Developing the Algebraic Hierarchy with Type Classes in Coq

Bas Spitters and Eelis van der Weegen

Radboud University Nijmegen

Abstract. We present a new formalization of the algebraic hierarchy in Coq, exploiting its new type class mechanism to make practical a solution formerly thought infeasible. Our approach addresses both traditional challenges as well as new ones resulting from our ambition to build upon this development a library of constructive analysis in which abstraction penalties inhibiting efficient computation are reduced to a bare minimum. To support mathematically sound abstract interfaces for $\mathbb{N}$, $\mathbb{Z}$, and $\mathbb{Q}$, our formalization includes portions of category theory and multisorted universal algebra.

1 Introduction

The development of libraries for formalized mathematics presents many software engineering challenges [4, 8], because it is far from obvious how the clean, idealized concepts from everyday mathematics should be represented using the facilitities provided by concrete theorem provers and their formalisms, in a way that is both mathematically faithful and convenient to work with.

For the algebraic hierarchy—a critical component in any library of formalized mathematics—these challenges include structure inference, handling of multiple inheritance, idiomatic use of notations, and convenient algebraic manipulation.

Several solutions have been proposed for the Coq theorem prover: dependent records [7] (a.k.a. telescopes), packed classes [6], and occasionally modules. We present a new approach based entirely on Coq's new type class mechanism, and show how its features together with a key design pattern let us effectively address the challenges mentioned above.

Since we intend to use this development as a basis for constructive analysis with practical certified exact real arithmetic, an additional objective and motivation in our design is to facilitate *efficient* computation. In particular, we want to be able to effortlessly swap implementations of number representations. Doing this requires that we have clean abstract interfaces, and mathematics tells us what these should look like: we represent $\mathbb{N}$, $\mathbb{Z}$, and $\mathbb{Q}$ as *interfaces* specifying an initial semiring, an initial ring, and a field of integral fractions, respectively.

To express these elegantly and without duplication, our development[1] includes an integrated formalization of parts of category theory and multi-sorted universal algebra, all expressed using type classes for optimum effect.

[1] The sources are available at: `http://www.eelis.net/research/math-classes/`

2 The Type-Classified Algebraic Hierarchy

Unlike Haskell's and Isabelle's second class type classes, Coq's type classes are first class: classes and their instances are realized as ordinary record types ("dictionaries") and registered constants of these types.

We represent each structure in the algebraic hierarchy as a type class. This immediately leads to the familiar question of which components of the structure should become parameters of the class, and which should become fields. By far the most important design choice in our development is the decision to turn all *structural* components (i.e. carriers, relations, and operations) into parameters, keeping only *properties* as fields. Type classes defined in this way are essentially predicates with automatically resolved proofs.

Conventional wisdom warns that while this approach is theoretically very flexible, one risks extreme inconvenience both in having to declare and pass around all these structural components all the time, as well as in losing notations (because we no longer project named operations out of records).

These are legitimate concerns that we avoid by exploiting the way Coq type classes and their support infrastructure work, using *operational type classes*: classes with a single field representing a single relation or operation in isolation. Such classes are treated specially by Coq in being translated to mere definitions rather than records, with the field projection becoming the identity function.

Class *Equiv A* := *equiv* : *relation A*.
Infix `"="` := *equiv* (*at level 70*, *no associativity*) .

These operational type classes serve to establish *canonical names*, which not only lets us bind notations to them, but also makes their declaration and use implicit in most contexts. For instance, using the following definition of semirings, all structural parameters (represented by operational classes declared with curly brackets) will be implicitly resolved by the type class mechanism rather than listed explicitly whenever we talk about semirings.

Class *SemiRing A* {*e* : *Equiv A*} {*plus* : *RingPlus A*} {*mult* : *RingMult A*}
{*zero* : *RingZero A*} {*one* : *RingOne A*} : *Prop* :=
{*semiring_mult_monoid* :> *Monoid A* (*op* := *mult*) (*unit* := *one*)
; *semiring_plus_monoid* :> *Monoid A* (*op* := *plus*) (*unit* := *zero*)
; *semiring_plus_comm* :> *Commutative plus*
; *semiring_mult_comm* :> *Commutative mult*
; *semiring_distr* :> *Distribute mult plus*
; *mult_0_l* : $\forall\ x, 0 * x\ =\ 0$ } .

The two key Coq features that make this work are implicit quantification (when declaring a semiring), and maximally inserted implicit arguments (when stating that something is a semiring, and when referencing operations and relations). Both were added specifically to support type classes.

Having argued that the *all-structure-as-parameters* approach *can* be made practical, we enumerate some of the benefits that make it worthwhile.

First, multiple inheritance becomes trivial: *SemiRing* inherits two *Monoid* structures on the same carrier and setoid relation, using ordinary named arguments (rather than dedicated extensions [9]) to achieve "manifest fields".

Second, because our terms are small and independent and never refer to proofs, we are invulnerable to concerns about efficiency and ambiguity of projection paths that plague existing solutions, obviating the need for extensions like the proposed coercion pullbacks [1].

