Math. Struct. in Comp. Science (2011), *vol.* 21, *pp.* 795–825. © Cambridge University Press 2011
doi:10.1017/S0960129511000119

Type classes for mathematics in type theory[†]

BAS SPITTERS and EELIS VAN DER WEEGEN

Radboud University Nijmegen, Nijmegen, The Netherlands
Email: spitters@cs.ru.nl;eelis@eelis.net

Received 2 July 2010; revised 15 December 2010

The introduction of first-class type classes in the Coq system calls for a re-examination of the basic interfaces used for mathematical formalisation in type theory. We present a new set of type classes for mathematics and take full advantage of their unique features to make practical a particularly flexible approach that was formerly thought to be unfeasible. Thus, we address traditional proof engineering challenges as well as new ones resulting from our ambition to build upon this development a library of constructive analysis in which any abstraction penalties inhibiting efficient computation are reduced to a minimum.

The basis of our development consists of type classes representing a standard algebraic hierarchy, as well as portions of category theory and universal algebra. On this foundation, we build a set of mathematically sound abstract interfaces for different kinds of numbers, succinctly expressed using categorical language and universal algebra constructions. Strategic use of type classes lets us support these high-level theory-friendly definitions, while still enabling efficient implementations unhindered by gratuitous indirection, conversion or projection.

Algebra thrives on the interplay between syntax and semantics. The Prolog-like abilities of type class instance resolution allow us to conveniently define a quote function, thus facilitating the use of reflective techniques.

1. Introduction

The development of libraries for formalised mathematics presents many software engineering challenges (Cruz-Filipe *et al.* 2004; Haftmann and Wenzel 2008) because it is far from obvious how the clean, idealised concepts of everyday mathematics should be represented using the facilities 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, which is a critical component in any library of formalised mathematics, these challenges include: structure inference; the handling of multiple inheritance; the equality of axiomatically posited and derived structure; the idiomatic use of notations; support for models based on quotient representations; and convenient algebraic manipulation (for example, rewriting). Several solutions have been proposed for the Coq theorem prover: dependent records (Geuvers *et al.* 2002), which are also known as telescopes; packed classes (Garillot *et al.* 2009); and, occasionally, modules. We present

[†] This work has been partially funded by the FORMATH project, nr. 243847, of the FET program within the 7th Framework program of the European Commission.

a new solution based entirely on the use of Coq's new type class facility to make fully 'unbundled' predicate representations of algebraic structures practical to work with.

Our development is not merely aimed at the formalisation of theory, and our choice of a system based on type theory is no accident. It is our explicit ambition that the interfaces and theory we develop be employed directly for the specification and parameterisation of efficiently executable procedures and data structures, implemented using type theory's native term reduction as a programming language. Thus, our work belongs in the long tradition of realising the promise of type theory to truly unite mathematical formalisation and certified (functional) programming, without making painful sacrifices on either side.

Because our 'ultimate' goal is to use this development as a basis for constructive analysis with practical certified exact real arithmetic, and because numerical structures are ideal test subjects for our algebraic hierarchy, we shall use these to motivate and demonstrate the key parts of our development. Since we are concerned with *efficient* computation, we want to be able to swap effortlessly between 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 interfaces elegantly and without duplication, our development[†] includes an integrated formalisation of parts of category theory and multi-sorted universal algebra, all expressed using type classes for optimum effect.

In this paper we focus on the Coq proof assistant. We conjecture that the methods can be transferred to any type theory based proof assistant supporting type classes, such as Matita (Asperti *et al.* 2007).

Outline of the paper

In Section 2, we briefly describe the Coq system and its implementation of type classes. Then, in Section 3, we give a very concrete introduction to the issue of *bundling*, arguably the biggest design dimension when building interfaces for abstract structures. In Section 4, we show how type classes can make practical the use of 'unbundled' purely predicate based interfaces for abstract structures.

In the rest of the paper, we make a tour through the key components in our development, leading up to the numerical interfaces. This will not only show the pleasant style of formalisation that rigorous use of type classes enables, but will also show that an eager adoption and incorporation of more abstract mathematical perspectives (which are traditionally often ignored when doing dependently typed programming on concrete data structures in type theory) is not only feasible but actually practical and beneficial.

In Section 5, we discuss our algebraic hierarchy implemented with type classes. In Sections 6 and 7 we give a taste of what category theory and universal algebra look like in our development, and in Section 8 we use these facilities to build abstract interfaces for numbers. In order to illustrate a very different use of type classes, in Section 9, we

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

discuss the implementation of a quoting function for algebraic terms in terms of type classes. In Section 10, we hint at an interface for sequences, but describe how a limitation in the current implementation of Coq makes its use problematic. Finally, we present our conclusions in Section 11.

2. Preliminaries

The Coq proof assistant is based on the calculus of inductive constructions (Coquand and Huet 1988; Coquand and Paulin 1990), which is a dependent type theory with (co)inductive types (Bertot and Castéran 2004 and Coq Development Team 2008). In true Curry–Howard fashion, it is both an excessively pure, if somewhat pedantic, functional programming language with an extremely expressive type system, and a language for mathematical statements and proofs. In the following sections we highlight some aspects of Coq that are of particular relevance to our development.

2.1. *Types and propositions*

Propositions in Coq are types (Martin-Löf 1982; Martin-Löf 1998), which themselves have types called *sorts*. Coq features a distinguished sort called Prop, which one may choose to use as the sort for types representing propositions. The distinguishing feature of the Prop sort is that terms of non-Prop type may not depend on the values of inhabitants of Prop types (that is, proof terms). This regime of discrimination establishes a weak form of proof irrelevance, in that changing a proof can never affect the result of value computations. At a very practical level, this lets Coq safely erase all Prop components when extracting certified programs to OCaml or Haskell.

Occasionally, there is some ambiguity as to whether a certain piece of information (such as a witness to an existential statement) is strictly 'proof matter' (and thus belongs in the Prop sort) or actually of further computational interest (and thus does *not* belong to the Prop sort). We will see one such case when we discuss the first homomorphism theorem in Section 7.3. Coq provides a modest level of *universe-polymorphism* so that we may avoid duplication when trying to support Prop-sorted and non-Prop-sorted content with a single set of definitions.

2.2. *Equality, setoids and rewriting*

The 'native' notion of equality in Coq is that of term convertibility, naturally reified as a proposition by the inductive type family eq: ∀ (T: Type), T → T → Prop with single constructor eq_refl:

```
eq_refl : ∀ (T: Type) (x: T), x ≡ x,
```

where 'a ≡ b' is notation for eq T a b. Here we diverge from Coq tradition and reserve the 'a = b' notation for *setoid* equality (to be discussed below), as this is the equality we will be working with most of the time.

Importantly, since convertibility is a congruence, a proof of a ≡ b lets us substitute b for a anywhere inside a term without further conditions. We mention this explicitly only because such rewriting *does* give rise to conditions when we depart from raw convertibility and introduce equivalence relations that express how possibly distinct (unconvertible) terms may represent the same conceptual object. Rational numbers represented by (non-reduced) formal integer fractions are a typical example. Rewriting a subterm using a proof of such an equality is permitted only if the subterm is argument to a function that has been proved to *respect* the equality. Such a function is called *proper* with respect to the equality in question, and propriety must be proved for each function in whose arguments we wish to enable rewriting.

Because the Coq type theory lacks quotient types (as it would make type checking undecidable), one usually bases abstract structures on a *setoid* ('Bishop set'): a type equipped with an equivalence relation (Bishop 1967; Hofmann 1997; Barthe *et al.* 2003). Palmgren (2009) shows that Bishop sets have pleasant categorical properties, which translate to a powerful implicit type structure. It would be of interest to actually provide machine support for this type structure. As we will see in Section 7, working with setoids pays off when working with notions such as quotient algebras.

Effectively keeping track of, resolving and combining proofs of equivalence-ness and propriety when the user attempts to substitute a given (sub)term using a given equality, is known as 'setoid rewriting', and requires non-trivial infrastructure and support from the system. The Coq implementation of these mechanisms was largely rewritten by Matthieu Sozeau in order to make it more flexible and to replace the old special-purpose setoid/morphism registration command with a clean type class based interface (Sozeau 2009).

The algebraic hierarchy of the SSREFLECT libraries (Garillot *et al.* 2009) uses an alternative approach. It simply requires canonical representation of all objects, so that setoid equality is not needed. Of course, this policy severely restricts the freedom one has when implementing models of abstract structures. Indeed, for some sets, there are no canonical representation schemes. The constructive reals, which are of particular interest to us, are an example of such a set.

2.3. *Type classes*

Type classes (Wadler and Blott 1989) have been a great success story in the Haskell functional programming language as a means of organising interfaces of abstract structures. Coq's type classes provide a superset of their functionality, but implemented in a different way.

