Research Proposal

Leveraging The Information Contained in Theory Presentations

Yasmine Sharoda

Supervisors: Jacques Carette
William Farmer

September 20, 2019

1 Introduction

Mathematical knowledge is highly structured and inter-related. Libraries of formalized mathematics aim to capture this knowledge in a way that preserves its structure. The work in [Farmer, 2004] considers building a universal digital mathematical library (UDML) a grand challenge for mathematical knowledge management (MKM). In [Piroi et al., 2007], the problem of organizing big collection of mathematical knowledge formalized within a logic is listed as one of the main problems facing MKM. [Kohlhase et al., 2010] suggests that theory graph structure is well-suited for "MKM in the large".

Libraries are constructed on top of some formal language that provides syntactic structure to the defined theories. A theory written in a formal languages is called a theory presentation. An axiomatic theory presentation consists of definitions of types, typed symbols, and axioms. Our research is based on two main observations related to theory presentations

Libraries of theories contain boilerplate The syntactic structure of a theory presentation gives rise to some related structures that can be mechanically generated. Consider, for example, the definitions of theories of monoids and groups, written in MathScheme language, Figure 1. The

```
Monoid := Theory {
    U : type;
    * : (U, U) -> U;
    e : U;
    axiom rightIdentity_*_e :
       forall x : U. (x * e) = (x);
    axiom leftIdentity_*_e :
       forall x : U. (e * x) = (x);
    axiom associative_* :
       forall x, y, z : U.
       ((x * y) * z) = (x * (y * z))
 }
```

```
Group := Theory {
    U : type;
    * : (U, U) -> U;
    e : U;
    inv : U -> U;
    axiom rightInverse_inv_*_e :
        forall x : U. ((inv x) * x) = (e);
    axiom rightIdentity_*_e :
        forall x : U. (x * e) = (x);
    axiom leftInverse_inv_*_e :
        forall x : U. (x * (inv x)) = (e);
    axiom leftIdentity_*_e :
        forall x : U. (e * x) = (x);
}
```

Figure 1: Definitions of monoid and group theories in MathScheme

signatures of those theories, defined in Figure 2, can be obtained by just dropping the axioms from the definitions of the theories.

The term language of both can be generated from the signatures by using the typed symbols as the constructors of the language. The term languages for the two theories are shown in Figure

```
Monoid := Theory {
    U : type;
    * : (U, U) -> U;
    e : U;
}
```

```
Group := Theory {
    U : type;
    * : (U, U) -> U;
    e : U;
    inv : U -> U;
}
```

Figure 2: Definitions of signatures of monoid and group theories

```
type monoidLang =
    | Mult of (monoidLang * monoidLang)
    | E
```

```
type groupLang =
    | Mult of (groupLang * groupLang)
    | E
    | Inv of groupLang
```

Figure 3: Definitions of term languages of monoid and group theories

3. By adding a module to represent variables, we can generate the open term language over an algebra, as in Figure 4.

```
module type Var = sig
  type t
end
```

```
type monoidLang =
    | Mult of (monoidLang * monoidLang)
    | E
    | Var of V.t
```

```
type groupLang =
    | Mult of (groupLang * groupLang)
    | E
    | Inv of groupLang
    | Var of V.t
```

Figure 4: Definitions of term languages with variables of monoid and group theories

This leads us to think about interpretation functions to give meaning to terms of the language, as in Figure 5. The definition of the function can also be mechanically constructed during the process of defining the language by mapping constructors of the language to objects of the theory presentation.

Also partial evaluators for the theories can be defined by adding a staged type and changing the types in the definitions a bit. Partial evaluators perform some direct computations that can be done before run-time based on knowledge of the program's input. The staged type is used to identify what parts can be evaluated before computations and what parts have to wait till runtime. The signature of the partial evaluator is shown in Figure 6. This can be used to write staged expressions to be used by the partial evaluator.

```
interp (x : monoidLang) : U =
    match x with
        | E -> e
        | Mult (a,b) -> interp a * interp b
```

```
interp (x : groupLang) : U =
     match x with
        | E -> e
        | Mult (a,b) ->
            interp a * interp b
        | Inv a -> inv (interp a)
```

If the language has variables, the interpretation function needs an assignment function for the variables.

```
interp (x : monoidLang)
       (env : Var.t -> U) : U =
    match x with
    | E -> e
    | Mult (a,b) ->
        interp a * interp b
    | Var v -> env v
```

```
interp (x : groupLang)
       (env : Var.t -> U) : U =
    match x with
      | E -> e
      | Mult (a,b) ->
          interp a * interp b
      | Inv a -> inv (interp a)
      | Var v -> env v
```

