Leveraging the Information Contained in Theory Presentations

Jacques Carette, William M. Farmer, and Yasmine Sharoda(✉)

Computing and Software, McMaster University, Hamilton, Canada
{carette,wmfarmer,sharodym}@mcmaster.ca
http://www.cas.mcmaster.ca/research/mathscheme/

Abstract. A theorem prover without an extensive library is much less useful to its potential users. Algebra, the study of algebraic structures, is a core component of such libraries. Algebraic theories also are themselves structured, the study of which was started as Universal Algebra. Various constructions (homomorphism, term algebras, products, etc.) and their properties are both universal and constructive. Thus they are ripe for being automated. Unfortunately, current practice still requires library builders to write these by hand. We first highlight specific redundancies in libraries of existing systems. Then we describe a framework for generating these derived concepts from theory definitions. We demonstrate the usefulness of this framework on a test library of 227 theories.

Keywords: Formal library · Algebraic hierarchy

1 Introduction

A theorem prover on its own is not nearly as useful for end-users as one equipped with extensive libraries. Most users have tasks to perform that are not related to new ideas in theorem proving. The larger the library of standard material, the faster that users can just get to work. However building large libraries is currently very labor intensive. Although some provers provide considerable automation for proof development, they do not the same for theory development.

This is the problem we continue [1,6,8,9] to tackle here, and that others [11] have started to look at as well. It is worthwhile noting that some programming languages already provide interesting features in this direction. For example, Haskell [22] provides the *deriving* mechanism that lets one get instances for some classes "for free"; recently, the *Deriving Via* mechanism [2] has been introduced, that greatly amplifies these features. Some libraries, such as the one for *Lens* [24], use *Template Haskell* [33] for the same purpose.

Libraries of algebra define algebraic structures, constructions on these, and properties satisfied by the structures and constructions. While structures like `Semigroup`, `Monoid`, `AbelianGroup`, `Ring` and `Field` readily come to mind, a look at compendiums [21,23] reveals a much larger zoo of hundreds of structures.

C. Benzmüller and B. Miller (Eds.): CICM 2020, LNAI 12236, pp. 55–70, 2020.
https://doi.org/10.1007/978-3-030-53518-6_4

Haskell
```
class Semiring a => Monoid a
 where
  mempty :: a
  mappend :: a -> a -> a
  mappend = (<>)
  mconcat :: [a] -> a
  mconcat =
   foldr mappend mempty
```
Coq
```
class Monoid {A : type}
 (dot : A → A → A)
 (one : A) : Prop := {
  dot_assoc : forall x y z : A,
  (dot x (dot y z)) =
  dot (dot x y) z
  unit_left : forall x,
  dot one x = x
  unit_right : forall x,
  dot x one = x
}
```
Alternative Definition:
```
Record monoid := {
 dom : Type;
 op : dom -> dom -> dom
  where "x * y" := op x y;
 id : dom where "1" := id;
 assoc : forall x y z,
  x * (y * z) = (x * y) * z;
 left_neutral : forall x,
  1 * x = x;
 right_neutal : forall x,
  x * 1 = x;
}
```
MathScheme
```
Monoid := Theory {
 U : type;
 * : (U,U) → U;
 e : U;
 axiom right_identity_*_e :
  forall x : U · (x * e) = x;
 axiom left_identity_*_e :
  forall x : U · (e * x) = x;
 axiom associativity_* :
  forall x,y,z : U ·
  (x * y) * z = x * (y * z);
}
```

Agda
```
record Monoid c ℓ :
   Set (suc (c ⊔ ℓ)) where
 infixl 7 _•_
 infix 4 _≈_
 field
  Carrier : Set c
  _≈_ : Rel Carrier ℓ
  _•_ : Op₂ Carrier
  isMonoid : IsMonoid _≈_ _•_ ε
where  IsMonoid  is defined as
record IsMonid (• : Op₂) (ε : A)
 : Set (a ⊔ ℓ) where
  field
   isSemiring : IsSemiring •
   identity : Identity ε
   identityˡ : LeftIdentity ε •
   identityˡ : proj₁ identity
   identityʳ : Rightdentity ε •
   identityʳ : proj₂ identity
```

MMT
```
theory Semigroup : ?NatDed =
 u : sort
 comp : tm u → tm u → tm u
  # 1 * 2 prec 40
 assoc : ⊢ ∀ [x, y, z]
  (x * y) * z = x * (y * z)
 assocLeftToRight :
  {x,y,z} ⊢ (x * y) * z
          = x * (y * z)
  = [x,y,z]
   allE (allE (allE assoc x) y) z
 assocRightToLeft :
  {x,y,z} ⊢ x * (y * z)
           = (x * y) * z
  = [x,y,z] sym assocLR
theory Monoid : ?NatDed
 includes ?Semigroup
 unit : tm u # e
 unit_axiom : ⊢ ∀ [x] = x * e = x
```

Fig. 1. Representation of `Monoid` theory in different languages.