In Haskell and Isabelle, type classes and their instances are second class. They are handled as specialised syntactic constructs whose semantics are given specifically by the type class apparatus. By contrast, the expressivity of dependent types and inductive families, as supported in Coq, combined with the use of pre-existing technology in the system (namely proof search and implicit arguments) enable a *first class* type class implementation (Sozeau and Oury 2008): classes are ordinary record types ('dictionaries'); instances are ordinary constants of these record types (registered as *hints* with the proof search machinery); class constraints are ordinary implicit parameters; and instance resolution is achieved by augmenting the unification algorithm to invoke ordinary proof search for implicit arguments of class type. Thus, type classes in Coq are realised using relatively minor syntactic aids that bring together existing facilities of the theory and the system into a coherent idiom, rather than by the introduction of a new category of qualitatively different definitions with their own dedicated semantics.

The basic idea of using type-class-like facilities for structuring computerised mathematics dates back to the AXIOM computer algebra system (Jenks *et al.* 1992). Weber and Klaeren (1993) pursued the analogy between AXIOM's so-called categories and type classes in Haskell. Santas (1995) pursued analogies between type classes, AXIOM categories and existential types. Existential types are present in Haskell, but absent from Coq.

3. Bundling is bad

Algebraic structures are expressed in terms of a number of carrier sets, a number of operations and relations on these carriers, together with a number of laws that the operations and relations satisfy. In a system like Coq, we have different options when it comes to representing the grouping of these components. At one end of the spectrum, we can simply define the (conjunction of) laws as an n-ary predicate over n components, forgoing explicit grouping altogether. For instance, for the mundane example of a reflexive relation, we could use

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

The curly brackets used for A mark it as an implicit argument.

More elaborate structures can also be expressed as predicates (expressing laws) over a number of carriers, relations and operations. While optimally flexible in principle, in practice, a *naive* adoption of this approach (that is, without using type classes) leads to substantial inconveniences in actual use: when *stating* theorems about abstract instances of such structures, one must enumerate all components along with the structure (predicate) of interest. And when *applying* such theorems, one must either enumerate any non-inferrable components, or let the system spawn awkward metavariables to be resolved at a later time. Importantly, this also hinders proof search for proofs of the structure predicates, making any non-trivial use of theorems a laborious experience. Finally, the lack of *canonical names* for particular components of abstract structures makes it impossible for us to provide them with idiomatic notations.

In the absence of type classes, these are all very real problems, and for this reason the two largest formalisations of abstract algebraic structures in Coq today, CoRN (Cruz-Filipe *et al.* 2004) and Ssreflect (Garillot *et al.* 2009), both use *bundled* representation schemes, using records with one or more of the components as fields instead of parameters. For reflexive relations, the following is a fully bundled representation, which represents the other end of the spectrum:

```
Record ReflexiveRelation: Type :=
  { rr_carrier: Type
  ; rr_rel: relation rr_carrier
  ; rr_proof: ∀ x, rr_rel x x }.
```

Superficially, this instantly solves the problems described above: reflexive relations can now be declared and passed as self-contained packages, and the rr_rel projection now constitutes a canonical name for relations that are known to be reflexive, and we could bind a notation to it. While there is no conventional notation for reflexive relations, the situation is the same in the context of, say, a semiring, where we would bind + and * notations to the record field projections for the addition and multiplication operations, respectively.

Unfortunately, despite its apparent virtues, the bundled representation introduces serious problems of its own, the most immediate and prominent being a lack of support for *sharing* components between structures, which is needed to cope with overlapping multiple inheritance.

In our example, the lack of sharing support rears its head as soon as we try to define EquivalenceRelation in terms of ReflexiveRelation and its hypothetical siblings bundling symmetric and transitive relations. For this, we would need some way to make sure that when we 'inherit' ReflexiveRelation, SymmetricRelation and TransitiveRelation by adding them as fields in our bundled record, they all refer to the same carrier and relation. Adding additional fields stating equalities between the three bundled carriers and relations is neither easily accomplished (because one would need to work with heterogenous equality) nor would it permit a natural use of the resulting structure (because one would constantly have to rewrite things back and forth).

Manifest fields (Pollack 2002) have been proposed to address exactly this problem. In fact, a semblance of this has been implemented in the Matita system (Sacerdoti Coen and Tassi 2008). We hope to convince the reader that type system extensions like this, which have been designed to mitigate particular symptoms of the bundled approach, are less elegant than a solution (described in the next section) that avoids the problem altogether by using predicate-like type classes in place of bundled records.

If we were to revert back to the predicate formulation of relations, we *could* still define EquivalenceRelation in a bundled fashion without the need for equalities:

```
Record EquivalenceRelation: Type :=
  { er_carrier: Type
  ; er_rel: relation er_carrier
  ; er_refl: ReflexiveRelation er_carrier er_rel
  ; er_sym: SymmetricRelation er_carrier er_rel
  ; er_trans: TransitiveRelation er_trans er_rel }.
```

However, as before, we conclude that EquivalenceRelation, should also be a predicate. Indeed, it would be rather strange for the interface of equivalence relations to differ qualitatively from the interface of reflexive relations.

Another attempt to recover some grouping might be to bundle the carrier with the relation into a (lawless) record, but this also hinders sharing. As soon as we try to define an algebraic structure with two reflexive relations on the same carrier, we need awkward hacks to establish equality between the carrier projections of two different (carrier, relation) bundles.

Even bundling just the operations of an algebraic structure together in a record (with the carrier as a parameter) leads to the same problem when, for example, one attempts to define a hypothetical algebraic structure with two binary relations and a constant such that both binary relations form a monoid with the constant.

A second problem with bundling is that as the bundled records are stacked to represent higher and higher structures, the projection paths for their components grow longer and longer, resulting in ever more unwieldy terms (though coercions and notations can make this less painful). Furthermore, if one tries to implement some semblance of sharing in a bundled representation, these projection paths additionally become non-canonical, and still more extensions have been proposed to address this symptom, for example, coercion pullbacks (Asperti *et al.* 2009).

Thus, bundled representations come at a substantial cost in flexibility. Historically, using bundled representations has, nevertheless, been an acceptable trade off, because:

(1) the unbundled alternative was such a pain; and
(2) the standard algebraic hierarchy (up to, say, fields and modules) is not all that wild.

In the next section, we show that type-classification of structure predicates and their component parameters has the potential to remedy the problems associated with the naive unbundled predicate approach.

One may wonder whether it might be beneficial to go one step further and unbundle proofs of laws and inherited substructures as well. This is not the case, because there is no point in sharing them. After all, by (weak) proof irrelevance, the ‘value’ of such proofs can be of no consequence anyway. Indeed, parameterising on proofs would be actively harmful because instantiations differing only in the proof argument would express the same thing yet be non-convertible, requiring awkward conversions and hindering automation.

4. Predicate classes and operational classes

To show that the fully unbundled approach with structures represented by predicates can be made feasible using type classes, we will tackle each of the problems traditionally associated with their use, starting with those encountered during theorem *application*.

Suppose we have defined SemiGroup as a structure predicate as follows[†]:

```
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 }.
```

Then by

(1) making SemiGroup a *class* (by replacing the Record keyword with the Class keyword),

[†] Note that defining SemiGroup as a record instead of as a straight conjunction does not make it any less of a predicate. The record form is simply more convenient in that it immediately gives us named projections for laws and substructures.

(2) marking its proofs as *instances* (by replacing the Lemma keyword with the Instance keyword), and
(3) marking the SemiGroup parameter of semigroup theorems as implicit (by using curly instead of round brackets),

we no longer have to pass SemiGroup proofs around manually ourselves, letting instance resolution do it for us instead. Because instance resolution is part of the unifier, this also works when the statement of the theorem we wish to apply only mentions some of the components (which admittedly does not make much sense for semigroups).

Next, we turn to problems concerning theorem *declaration*. Our ideal would be the common mathematical vernacular, where one simply says:

Theorem: For x, y, z in a semigroup G, $x * y * z = z * y * x$.

(This silly statement allows us to present the syntax clearly.)

Without further support from the system, this would have to be written as

```
Theorem example G e op {P: SemiGroup G e op}:
  ∀ x y z, e (op (op x y) z) (op (op z y) x).
```

Because e and op are freshly introduced local names, we cannot bind notations to them prior to this theorem. Hence, if we want notations, what we really need are canonical names for these components. This is easily accomplished with single-field type classes containing one component each, which we will call *operational type classes*[‡]:

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

Infix "=" := equiv: type_scope.
Infix "&" := sg_op (at level 50, left associativity).
```

We use & here, and reserve the notation $*$ for (semi)ring multiplication.

As an aside, note that the distinction between the class field name and the infix operator notation bound to it is really just a mildly awkward Coq artifact. In Haskell, where operators can themselves be used as names, there would be no need to have the equiv and sg_op names in addition to the operator 'names'.

If we now retype SemiGroup as

```
∀ (G: Type) (e: Equiv G) (op: SemiGroupOp G), Prop
```

we can declare the theorem with

```
Theorem example G e op {P: SemiGroup G e op}:
  ∀ x y z, x & y & z = z & y & x.
```

This works because instance resolution, invoked by the use of = and &, will find e and op, respectively. Hence, the above is really

[‡] These single-field type classes are used in the same way in the Clean standard library (Brus *et al.* 1987).

