** `cons 1 nil : list ?X`
- **Desugar:** `cons (S O) nil : list ?X`
- **Implicits:** `@cons _ (S O) (@nil _) : list ?X`
Unification for implicit arguments

**Implicit arguments:**
At each application of a constant, turn the argument into an existential variable to be resolved.

**Example:**

```plaintext
cons : \forall \{A\}, A \rightarrow list A \rightarrow list A  
nil : \forall \{A\}, list A
```

- Source: `cons 1 nil : list ?X`
- Desugar: `cons (S O) nil : list ?X`
- Implicits: `@cons _ (S O) (@nil _) : list ?X`
- Type-checking:
  `@cons (?A : Type) (S O) (@nil (?B : Type)) : list ?X`

Unification for implicits

Implicit arguments:
At each application of a constant, turn the argument into an existential variable to be resolved.

Example:

```
cons : \forall \{A\}, A \to list A \to list A
nil : \forall \{A\}, list A
```

- Source: `cons 1 nil : list ?X`
- Desugar: `cons (S O) nil : list ?X`
- Implicits: `@cons _ (S O) (@nil _) : list ?X`
- Type-checking:
  `@cons (?A : Type) (S O) (@nil (?B : Type)) : list ?X`
- Type-checking and Unification:
  `@cons ?A (S O) (@nil ?A) : list ?X \leadsto A := nat; ?X := A`

Unification for implicit arguments

Implicit arguments:
At each application of a constant, turn the argument into an existential variable to be resolved.

Example:

\[
\text{cons} : \forall \{A\}, A \to \text{list} \ A \to \text{list} \ A \\
\text{nil} : \forall \{A\}, \text{list} \ A
\]

- Source: \texttt{cons 1 nil : list ?X}
- Desugaring: \texttt{cons (S O) nil : list ?X}
- Implicit arguments: \texttt{@cons _ (S O) (@nil _): list ?X}
- Type-checking:
  \[
  \texttt{@cons (?A : Type) (S O) (@nil (?B : Type)): list ?X}
  \]
- Type-checking and unification:
  \[
  \]
- Final term \texttt{@cons nat (S O) (@nil nat): list nat}
1 Mathematical Structures

2 Elaboration: Coercions, Unification and Implicit Arguments

3 Type Classes and Modular Developments
   - Modules vs Type Classes in Coq
   - Type Classes in theory
   - Type Classes in practice
   - Variations on type classes
   - Additional exercises
Different concepts

Modules are:

- Second-class:

  Module Type Set. Parameter elt : Type. ...


Good for:

- Namespace management
- Implementation hiding (type and value abstraction)
- Seldom used in mathematical developments, more common in computer science developments

Implementation note: a considerable part of the Coq kernel.
Different concepts

**Type-classes:**

- First-class citizens of the core type theory:
  
  Class Set (A : Type) := ...
  
  Record Set (A : Type) := ...

- Haskell-style: proof-search for structures, restricted specialization `Set nat`.

- Overloading (compile-time binding of names).
- Lightweight generic code (no functorization needed).
- Closer to informal math practice.

Records are the standard representation of algebraic structures in type theories.
Enhancing type inference

Overloading:

- **Generic programming** with interfaces instead of implementations.
- **Generic proving**: refer to semantic concepts rather than specific names. E.g. `reflexivity (R:=R)` instead of `R_refl`.
- We use arbitrary proof search to find instances and resolve overloading, mimicking math practice.

Inference of *arbitrary* additional structure on types or values.
Making *ad-hoc* polymorphism less *ad hoc*

In *HASKELL*, Wadler & Blott, POPL’89.
In *ISABELLE*, Nipkow & Snelting, FPCA’91.

```haskell
class Eq a where
    (==) :: a → a → Bool
```
Making *ad-hoc* polymorphism less *ad hoc*

In **HASKELL**, Wadler & Blott, POPL’89.
In **ISABELLE**, Nipkow & Snelting, FPCA’91.

```haskell
class Eq a where
  (==) :: a → a → Bool

instance Eq Bool where
  x == y = if x then y else not y
```
Making *ad-hoc* polymorphism less *ad hoc*

In HASKELL, Wadler & Blott, POPL’89.
In ISABELLE, Nipkow & Snelting, FPCA’91.

```haskell
class Eq a where
  (==) :: a → a → Bool

instance Eq Bool where
  x == y = if x then y else not y
```

```haskell
in :: Eq a ⇒ a → [a] → Bool
in x [] = False
in x (y : ys) = x == y || in x ys
```
Parameterizing

