Type Classes for Mathematics

Robbert Krebbers
Joint work with Bas Spitters and Eelis van der Weegen[1]

Radboud University Nijmegen

March 31, 2011

[1]The research leading to these results has received funding from the European Union's 7th Framework Programme under grant agreement nr. 243847 (ForMath).

Goal

Build theory and programs on top of abstract interfaces instead of concrete implementations.

- Cleaner.
- Mathematically sound.
- Can swap implementations.

For example:

- Real number arithmetic based on an abstract interface for underlying dense ring.

Interfaces for mathematical structures

We need solid interfaces for:

- Algebraic hierarchy (groups, rings, fields, ...)
- Relations, orders, ...
- Categories, functors, universal algebra, ...
- Numbers: $\mathbb{N}$, $\mathbb{Z}$, $\mathbb{Q}$, ...
- Operations, ...

Interfaces for mathematical structures

Engineering challenges:

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

Solutions in Coq

Existing solutions:

- Dependent records
- Packed classes (Ssreflect)
- Modules

New solution: use type classes!

Fully unbundled

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

Flexible in theory, inconvenient in practice:

- Nothing to bind notations to
- Declaring/passing inconvenient
- No structure inference

Fully bundled

```
Record SemiGroup : Type := {
 sg_car :> Setoid ;
 sg_op : sg_car → sg_car → sg_car ;
 sg_proper : Proper ((=) ==> (=) ==> (=)) sg_op ;
 sg_ass : ∀ x y z, sg_op x (sg_op y z) = sg_op (sg_op x y) z) }
```

Problems:

- Prevents sharing, e.g. group together two CommutativeMonoids to create a SemiRing.
- Multiple inheritance (diamond problem).
- Long projection paths.

Unbundled using type classes

```
Class Equiv A := equiv: relation A.
Infix "=" := equiv: type_scope.
Class RingPlus A := ring_plus: A → A → A.
Infix "+" := ring_plus.

Class SemiRing A {e : Equiv A} {plus: RingPlus A}
  {mult: RingMult A} {zero: RingZero A} {one: RingOne A} : Prop := {
 semiring_mult_monoid :> @CommutativeMonoid A e mult one ;
 semiring_plus_monoid :> @CommutativeMonoid A e plus zero ;
 semiring_distr :> Distribute (.*.) (+) ;
 semiring_left_absorb :> LeftAbsorb (.*.) 0 }.
```

Changes:

1. Make SemiRing a type class ("predicate class").
2. Use *operational type classes* for relations and operations.

Examples

Instance syntax

```
Instance nat_equiv: Equiv nat := eq.
Instance nat_plus: RingPlus nat := plus.
Instance nat_0: RingZero nat := 0%nat.
Instance nat_1: RingOne nat := 1%nat.
Instance nat_mult: RingMult nat := mult.

Instance: SemiRing nat.
Proof.
  ...
Qed.
```

Examples

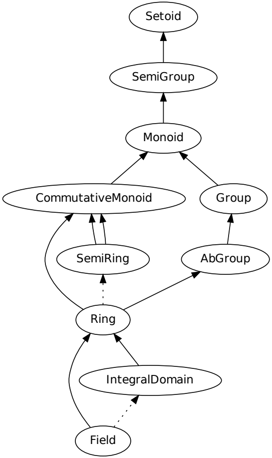
Usage syntax

```
(* z & x = z & y → x = y *)
Instance group_cancel `{Group G} : ∀ z, LeftCancellation (&) z.
Proof. ... Qed.

Lemma preserves_inv `{Group A} `{Group B}
 `{!Monoid_Morphism (f : A → B)} x : f (−x) = −f x.
Proof.
 apply (left_cancellation (&) (f x)). (* f x & f (-x) = f x - f x *)
 rewrite ← preserves_sg_op. (* f (x - x) = f x - f x *)
 rewrite 2!right_inverse. (* f unit = unit *)
 apply preserves_mon_unit.
Qed.

Lemma cancel_ring_test `{Ring R} x y z : x + y = z + x → y = z.
Proof.
 intros. (* y = z *)
 apply (left_cancellation (+) x). (* x + y = x + z *)
 now rewrite (commutativity x z).
Qed.
```

Algebraic hierarchy



Features:

- No distinction between axiomatic and derived inheritance.
- No sharing/multiple inheritance problems.
- No rebundling.
- No projection paths.
- Instances opaque.
- Terms never refer to proofs.
- Overlapping instances harmless.
- Seamless setoid/rewriting support.
- Seamless support for morphisms between structures.

Number structures

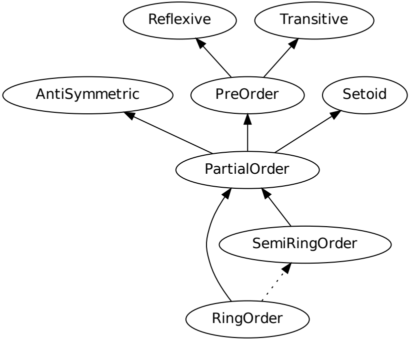
Our specifications:

- Naturals: initial semiring.
- Integers: initial ring.
- Rationals: field of fractions of $\mathbb{Z}$.