```
Theorem example G e op {P: SemiGroup G e op}:
  ∀ x y z, equiv e (sg_op op (sg_op op x y) z) (sg_op op (sg_op op z y) x).
```

where e and the ops are filled in by instance resolution.

At this point, a legitimate worry might be that the Equiv/SemiGroup classes and their equiv/sg_op projections imply constant construction and deconstruction of records, harming the simplicity and flexibility of the predicate approach that we are trying so hard to preserve. However, no such construction and destruction takes place because type classes with only a single field are not desugared into an actual record with field projections in the same way as classes with any other number of fields are. Instead, both class itself and its field projection are defined as the identity function with a fancy type. Thus, the introduction of these canonical names is essentially free; the structure predicate's new type reduces straight back to what it was before.

A remaining eyesore in the theorem declaration is the enumeration of e and op. To remove these, we use a new parameter declaration feature called *implicit generalisation*, which was introduced in Coq specifically to support type classes. Using implicit generalisation, we can write

```
Theorem example `{SemiGroup G}: ∀ x y z: G, x & y & z = z & y & x.
```

The backtick tells Coq to insert implicit declarations of further parameters to SemiGroup G, namely those declared as e and op above. It also lets us omit a name for the SemiGroup G parameter itself. All of these will be given automatically generated names, which we will never refer to.

Thus, we have reached the mathematical ideal we aimed for.

While we are on the topic of implicit generalisation, we should mention one inadequacy concerning their current implementation that we feel should be addressed for the facility to be a completely satisfying solution. While the syntax already supports variants (not shown above) that differ in how exactly different kinds of arguments are inferred and/or generalised, there is no support for an argument to be ‘inferred if possible, and generalised otherwise’. The need for such a policy arises naturally when declaring a parameter of class type in a context where *some* of its components are already available, while others are to be newly introduced. The current workaround in these cases involves providing names for components that are then never referred to, which is a bit awkward.

One aspect of the predicate approach we have not mentioned thus far is that in proofs parameterised by abstract structures, all components become hypotheses in the context. For the theorem above, the context looks like

```
G: Type
e: Equiv G
op: SemiGroupOp G
P: SemiGroup G e op
```

We are not particularly worried about overly large contexts, especially because most of the ‘extra’ hypotheses we have compared with bundled approaches are declarations of relations, operators and constants, which are all in some sense inert with respect to proof search. Hence, we do not foresee problems with large contexts for any but the most complex formalisations.

4.1. *Implicit syntax-directed algorithm composition*

Before we proceed to discuss the algebraic hierarchy based on predicate classes and operational classes, in this section we will briefly highlight one specific operational type class because we will use it later, and because it is a particularly nice illustration of another neat application of operational type classes. The operation in question is that of deciding a proposition:

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

Here, sumbool is just the (informative) sum of proofs.

Decision is a very general-purpose type class, which also works for predicates. For instance, to declare a parameter expressing decidability of, say, (setoid) equality on a type X, we write ‘{∀ a b: X, Decision (a = b)}. To then use this (unnamed) decider to decide a particular equality, we simply say decide (x = y), and instance resolution will resolve the decider we declared.

With Decision as a type class, we can very easily define composite deciders for things like conjunctions and quantifications over (finite) domains:

```
Instance decide_conj ‘{Decision P} ‘{Decision Q}: Decision (P ∧ Q).
```

With these in place, we can just say decide (x = y ∧ p = q) and let instance resolution automatically compose a decision procedure that can decide the specified proposition. This style of syntax-directed implicit composition of algorithms is very convenient and highly expressive.

5. The algebraic hierarchy

We have developed an algebraic hierarchy composed entirely out of predicate classes and operational classes as described in the previous section. For instance, our semiring interface looks as follows:

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

All of Equiv, RingPlus, RingMult, RingZero and RingOne are operational (single-field) classes, with bound notations =, +, ∗, 0 and 1, respectively. We will now briefly highlight some additional aspects of this style of structure definition in more detail.

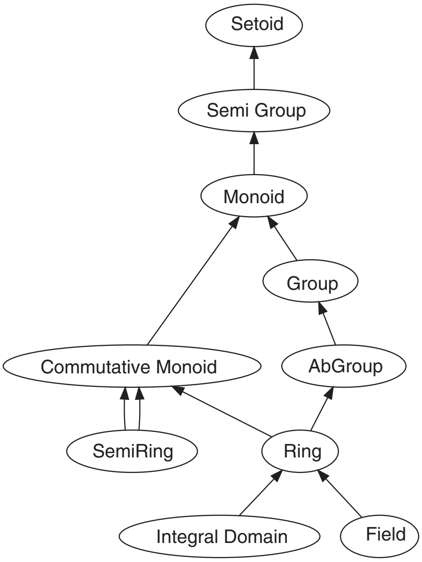

Fig. 1. Inheritance diagram

Fields declared with :> are registered as hints for instance resolution, so that in any context where (A, =, +, 0, ∗, 1) is known to be a SemiRing, (A, =, +, 0) and (A, =, ∗, 1) are automatically known to be CommutativeMonoids (and so on, transitively, because instance resolution is recursive). In our hierarchy, these substructures by themselves establish the inheritance diagram in Figure 1.

However, we can easily add additional inheritance relations by declaring corresponding class instances. For instance, while our Ring class does not have a SemiRing field, the following instance declaration has the exact same effect for the purposes of instance resolution (at least once proved, which is trivial):

```
Instance ring_as_semiring ‘{Ring R}: SemiRing R.
```

Thus, axiomatic structural properties and inheritance have precisely the same status as separately proved structural properties and inheritance, reflecting natural mathematical ideology. Again, contrast this with bundled approaches, where axiomatic inheritance relations determine projection paths, and where additional inheritance relations require rebundling and lead to additional and ambiguous projection paths for the same operations.

The declarations of the two inherited CommutativeMonoid structures in SemiRing nicely illustrate how predicate classes naturally support not just multiple inheritance, but *overlapping* multiple inheritance, where the inherited structures may share components (in this case carrier and equivalence relation). The carrier A, being an explicit argument, is specified as normal. The equivalence relation, being an implicit argument of class type, is resolved automatically to e. The binary operation and constant would normally be automatically resolved as well, but we override the inference mechanism locally using Coq's existing named argument facility (which is only syntactic sugar of the most superficial kind) in order to explicitly pair multiplication with 1 for the first CommutativeMonoid substructure, and addition with 0 for the second CommutativeMonoid substructure. Again, contrast this with type system extensions such as Matita's manifest records, which are required to make this work when the records bundle components such as op and unit as *fields* instead of parameters.

Since CommutativeMonoid indirectly includes a SemiGroup field, which in turn includes an Equivalence field, having a SemiRing proof means having two distinct proofs that the equality relation is an equivalence. This kind of redundant knowledge (which arises naturally) is never a problem in our setup, because the use of operational type classes ensures that terms composed of algebraic operations and relations never refer to structure proofs. We find this to be a tremendous relief compared with approaches that *do* intermix the two and where one must be careful to ensure that such terms refer to the *right* proofs of properties. There, even *strong* proof irrelevance (which would make terms convertible that differ only in what proofs they refer to) would not make these difficulties go away entirely, because high-level tactics that rely on quotation of terms require syntactic identity (rather than 'mere' convertibility) to recognise identical subterms.

Because predicate classes only provide contextual information and are insulated from the actual algebraic expressions, their instances can always be kept entirely opaque – only their existence matters. Together, these properties largely defuse an argument occasionally voiced against type classes concerning a perceived unpredictability of instance resolution. While it is certainly true that in contexts with redundant information it can become hard to predict which instance of a predicate class will be found by proof search, it simply *does not matter* which one is found. Moreover, for operational type classes, the issue rarely arises because their instances are not nearly as abundant, and are systematically shared.

We use names for properties like distributivity and absorption, because these are type classes as well (which is why we declare their instances with :>). It has been our experience that almost any generic predicate worth naming is worth representing as a predicate type class so that its proofs will be resolved as instances behind the scenes whenever possible. Doing this consistently minimises administrative noise in the code, bringing us closer to ordinary mathematical vernacular. Indeed, we believe that type classes provide an elegant and apt formalisation of the seemingly casual manner in which ordinary mathematical presentation assumes implicit administration and use of a 'database' of properties previously proved.

The operational type classes used in SemiRing for zero, one, multiplication and addition, are the same ones used by Ring and Field (not shown). Thus, the realisation that a particular semiring is in fact a ring or field has no bearing on how one refers to the operations in question, which is as it should be. However, the realisation that a particular semigroup is part of a semiring *does* call for a new (canonical) name, simply because of the need for disambiguation. The introduction of these additional names for the same operation is quite harmless in practice, because canonical names established by operational type class fields are identity functions, so in most contexts the distinction reduces away instantly.

The hierarchy of predicate classes for the abstract structures themselves is mirrored by a hierarchy of predicate classes for morphisms. For instance

```
Context ‘{Monoid A} ‘{Monoid B}.