Picking `Monoid` as an example, it is a structure with a carrier set, an associative binary operation and an identity element for the binary operation. Different

systems implement `Monoid` in different ways (see Fig. 1). Other than layout and vocabulary, different libraries also make more substantial choices:

- Whether declarations are arguments or fields.
- The packaging structure — whether theory, record, locale, etc.
- The underlying notion of equality.

Some of these choices are mathematically irrelevant—in the sense that the resulting theories can be proved to be equivalent, internally or externally—while others are more subtle, such as the choice of equality.

A useful construction on top of `Monoid` is the homomorphism between two of its instances, which maps elements of the carrier of the first instance to that of the second one such that structure is preserved. For an operation `op` and a function `hom`, the preservation axiom has the form

$$\texttt{hom (op } x_1 \texttt{ .. } x_n\texttt{) = op (hom } x_1\texttt{) .. (hom } x_n\texttt{)}$$

One can see that this definition can be "derived" from that of `Monoid`. And that, in fact, this derivation is uniform in the "shape" of the definition of `Monoid`, so that this construction applies to any single-sorted equational theory. This observation is one of the cornerstones of Universal Algebra [35].

There are other classical constructions that can also be generated. This poses a number of questions:

- What other information can be generated from theory presentations?
- How would this affect the activity of library building?
- What pieces of information are needed for the system to generate particular constructions?

Theories written in equational logic that describe algebraic structures are rich in implicit information that can be extracted automatically.

There are obstacles to this automation. For example, definitional and "bundling" choices can make reuse of definitions from one project in another with different aims difficult. Thus users resort to redefining constructs that have already been formalized. We then end up with multiple libraries for the same topic in the same system. For example, there are at least four algebra libraries in Coq [17,18,30,34], and even more for Category Theory [19]. In [17], the authors mention, referring to other libraries:

> "In spite of this body of prior work, however, we have found it difficult to make practical use of the algebraic hierarchy in our project to formalize the Feit-Thompson Theorem in the Coq system."

Universal Algebra [29,31,35] provides us with tools and abstractions well-suited to this task. It is already used in providing semantics and specifications of computer systems [14,15,32] and has been formalized in Coq [3] and Agda [20]. We use Universal Algebra abstractions as basis for our framework to automate the generation of useful information from the definition of a theory. We use Tog

to realize our framework[1]. Tog is a small implementation of a dependent type theory, in the style of Agda, Idris and Coq. It serves well as an abstraction over the design details of different systems. Studying theory presentations at this level of abstraction is the first step to generating useful constructions for widely used systems, like Agda, Coq, Isabelle and others.

In Sect. 2 we highlight some of the redundancies in current libraries. We present our framework for mechanizing the generation of this information in Sect. 3. We follow this with a discussion of related work in Sect. 4 and a conclusion and future work in Sect. 5.

2 Algebra in Current Libraries

Our first observation is that current formalizations of Algebra contain quite a bit of information that is "free" in the sense that it can be mechanically generated from basic definitions. For example, given a theory X, it is mechanical to define X-homomorphisms. To do this within a system is extremely difficult, as it would require introspection and for theory *definitions* to be first-class citizens, which is not the case for any system based on type-theory that we are aware of. Untyped systems in the Lisp tradition do this routinely, as does Maude [10], which is based on *rewriting logic*; the downside is that there is no difference between meaningful and meaningless transformations in these systems, only between "runs successfully" and "crashes". However, these constructions are fully typeable and, moreover, are not system-specific (as they can be phrased meta-theoretically within Universal Algebra), even though an implementation has to be aware of the syntactic details of each system.

Lest the reader think that our quest is a little quixotic, we first look at current libraries from a variety of systems, to find concrete examples of human-written code that could have been generated. We look at Agda, Isabelle/HOL and Lean in particular. More specifically, we look at version 1.3 of the Agda standard library, the 2019 release of the Isabelle/HOL library and Lean's mathlib, where we link to the proper release tag.

We use the theory `Monoid` as our running example, and we highlight the reusable components that the systems use to make writing the definitions easier and more robust.

2.1 Homomorphism

How do the libraries of our three systems[2] represent homomorphism?

Agda defines `Monoid` homomorphism, indirectly, in two ways. First, a predicate encapsulating the proof obligations is defined, which is layered on top of the

[1] The implementation is available at https://github.com/ysharoda/tog.