Parametrized instances

instance (Eq a) ⇒ Eq [a] where
  [] == [] = True
  (x : xs) == (y : ys) = x == y && xs == ys
  _ == _ = False
Parameterizing

Parametrized instances

```haskell
instance (Eq a) ⇒ Eq [a] where
    [] == [] = True
    (x : xs) == (y : ys) = x == y && xs == ys
    _ == _ = False
```

Super-classes

```haskell
class Num a where
    (+) :: a → a → a …

class (Num a) ⇒ Fractional a where
    (/) :: a → a → a …
```
3 Type Classes and Modular Developments

- Modules vs Type Classes in Coq
- Type Classes in theory
  - Motivation and setup
  - Type Classes from Haskell
  - Type Classes in Coq
- Type Classes in practice
- Variations on type classes
- Additional exercises
A cheap implementation

- Parametrized dependent records

\[
\text{Class } \text{Id } (\alpha_1 : \tau_1) \cdots (\alpha_n : \tau_n) := \\
\{ f_1 : \phi_1 ; \cdots ; f_m : \phi_m \}.
\]
A cheap implementation

- Parametrized dependent records

\[
\text{Record } \operatorname{Id} (\alpha_1 : \tau_1) \cdots (\alpha_n : \tau_n) := \\
\{ f_1 : \phi_1 ; \cdots ; f_m : \phi_m \}.
\]
A cheap implementation

- Parametrized dependent records
  \[
  \text{Record } \text{Id} \ (\alpha_1 : \tau_1) \cdots (\alpha_n : \tau_n) := \\
  \{ f_1 : \phi_1 ; \cdots ; f_m : \phi_m \}.
  \]
  Instances are just definitions of conclusion \text{Id} \ t_n.
A cheap implementation

- Parametrized dependent records
  \[
  \text{Record } \text{Id} \ (\alpha_1 : \tau_1) \cdots (\alpha_n : \tau_n) := \\
  \{ f_1 : \phi_1 ; \cdots ; f_m : \phi_m \}.
  \]
  Instances are just definitions of conclusion \( \text{Id} \ t_n \).

- Custom implicit arguments of projections
  \[
  f_1 : \forall \overrightarrow{\alpha_n : \tau_n} , \ \text{Id} \overrightarrow{\alpha_n} \rightarrow \phi_1
  \]
A cheap implementation

- Parametrized dependent records

  \[ \text{Record } \text{Id} \ (\alpha_1 : \tau_1) \cdots (\alpha_n : \tau_n) := \{ f_1 : \phi_1 ; \cdots ; f_m : \phi_m \}. \]

  Instances are just definitions of conclusion \( \text{Id} \ t_n \).

- Custom implicit arguments of projections

  \[ f_1 : \forall \{ \alpha_n : \tau_n \}, \{ \text{Id} \ \alpha_n \} \rightarrow \phi_1 \]
Elaboration with classes, an example

\[ \lambda x \ y : \text{bool. } \text{eqb} \ x \ y \]
Elaboration with classes, an example

\[ \lambda x \, y : \text{bool. } \text{eqb} \; x \; y \]

\[ \rightsquigarrow \{ \text{Implicit arguments} \} \]

\[ \lambda x \, y : \text{bool. } @\text{eqb} \; _\_ \; x \; y \]
Elaboration with classes, an example

\[
\lambda x \ y : \text{bool}. \ eqb \ x \ y
\]

\[\leadsto \{ \text{Implicit arguments} \}\]

\[
\lambda x \ y : \text{bool}. \ @eqb \_ \_ \ x \ y
\]

\[\leadsto \{ \text{Typing} \}\]

\[
\lambda x \ y : \text{bool}. \ @eqb \ (?A : \text{Type}) \ (?eq : \text{Eq} \ ?A) \ x \ y
\]
Elaboration with classes, an example

\[ \lambda x \ y : \text{bool. } \text{eqb } x \ y \]
\[ \Rightarrow \{ \text{Implicit arguments} \} \]
\[ \lambda x \ y : \text{bool. } \text{eqb} \_ \_ x \ y \]
\[ \Rightarrow \{ \text{Typing} \} \]
\[ \lambda x \ y : \text{bool. } \text{eqb} \ (?A : \text{Type}) \ (?eq : \text{Eq } ?A) \ x \ y \]
\[ \Rightarrow \{ \text{Unification} \} \]
\[ \lambda x \ y : \text{bool. } \text{eqb bool} \ (?eq : \text{Eq bool}) \ x \ y \]
Elaboration with classes, an example

