Type Classes for Mathematics

Eelis van der Weegen
J.w.w. Bas Spitters

Radboud University Nijmegen
Formath EU project

Coq Workshop 2010

Interfaces for mathematical structures:

- Algebraic hierarchy (groups, rings, fields, ...)
- Relations, orders, ...
- Categories, functors, ...
- Algebras over equational theories
- Numbers ($\mathbb{N}$, $\mathbb{Z}$, $\mathbb{Q}$, $\mathbb{R}$, ...)

Need solid representations of these.

Representing interfaces in Coq

Engineering challenges:

- Structure inference
- Multiple inheritance/sharing
- Convenient algebraic manipulation (e.g. rewriting)
- Idiomatic use of notations

Solutions in Coq

Existing solutions:

- Dependent records
- Packed classes (Ssreflect)
- Modules

All of these have problems.

New solution: Use type classes!

Solutions in Coq

Existing solutions:

- Dependent records
- Packed classes (Ssreflect)
- Modules

All of these have problems.

New solution: Use type classes!

Type classes

Implementation in Coq is *first class*:

- *Classes*: records (“dictionaries”)
- Class *instances*: constants of these record types
- ... registered as hints for instance resolution.
- Class *constraints*: implicit parameters
- ... resolved during unification using instance hints.

Bundling

Core principle in our approach:

> Represent algebraic structures as predicates,
>
> ... over fully *unbundled* components.

Fully unbundled:

```
Definition reflexive {A: Type} (R: relation A): Prop
  := Π a, R a a.
```

- Very flexible *in theory*
- Inconvenient *in practice* (without type classes!):
  - Nothing to bind notations to
  - Declaring/passing inconvenient
  - No structure inference
- Hence: existing solutions choose to bundle.

Fully unbundled:

```
Definition reflexive {A: Type} (R: relation A): Prop
  := Π a, R a a.
```

- Very flexible *in theory*
- Inconvenient *in practice* (without type classes!):
  - Nothing to bind notations to
  - Declaring/passing inconvenient
  - No structure inference
- Hence: existing solutions choose to bundle.

Bundling is bad

Fully bundled (the other end of the spectrum):

```
Record ReflexiveRelation: Type :=
  { A: Type;   R: relation A;   proof: Π a, R a a }.
```

Addresses *some* of the problems:

- Structure inference
- Notations
- Declaring/passing

But also introduces new ones:

- Prevents sharing
- Multiple inheritance (diamond problem)
- Long projection paths

Bundling is bad

Fully bundled (the other end of the spectrum):

```
Record ReflexiveRelation: Type :=
  { A: Type;   R: relation A;   proof: Π a, R a a }.
```

Addresses *some* of the problems:

- Structure inference
- Notations
- Declaring/passing

But also introduces new ones:

- Prevents sharing
- Multiple inheritance (diamond problem)
- Long projection paths

Solving problems with type classes

Slightly more interesting example:

```
Record SemiGroup
    (G: Type) (e: relation G) (op: G → G → G): Prop :=
  { sg_setoid: Equivalence e
  ; sg_ass: Associative op
  ; sg_proper: Proper (e ⇒ e ⇒ e) op }.
```

Modifications we make:

1. Make it a type class ("predicate class")
2. Use *operational type classes* for e and op

Solving problems with type classes

Slightly more interesting example:

```
Record SemiGroup
    (G: Type) (e: relation G) (op: G → G → G): Prop :=
  { sg_setoid: Equivalence e
  ; sg_ass: Associative op
  ; sg_proper: Proper (e ⇒ e ⇒ e) op }.
```

Modifications we make:

1. Make it a type class (“predicate class”)
2. Use *operational type classes* for e and op

Solving problems with type classes (cont'd)

Revised SemiGroup:

```
Class Equiv (A: Type) := equiv: relation A.
Class SemiGroupOp (A: Type) := sg_op: A → A → A.

Infix "=" := equiv.
Infix "&" := sg_op.

Class SemiGroup
    (G: Type) {e: Equiv G} {op: SemiGroupOp G}: Prop :=
  { sg_setoid:> Equivalence e
  ; sg_ass:> Associative op
  ; sg_proper:> Proper (e ⇒ e ⇒ e) op }.
```

More syntax

Theorem syntax:

```
Lemma bla `{SemiGroup G}:
  Π x y z: G, x & (y & z) = (x & y) & z.
```

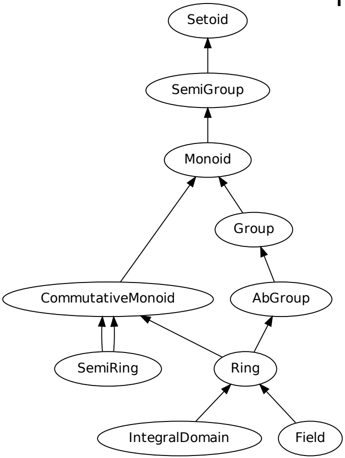
Usage syntax:

```
apply bla
rewrite bla
```

Instance syntax:

```
Instance: Equiv nat := @eq nat.
Instance: SemiGroupOp nat := plus.
Instance: SemiGroup nat.
Proof. ... Qed.
```

Algebraic hierarchy



Features:

- No distinction between axiomatic and derived inheritance.
- No sharing/multiple inheritance problems.
- No rebundling.
- No projection paths (hence, no ambiguous projection paths).
- Instances opaque.
- Terms never refer to proofs.
- Overlapping instances harmless.
- Seamless setoid/rewriting support.