[2] We do not have enough room to give an introduction to each system; hopefully each system's syntax is clear enough for the main ideas to come through.

predicate for Semigroup homomorphism. This is then used to define homomorphisms themselves.

```
module _ {c₁ ℓ₁ c₂ ℓ₂}
         (From : Monoid c₁ ℓ₁)
         (To   : Monoid c₂ ℓ₂) where

 private
  module F = Monoid From
  module T = Monoid To

 record IsSemigroupMorphism (⟦_⟧:Morphism)
        : Set(c₁ ⊔ ℓ₁ ⊔ c₂ ⊔ ℓ₂) where
  field
   ⟦⟧-cong : ⟦_⟧ Preserves F._≈_ → T._≈_
   ·-homo  : Homomorphic₂ ⟦_⟧ F._·_ T._·_
   ...
 record IsMonoidMorphism (⟦_⟧:Morphism)
        : Set(c₁ ⊔ ℓ₁ ⊔ c₂ ⊔ ℓ₂) where
  field
   sm-homo : IsSemigroupMorphism F.semigroup T.semigroup ⟦_⟧
   ε-homo  : Homomorphic₀ ⟦_⟧ F.ε T.ε

 open IsSemigroupMorphism sm-homo public
```

There are many design decisions embedded in the above definitions. These decisions are not canonical, so we need to understand them to later be able to both abstract them out and make them variation points in our generator. Namely, these decisions are:

- The choice of which declarations are parameters and which are fields. The monoids (`From` and `To`) over which we define homomorphism are parameters, not fields, as is the function ⟦_⟧.
- The preservation axioms can be defined based on their arity patterns, as type-level function such as `Homomorphic₂`:

  ```
  Homomorphic₂ : (A → B) → Op₂ A → Op₂ B → Set _
  Homomorphic₂ ⟦_⟧ _·_ _∘_ =
    ∀ x y → ⟦ x · y ⟧ ≈ (⟦ x ⟧ ∘ ⟦ y ⟧)
  ```

  The library also provides shortcuts for 0-ary and 1-ary function symbols, the most common cases.
- The definition of structures over setoids. Thus equalities need to be preserved, and that is what the `⟦⟧-cong` axiom states.

Isabelle/HOL provides the following definition of monoid homomorphism:

```
definition
hom :: "_ ⇒ _ ⇒ ('a ⇒ 'b) set" where
  "hom G H =
  {h · h ∈ carrier G → carrier H ∧
```

```
  (∀ x ∈ carrier G · ∀ y ∈ carrier G ·
    h (x ⊕G y) = h x ⊕H h y)}"
```

The reader might notice a discrepancy in the above: unit preservation is missing. The Isabelle library does not provide this version. There is, however, a proof that such a multiplication-preserving homomorphism necessarily maps the source unit to a unit of the image (sub)monoid, but that unit is not necessarily that of the full image. The above definition is also used to define group homomorphism and other structures. We consider this to be missing information in the library.

Lean's definition of monoid homomorphism is the one that most resembles the one found in textbooks.

```
structure monoid_hom (M : Type*) (N : Type*)
 [monoid M] [monoid N] :=
  (to_fun : M → N)
  (map_one' : to_fun 1 = 1)
  (map_mul' : ∀ x y, to_fun (x * y) = to_fun x * to_fun y)
```

However, in the same file, there is another definition of `add_monoid_hom` that looks "the same" up to renaming. This points to a weakness of Lean: there is no renaming operation on `structure`, and for a `Ring` to contain two "monoids", one is forced to duplicate definitions. This redundancy is unpleasant.

2.2 Term Language

The "term language" of a theory is the (inductive) data type that represents the syntax of well-formed terms of that theory, along with an interpretation function from *expressions* to the carrier of the (implicitly single-sorted) given theory, i.e. its denotational semantics.

In Agda, the definition of `Monoid` term language is straightforward:

```
data Expr (n : ℕ) where
  var : Fin n → Expr n
  id : Expr n
  _⊕_ : Expr n → Expr n → Expr n
```

Defining the interpretation function requires the concept of an environment. An environment associates a value to every variable, and the semantics associates a value (of type `Carrier`) to each expression of `Expr`.

```
Env : Set _
Env = λ n → Vec Carrier n

⟦_⟧ : ∀ {n} → Expr n → Env n → Carrier
⟦ var x ⟧ ρ = lookup ρ x
⟦ id ⟧ ρ = ϵ
⟦ e₁ ⊕ e₂ ⟧ ρ = ⟦ e₁ ⟧ ρ · ⟦ e₂ ⟧ ρ
```

In Agda, these definitions are not found with the definitions of the algebraic structures themselves, but rather as part of the *Solver* for equations over that theory. Here, we find more duplication, as the above definitions are repeated for the following three highly related structures: `Monoid`, `CommutativeMonoid` and `IdempotentCommutativeMonoid`.