Class Monoid_Morphism (f: A → B) :=
  { monmor_from: Monoid A
  ; monmor_to: Monoid B
  ; monmor_sgmor:> SemiGroup_Morphism f
  ; preserves_mon_unit: f mon_unit = mon_unit }.
```

Some clarification is in order to explain the role of the Context declaration of the two monoids. While Monoid_Morphism appears to depend on monoid-ness proofs (which would be a gross violation of our idiom), in fact, it is only parameterised on the monoid *components* declared through implicit generalisation of the Monoid declarations, because it only refers to those. Here, we use declarations of predicate class parameters merely as convenient shorthands to declare their components.

Notice that f is *not* made into an operational type class. The reason for this is that the role of f is analogous to the carrier type in the previous predicate class definitions in that it serves as the primary identification for the structure, and should therefore not be inferred.

We include the monmor_to and monmor_from fields because it does not make much sense to talk about monoid morphisms between non-monoids, and having these fields removes the need for Monoid class constraints when we are already parameterising definitions or theory on a Monoid_Morphism. On the other hand, we will also wish to talk about monoid morphisms between *known* monoids, and in these cases the fields will be strictly redundant. As mentioned earlier, it is a strength of our approach that such redundant knowledge is entirely harmless, so we may freely posit these structural properties whenever they make sense and provide convenience, and without risking rebundling tar-pits or projection path ambiguities down the line.

Unfortunately, there is actually an annoying wrinkle here, which also explains why we do not register these two fields as instance resolution hints (by declaring them with :>). What we really want these fields to express is '*if* in a certain context we know something to be a Monoid_Morphism, *then* realise that the source and target are Monoids'. However, the current instance resolution implementation has little support for this style of *forward* reasoning, and is really primarily oriented towards *backward* reasoning: had we registered monmor_to and monmor_from as instance resolution hints, we would in fact be saying '*if* trying to establish that something is a Monoid, *then* try finding a Monoid_Morphism to or from it', which quickly degenerates into a wild goose chase. We will return to this point in Section 11.

Having described the basic principles of our approach, in the remainder of this paper we present a tour around other parts of our development, further illustrating what a state of the art formal development of foundational mathematical structures can look like with a modern proof assistant based on type theory.

These parts were originally motivated by our desire to express cleanly the interfaces for basic numeric data types such as ℕ and ℤ in terms of their categorical characterisation as initial objects in the categories of semirings and rings, respectively. We will start, therefore, with basic category theory.

6. Category theory

Following our idiom, we introduce operational type classes for the *components* of a category:

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

Infix "⟶ " := Arrow (at level 90, right associativity).
Infix "⊙" := comp (at level 40, left associativity).
```

(The categorical arrow is distinguished from the primitive function space arrow by its length.)

With these in place, our type class for categories follows the usual type-theoretical definition of a category (Huet and Saibi 1995):

```
Class Category (O: Type) '{Arrows O} '{∀ x y: O, Equiv (x ⟶ y)}
    '{CatId O} '{CatComp O}: Prop :=
  { arrow_equiv:> ∀ x y, Setoid (x ⟶ y)
  ; comp_proper:> ∀ 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 }.
```

This definition is based on the 2-categorical idea of having equality only on arrows, and not on objects.

Initiality, too, is defined by a combination of an operational and a predicate class:

```
Context '{Category X}.
Class InitialArrows (x: X): Type := initial_arrow: ∀ y, x ⟶ y.
Class Initial (x: X) '{InitialArrows x}: Prop :=
  initial_arrow_unique: ∀ y (a: x ⟶ y), a = initial_arrow y.
```

The operational class InitialArrows designates the arrows that originate from an initial object x by virtue of it being initial. The Initial class itself further requires these 'initial arrows' to be unique. Having InitialArrows as an operational type class means that we can always simply say initial_arrow y whenever y is known to be an object in a category known to have an initial object (where 'known' should be read as 'can be determined by instance resolution').

Strictly speaking, the above is all we need in order to continue with the story line leading up to the numerical interfaces, but just to give a further taste of what category theory with this setup looks like in practice, we briefly mention a few more definitions and theorems.

6.1. *Functors*

In our definition of functors, we again see the by now familiar refrain:

```
Context ‘{Category C} ‘{Category D} (map_obj: C → D).

Class Fmap: Type :=
  fmap: ∀ {v w: C}, (v ⟶ w) → (map_obj v ⟶ map_obj w).

Class Functor ‘{Fmap}: Prop :=
  { functor_from: Category C
  ; functor_to: Category D
  ; functor_morphism:> ∀ a b: C, Setoid_Morphism (@fmap _ a b)
  ; preserves_id: ‘(fmap (cat_id: a ⟶ a) = cat_id)
  ; preserves_comp ‘(f: y ⟶ z) ‘(g: x ⟶ y):
    fmap (f ⊙ g) = fmap f ⊙ fmap g }.
```

We ought to say a few words about our use of fmap. The usual mathematical notational convention for functor application is to use the name of the functor to refer to both its object map and its arrow map, relying on additional conventions regarding object/arrow names for disambiguation: F x and F f map an object and an arrow, respectively, because x and f are conventional names for objects and arrows, respectively.

In Coq, for a term F to function as though it has two different types simultaneously (namely, the object map and the arrow map), either:

(1) there must be coercions from the type of F to either function, or
(2) F must be (coercible to) a single function that is able to consume both object and arrow arguments.

In addition to not being supported by Coq, option (1) would violate our policy of leaving components unbundled.

For (2), if it could be made to work at all, F would need a pretty egregious type considering that arrow types are indexed by objects, and that the type of the arrow map

```
∀ x y, (x ⟶ y) → (F x ⟶ F y)
```

must refer to the object map.

We feel that these issues are not limitations of the Coq system, but merely reflect the fact that notationally identifying these two distinct and interdependent maps is an abuse of notation of sufficient severity to make it ill-suited to a formal development where software engineering concerns apply. Hence, we do not adopt this practice, and use fmap F (which is a name taken from the Haskell standard library) to refer to the arrow map of a functor F.

6.2. *Natural transformations and adjunctions*

We introduce a convenient notation for the type of the computational content of a natural transformation between two functors:

```
Notation ”F ⟹ G” := (∀  x, F x ⟶ G x).
```

We now assume the following context:

```
Context ‘{Category C} ‘{Category D}
  ‘{Functor (F: C → D)} ‘{Functor (G: D → C)}.
```

The naturality property is easy to write:

```
Class NaturalTransformation (η: F ⟹ G): Prop :=
  { naturaltrans_from: Functor F
  ; naturaltrans_to: Functor G
  ; natural: ∀ ‘(f: x ⟶ y), η y ⊙ fmap F f = fmap G f ⊙ η x }.
```

Adjunctions can be defined in different ways – a nice symmetric definition is

```
Class Adjunction (φ: ∀ ‘(F c ⟶ d), (c ⟶ G d)): Prop :=
  { adjunction_left_functor: Functor F _
  ; adjunction_right_functor: Functor G _
  ; natural_left ‘(f: d ⟶ d’) c: (fmap G f ⊙) ∘ φ = φ(c:=c) ∘ (f ⊙)
  ; natural_right ‘(f: c’ ⟶ c) d: (⊙ f) ∘ φ(d:=d) = φ ∘ (⊙ fmap F f) }.
```

An alternative definition is

```
Class AltAdjunction (η: id ⟹ G ∘ F) (φ: ∀ ‘(f: c ⟶ G d), F c ⟶ d): Prop :=
  { alt_adjunction_natural_unit: NaturalTransformation η
  ; alt_adjunction_factor: ∀ ‘(f: c ⟶ G d),
      is_sole ((f =) ∘ (⊙ η c) ∘ fmap G) (φ f) }.
```

Formalising the (non-trivial) proof that these two definitions are equivalent provides a nice test for our definitions. As a first step, we have constructed the unit and co-unit of the adjunction, thereby proving Mac Lane's Theorem 1 (Mac Lane 1998) – we have followed his proof concisely and closely.

7. Universal algebra

To specify the natural numbers and the integers as initial objects in the categories of semirings and rings, respectively, definitions of these categories are needed. While one could define both of them manually, greater economy can be achieved by recognising that both semirings and rings can be defined by equational theories, for which *varieties* can be defined generically. Varieties are categories consisting of models for a fixed theory with homomorphisms between them.