Third, since our structural type classes are mere predicates, overlap between their instances is a non-issue. Together with the previous point, this gives us tremendous freedom to posit multiple structures on the same operations and relations, including ones derived implicitly via subclasses: by simply declaring a *SemiRing* class instance showing that a ring is a semiring, results about semirings immediately apply implicitly to any known ring, without us having to explicitly encode this relation in the hierarchy definition itself, and without needing any projection or translation of carriers or operations.

3 Category Theory and Universal Algebra

Motivated originally by our desire to cleanly express interfaces for basic numeric data types such as $\mathbb{N}$ and $\mathbb{Z}$ in terms of their categorical characterization as initial objects in the categories of semirings and rings, respectively, we initially introduced only the very basics of category theory into our development, again using type classes where possible to achieve the same benefits mentioned before.

Realizing that much code duplication for the various algebraic structures in the hierarchy could be avoided by employing universal algebra constructions, we then proceeded to formalize some of the theory of multisorted universal algebra and equational theories, using it to automatically construct varieties of algebras. We avoided existing formalizations [3, 5] of universal algebra, because we aimed to find out what level of elegance, convenience, and integration can be achieved using the state of the art technology (of which type classes are the most important instance).

At the time of writing, our development includes a fully integrated formalization of a nontrivial portion of category theory and multisorted universal algebra, including various categories (e.g. the category *Cat* of categories, and generic variety categories which we instantiate to obtain the categories of monoids, semirings, and rings), functors (including automatically generated forgetful functors), natural transformations, adjunctions, initial models of equational theories constructed from term algebras, transference of proofs between isomorphic models of equational theories, subalgebras, congruences, quotients, products, and the first homomorphism theorem.

There is an interesting interplay in our development between concrete algebraic structure type classes and their expressions on the one hand, and models of universal algebras and varieties instantiated with equational theories on the other. While occasionally a source of tension in that translation in either direction is not (yet) fully automatic, this duality also opens the door to the possibility

of fully internalized implementations of generic tactics for algebraic manipulation, no longer requiring plugins. One missing piece in this puzzle is automatic quotation of concrete expressions into universal algebra expressions. We have already implemented a proof of concept showing that like unification hints [1], type classes can be used to implement Ltac/OCaml-free quotation.

4 Conclusions

Our development (which according to `coqwc` consists of about 5K lines of specifications and 1K lines of proofs) shows that the first class type class implementation in Coq is already an extremely powerful piece of technology which enables new practical and elegant solutions to old problems.

In our work we push the type class implementation and the new generalized rewriting infrastructure [10] to their limits, revealing both innocent bugs as well as more serious issues (concerning both efficiency and functionality) that the Coq development team is already working on (for instance with the soon to be revealed new proof engine).

Acknowledgements. We would like to thank Matthieu Sozeau for discussions and quickly solving numerous small bugs and feature requests.

References

1. A. Asperti, W. Ricciotti, C.S. Coen, and E. Tassi. Hints in unification. In *Proceedings of the 22nd International Conference on Theorem Proving in Higher Order Logics*, page 98. Springer, 2009.
2. S. Berardi, F. Damiani, and U. de'Liguoro, editors. *Types for Proofs and Programs, International Conference, TYPES 2008, Torino, Italy, March 26-29, 2008, Revised Selected Papers*, volume 5497 of *LNCS*. Springer, 2009.
3. V. Capretta. Universal algebra in type theory. In Y. Bertot, G. Dowek, A. Hirschowitz, C. Paulin, and L. Théry, editors, *TPHOLs*, volume 1690 of *LNCS*, pages 131–148. Springer, 1999.
4. L. Cruz-Filipe, H. Geuvers, and F. Wiedijk. C-CoRN, the Constructive Coq Repository at Nijmegen. In A. Asperti, G. Bancerek, and A. Trybulec, editors, *MKM*, volume 3119 of *LNCS*, pages 88–103. Springer, 2004.
5. C. Domınguez. Formalizing in Coq Hidden Algebras to Specify Symbolic Computation Systems. In *AISC*, pages 270–284. Springer, 2008.
6. F. Garillot, G. Gonthier, A. Mahboubi, and L. Rideau. Packaging mathematical structures. In S. Berghofer, T. Nipkow, C. Urban, and M. Wenzel, editors, *TPHOLs*, volume 5674 of *LNCS*, pages 327–342. Springer, 2009.
7. H. Geuvers, R. Pollack, F. Wiedijk, and J. Zwanenburg. A constructive algebraic hierarchy in Coq. *J. Symb. Comput.*, 34(4):271–286, 2002.
8. F. Haftmann and M. Wenzel. Local theory specifications in Isabelle/Isar. In Berardi et al. [2], pages 153–168.
9. Z. Luo. Manifest fields and module mechanisms in intensional type theory. In Berardi et al. [2], pages 237–255.
10. M. Sozeau. A new look at generalized rewriting in type theory. *Journal of Formalized Reasoning*, 2(1):41–62, 2009.