Despite its usefulness, we were not able to find the definition of the term language of a theory in Isabelle/HOL or Lean.

2.3 Product

Until recently, there was no definition of the product of algebraic structures in the Agda library. A recent pull request has suggested adding these, along with other constructions. The following hand-written definition has now been added:

```
rawMonoid : RawMonoid c cℓ → RawMonoid d dℓ →
RawMonoid (c ⊔ d) (cℓ ⊔ dℓ)
rawMonoid M N = record
  { Carrier = M.Carrier × N.Carrier
  ; _≈_ = Pointwise M._≈_ N._≈_
  ; _·_ = zip M._·_ N._·_
  ; ε = M.ε , N.ε
  }
  where
  module M = RawMonoid M
  module N = RawMonoid N
```

These could have been mechanically generated from the definition of `Monoid`.

Both Isabelle/HOL and Lean provide definitions of product algebras for monoids, which we omit for space. It is worth mentioning that the Lean library has 15 definitions for products of structures that look very similar and could be generated.

2.4 More Monoid-Based Examples

We have presented three concrete examples, based on monoid, of human-written code in current libraries that could have instead been generated. There are many more that could be, although these are sparsely found in current libraries. We continue to use monoid as our guiding example, and also briefly discuss how they can be generalized to a larger algebraic context and why they are useful. These are presented in a syntax that closely resembles that of Agda (and is formally Tog syntax), which should be understandable to anyone familiar with dependently-typed languages.

Trivial Submonoid. Given a monoid `M`, we can construct the trivial monoid, also called the zero monoid[3] (containing only the identity element) in the same language as `M`.

[3] As it is both initial and terminal in the corresponding Category.

```
record TrivialSubmonoid {A : Set} (M : Monoid A) : Set
where
  constructor trivialSubmonoid
  field
    singleton : {x : A} → x == M.e
```

One can easily proceed to show that this predicate on a monoid induces a new (sub)monoid. In fact, we do not need associativity for this; in other words, already a unital magma induces a trivial monoid.

Flipped Monoid. Given a monoid `M`, we can construct a new monoid where the binary operation is that of `M` but applied in reverse order.

The construction here is direct, in that the result is a `Monoid`.

```
record FlippedMonoid : {A : Set} → Monoid A → Monoid A
record FlippedMonoid m = {
  A = M.A,
  e = M.e,
  op = (x y : A) → M.op y x,
  lunit = M.runit,
  runit = M.lunit,
  assoc = sym M.assoc
}
```

This example can be generalized from a monoid to a magma.

Monoid Action. This example constructs, from a Monoid `M` and a set `B`, a monoid action of `M` on `B`.

```
record MonoidAction {A : Set} (M : Monoid A)
                    (B : Set) : Set where
  constructor monoidAction
  field
    act     : A → B → B
    actunit : {b : B} → (act M.e b) == b
    actop   : {x y : A} → {b : B} →
                (act (M.op x y) b) == (act x (act y b))
```

Monoid actions are extremely useful for expressing ideas in group theory, and in automata theory. They are only defined in the presence of a monoid structure, which can be easily checked at the meta level.

Subsets Action. The fourth example construct, from a Monoid `M`, the monoid on the subsets of `M`. Note that the following is pseudo-code written in an imagined Set-theoretic extension of dependent type theory.

```
record SubsetsAction {A : Set} (M : Monoid A) : Set
where
  constructor subsetsAction
```

```
  field
    S       : (powerset A)
    e'      : S
    op'     : S → S → S
    e'def   : e' == {M.e}
    op'def  : {x y : S} → (op' x y)
                == {(M.op a b) | a ∈ x and b ∈ y}
```

The subsets monoid is used extensively in automata theory and group theory.

The above can also be written as a construction of a new monoid, in dependent type theory, where the carrier is the set of unary relations on A.

Monoid Cosets. The next example constructs, from a Monoid M, the cosets of M. This is also pseudo-code, as above.

```
record MonoidCosets {A : Set} (M : Monoid A) : Set
where
  constructor monoidCosets
  field
    S       : (powerset A)
    e'      : S
    op'     : A → S → S
    e'def   : e' == {M.e}
    op'def  : {a : A} → {x : S} → (op' a x)
                == {(M.op a b) | b ∈ x}
```

Monoid cosets are extensively used in group theory.

3 Constructions for Free!

A meta-theory (either a logic or a type theory) provides us with a concrete language in which to represent axiomatic theories. Through having a uniform syntactic representation of the components of axiomatic theories, we can manipulate them, and eventually generate new ones from them.