To this end, we have formalised some of the theory of multisorted universal algebra and equational theories. We chose not to revive existing formalisations (Capretta 1999; Domınguez 2008) of universal algebra, because an important aim for us has been to find out what level of elegance, convenience and integration can be achieved by leveraging the state of the art in Coq facilities (of which type classes are the most important example).

7.1. *Signatures and algebras*

A multisorted signature enumerates sorts and operations, and specifies the ‘types’ of the operations as non-empty lists of sorts, where the final element denotes the result type:

```
Inductive Signature: Type :=
  { sorts: Set
  ; operation:> Set
  ; operation_type:> operation → ne_list sorts }.
```

Given an interpretation of the sorts (mapping each symbolic sort to a carrier type), interpretations of the operations are easily represented by an operational type class:

```
Variables (σ: Signature) (carriers: sorts σ → Type).

Class AlgebraOps :=
  algebra_op: ∀ o: operation σ, fold (→) (map carriers (operation_type σ o)).
```

Because our carriers will normally be equipped with a setoid equality, we further define the predicate class Algebra, stating that each of the operations respects the setoid equality on the carriers:

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

The (=) referred to in algebra_propers is an automatically derived Equiv instance expressing setoid-respecting extensionality for the function types produced by the fold in AlgebraOps.

We do not unbundle Signature because it represents a triple that will always be specifically constructed for subsequent use with the universal algebra facilities. We have no ambition to recognise signature triples ‘in the wild’, nor will we ever talk about multiple signatures sharing sort- or operation enumerations.

7.2. *Equational theories and varieties*

In order to characterise such structures as semirings and rings adequately, we need not just a signature that enumerates and gives the types of their operations, but also a specification of the axioms (laws) that these operations must satisfy. For this, we define EquationalTheory as a signature together with a set of laws, the latter represented by a predicate over equality entailments:

```
Record EquationalTheory :=
  { eqt_sig:> Signature
  ; eqt_laws:> EqEntailment eqt_sig → Prop }.
```

An EqEntailment consists of premises and a conclusion represented by an inductively defined statement grammar, which in turn uses an inductively defined term grammar. However, a detailed discussion of these definitions and the theory developed for them is beyond the scope of this paper.

We now introduce a predicate class designating algebras that satisfy the laws of an equational theory:

```
Class InVariety
  (et: EquationalTheory) (carriers: sorts et → Type)
  {e: ∀ a, Equiv (carriers a)} ‘{AlgebraOps et carriers}: Prop :=
  { variety_algebra:> Algebra et carriers
  ; variety_laws: ∀ s, eqt_laws et s → (∀ vars, eval_stmt et vars s) }.
```

We still need to show that carrier sets together with Equivs and AlgebraOps satisfying InVariety for a given EquationalTheory do indeed form a Category (the ‘variety’). Since we need a type for the objects in the Category, at this point we have no choice but to bundle components and proof together in a record:

```
Variable et: EquationalTheory.

Record ObjectInVariety: Type := object_in_variety
  { variety_carriers:> sorts et → Type
  ; variety_equiv: ∀ a, Equiv (variety_carriers a)
  ; variety_op: AlgebraOps et variety_carriers
  ; variety_proof: InVariety et variety_carriers }.
```

The arrows will be homomorphisms, which are also defined generically for any equational theory:

```
Instance: Arrows Object := λ X Y: Object ⇒ sig (HomoMorphism et X Y).
```

The instance definitions for identity arrows, arrow composition, arrow setoid equality and composition propriety are all trivial, as is the final Category instance:

```
Instance: Category ObjectInVariety.
```

In addition to this variety category, we also have categories of lawless algebras, as well as forgetful functors from the former to the latter, and from the latter to the category of setoids.

7.3. *The first homomorphism theorem*

To give a further taste of what universal algebra in our development looks like, we consider the definitions involved in the first homomorphism theorem (Meinke and Tucker 1993) in more detail.

Theorem 7.1 (first homomorphism theorem). If A and B are algebras, and f is a homomorphism from A to B, then the equivalence relation $\sim$ defined by ‘$a \sim b \leftrightarrow$

$f(a) = f(b)$' is a congruence on A, and the quotient algebra $A/\sim$ is isomorphic to the image of f, which is a subalgebra of B.

A set of relations e (one for each sort) is a congruence for an existing algebra if:

(1) e respects that algebra's existing setoid equality, and
(2) the operations with e again form an algebra (namely the quotient algebra):

```
Context ‘{Algebra σ A}.
Class Congruence (e: ∀ s: sorts σ, relation (v s)): Prop :=
  { congruence_proper:> ∀ s, Proper (equiv ⇒ equiv ⇒ iff) (e s)
  ; congruence_quotient:> Algebra σ v (e:=e) }.
```

We have proved that this natural and economical type-theoretic formulation, which leverages our systematic integration of setoid equality, is equivalent to the traditional definition of congruences as relations that, represented as sets of pairs, form a subalgebra of the product algebra.

For the homomorphism theorem, we begin by declaring our *dramatis personae*:

```
Context ‘{HomoMorphism σ A B f}.
```

With $\sim$ defined as indicated, the first part of the proof is simply the definition of the following instance:

```
Instance co: Congruence σ (∼).
```

For the second part, we describe the image of f as a predicate over B, and show that it is closed under the operations of the algebra:

```
Definition image s (b: B s): Type := sigT (λ a ⇒ f s a = b).

Instance: ClosedSubset image.
```

The sigT type constructor is a Type-sorted existential quantifier. ClosedSubset is defined elsewhere as

```
Context ‘{Algebra σ A} (P: ∀ s, A s → Type).
Class ClosedSubset: Type :=
  { subset_proper: ∀ s x x’, x = x’ → iffT (P s x) (P s x’)
  ; subset_closed: ∀ o, op_closed (algebra_op o) }.
```

Here, op_closed is defined by recursion over the symbolic operation types.

The reason we define image and ClosedSubset in Type rather than in Prop is that since the final goal of the proof is to establish an isomorphism in the category of -algebras (where arrows are algebra homomorphisms), we will eventually need to map elements in the subalgebra defined by image back to their pre-image in A.

However, there are contexts (in other proofs) where Prop-sorted construction of subalgebras really *is* appropriate. Unfortunately, Coq's universe polymorphism is not yet up to the task of letting us use a single set of definitions to handle both cases. In particular, there is no universe polymorphism for ordinary definitions (as opposed to inductive definitions) yet. We will return to this point later. In our development, we have two sets of definitions, one for Prop and one for Type, resulting in duplication of about a hundred lines of code.

For the main theorem, we now bundle the quotient algebra and the subalgebra into records akin to ObjectInVariety from Section 7.2:

```
Definition quot_obj: algebra.Object σ:=
  algebra.object σA (algebra_equiv:=(∼)).
Definition subobject: algebra.Object σ :=
  algebra.object σ(ua_subalgebraT.carrier image).
```

Here, algebra is the module defining the bundled algebra record Object with constructor object. The module ua_subalgebraT constructs subalgebras.

Finally, we define a pair of arrows between the two and show that these arrows form an isomorphism:

```
 Program Definition back: subobject ⟶ quot_obj
  := λ _ X ⇒ projT1 (projT2 X).

Program Definition forth: quot_obj ⟶ subobject
  := λ a X ⇒ existT _ (f a X) (existT _ X (reflexivity _)).

Theorem first_iso: iso_arrows back forth.
```

The Program command generates proof obligations (not shown) expressing the fact that these two arrows are indeed homomorphisms. The proof of the theorem itself is trivial.

8. Numerical interfaces

EquationalTheory's for semirings and rings are easy to define, and so from Section 7.2 we get corresponding categories in which we can postulate initial objects:

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

Although succinct, this definition is not a satisfactory abstraction because the use of ObjectInVariety for the type of the A component ‘leaks’ the fact that we used this one particular universal algebraic construction of the category, which is just an implementation choice. Furthermore, this definition needs an additional layer of class instances to relate it to the SemiRing class from our algebraic hierarchy.

What we *really* want to say is that an implementation of the natural numbers ought to be an *a priori* SemiRing that, when *bundled* into an ObjectInVariety semiring_theory, is initial in said category. This is a typical example where conversion functions between concrete classes such as SemiRing and instantiations of more abstract classes such as InVariety and Category are required in our development in order to leverage and apply concepts and theory defined for the latter to the former. While sometimes a source of some tension in that these conversions are not yet applied completely transparently whenever needed, the ability to move between ‘down to earth’ and ‘high in the sky’ perspectives on the same abstract structures has proved invaluable in our development, and we will give more examples of this in a moment.