\[ \lambda x \ y : \text{bool}. \ \text{eqb} \ x \ y \]
\[ \rightsquigarrow \{ \text{Implicit arguments} \} \]
\[ \lambda x \ y : \text{bool}. \ \text{@eqb} \ _\ _ \ x \ y \]
\[ \rightsquigarrow \{ \text{Typing} \} \]
\[ \lambda x \ y : \text{bool}. \ \text{@eqb} \ (?_A : \text{Type}) \ (?_eq : \text{Eq} \ ?_A) \ x \ y \]
\[ \rightsquigarrow \{ \text{Unification} \} \]
\[ \lambda x \ y : \text{bool}. \ \text{@eqb bool} \ (?_eq : \text{Eq bool}) \ x \ y \]
\[ \rightsquigarrow \{ \text{Proof search for Eq bool returns Eq_bool} \} \]
\[ \lambda x \ y : \text{bool}. \ \text{@eqb bool} \ \text{Eq_bool} \ x \ y \]
Type Class resolution

Proof-search tactic with instances as lemmas:

\[ A : \text{Type},\ eqa : \text{Eq}\ A \vdash \ ? : \text{Eq}\ (\text{list}\ A) \]

- Simple depth-first search with higher-order unification
  - Returns the first solution only
- Extensible through LTAC
1 Mathematical Structures

2 Elaboration: Coercions, Unification and Implicit Arguments

3 Type Classes and Modular Developments
   - Modules vs Type Classes in CoQ
   - Type Classes in theory
   - Type Classes in practice
     - Playing with numbers
     - Instance Inference
     - Dependent Classes
     - A programming example: Generic exponentiation
     - Monadic Programming with Type Classes
     - Exercise
     - Logic programming: Reification
   - Variations on type classes
   - Additional exercises
Numeric overloading

Class \texttt{Num }\(\alpha\) := \{ \text{zero} : \alpha \; ; \text{one} : \alpha \; ; \text{plus} : \alpha \rightarrow \alpha \rightarrow \alpha \}. \]
Numeric overloading

\textbf{Class} \texttt{Num} \( \alpha \) \(:=\) \{ \( \text{zero} \) : \( \alpha \) ; \( \text{one} \) : \( \alpha \) ; \( \text{plus} \) : \( \alpha \rightarrow \alpha \rightarrow \alpha \) \}. \\
\textbf{Instance} \texttt{nat\_num} : \texttt{Num} \texttt{nat} := \\
\hspace{1em} \{ \( \text{zero} \) := 0\%nat ; \( \text{one} \) := 1\%nat ; \( \text{plus} \) := \texttt{Peano\_plus} \}. \\
\textbf{Instance} \texttt{Z\_num} : \texttt{Num} \texttt{Z} := \\
\hspace{1em} \{ \( \text{zero} \) := 0\%Z ; \( \text{one} \) := 1\%Z ; \( \text{plus} \) := \texttt{Zplus} \}. \\
\text{Notation} \texttt{"0"} := \texttt{zero}. \\
\text{Notation} \texttt{"1"} := \texttt{one}. \\
\text{Infix} \texttt{"+"} := \texttt{plus}. \\
\text{Check} (\lambda x: \texttt{nat}, x + (1 + 0 + x)). \\
\text{Check} (\lambda x: \texttt{Z}, x + (1 + 0 + x)). \\
\text{(* Allows Defaulting *) \text{Check} (\lambda x, x + 1). \texttt{Z \rightarrow Z}}. \\
\text{(* Or preventing ambiguity *) \text{Hint} \text{Mode} \texttt{Num +} : \texttt{typeclass}}.
Numeric overloading

Class **Num** \(\alpha\) := \{ zero : \alpha ;\ one : \alpha ;\ plus : \alpha \rightarrow \alpha \rightarrow \alpha \}.

Instance **nat_num** : Num nat :=

\{ zero := 0\%nat ; one := 1\%nat ; plus := Peano.plus \}.

Instance **Z_num** : Num \(\mathbb{Z}\) :=

\{ zero := 0\%\(\mathbb{Z}\) ; one := 1\%\(\mathbb{Z}\) ; plus := Zplus \}.