Toward numerical interfaces

Goal: Build theory/programs on *abstract* numerical interfaces instead of concrete implementations.

- Cleaner
- Mathematically sound
- Can swap implementations

Example:

Characterize $\mathbb{N}$ as *initial semiring*.

Need a bit of category theory.

Category theory

Again, begin with operational type classes:

```
Class Arrows (O: Type): Type :=
  Arrow: O → O → Type.
Class CatId O `{Arrows O}: Type :=
  cat_id: `(x ⟶ x).
Class CatComp O `{Arrows O}: Type :=
  comp: Π {x y z}, (y ⟶ z) → (x ⟶ y) → (x ⟶ z).
```

... with notations bound to them:

```
Infix "⟶" := Arrow.
Infix "⊙" := comp.
```

Category theory (cont'd)

```
Class Category (O: Type)
    `{Arrows O} `{Π x y: O, Equiv (x ⟶ y)}
    `{CatId O} `{CatComp O}: Prop :=
  { arrow_equiv:> Π x y, Setoid (x ⟶ y)
  ; comp_proper:> Π x y z,
      Proper (equiv ⇒ equiv ⇒ equiv) comp
  ; comp_assoc w x y z (a: w ⟶ x) (b: x ⟶ y) (c: y ⟶ z):
      c ⊙ (b ⊙ a) = (c ⊙ b) ⊙ a
  ; id_l `(a: x ⟶ y): cat_id ⊙ a = a
  ; id_r `(a: x ⟶ y): a ⊙ cat_id = a }.
```

Next up: Building categories.

Could define category of semirings (etc) manually...

Nicer: *generate* category of equational theory of semirings.

Need a bit of universal algebra.

Universal algebra

We formalize:

- multisorted universal algebra
- equational theories
- categories of algebras, equational theories
- forgetful functors
- open/closed term algebras
- generic construction of initial objects
- subalgebras/varieties, quotients
- theory transference between isomorphic models

All of it using type classes for optimum effect.

Universal algebra (cont'd)

Operational type class:

```
Variables (ϕ: Signature) (carriers: sorts ϕ→ Type).
Class AlgebraOps: Type :=
  algebra_op: Π o: operation ϕ, op_type carriers (ϕ o).
```

Predicate class:

```
Class Algebra
  `{Π  a, Equiv (carriers a)} `{AlgebraOps ϕ carriers}: Prop :=
  { algebra_setoids:> Π a, Setoid (carriers a)
  ; algebra_propers:> Π o: ϕ, Proper (=) (algebra_op o) }.
```

Numerical interfaces

Minimalistic interface for $\mathbb{N}$:

```
Class Naturals (A: ObjectInVariety semiring_theory)
    `{InitialArrows A}: Prop :=
  { naturals_initial:> Initial A }.
```

More convenient:

```
Context `{SemiRing A}.
Class Naturals `{NaturalsToSemiRing A}: Prop :=
  { naturals_ring:> SemiRing A
  ; naturals_to_semiring_mor:> Π `{SemiRing B},
      SemiRing_Morphism (naturals_to_semiring A B)
  ; naturals_initial:> Initial (bundle_semiring A) }.
```

Specialization

Suppose you want to calculate things:

Definition calc ‘{Naturals N} (n m: N) := ... decide (n = m) ...

Generic instance:

Instance: Π ‘{Naturals N} (n m: N): Decision (n = m) | 9 := ...

Works, but inefficient.

Specialized instance for nat:

Instance: Π n m: nat, Decision (n = m).

Extra parameterization:

Definition calc ‘{Naturals N} ‘{Π n m: N, Decision (n = m)}
(a b: nat) := ... decide (a = b) ... .

Specialization

Suppose you want to calculate things:

Definition calc ‘{Naturals N} (n m: N) := ... decide (n = m) ...

Generic instance:

Instance: Π ‘{Naturals N} (n m: N): Decision (n = m) | 9 := ...

Works, but inefficient.

Specialized instance for nat:

Instance: Π n m: nat, Decision (n = m).

Extra parameterization:

Definition calc ‘{Naturals N} ‘{Π n m: N, Decision (n = m)}
(a b: nat) := ... decide (a = b) ... .

Quoting

Quoting:

- Find syntactic representation of semantic expression
- Required for proof by reflection (ring, omega)

Usually implemented at meta-level (Ltac, ML).

Alternative: object level quoting.

- Unification hints (Matita)
- Canonical structures (Ssreflect)

Quoting (cont'd)

Our implementation: type classes!

Instance resolution

- Syntax-directed
- Prolog-style resolution
- Unification-based programming language

Implementation in terms of type classes:

- Straightforward
- Plan: integrate with universal algebra term types

Conclusions

Predicate type classes for mathematics:

- Works well in practice
- Match mathematical practice
- Compatible with efficient computation
- Plan: use as basis for computational analysis (Formath)

Pending issues:

- instance resolution efficiency
- universe polymorphism
- “infer if possible, generalize otherwise”

Sources/papers:

Google keywords: `coq math classes`