Our meta-theory is Martin-Löf Type Theory, as implemented in Tog [27]. Tog is developed by the implementors of Agda for the purpose of experimenting with new ideas in (implementations of) dependent type theories. It has mainly been used to experiment with type checking through unification [26]. Tog is minimalistic, and serves our purpose of being independent of the design details of many of the large proof languages. It also gives us a type checker.

The following implementation details of Tog are worth pointing out:

- It has one universe `Set`, which is the kind of all sorts.
- Functions are represented as curried lambda expressions: `Fun Expr Expr`.
- Axioms are represented as Π-types: `Pi Telescope Expr`. They use the underlying propositional equality: `Eq Expr Expr`.
- Theories are represented as parameterized dependent records, Σ-types.

- A parameter to the record has the type `Binding`. It can be hidden using `HBind [Arg] Expr`, or explicit using `Bind [Arg] Expr`.
- A declaration within the record has the type `Constr Name Expr`.

In Universal Algebra, an algebraic theory consists of sorts, function symbols (with their arities) and a list of axioms, often denoted as a theory `T` having three components `(S,F,E)`. We assume a single sort. This can be internalized, in the Haskell implementation of Tog, as

```
data EqTheory = EqTheory  {
  thryName    :: Name_    ,
  sort        :: Constr   ,
  funcTypes   :: [Constr],
  axioms      :: [Constr],
  waist       :: Int      }
```

where:

– `sort`, `funcTypes`, and `axioms` are treated as elements of a telescope [13]. Therefore, the order in which they are defined matters.
– The `waist` is a number referring to how many of the declarations within the telescope are parameters. The notation is taken from [1]. This information is needed in generating some constructions, like homomorphism.

Given a Tog record type that exhibits an equational theory structure, like that of `Monoid` in Sect. 1, we convert it into an instance of `EqTheory`. We, then, proceed with generating useful information from the theory. Finally, we convert this information into Tog records and data types, so they can be type checked by Tog, i.e. our approach builds on Tog, without changing its syntax or type checker. In the sequel of this section, we describe the constructions we generate.

3.1 Signature

Given a theory `T = (S,F,E)`, the signature of the theory is `Sig(T) = (S,F)`. A signature is obtained from an `EqTheory` as follows:

```
signature_ :: Eq.EqTheory -> Eq.EqTheory
signature_ =
  over Eq.thyName (++ "Sig") . set Eq.axioms [] .  gmap ren
```

For a theory with name `X`, the signature is an `EqTheory` with the name `XSig` and an empty axioms list. The theory and its signature exists in the same module. Tog requires that they have different field names. We use `gmap ren` to apply this renaming. We discuss this in more details in Sect. 3.5.

3.2 Product Algebra

Given a theory `T = (S,F,E)`, we obtain the product theory `Prod(T) = (S×S, F′, E′)` by replacing each occurrence of the type `S` by `S×S`. The modification to the function symbols and axioms is straightforward.

```
productThry :: Eq.EqTheory -> Eq.EqTheory
productThry t =
  over Eq.thyName (++ "Prod") $
  over Eq.funcTypes (map mkProd) $
  over Eq.axioms (map mkProd) $
  gmap ren t
```

Similar to what we did with signatures, the `ren` function renames the fields of the input theory. `mkProd` changes the sort to be an instance of `Prod`, with the sort of the input theory as the type parameter.

3.3 Term Language

For a theory `T = (S,F,E)`, the closed term language is generated by converting every function symbol to a constructor, with the same arity. The axioms are dropped.

```
termLang t =
 let constructors =
  gmap (ren (getConstrName $ t^.Eq.sort) nm) $ t^.Eq.funcTypes
 in Data (mkName $ t^.thyName ++ "Lang") NoParams $
  DataDeclDef setType constructors
```

Constructors are generated by substituting the name of the language type for a sort `A`. Term languages are realized as Tog data declarations using the constructor `Data`.

Generating the closed term language is a first step to generating an open term language (i.e. a term language parametrized by a type of variables), and an interpreter.

For some kinds of axioms, namely those that can be *oriented*, we can turn these into *simplification rules*, i.e. into (unconditional) rewrite rules. The resulting simplifier can be shown to be meaning preserving. These two pieces, the evaluator and simplifier, can be attached to each other to form a *partial evaluator*, using the "finally tagless" [7] method. Eventually, we would like to be able to automate the majority of the hand-written code for a generative geometry library [4], which is indeed quite amenable to such techniques. Unfortunately, the details will have to wait for a future paper.