Notation "0" := zero. Notation "1" := one. Infix "+" := plus.
Numeric overloading

Class **Num** \( \alpha \) := \{ zero : \alpha ; one : \alpha ; plus : \alpha \to \alpha \to \alpha \}.

Instance **nat**_num : Num nat :=

\{ zero := 0\%nat ; one := 1\%nat ; plus := Peano.plus \}.

Instance **Z**_num : Num Z :=

\{ zero := 0\%Z ; one := 1\%Z ; plus := Zplus \}.

Notation "0" := zero. Notation "1" := one. Infix "+" := plus.

Check (\( \lambda x : \text{nat}, x + (1 + 0 + x) \)).
Check (\( \lambda x : \text{Z}, x + (1 + 0 + x) \)).
Numeric overloading

Class **Num** $\alpha := \{ \text{zero} : \alpha ; \text{one} : \alpha ; \text{plus} : \alpha \rightarrow \alpha \rightarrow \alpha \}.$

Instance **nat_num** : **Num** nat :=
   \{ zero := 0%nat ; one := 1%nat ; plus := Peano.plus \}.

Instance **Z_num** : **Num** Z :=
   \{ zero := 0%Z ; one := 1%Z ; plus := Zplus \}.

Notation "0" := zero. Notation "1" := one. Infix "+" := plus.

Check ($\lambda x : \text{nat}, x + (1 + 0 + x))$.
Check ($\lambda x : \text{Z}, x + (1 + 0 + x))$.

(* Allows Defaulting *) Check ($\lambda x, x + 1). \text{Z} \rightarrow \text{Z}$
(* Or preventing ambiguity *)

**Hint** **Mode** **Num** + : typeclass_instances.
**Fail** **Definition** foo := (fun x ⇒ x + 1).
Instance inference

Class `EqDec A := eq_dec : \( \forall x y : A, \{x = y\} + \{x \neq y\}\).`

Instance: `EqDec nat := \{ eq_dec := eq_nat_dec \}.`
Instance: `EqDec bool := \{ eq_dec := bool_dec \}.`
Instance inference

Class EqDec $A := \text{eq\_dec} : \forall x \ y : A, \{x = y\} + \{x \neq y\}$.

Instance: EqDec nat := \{ \text{eq\_dec} := \text{eq\_nat\_dec} \}.
Instance: EqDec bool := \{ \text{eq\_dec} := \text{bool\_dec} \}.

Program Instance: $\forall A, \text{EqDec }A \rightarrow \text{EqDec } (\text{option }A) := \{
\text{eq\_dec }x \ y := \text{match }x, y\text{ with}
| \text{None, None } \Rightarrow \text{in\_left}
| \text{Some }x, \text{Some }y \Rightarrow \text{if eq\_dec }x \ y\text{ then in\_left else in\_right}
| \_ , \_ \Rightarrow \text{in\_right}
end \}.$
Instance inference

Class `EqDec A := eq_dec : ∀ x y : A, {x = y} + {x ≠ y}.`

Instance: `EqDec nat := { eq_dec := eq_nat_dec }.`
Instance: `EqDec bool := { eq_dec := bool_dec }.`

Program Instance: `∀ A, EqDec A → EqDec (option A) := { eq_dec x y := match x, y with
  | None, None ⇒ in_left
  | Some x, Some y ⇒ if eq_dec x y then in_left else in_right
  | _, _ ⇒ in_right
end }.

Check (λ x : option (option nat), eq_dec x None).
: ∀ x : option (option nat), {x = None} + {x ≠ None}

Eval compute in (eq_dec (Some (Some 0)) (Some None)).
= in_right : {Some (Some 0) = Some None} + {Some (Some 0) ≠ Some None}`
Dependent classes

Class Reflexive \{ A \} (R : relation A) :=
  refl : \forall x, R x x.
Dependent classes

Class Reflexive \{A\} (R : relation A) :=
     refl : \forall x, R x x.

Instance eq_refl A : Reflexive (@eq A) := @refl_equal A.
Instance iff_refl : Reflexive iff.
Proof. red. tauto. Qed.
Dependent classes

Class Reflexive \{A\} (R : relation A) :=
  refl : \forall x, R x x.

Instance eq_refl A : Reflexive (@eq A) := @refl_equal A.

Instance iff_refl : Reflexive iff.
Proof. red. tauto. Qed.

Goal \forall P, P \leftrightarrow P.
Proof. apply refl. Qed.