Taking these conversion functions for granted, we will also need a ‘down to earth’ representation of the initiality arrows if we are to give a SemiRing-based definition of the interface for natural numbers. Once again, we introduce an operational type class to represent this particular component:

```
  Class NaturalsToSemiRing (A: Type) :=
    naturals_to_semiring: ∀ B ‘{RingMult B} ‘{RingPlus B} ‘{RingOne B}
      ‘{RingZero B}, A → B.
```

The instance for nat is defined as follows:

```
Instance nat_to_semiring: NaturalsToSemiRing nat :=
  λ _ _ _ _ _ ⇒ fix f (n: nat) := match n with 0 ⇒ 0 | S m ⇒ f m + 1 end.
```

To use NaturalsToSemiRing with Initial, we define an additional conversion instance that takes a NaturalsToSemiRing along with a proof showing that it yields SemiRing_Morphisms and builds an InitialArrow instance out of it. This conversion instance in turn invokes another conversion function that translates concrete SemiRing_Morphism proofs into univeral algebra Homomorphisms instantiated with the semiring signature, which make up the arrows in the category.

With these instances in place, we can now define the improved natural numbers specification:

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

Basing theory and programs on this abstract interface instead of on a specific implementation (such as the ubiquitous Peano naturals nat in the Coq standard library) is not only cleaner mathematically, but also facilitates easy swapping between implementations. And this benefit is far from theoretical, as diverse representations of the natural numbers abound; for instance, unary, binary, factor multisets and arrays of native machine words.

Since initial objects in categories are isomorphic, we can easily derive the fact that naturals_to_semiring gives isomorphisms between different Naturals implementations:

```
  Lemma iso_naturals ‘{Naturals A} ‘{Naturals B}:
    ∀ a: A, naturals_to_semiring B A (naturals_to_semiring A B a) = a.
```

This is very useful because some properties of naturals are more easily proved, and operations on them more easily defined, for concrete implementations (such as nat) and then *lifted* to the abstract Naturals interface so that they work for arbitrary implementations. For example, while showing decidability directly for an arbitrary Naturals implementation is tricky, it is very easy to show decidability for nat. Using iso_naturals, the latter can be very straightforwardly used to implement the former.

To lift properties such as injectivity of partially applied addition and multiplication from nat to arbitrary Naturals implementations, we take a longer detour. As part of our universal algebra theory, we have proved that proofs of statements in the language of an equational theory can be transferred between isomorphic implementations. Hence, we can transfer proofs of such statements between implementations of Naturals, requiring only that we reflect the concrete statement (expressed in terms of the operational type classes) to a symbolic statement in the language of semirings. We intend eventually to make this reflection completely automatic using type class based quotation techniques along the lines of those described in Section 9.

Thanks to our close integration of universal algebra, we can actually obtain a Naturals implementation completely automatically by invoking a generic construction of initial models built from the closed term algebra for the signature along with a setoid equality expressing the congruence closure of the identities in the equational theory. However, this implementation is not very useful, neither in terms of efficiency, nor as a canonical implementation (to be used as the basis for theory and programs that are then subsequently lifted). For example, defining a normalisation procedure to decide the aforementioned setoid equality is far harder than deciding equality for, say, nat.

8.1. *Specialisation*

The generic Decision instance for Naturals equality implemented by mapping to nat will typically be far less efficient than a specialised implementation for a particular representation of the natural numbers. Fortunately, with Coq's type classes, it is no problem for instances overlapping in this way to co-exist. We can even deprioritise the generic instance so that instance resolution will always pick the specialisation when the representation is known.

To permit a *generic* function operating on naturals to take advantage of specialised operations, we simply introduce an additional instance parameter:

```
Definition calculate_things ‘{Naturals N} ‘{∀ n m: N, Decision (n = m)}
  (a b: nat): ... := ... decide (a = b) ... .
```

Without the Decision parameter, calculate_things would be equally correct, but could be less efficient. Thus, using this scheme, one can start by writing correct-but-possibly-inefficient programs that make use of generic operation instances, and then selectively improve efficiency of key algorithms simply by adding additional operational type class instance parameters where profiling shows it to make a significant difference, and without changing their definition body.

Other examples of operations on natural numbers that are sensible choices for specialisation include subtraction, distance, and division and multiplication by 2.

8.2. *Integers, rationals and polynomials*

The abstract interface for integers is completely analogous to the one for natural numbers:

```
Context ‘{Ring A} ‘{IntegersToRing A}.
Class Integers: Prop :=
  { integers_ring:> Ring A
  ; integers_to_ring_mor:> ∀ ‘{Ring B},
      Ring_Morphism (integers_to_ring A B)
  ; integers_initial:> Initial (ring.object A) }.
```

The rationals are characterised as a decidable field with an injective ring morphism from a canonical implementation of the integers and a surjection of fractions of such integers:

```
Context ‘{Field A} ‘{∀ x y: A, Decision (x = y)} {inj_inv}.
Class Rationals: Prop :=
  { rationals_field:> Field A
  ; rationals_frac: Surjective
      (λ  p  ⇒ integers_to_ring (Z nat) A (fst p) ∗
        / integers_to_ring (Z nat) A (snd p)) (inv:=inj_inv)
  ; rationals_embed_ints: Injective (integers_to_ring (Z nat) A) }.
```

Here, Z is an Integers implementation paramerised by a Naturals implementation, for which we just take nat. The choice of Z nat here is immaterial; we could have picked another, or even a generic, implementation of Integers, but doing so would provide no benefit.

In our development, we prove that the standard library's default rationals do indeed implement Rationals, as do implementations of the QType module interface. While the latter is rather *ad hoc* from a theoretical perspective, it is nevertheless of great practical interest because it is used for the very efficient BigQ rationals based on machine integers (Armand *et al.* 2010). Hence, the theory and programs developed on our Rationals interface applies and we can make immediate use of these efficient rationals. We plan to rebase the computable real number implementation (O'Connor 2008) on this interface, precisely so that it may be instantiated with efficient implementations like these.

We also plan to provide an abstract interface for polynomials as a free commutative algebra. This would unify existing implementations such as coefficient lists and Bernstein polynomials – see Zumkeller (2008) for the latter.

9. Quoting with type classes

A common need when interfacing generic theory and utilities developed for algebraic structures (such as normalisation procedures) with concrete instances of these structures is to take a concrete expression or statement in a model of a particular algebraic structure, and translate it to a symbolic expression or statement in the language of the algebra's signature so that its structure can be inspected.

Traditionally, proof assistants such as Coq have provided sophisticated tactics or built-in commands to support such *quoting*. Unification hints (Asperti *et al.* 2009), a very general way of facilitating user-defined extensions to term and type inference, can be used to semi-automatically build quote functions without dropping to a meta-level[†]. This feature is absent from Coq, but, fortunately, type classes also allow us to do this, as we will now show.

For ease of presentation, we will only show a proof of concept for a very concrete language. We are currently working to integrate this technique with our existing universal algebra infrastructure. In particular, the latter's term data type should be ideally suited to serve as a generic symbolic representation of terms in a wide class of algebras. This should let us implement the basic setup of the technique once and for all so that quotation for new algebraic structures can be enabled with minimal effort.

For the present example, we define an *ad hoc* term language for monoids:

```
Inductive Expr (V: Type) := Mult (a b: Expr V) | One | Var (v: V).
```

The expression type is parameterised over the set of variable indices. In the following, we use an implicitly defined heap of such variables. Hence, we diverge from Asperti *et al.* (2009), which uses nat for variable indices, thereby introducing a need for dummy variables for out-of-bounds indices.

Suppose now that we want to quote nat expressions built from 1 and multiplication. To describe the relation we want the symbolic expression to have to the original expression, we first define how symbolic expressions evaluate to values (given a variable assignment):

```
Definition Value := nat.
Definition Env V := V → Value.

Fixpoint eval {V} (vs: Env V) (e: Expr V): Value :=
  match e with
  | One ⇒ 1
  | Mult a b ⇒ eval vs a ∗ eval vs b
  | Var v ⇒ vs v
  end.
```

We can now state our goal: given an expression of type nat, we seek to construct an Expr V for some appropriate V along with a variable assignment such that evaluation of the latter yields the former. Because we will be doing this incrementally, we introduce a few simple variable 'heap combinators':

```
Definition novars: Env False := False_rect _.
Definition singlevar (x: Value): Env unit := λ _ ⇒ x.
Definition merge {A B} (a: Env A) (b: Env B): Env (A+B) :=
  λ i ⇒ match i with inl j ⇒ a j | inr j ⇒ b j end.
```

These last two combinators are the 'constructors' of an implicitly defined subset of Gallina terms, representing heaps, for which we will implement syntactic lookup with type classes in a moment. The heap can also be defined explicitly, with no essential change in the code.

[†] Gonthier provides similar functionality through an ingenious use of canonical structures.