3.4 Homomorphism

For a theory `T = (S,F,E)`, with instances $\mathtt{T}_1$ and $\mathtt{T}_2$, the homomorphism of `T` consists of

1. a function mapping the carrier of $\mathtt{T}_1$ to that of $\mathtt{T}_2$,
2. a set of axioms asserting that operations (i.e. elements of `F`) are preserved.

Our definition of homomorphism is parameterized by the instances $\mathtt{T}_1$ and $\mathtt{T}_2$. The parameters of `T`, if `waist` > 0, are lifted out as parameters to the resulting homomorphism, and used to define the instances of the theory.

```
homomorphism :: Eq.EqTheory -> Decl
homomorphism t =
 let nm = t ^. Eq.thyName ++ "Hom"
     a = Eq.args t
     (psort,pfuncs,_) = mkPConstrs t
     ((i1, n1), (i2, n2)) = createThryInsts t
     homFnc = genHomFunc psort n1 n2
     axioms = map (oneAxiom fnc psort n1 n2) pfuncs
 in Record (mkName nm)
    (mkParams $ (map (recordParams Bind) a) ++ [i1,i2])
    (RecordDeclDef setType
                (mkName  $ nm ++ "C")
                (mkField $ fnc : axioms))
```

The `genHomFunc` function generates the homomorphism function. Each preservation axiom is created using the `oneAxiom` function.

Other kinds of morphisms can also be generated by providing more axioms to describe properties of the functions. For example a monomorphism would have the same definition with one more axiom stating that the function is injective. An endomorphism is a self-homomorphism, and thus can be parametrized by a single theory.

3.5 Discussion

The above are a small sample of what can be done. We've found at least 30 constructions that should be amenable to such a treatment and are currently implementing them, including quotient algebras and induction axioms. Figure 2 shows the generated constructions. The input is the theory of `Monoid` represented as a Tog record type (illustrated on the left with the blue background). For this, we generate the four constructions discussed above (illustrated with pink background). The names of carriers $\texttt{A}_1$ and $\texttt{A}_2$, names of instances $\texttt{Mo}_1$ and $\texttt{Mo}_2$ are machine generated based on the names used by the input theory, which are given by the user. A somehow unpleasant restriction is that all field names need to be distinct, even if the fields belong to different records. That is the reason we have names like `eL` in `MonoidLang` and `eS` in `MonoidSig`. This is still a minor inconvenience, given that we are working on an abstract level, from which more readable and usable code will be generated.

4 Related Work

Many algebraic hierarchies have been developed before. [18] documents the development of the algebra needed for proving the fundamental theorem of algebra. [17] formalizes the same knowledge in Coq, but suggests a packaging structure alternative to telescopes, to support multiple inheritance. [11] addresses the important problem of library maintainability, especially when dealing with changes to the hierarchy. We have proposed an alternate solution in [9], based on the categorical structures already present in dependent type theories.

```
record Monoid (A : Set) : Set
 where
  constructor monoid
  field
   e  : A
   op : A → A → A
   lunit : {x : A} → (op e x) == x
   runit : {x : A} → (op x e) == x
   assoc : {x y z : A} →
     op x (op y z) == op (op x y) z
```

```
record MonoidHom
    (A1 : Set) (A2 : Set)
    (Mo1 : Monoid A1)
    (Mo2 : Monoid A2) : Set where
  constructor MonoidHomC
  field
    hom : A1 → A2
    pres-e : hom (e Mo1) == e Mo2
    pres-op :
      (x1 : A1) (x2 : A1) →
      hom (op Mo1 x1 x2)
      == op Mo2 (hom x1) (hom x2)
```

```
data MonoidLang : Set where
 eL : MonoidLang
 opL : MonoidLang → MonoidLang
       → MonoidLang


record MonoidSig (AS : Set) : Set
 where
  constructor MonoidSigSigC
  field
   eS : AS
   opS : AS → AS → AS


record MonoidProd (AP : Set)
   : Set
 where
  constructor MonoidProdC
  field
   eP : Prod AP AP
   opP : Prod AP AP → Prod AP AP
       → Prod AP AP
   lunit_eP : (xP : Prod AP AP)
       → opP eP xP == xP
   runit_eP : (xP : Prod AP AP)
       → opP xP eP == xP
   associative_opP :
       (xP : Prod AP AP)
       (yP : Prod AP AP)
       (zP : Prod AP AP)
       → opP (opP xP yP) zP
         == opP xP (opP yP zP)
```

Fig. 2. The generated constructions from `Monoid` theory (Color figure online)

The algebraic library of Lean [12] is of particular interest, as its developers are quite concerned with automation. But this automation, also done via metaprogramming, is largely oriented to proof automation via tactics. We instead focus on automating the generation of structures.