Goal \forall A (x : A), x = x.
Proof. intros A ; apply refl. Qed.
Class **Reflexive** \{A\} (R : relation A) :=
refl : \( \forall \ x, \ R \ x \ x \).

**Instance** eq_refl A : Reflexive (@eq A) := @refl_equal A.

**Instance** iff_refl : Reflexive iff.
**Proof.** red. tauto. Qed.

**Goal** \( \forall \ P, \ P \leftrightarrow \ P \).
**Proof.** apply refl. Qed.

**Goal** \( \forall \ A (x : A), \ x = x \).
**Proof.** intros A ; apply refl. Qed.

**Lemma** foo \{'Reflexive nat R\} : R 0 0.
**Proof.** intros. apply refl. Qed.
Implicit Generalization

An old convention: the free variables of a statement are implicitly universally quantified. E.g., when defining a set of equations:

\[ x + y = y + x \quad \forall x, y \in \mathbb{N} \]
\[ x + 0 = 0 \quad \forall x \in \mathbb{N} \]
\[ x + S\,y = S\,(x + y) \quad \forall x, y \in \mathbb{N} \]
Implicit Generalization

An old convention: the free variables of a statement are implicitly universally quantified. E.g., when defining a set of equations:

\[
\begin{align*}
    x + y & = y + x \quad \forall x, y \in \mathbb{N} \\
    x + 0 & = 0 \quad \forall x \in \mathbb{N} \\
    x + S y & = S(x + y) \quad \forall x, y \in \mathbb{N}
\end{align*}
\]

We introduce new syntax to automatically generalize the free variables of a given term or binder, as implicit arguments:

\[
\begin{align*}
\Gamma \vdash 't : \text{Type} & \triangleq \Gamma \vdash \Pi_{\text{FV}(t) \setminus \Gamma}, t \\
\Gamma \vdash '(t) : T : \text{Type} & \triangleq \Gamma \vdash \lambda_{\text{FV}(t) \setminus \Gamma}, t \\
(x_i : \tau_i) \{y : T\} & \triangleq (x_i : \tau_i) \{\text{FV}(T) \setminus \overrightarrow{x_i}\} \{y : T\} \\
(x_i : \tau_i) '(y : T) & \triangleq (x_i : \tau_i) \{\text{FV}(T) \setminus \overrightarrow{x_i}\} (y : T)
\end{align*}
\]
Implicit Generalization example

When writing:

\textbf{Lemma} foo ‘\{Reflexive nat } R} \colon R 0 0.

Elaboration introduces a new quantifier for } R: \textbf{ Lemma} foo \{R : relation nat\} \{Reflexive nat } R} \colon R 0 0.
Outline

1. Mathematical Structures

2. Elaboration: Coercions, Unification and Implicit Arguments

3. Type Classes and Modular Developments
   - Type Classes in practice
     - A programming example: Generic exponentiation
The following definition is very naïve, but obviously correct:

```lisp
Fixpoint power (a : Z) (n : nat) :=
    match n with
    | 0%nat ⇒ 1
    | S p ⇒ a × power a p
    end.

eval vm_compute in power 2 40.
= 1099511627776 : Z
```
An efficient tail-recursive version

This is more efficient but relies on a more elaborate property:

Function \textit{binary\_power\_mult} (\textit{acc} \textit{x} : \textit{Z}) (\textit{n} : \textit{nat})
{\textit{measure (fun i \mapsto i) n}} : \textit{Z} :=
match \textit{n} with
  | 0%nat \Rightarrow \text{acc}
  | _ \Rightarrow \text{if Even.even_odd_dec} \textit{n}
    then \text{binary\_power\_mult} \textit{acc} (\textit{x} \times \textit{x}) (\text{div2} \textit{n})
    else \text{binary\_power\_mult} (\textit{acc} \times \textit{x}) (\textit{x} \times \textit{x}) (\text{div2} \textit{n})
end.

Definition \textit{binary\_power} (\textit{x}:\textit{Z}) (\textit{n}:\textit{nat}) :=
\text{binary\_power\_mult} 1 \textit{x} \textit{n}.

Eval vm\_compute in \textit{binary\_power} 2 40.
= 1099511627776 : \textit{Z}

Check \textit{eq\_refl} : (\textit{binary\_power} 2 234 = \textit{power} 2 234).
Questions

- Is `binary_power` correct (w.r.t. `power`)?
Questions

- Is `binary_power` correct (w.r.t. `power`)?
- Is it worth proving this correctness only for powers of integers?
Questions

- Is `binary_power` correct (w.r.t. `power`)?
- Is it worth proving this correctness only for powers of integers?
- And prove it again for powers of real numbers, matrices?
Questions

- Is `binary_power` correct (w.r.t. `power`)?
- Is it worth proving this correctness only for powers of integers?
- And prove it again for powers of real numbers, matrices?