With these, we can define the primary ingredient, the Quote class:

```
Class Quote {V} (l: Env V) (n: Value) {V'} (r: Env V'): Type :=
  { quote: Expr (V + V')
  ; eval_quote: eval (merge l r) quote = n }.
```

We can think of Quote as the type for a family of Prolog-like syntax-directed resolution functions, which will take as input V and l representing previously encountered holes (opaque subexpressions that could not be destructured further) and their values, along with a concrete term n to be quoted. Their 'output' will consist not only of the fields in the class, but also of V' and r representing additional holes and their values. Hence, a type class constraint of the form Quote x y z should be read as 'quoting y with existing heap x generates new heap z'.

The Quote instance for 1 illustrates the basic idea:

```
Instance quote_one V (v: Env V): Quote v 1 novars := { quote := One }.
```

The expression '1' can be quoted in any context (V, v) – it introduces no new variables, and the symbolic term representing it is just One. The eval_quote field is turned into a trivial proof obligation.

The Quote instance for multiplication is a little more subtle, but really only does a bit of heap juggling:

```
Instance quote_mult V (v: Env V) n V' (v': Env V') m V'' (v'': Env V'')
  `{Quote v n v'} `{Quote (merge v v') m v''}:
    Quote v (n ∗ m) (merge v' v'') :=
    { quote :=
      Mult (map_var shift (quote n)) (map_var sum_assoc (quote m)) }.
```

These two instances specify how 1 and multiplications are to be quoted, but what about other expressions? For these, we want to distinguish between expressions we have seen before, and those we have not. To make this distinction, we need to be able to look up expressions in variable heaps to see if they are already there. Importantly, we must not do this by comparing the values they evaluate to, but by actually browsing the term denoting the variable heap – that is, a composition from novars, singlevar and merge. This, too, is a job for a type class:

```
Class Lookup {A} (x: Value) (v: Env A) := { key: A; key_correct: v key = x }.
```

Our first Lookup instance states that x can be looked up in singlevar x:

```
Instance singlevar_lookup (x: Value): Lookup x (singlevar x) := { key := tt }.
```

Finally, if an expression can be looked up in a pack, then it can also be looked up when that pack is merged with another pack:

```
Context (x: Value) {A B} (va: Env A) (vb: Env B).

Instance lookup_left `{Lookup x va}: Lookup x (merge va vb)
  := { key := inl (key x va) }.
```

```
Instance lookup_right '{Lookup x vb}: Lookup x (merge va vb)
  := { key := inr (key x vb) }.
```

With Lookup, we can now define a Quote instance for previously encountered expressions:

```
Instance quote_old_var V (v: Env V) x {Lookup x v}:
  Quote v x novars | 8 := { quote := Var (inl (key x v)) }.
```

If none of the Quote instances defined so far apply, the term in question is a newly encountered hole. For this case, we define a catch-all instance with a low priority, which yields a singleton heap containing the expression:

```
Instance quote_new_var V (v: Env V) x: Quote v x (singlevar x) | 9
  := { quote := Var (inr tt) }.
```

And with that, we can now start quoting:

```
Goal ∀ x y (P: Value → Prop), P ((x ∗ y) ∗ (x ∗ 1)).
  intros.
  rewrite ← eval_quote.
```

The rewrite rewrites the goal to (something that reduces to):

```
P (eval
    (merge novars
       (merge (merge (singlevar x) (singlevar y)) (merge novars novars)))
    (Mult (Mult (Var (inr (inl (inl ())))) (Var (inr (inl (inr ())))))
       (Mult (Var (inr (inl (inl ())))) One)))
```

The following additional utility lemma lets us quote equalities with a shared heap (so that an opaque expression that occurs on both sides of the equation is not represented by two distinct variables):

```
Lemma quote_equality {V} {v: Env V} {V'} {v': Env V'} (l r: Value)
  '{Quote novars l v} '{Quote v r v'}:
    let heap := merge v v' in
    eval heap (map_var shift quote) = eval heap quote → l = r.
```

Notice that we have not made any use of Coq's tactic language Ltac. Instead, we have used instance resolution as a unification-based programming language to steer the unifier into inferring the symbolic quotation.

10. Sequences and universes

Finite sequences are another example of a concept that can be represented in many different ways: as cons lists; maps from bounded naturals; array-queues; and so on. Here, too, the introduction of an abstract interface facilitates implementation independence.

Mathematically, finite sequences can be characterised as free monoids over sets. A categorical way of expressing this is in terms of adjunctions. As with the numeric interfaces, we *could* fully embrace this perspective, paying no heed to the practicality of implementation and usage, and define a relatively succinct type class for sequences as follows:

```
Class PoshSequence
  (free: setoid.Object → monoid.Object) ‘{Fmap free}
  (singleton: id ⟹ monoid.forget ∘ free)
  (extend: ‘((x ⟶ monoid.forget y) → (free x ⟶ y))): Prop :=
  { sequence_adjunction: AltAdjunction singleton extend
  ; extend_morphism: ‘(Setoid_Morphism (extend x y)) }.
```

Here, monoid.forget is the forgetful functor from monoids to sets.

However, we *do* care about practicality, so we will again take a more concrete perspective, starting with operational type classes for the characteristic operations:

```
Context ‘{Functor (seq: Type → Type)}.

Class Extend := extend: ∀ {x y} ‘{SemiGroupOp y} ‘{MonoidUnit y},
  (x → y) → (seq x → y).
Class Singleton := singleton: ∀ x, x → seq x.
```

With these, we can now define the predicate class for sequences:

```
Class Sequence
 ‘{∀ a, MonoidUnit (seq a)} ‘{∀ a, SemiGroupOp (seq a)}
 ‘{∀ a, Equiv a → Equiv (seq a)} ‘{Singleton} ‘{Extend}: Prop := ...
```

On top of this interface, we can build theory about typical sequence operations such as maps, folds, their relation to singleton and extend, and so on. We can also generically define ‘big operators’ for sums ($\sum$) and products ($\prod$) of sequences, and easily show properties like distributivity, all without ever mentioning cons lists.

Unfortunately, disaster strikes when, after having defined this theory, we try to show that regular cons lists implement the abstract Sequence interface. When we get to the point where we want to define the Singleton operation, Coq emits a universe inconsistency error. The problem is that because of the categorical constructions involved, the theory forces Singleton to inhabit a relatively high universe level, making it incompatible with lowly list.

In principle, universe polymorphism could probably be used to solve this problem, but its current implementation in Coq only supports universe polymorphic *inductive* definitions, while Singleton is a regular definition. Historically, universe polymorphic regular definitions have not been supported in Coq, primarily because of efficiency concerns. However, we have taken up the issue with the Coq development team, and they have agreed to introduce a mechanism for voluntarily turning on universe polymorphism for definitions on a per-definition basis. Using this functionality, we could make Singleton universe polymorphic, and hopefully resolve these problems.

We have encountered universe inconsistencies in other places in our development that could be traced back to universe monomorphic definitions being forced into disparate universes (Equiv being a typical example). Hence, we consider the support for universe polymorphic definitions that is currently being implemented to be of great importance to the general applicability and scalability of our approach.

11. Conclusions

While bundling operational and propositional components of abstract structures into records may seem natural at first, doing so actually introduces many serious problems. With type classes, we avoid these problems by avoiding bundling altogether.

It has been suggested that canonical structures are more robust because of their more restricted nature compared to the wild and open-ended proof search of instance resolution. However, these restrictions force one into bundled representations, and, moreover, their more advanced usage requires significant ingenuity, whereas type class usage is straightforward. Furthermore, wild and open-ended proof search is harmless for predicate classes, for which only existence, and not identity, matters.

Unification hints are a more general mechanism than type classes, and could provide a more precise account of the interaction between implicit argument inference and proof search. It is not a great stretch to conjecture that a fruitful approach might be to use unification hints as the underlying mechanism, with type classes as an end-user interface encapsulating a particularly convenient idiom for using them.

There are really only two pending concerns that keeps us from making an unequivocal endorsement of type classes as a versatile, expressive and elegant means of organising proof developments. The first, and lesser, of the two is universe polymorphism for definitions as described in the previous section. The second is instance resolution efficiency. In more complex parts of our development, we are now experiencing increasingly serious efficiency problems, despite having already made sacrifices by artificially inhibiting many natural class instances in order not to further strain instance resolution. Fortunately, there is plenty of potential room for improvement of the current instance resolution implementation. One source is the vast literature on efficient implementation of Prolog-style resolution, which the hint-based proof search used for instance resolution greatly resembles. We emphasise that these efficiency problems only affect type checking; the efficiency of computation using type-checked terms is not affected.

We are currently in the process of retrofitting the rationals interface into CoRN. In future work, we aim to base our development of its reals on an abstract dense set, allowing us to use the efficient dyadic rationals (Boldo *et al.* 2009) as a base for exact real number computation in Coq (O'Connor 2008; O'Connor and Spitters 2010). The use of category theory has been important in these developments.