Universal Algebra constructions are grounded in set theory, yet is nevertheless quite constructive. It has been formalized in Coq [3,34] and Agda [20]. [34] is notable for the use of type classes to formalize the algebraic hierarchy.

While the work in interactive provers has been mainly manual, the programming languages community has been actively investigating the generation of various utilities derived from the definition of algebraic data types. Haskell's *deriving* mechanism has already been mentioned. This has been greatly extended twice, first in [25], to allow more generic deriving, and then in [2] allowing the users to define new patterns. The usefulness of these mechanisms has been of great inspiration to us. We would like to provide similar tools for library developers of interactive proof systems.

5 Conclusion and Future Work

Building large libraries of mathematical knowledge can greatly enhance the usefulness of interactive proof systems. Currently, the larger the library, the more labor intensive it becomes. We suggest automating some of the definitions of concepts derivable via known techniques. We have tested our implementation on a library of 227 theories, including `Ring` and `BoundedDistributedLattice`, built using the tiny theories approach [5] and the combinators of [9]. A theory defined declaratively using the combinators elaborate into a Tog record, which is then manipulated to generate the constructions presented in Sect. 3. From the declarative description of the 227 theories, we were able to generate a much larger library which contains 1132 definitions and, when pretty-printed, spanned 14811 lines, containing theories and data types representing the structures we discussed in Sect. 3. We are adding more derived theories, and can then get a multiplicative factor, as each time we do, we get 227 new theories.

While the knowledge representable in single-sorted equational logic is still impressive (e.g. it covers most of Algebra), we are also interested in generating the same structures (and more) for theories represented in more sophisticated logics [28], such as category theory represented in dependent type theory.

We currently generate all constructions for all theories in a given library. As more structures get generated, we would want to give developers more control over what to generate. Thus we intend to provide a scripting language for referring to theories, or groups of theories, and specifying what constructions to apply. This could also include an "on demand" version, similar to how the deriving mechanism of Haskell works. We are also interested in generating morphisms, as explained in [16], between theories. Even for our constructions, some of these morphism are not obvious, but are needed to transport results.

We envision using our current implementation as a meta-language to generate definitions for existing, full-featured systems, such as Isabelle/HOL and Agda. To achieve this, we will need to reintroduce certain details (such as notations) that we elided. The scripting language described above will need to be extended to cover different kinds of *design decisions*.

We envision a framework in which the contents of the library can be defined succinctly, and elaborated to a large reusable and flexible body of standardized mathematics knowledge.

References

1. Al-hassy, M., Carette, J., Kahl, W.: A language feature to unbundle data at will (short paper). In: Proceedings of the 18th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences, GPCE 2019, pp. 14–19. ACM, New York (2019)
2. Blöndal, B., Löh, A., Scott, R.: Deriving via: or, how to turn hand-written instances into an anti-pattern. In: Proceedings of the 11th ACM SIGPLAN International Symposium on Haskell, Haskell 2018, pp. 55–67. Association for Computing Machinery, New York (2018)