**NO!**

Program with interfaces, here a monoid.
Class Monoid \(\{A:\text{Type}\} (\text{dot} : A \to A \to A) (\text{one} : A) : \text{Type} := \)
\[
\begin{align*}
\text{dot_assoc} & : \forall x y z : A, \text{dot} x (\text{dot} y z) = \text{dot} (\text{dot} x y) z; \\
\text{one_left} & : \forall x, \text{dot} \text{one} x = x; \\
\text{one_right} & : \forall x, \text{dot} x \text{one} = x.
\end{align*}
\]

Operations as parameters, distinguishing Monoid plus 0 and Monoid mult 1.
Implicit Generalization

Quantification on parameters:

**Definition two** \( \{ A \cdot one \} \{ M : @Monoid A \cdot one \} := dot \cdot one \cdot one. \)

Using implicit generalization:

**Generalizable Variables** \( A \cdot one. \)

**Definition three** \( \{ Monoid A \cdot one \} := dot \cdot two \cdot one. \)
Generic notations

Global names for parameters:

Definition `monop` `{Monoid A dot one} := dot.
Definition `monunit` `{Monoid A dot one} := one.

Generic notations:

Infix "×" := monop.
Notation "1" := monunit.
Generic power and binary_power.

Section Power.

Context `{Monoid A ?dot ?one}`.

Fixpoint power (a : A) (n : nat) : A :=
  match n with
  | 0%nat ⇒ 1 ← lookup for a Monoid A ?dot ?one
  | S p ⇒ a × (power a p)
  end.

Lemma power_of_unit : ∀ n : nat, power 1 n = 1.
Proof. ... Qed.
Generic binary exponentiation

Function `binary_power_mult (acc x : A) (n : nat)`

```plaintext
{measure (fun i⇒i) n} : A :=
match n with
    | 0%nat ⇒ acc
    | _ ⇒ if Even.even_odd_dec n
        then binary_power_mult acc (x × x) (div2 n)
        else binary_power_mult (acc × x) (x × x) (div2 n)
end.
```

Definition `binary_power (x : A) (n : nat) :=
    binary_power_mult 1 x n.`

Lemma `binary_spec x n : power x n = binary_power x n.`
Proof. ... Qed.

End Power.
Instantiation

A Monoid instance is just a value of the record type:

\[
\text{Instance } \text{ZMult} : \text{Monoid Zmult } 1\%\mathbb{Z}.
\]

Proof.

\[
\text{split.}
\]

\text{subgoal 1 is:}

\[
\forall x, y, z : \mathbb{Z}, x \times (y \times z) = x \times y \times z
\]

\text{subgoal 2 is:}

\[
\forall x : \mathbb{Z}, 1 \times x = x
\]

\text{subgoal 3 is:}

\[
\forall x : \mathbb{Z}, x \times 1 = x
\]

... Qed.
power is parameterized by a **Monoid** implementation.

About power.

\[ \forall (A : \text{Type}) \ (dot : A \to A \to A) \ (one : A), \ \text{Monoid} \ dot \ one \to A \to \text{nat} \to A \]

*Arguments* \(A\), \(dot\), \(one\), \(H\) are implicit and maximally inserted.

Check `power 2 100`.

```
power Z Z.mul 1 ZMult 2 100 : Z
```

Compute `power 2 100`.

```
= 1267650600228229401496703205376 : Z
```
Instantiation

`power` is parameterized by a `Monoid` implementation.

About `power`.
: \(\forall (A : \text{Type}) (dot : A \rightarrow A \rightarrow A) (one : A), \text{Monoid} \ dot \ one \rightarrow A \rightarrow \text{nat} \rightarrow A\)

Arguments `A`, `dot`, `one`, `H` are implicit and maximally inserted

Set Printing Implicit.
Check `power 2 100`.
@`power Z Z.mul 1 ZMult 2 100 : Z`
power is parameterized by a Monoid implementation.

About power.

\( \forall (A : \text{Type}) \ (dot : A \rightarrow A \rightarrow A) \ (one : A), \ Monoid \ dot \ one \rightarrow A \rightarrow \text{nat} \rightarrow A \)

Arguments \( A, \ dot, \ one, \ H \) are implicit and maximally inserted

Set Printing Implicit.