Remarks:

- Use some category theory and universal algebra for initiality.
- Models of these structures are unique up to isomorphism.
- Stdlib structures, nat, N, Z, bigZ, Q, bigQ are models.

Order theory



Features:

- Interacts well with algebraic hierarchy.
- Support for order morphisms.
- Default orders on $\mathbb{N}$, $\mathbb{Z}$ and $\mathbb{Q}$.
- Total semiring order uniquely specifies the order on $\mathbb{N}$.
- Total ring order uniquely specifies the order on $\mathbb{Z}$ and $\mathbb{Q}$.

Basic operations

- Common definitions:
  - nat_pow: repeated multiplication,
  - shiftl: repeated multiplication by 2.
- Implementing these operations this way is too slow.
- We want different implementations for different number representations.
- And avoid definitions and proofs becoming implementation dependent.

Hence we introduce abstract specifications for operations.

Abstract specifications of operations

Using Σ-types

- Well suited for simple functions.
- An example:

  Class Abs A ‘{Equiv A} ‘{Order A} ‘{RingZero A} ‘{GroupInv A}
  := abs_sig: ∀ x, { y | (0 ≤ x → y = x) ∧ (x ≤ 0 → y = −x)}.
  Definition abs ‘{Abs A} := λ x : A, ‘ (abs_sig x).

- Program allows to create instances easily.

  Program Instance: Abs Z := Zabs.

- But unable to quantify over all possible input values.

Abstract specifications of operations

Bundled

- For example:

```
Class ShiftL A B ‘{Equiv A} ‘{Equiv B} ‘{RingOne A} ‘{RingPlus A}
  ‘{RingMult A} ‘{RingZero B} ‘{RingOne B} ‘{RingPlus B} := {
 shiftl : A → B → A ;
 shiftl_proper : Proper ((=) ⟹ (=) ⟹ (=)) shiftl ;
 shiftl_0 :> RightIdentity shiftl 0 ;
 shiftl_S : ∀ x n, shiftl x (1 + n) = 2 ∗ shiftl x n }.
Infix "≪" := shiftl (at level 33, left associativity).
```

- Here shiftl is a δ-redex, hence simpl unfolds it.
- For BigN, x ≪ n becomes BigN.shiftl x n.
- As a result, rewrite often fails.

Abstract specifications of operations

Unbundled

- For example:

```
Class ShiftL A B := shiftl: A → B → A.
Infix "≪" := shiftl (at level 33, left associativity).

Class ShiftLSpec A B (sl : ShiftL A B) `{Equiv A} `{Equiv B}
  `{RingOne A} `{RingPlus A} `{RingMult A}
  `{RingZero B} `{RingOne B} `{RingPlus B} := {
 shiftl_proper : Proper ((=) ==> (=) ==> (=)) (≪) ;
 shiftl_0 :> RightIdentity (≪) 0 ;
 shiftl_S : ∀ x n, x ≪ (1 + n) = 2 * x ≪ n }.
```

- The δ-redex is gone due to the operational class.
- Remark: not `shiftl x n := x * 2 ^ n` since we cannot take a negative power on the dyadics.

Theory on basic operations

- Theory on shifting with exponents in $\mathbb{N}$ and $\mathbb{Z}$ is similar.
- Want to avoid duplication of theorems and proofs.

```
Class Biinduction R ‘{Equiv R}
  ‘{RingZero R} ‘{RingOne R} ‘{RingPlus R} : Prop
 := biinduction (P: R → Prop) ‘{!Proper ((=) ⟹ iff) P} :
      P 0 → (∀ n, P n ↔ P (1 + n)) → ∀ n, P n.
```

- Some syntax:

```
Section shiftl.
 Context ‘{SemiRing A} ‘{!LeftCancellation (.∗.) (2:A)}
   ‘{SemiRing B} ‘{!Biinduction B} ‘{!ShiftLSpec A B sl}.

 Lemma shiftl_base_plus x y n : (x + y) ≪ n = x ≪ n + y ≪ n.

 Global Instance shiftl_inj: ∀ n, Injective (≪n).
End shiftl.
```

Decision procedures

The Decision class collects types with a decidable equality.

```
Class Decision P := decide: sumbool P (¬ P).
```

- Declare a parameter `{∀ x y, Decision (x ≤ y)},
- Use decide (x ≤ y) to decide whether x ≤ y or ¬ x ≤ y.
- Canonical names for deciders.
- Easily define/compose deciders.

Decision procedures

Eager evaluation

Consider:

```
Record Dyadic := dyadic { mant : Int ; expo : Int }. (* m ∗ 2^e *)
Global Instance dy_precedes: Order Dyadic := λ x y,
  ZtoQ (mant x) ∗ 2 ^ (expo x) ≤ ZtoQ (mant y) ∗ 2 ^ (expo y)
```

Problem:

- decide (x ≤ y) is actually @decide Dyadic (x ≤ y) dyadic_dec.
- x ≤ y is evaluated due to eager evaluation (in Prop).