An obvious topic for future research is the extension from equational logic with dependent types (Cartmell 1978; Palmgren and Vickers 2007). Another topic would be to fully, but practically, embrace the categorical approach to universal algebra (Pitts 2001).

According to coqwc, our development consists of 5660 lines of specifications and 937 lines of proofs.

Acknowledgements

This research was stimulated by the new possibilities provided by the introduction of type classes in Coq. In fact, this seems to be the first substantial development that tries to exploit all their possibilities. As a consequence, we found many small bugs and unintended behaviour in the type class implementation. All these issues were quickly solved by Matthieu Sozeau.

Discussions with Georges Gonthier and Claudio Sacerdoti Coen have helped us to sharpen our understanding of the relation with Canonical Structures and with Unification Hints.

Finally, we are grateful to James McKinna, who suggested many improvements.

References

Armand, M., Grégoire, B., Spiwack, A. and Théry, L. (2010) Extending Coq with imperative features and its application to SAT verification. In: Kaufmann, M. and Paulson, L. (eds.) Proceedings, Interactive Theorem Proving, ITP 2010. *Springer-Verlag Lecture Notes in Computer Science* **6172** 83–98.

Asperti, A., Ricciotti, W., Sacerdoti Coen, C. and Tassi, E. (2009) Hints in unification. In: Berghofer, S., Nipkow, T. Urban, C. and Wenzel, M. (eds.) Theorem Proving in Higher Order Logics, 22nd International Conference (TPHOLs 2009). *Springer-Verlag Lecture Notes in Computer Science* **5674** 84–98.

Asperti, A., Sacerdoti Coen, C., Tassi, E. and Zacchiroli, S. (2007) User interaction with the Matita proof assistant. *Journal of Automated Reasoning* **39** (2) 109–139.

Barthe, G., Capretta, V. and Pons, O. (2003) Setoids in type theory. *Journal of Functional Programming* **13** (2) 261–293. (Special issue on 'Logical frameworks and metalanguages'.)

Bertot, Y. and Castéran, P. (2004) *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*, Texts in Theoretical Computer Science, Springer-Verlag.

Bishop, E. A. (1967) *Foundations of constructive analysis*, McGraw-Hill.

Boldo, S., Filliâtre, J. and Melquiond, G. (2009) Combining Coq and Gappa for Certifying Floating-Point Programs. In: Carette, J., Dixon, L., Coen, C. and Watt, S. (eds.) Intelligent Computer Mathematics: Proceedings of the 16th Symposium, Calculemus 2009, held as Part of CICM 2009. *Springer-Verlag Lecture Notes in Computer Science* **5625** 59–74.

Brus, T. H., van Eekelen, C. J. D., van Leer, M. O. and Plasmeijer, M. J. (1987) Clean – A language for functional graph rewriting. In: Kahn, G. (ed.) Functional programming languages and computer architecture. *Springer-Verlag Lecture Notes in Computer Science* **274** 364–384.

Capretta, V. (1999) Universal algebra in type theory. In: Bertot, Y., Dowek, G., Hirschowitz, A., Paulin, C. and Théry, L. (eds.) Theorem Proving in Higher Order Logics (TPHOLs 1999) *Springer-Verlag Lecture Notes in Computer Science* **1690** 131–148.

Cartmell, J. (1978) *Generalized algebraic theories and contextual categories*, Ph.D. thesis, University of Oxford.

Coq Development Team (2008) The Coq Proof Assistant Reference Manual, INRIA-Rocquencourt.

Coquand, T. and Huet, G. (1988) The calculus of constructions. *Information and Computation* **76** (2-3) 95–120.

Coquand, T. and Paulin, C. (1990) Inductively defined types. In: Martin-Löf, P. and Mints, G. (eds.) COLOG-88. *Springer-Verlag Lecture Notes in Computer Science* **417** 50–66.

Cruz-Filipe, L., Geuvers, H. and Wiedijk, F. (2004) C-CoRN, the Constructive Coq Repository at Nijmegen. In: Asperti, A., Bancerek, G. and Trybulec, A. (eds.) Proceedings of MKM2004. *Springer-Verlag Lecture Notes in Computer Science* **3119** 88–103.

Domınguez, C. (2008) Formalizing in Coq Hidden Algebras to Specify Symbolic Computation Systems. In: Autexier, S. *et al.* (eds.) Intelligent Computer Mathematics: Proceedings 9th International Conference, AISC 2008. *Springer-Verlag Lecture Notes in Computer Science* **5144** 270–284.

Garillot, F., Gonthier, G., Mahboubi, A. and Rideau, L. (2009) Packaging mathematical structures. In: Berghofer, S., Nipkow, T., Urban, C. and Wenzel, M. (eds.) Theorem Proving in Higher Order Logics, 22nd International Conference (TPHOLs 2009). *Springer-Verlag Lecture Notes in Computer Science* **5674** 327–342.

Geuvers, H., Pollack, R., Wiedijk, F. and Zwanenburg, J. (2002) A constructive algebraic hierarchy in Coq. *Journal of Symbolic Computation* **34** (4) 271–286.

Haftmann, F. and Wenzel, M. (2008) Local theory specifications in Isabelle/Isar. In: Berardi, S., Damiani, F. and de'Liguoro, U. (eds.) Types for Proofs and Programs International Conference, TYPES 2008. *Springer-Verlag Lecture Notes in Computer Science* **5497** 153–168.

Hofmann, M. (1997) *Extensional constructs in intensional type theory*, CPHC/BCS Distinguished Dissertations, Springer-Verlag.

Huet, G. and Saibi, A. (1995) Constructive category theory. In: *Proceedings of the Joint CLICS-TYPES Workshop on Categories and Type Theory, Göteborg.*

Jenks, R., Sutor, R. and Morrison, S. (1992) *AXIOM: the scientific computation system*, Springer-Verlag.

Mac Lane, S. (1998) *Categories for the working mathematician*, Springer-Verlag.

Martin-Löf, P. (1982) Constructive mathematics and computer programming. In: Cohen, L. J., Los, J., Pfeiffer, H. and Podewski, K.-P. (eds.) *Logic, methodology and philosophy of science, VI*, Studies in Logic and the Foundations of Mathematics, North-Holland **104** 153–175.

Martin-Löf, P. (1998) An intuitionistic theory of types. In: *Twenty-five years of constructive type theory*, Oxford Logic Guides **36**, Oxford University Press 127–172.

Meinke, K. and Tucker, J. (1993) Universal algebra. In: Abramsky, S., Gabbay, D. and Maibaum, T. (eds.) *Handbook of logic in computer science (volume 1)*, Oxford University Press 189–411.

O'Connor, R. (2008) Certified exact transcendental real number computation in Coq. Mohamed, O. A., Muñoz, C. and Tahar, S. (eds.) Proceedings of the 21st International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2008). *Springer-Verlag Lecture Notes in Computer Science* **5170** 246–261.

O'Connor, R. and Spitters, B. (2010) A computer verified, monadic, functional implementation of the integral. *Theoretical Computer Science* **411** (37) 3386–3402.

Palmgren, E. (2009) Constructivist and Structuralist Foundations: Bishops and Lawveres Theories of Sets. Technical Report 4.

Palmgren, E. and Vickers, S. (2007) Partial Horn logic and cartesian categories. *Annals of Pure and Applied Logic* **145** (3) 314–353.

Pitts, A. (2001) Categorical logic. *Handbook of Logic in Computer Science: Logic and algebraic methods (volume 5)*, Oxford University Press 39–123.

Pollack, R. (2002) Dependently typed records in type theory. *Formal Aspects of Computing* **13** 386–402.

Sacerdoti Coen, C. and Tassi, E. (2008) Working with mathematical structures in type theory. In: Miculan, M., Scagnetto, I. and Honsell, F. (eds.) Types for Proofs and Programs. *Springer-Verlag Lecture Notes in Computer Science* **4941** 157–172.

Santas, P. (1995) A type system for computer algebra. *Journal of Symbolic Computation* **19** (1-3) 79–109.

Sozeau, M. (2009) A new look at generalized rewriting in type theory. *Journal of Formalized Reasoning* **2** (1) 41–62.

Sozeau, M. and Oury, N. (2008) First-class type classes. In: Mohamed, O. A., Muñoz, C. and Tahar, S. (eds.) Proceedings of the 21st International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2008). *Springer-Verlag Lecture Notes in Computer Science* **5170** 278–293.

Wadler, P. and Blott, S. (1989) How to make ad-hoc polymorphism less ad hoc. In: *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, ACM 60–76.

Weber, A. and Klaeren, H. (1993) Type systems for computer algebra. *Relation* **10** (1.54) 2615.

Zumkeller, R. (2008) *Global Optimization in Type Theory*, Ph.D. thesis, École Polytechnique, Paris.