Check \( \text{power 2 100.} \)

@\text{power Z Z.mul 1 ZMult 2 100 : Z}

Compute \( \text{power 2 100.} \)

\( = 1267650600228229401496703205376 : Z \)
Outline

1. Mathematical Structures

2. Elaboration: Coercions, Unification and Implicit Arguments

3. Type Classes and Modular Developments
   - Type Classes in practice
     - Monadic Programming with Type Classes
Monadic programming: ML vs HASKELL

- **Direct effects in ML:**
  ```ml
  val counter : unit → nat
  let counter =
    let x = ref 0 in
    fun () => let x' = !x in x := x' + 1; x'
  ```

- **Monadic effects in HASKELL:**
  ```haskell
  type State α = nat → α × nat

  return :: α → State α
  return x = \s -> (x, s)
  bind :: State α → (α → State β) → State β
  bind m f = \s -> let (a, s') = m s in f a s'

  get :: State nat
  put :: nat → State ()

  counter :: State nat
  counter = do x' <- get; put (x' + 1); return x'
  ```
In our setting, a monad will be defined as:

Class Monad (M : Type -> Type) := {

    return: forall {A}, A -> M A;

    bind : forall {A B}, M A -> (A -> M B) -> M B;

    bind_assoc {A B f g} (m : M A) :
        bind (bind m f) g = bind m (fun x : A => bind (f x) g);

    bind_return_left : ...; bind_return_right : ... }
Monad Instances

Instances of the monad interface:

- **Identity monad**: `id`
- **Error / partiality monad**: `option`, with `error` action
- **“nondeterminism” monad**: `list` (unit `x = [x]`, bind `m f = concat (map f m)`)
- **Continuations (with `callcc`)**, “computation” ...
Exercise

1. Define \textbf{Monad} as a type class with overloaded bind and return/unit operations.

2. Define monadic multiplication “mult” of type
   \[
   \forall A, M(MA) \rightarrow MA \text{ for any monad } M.
   \]

3. State and prove what is its relation to the return/unit operation.

4. Define the identity monad instance.

5. Define the \textbf{option} instance (return is injection, bind propagates “errors” represented as \textbf{None}), with its "error" action of type \textbf{option} A. You will need the functional extensionality axiom
   (\texttt{Coq.Logic.FunctionalExtensionality}).

6. Define the generic \textbf{mapM} operator for any monad (it processes the effects from left to right):
   \[
   \text{mapM: } \forall A, \text{ list } (m A) \rightarrow m A.
   \]
Boolean formulas

\[\text{Inductive } \text{formula} ::= \]
\[\mid \text{cst} : \text{bool} \to \text{formula} \]
\[\mid \text{not} : \text{formula} \to \text{formula} \]
\[\mid \text{and} : \text{formula} \to \text{formula} \to \text{formula} \]
\[\mid \text{or} : \text{formula} \to \text{formula} \to \text{formula} \]
\[\mid \text{impl} : \text{formula} \to \text{formula} \to \text{formula}. \]
Boolean formulas

Inductive `formula` :=
| `cst` : `bool` → `formula`
| `not` : `formula` → `formula`
| `and` : `formula` → `formula` → `formula`
| `or` : `formula` → `formula` → `formula`
| `impl` : `formula` → `formula` → `formula`.

Fixpoint `interp f` :=
  match `f` with
  | `cst` `b` ⇒ `if b` then `True` else `False`
  | `not` `b` ⇒ ¬ `interp b`
  | `and` `a` `b` ⇒ `interp a` ∧ `interp b`
  | `or` `a` `b` ⇒ `interp a` ∨ `interp b`
  | `impl` `a` `b` ⇒ `interp a` → `interp b`
end.
Class \textbf{Reify} \((prop : \text{Prop}) := \)
\{
  \text{reification} : \text{formula} ;
  \text{reify\_correct} : \text{interp reification} \leftrightarrow prop
\}. 
Reification

Class **Reify** \((prop : Prop) :=\)

\[
\{ \text{reification : formula ; } \text{reify\_correct : interp reification } \leftrightarrow prop \}.\]

Check \((\text{@reification : } \forall \ prop : Prop, \text{Reify } prop \rightarrow \text{formula}).\)

*Arguments* reification prop \{Reify\}.
Reification

Class Reify \((prop : \text{Prop}) :\) :=
\[
\begin{align*}
\{ \text{reification} : \text{formula} ; \\
\text{reify\_correct} : \text{interp reification} \leftrightarrow prop \}
\end{align*}
\]

Check (@reification : \(\forall prop : \text{Prop}, \text{Reify} prop \rightarrow \text{formula}\)).