3. Capretta, V.: Universal algebra in type theory. In: Bertot, Y., Dowek, G., Théry, L., Hirschowitz, A., Paulin, C. (eds.) TPHOLs 1999. LNCS, vol. 1690, pp. 131–148. Springer, Heidelberg (1999). https://doi.org/10.1007/3-540-48256-3_10
4. Carette, J., Elsheikh, M., Smith, S.: A generative geometric kernel. In: Proceedings of the 20th ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation, pp. 53–62. ACM (2011)
5. Carette, J., et al.: The MathScheme library: Some preliminary experiments. arXiv preprint arXiv:1106.1862, June 2011
6. Carette, J., Farmer, W.M., Kohlhase, M., Rabe, F.: Big math and the one-brain barrier a position paper and architecture proposal. arXiv preprint arXiv:1904.10405 (2019)
7. Carette, J., Kiselyov, O., Shan, C.: Finally tagless, partially evaluated: tagless staged interpreters for simpler typed languages. J. Funct. Program. **19**(5), 509–543 (2009)
8. Carette, J., O'Connor, R.: Theory presentation combinators. In: Jeuring, J., Campbell, J.A., Carette, J., Dos Reis, G., Sojka, P., Wenzel, M., Sorge, V. (eds.) CICM 2012. LNCS (LNAI), vol. 7362, pp. 202–215. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-31374-5_14
9. Carette, J., O'Connor, R., Sharoda, Y.: Building on the diamonds between theories: theory presentation combinators. arXiv preprint arXiv:1812.08079 (2018)
10. Clavel, M., Eker, S., Lincoln, P., Meseguer, J.: Principles of Maude. In: Meseguer, J. (ed.) Proceedings of the First International Workshop on Rewriting Logic, vol. 4, pp. 65–89 (1996)
11. Cohen, C., Sakaguchi, K., Tassi, E.: Hierarchy builder: algebraic hierarchies made easy in Coq with Elpi. https://hal.inria.fr/hal-02478907 (2020). working paper or preprint
12. The Mathlib Community. The lean mathematical library. arXiv preprint arXiv: 1910.09336 (2019).
13. de Bruijn, N.G.: Telescopic mappings in typed lambda calculus. Inf. Comput. **91**(2), 189–204 (1991)
14. Denecke, K., Wismath, S.L.: Universal Algebra and Applications in Theoretical Computer Science. Taylor & Francis, New York (2002)
15. Ehrig, H., Mahr, B.: Fundamentals of Algebraic Specification 1: Equations and Initial Semantics. Monographs in Theoretical Computer Science. An EATCS Series. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-69962-7
16. Farmer, W.M., Guttman, J.D., Javier Thayer, F.: Little theories. In: Kapur, D. (ed.) CADE 1992. LNCS, vol. 607, pp. 567–581. Springer, Heidelberg (1992). https://doi.org/10.1007/3-540-55602-8_192
17. Garillot, F., Gonthier, G., Mahboubi, A., Rideau, L.: Packaging mathematical structures. In: Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (eds.) TPHOLs 2009. LNCS, vol. 5674, pp. 327–342. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-03359-9_23
18. Geuvers, H., Pollack, R., Wiedijk, F., Zwanenburg, J.: A constructive algebraic hierarchy in Coq. J. Symb. Comput. **34**(4), 271–286 (2002)
19. Gross, J., Chlipala, A., Spivak, D.I.: Experience implementing a performant category-theory library in Coq. In: Klein, G., Gamboa, R. (eds.) ITP 2014. LNCS, vol. 8558, pp. 275–291. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08970-6_18
20. Gunther, E., Gadea, A., Pagano, M.: Formalization of universal algebra in Agda. Electron. Not. Theor. Comput. Sci. **338**, 147–166 (2018). The 12th Workshop on Logical and Semantic Frameworks, with Applications (LSFA 2017)

21. Halleck, J.: Logic system interrelationships. http://www.horizons-2000.org/2.%20Ideas%20and%20Meaning/John%20Halleck%27s%20Logic%20System%20Interrelationships.html. Accessed 20 Mar 2020
22. Haskell; an advanced, purely functional programming language. https://www.haskell.org/. Accessed 22 Mar 2020
23. Jipsen, P.: List of mathematical structures. http://math.chapman.edu/~jipsen/structures/doku.php. Accessed 20 Mar 2020
24. Haskell lens library. https://hackage.haskell.org/package/lens. version 4.19.1. Accessed 22 Mar 2020
25. Magalhães, J.P., Dijkstra, A., Jeuring, J., Löh, A.: A generic deriving mechanism for Haskell. ACM SIGPLAN Not. **45**(11), 37–48 (2010)
26. Mazzoli, F., Abel, A.: Type checking through unification. arXiv preprint arXiv:1609.09709 (2016)
27. Mazzoli, F., Danielsson, N.A., Norell, U., Vezzosi, A., Abel, A.: Tog, a prototypical implementation of dependent types. https://github.com/bitonic/tog
28. Meinke, K.: Universal algebra in higher types. Theor. Comput. Sci. **100**(2), 385–417 (1992)
29. Meinke, K., Tucker, J.V.: Universal algebra. University of Wales (Swansea). Mathematics and Computer Science Division (1991)
30. Pottier, L.: Coq user contributions - algebra library. https://github.com/coq-contribs/algebra
31. Sankappanavar, H.P., Burris, S.: A Course in Universal Algebra. Graduate Texts Math, vol. 78. Springer, New York (1981)
32. Sannella, D., Tarlecki, A.: Universal algebra. In: Foundations of Algebraic Specification and Formal Software Development. Monographs in Theoretical Computer Science. An EATCS Series, pp. 15–39. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-17336-3_1
33. Sheard, T., Jones, S.P.: Template meta-programming for Haskell. In: Proceedings of the 2002 ACM SIGPLAN Workshop on Haskell, Haskell 2002, pp. 1–16. Association for Computing Machinery, New York (2002)
34. Spitters, B., van der Weegen, E.: Developing the algebraic hierarchy with type classes in Coq. In: Kaufmann, M., Paulson, L.C. (eds.) ITP 2010. LNCS, vol. 6172, pp. 490–493. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-14052-5_35
35. Whitehead, A.N.: A Treatise on Universal Algebra: With Applications. Cornell University Library Historical Math Monographs. The University Press (1898)