We avoid this problem introducing a λ-abstraction:

```
Definition decide_rel `(R : relation A) {dec : ∀ x y, Decision (R x y)}
  (x y : A) : Decision (R x y) := dec x y.
```

Decision procedures

Example

```
Context `{!PartialOrder (≤) } {!TotalOrder (≤) } `{∀ x y, Decision (x ≤ y)}.
Global Program Instance sprecedes_dec: ∀ x y, Decision (x < y) | 9 := λ x y,
  match decide_rel (≤) y x with
  | left E ⇒ right _
  | right E ⇒ left _
  end.
```

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

Our implementation: type classes!
Instance resolution:

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

Quoting
Example

Trivial example:

```
Class Quote (x : A) := { quote : Exp ; eval_quote : x ≡ Denote quote }.

Instance q_unit: Quote mon_unit := { quote := Unit }.
Instance q_op `(q1 : Quote t1) `(q2 : Quote t2) : Quote (t1 & t2)
  := { quote := Op (quote t1) (quote t2) }.
```

More interestingly: use type classes to represent heaps.

Quoting

- Automatically rewrite to point-free.
- Automatically derive uniform continuity.
- Plan: integrate with universal algebra.

Implementation of the reals

- Define the reals over a dense set A as [O'Connor]:

$$\mathbb{R} := \mathfrak{C}A := \{f : \mathbb{Q}_+ \to A \mid f \textit{ is regular}\}$$

- $\mathfrak{C}$ is a monad.
- To define a function $\mathbb{R} \to \mathbb{R}$: define a *uniformly continuous function* $f : A \to \mathbb{R}$, and obtain $\check{f} : \mathbb{R} \to \mathbb{R}$.
- Efficient combination of proving and programming.

Need an abstract specification of the dense set.

Implementation of the reals

Approximate rationals

```
Class AppDiv AQ := app_div : AQ → AQ → Z → AQ.
Class AppApprox AQ := app_approx : AQ → Z → AQ.

Class AppRationals AQ {e plus mult zero one inv} `{!Order AQ}
  {AQtoQ : Coerce AQ Q_as_MetricSpace} `{!AppInverse AQtoQ}
  {ZtoAQ : Coerce Z AQ} `{!AppDiv AQ} `{!AppApprox AQ}
  `{!Abs AQ} `{!Pow AQ N} `{!ShiftL AQ Z}
  `{∀ x y : AQ, Decision (x = y)} `{∀ x y : AQ, Decision (x ≤ y)} : Prop := {
  aq_ring :> @Ring AQ e plus mult zero one inv ;
  aq_order_embed :> OrderEmbedding AQtoQ ;
  aq_ring_morphism :> SemiRing_Morphism AQtoQ ;
  aq_dense_embedding :> DenseEmbedding AQtoQ ;
  aq_div : ∀ x y k, B_{2^k}('app_div x y k) ('x / 'y) ;
  aq_approx : ∀ x k, B_{2^k}('app_approx x k) ('x) ;
  aq_shift :> ShiftLSpec AQ Z (≪) ;
  aq_nat_pow :> NatPowSpec AQ N (^) ;
  aq_ints_mor :> SemiRing_Morphism ZtoAQ }.
```

Implementation of the reals

Verified versions of:

- Basic field operations ($+$, $*$, -, $/$)
- Exponentiation by a natural.
- Computation of power series.
- exp, arctan, sin and cos.
- $\pi := 176 * \mathsf{arctan}\frac{1}{57} + 28 * \mathsf{arctan}\frac{1}{239} - 48 * \mathsf{arctan}\frac{1}{682} + 96 * \mathsf{arctan}\frac{1}{12943}$.
- Square root using Wolfram iteration.

Implementation of the reals

Benchmarks

- Our HASKELL prototype is ∼15 times faster.
- Our COQ implementation is ∼100 times faster.
- Now able to compute 2,000 decimals of π and 425 decimals of $\exp \pi - \pi$ within one minute in COQ!
- (Previously 300 and 25 decimals)
- Type classes only yield a 3% performance loss.
- COQ is still too slow compared to unoptimized HASKELL (factor 30 for Wolfram iteration).

Implementation of the reals

Improvements

- FLOCQ: more fine grained floating point algorithms.
- Type classified theory on metric spaces.
- native_compute: evaluation by compilation to OCAML.
- Newton iteration to compute the square root.

Conclusions

- Works well in practice.
- Match mathematical practice.
- Abstract interfaces allow to swap implementations and share theory and proofs.
- Type classes yield no apparent performance penalty.
- Nice notations with unicode symbols.
- Greatly improved the performance of the reals.

Issues

- Type classes are quite fragile.
- Instance resolution is too slow.
- Need to adapt definitions to avoid evaluation in Prop.
- Universe polymorphism (finite sequences as free monoid).
- Setoid rewriting with relations in Type.
- Dependent pattern match (quoting to UA-terms).

Sources

http://robbertkrebbers.nl/research/reals/