Arguments reification prop \{Reify\}.

Program Instance true\_reif : Reify True :=
\[
\begin{align*}
\{ \text{reification} := \text{cst true} \}
\end{align*}
\]

Program Instance not\_reif \((Rb : \text{Reify} b) : \text{Reify} (\neg b) :=
\[
\begin{align*}
\{ \text{reification} := \text{not} (\text{reification} b) \}
\end{align*}
\]
Reification

Class Reify \( (\text{prop} : \text{Prop}) := \)
\[
\{ \text{reification} : \text{formula} ;
\quad \text{reify}_{-}\text{correct} : \text{interp \ reification} \leftrightarrow \text{prop} \}.
\]

Check \((@\text{reification} : \forall \text{prop} : \text{Prop}, \text{Reify prop} \rightarrow \text{formula}).\)

**Arguments** reification prop \{Reify\}.

Program Instance \text{true\_reif} : Reify True :=
\[
\{ \text{reification} := \text{cst true} \}.
\]

Program Instance \text{not\_reif} ‘(\(Rb : \text{Reify b}\)) : Reify (\neg b) :=
\[
\{ \text{reification} := \text{not (reification b)} \}.
\]

Example \text{example\_prop} :=
\[
\text{reification ((True \land \neg False) \rightarrow \neg \neg False)}.
\]

Check \((\text{refl\_equal} _ : \text{example\_prop} =
\quad \text{impl (and (cst true) (not (cst false))) (not (not (cst false))))).\)
Implement domain-specific proof-automation:

- Discharge separation logic disjointness side-conditions (Nanevsky et al, ICFP’11)
- Generalized rewriting tactic using proof-search for morphisms (Sozeau, JFR’09)
- Derive continuity, monotonicity conditions...
Outline

1. Mathematical Structures

2. Elaboration: Coercions, Unification and Implicit Arguments

3. Type Classes and Modular Developments
   - Modules vs Type Classes in Coq
   - Type Classes in theory
   - Type Classes in practice
   - Variations on type classes
   - Additional exercises
Variations on type classes

Type Classes implementations:

- In Haskell by Wadler et al. (POPL’89, second class)
- In Isabelle by Nipkow et al. (POPL’93, same)
- In Agda by Devriese and PiebSENS (ICFP’11, non-recursive proof search)
- In Lean and Idris as well

In Coq and Matita:

- Coercive Subtyping and Canonical Structures (Saïbi, POPL’97). Used by Gonthier et al. (TPHOLs’09), Nanevski et al. (ICFP’11).
- Unification hints, a more general framework studied by Asperti et al. (TPHOLs’09).
Another way to use records to represent structures, capable of doing much the same as type classes

Based on unification instead of general proof search.

Scales better than type-classes but requires expert knowledge of unification and type-checking.

At the basis of the Ssreflect methodology and the Mathematical Components library (largest and broadest development of mathematics in Coq and probably any proof assistant to date).

See Mahboubi and Tassi [ITP’13] for a gentle introduction and comparison, and Ziliani et al [ICFP’11] for examples using both representations.
Experiments and formalizations in Coq

- Sets, Maps etc... (LETOUZÉY, LESCUYER ...)
- Domain theory, probability monad (PAULIN, ...)
- Generalized rewriting (SOZEAU, JFR’09)
- ACI rewriting (BRAIBANT & POUS, ITP’11)
- Universal algebra, category theory and computable reals (SPITTERS et al., ITP’10)
- ...

See DeepSpec Summer School lecture on Type Classes by B. C. Pierce for tips and tricks.

https://youtu.be/lj0Hy7nJoQc
Exercise: reification of monoids

We want to write a reflexive tactic for simplifying monoidal expressions in any monoid, inspired by the simplification of boolean expressions we just saw. Complete the partial script in lecture7_monoid_reification.
Optional Exercises

1. Complete the proof of correctness of `fibonacci` and give an efficient implementation based on matrix computations, following the generic exponentiation development from 2: `lecture7_matrices.v`

2. Define the state monad and prove a tree labeling algorithm with it: `lecture7_monad.v`