Figure 5: Definitions of Interpretation function for term languages of monoid and group theories.

```
type 'a Staged = Now of 'a | Later of 'a code
```

```
Monoid := Theory {
    U : type;
    Ust : Staged U;
    * : (Ust, Ust) -> Ust;
    e : Ust;
}
```

```
Group := Theory {
    U : type;
    Ust : Staged U;
    * : (Ust, Ust) -> Ust;
    e : Ust;
    inv : Ust -> Ust;
}
```

Figure 6: Definitions of the signatures of partial evaluators for monoid and group theories

```
locale submonoid =
  fixes H and G (structure)
  assumes subset: "H ⊆ carrier G"
    and m_closed [intro, simp]: "⟦x ∈ H; y ∈ H⟧ ⟹ x ⊗ y ∈ H"
    and one_closed [simp]: "1 ∈ H"

locale subgroup =
  fixes H and G (structure)
  assumes subset: "H ⊆ carrier G"
    and m_closed [intro, simp]: "⟦x ∈ H; y ∈ H⟧ ⟹ x ⊗ y ∈ H"
    and one_closed [simp]: "1 ∈ H"
    and m_inv_closed [intro,simp]: "x ∈ H ⟹ inv x ∈ H"
```

Figure 7: The definition of submonoid and subgroup as defined in Isabelle library. Note how the definitions are repeated, although subgroup can be defined as an extension of submonoid.

The theories of `monoid` and `group` are different, but the algorithms used to generate the signatures, languages, partial evaluators and other important related constructions are the same. This implies that by having the theory as input to the algorithm, these definitions can be mechanically generated. Having to write down every definition of these by hand is a burden that can be lifted so library builders can focus on more interesting parts of their task. We believe that a big list of these operations exist. Universal algebra is all about studying the common structures of algebraic theories and we have made an initial list that can be found in Section 3. Since there is a big number of operations (our initial list contains 25), a big number of theories within a library (MathScheme library has over a thousand theory) and a big number of formal languages in which a theory can be expressed (We are considering 8 of them), there is indeed a lot of boilerplate associated with library building. Besides being boring, boilerplate code is usually error-prone and not reusable.

We given an example of the existence of boilerplate in libraries of formal systems by looking into the library of Isabelle[1]. Considering the formalization of `Monoid`s and `Group`s in `HOL/Algebra/Group.thy`, we find redundancy in the definitions of `submonoid` and `subgroup` as in Figure 7.

The same can also be seen in definitions of products of both theories, shown in Figure 8. Homomorphisms[2] and quotient algebras[3] are also examples of constructions that are defined in multiple places, we show these definitions in Figure 9. The formalization of these definitions in

[1]source: https://isabelle.in.tum.de/dist/library/HOL/HOL-Algebra/index.html

[2]defined in `HOL/Algebra/Group.thy` and `HOL/Algebra/Ring.thy`

[3]defined in `HOL/Algebra/Coset.thy` and `HOL/Algebra/QuotRing.thy`

```
DirProd :: "_ ⇒ _ ⇒ ('a × 'b) monoid" (infixr "××" 80)
  where
        "G ×× H =
      ⦇ carrier = carrier G × carrier H,
        mult = (λ(g, h) (g', h'). (g ⊗G g', h ⊗H h')),
        one = (1G, 1H ⦈"

lemma DirProd_monoid:
  assumes "monoid G" and "monoid H"
  shows "monoid (G ×× H)"

lemma DirProd_group:
  assumes "group G" and "group H"
  shows "group (G ×× H)"
```

Figure 8: The definition of cartesian product of theories in Isabelle library

the library signifies their usefulness. Hand-writing them restricts their availability to only a small number of theories. In case users need them for a new theory they defined, they have to hand write them.

Theory presentations are pervasive Different formal systems have different ways for expressing mathematical knowledge. Whether they are called theories, locales, classes, modules or records, they all contain the same pieces of information. Figure 10 shows the representation of `Monoid` theory in 5 different formal systems. The representation in MathScheme (MSL) states the components of `Monoid` as declarations in a theory. MMT uses the same approach, but instead of stating the six pieces of `Monoid`, it declares `Monoid` as extension of `Semigroup`, So the components of `Monoid` is distributed between the two theories. Despite the usefulness of knowing that `Monoid` is an extension of `Semigroup`, in some cases the user would really want to see the declarations within the theory. MathScheme provides this view of theories, while keeping track of the connection between them using the arrows of the graph. In Coq, we can see two different representations, both are useful in different ways. The `Class` representation expects the function symbols as input to the definition with the axioms being the members of the class. The carrier `A` is deduced and need not be passed by the user. This allows one to easily define `Monoid`s over the same carrier, operation or unit element, which can be beneficial for proving purposes. The `Record` representation does not have the same property. Every declaration of monoid is defined as a field in the record. In the case of defining two records over the same carrier, the user needs to actually prove they are the same. Haskell, as a programming language, does not define axioms or

```
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)}
definition
   ring_hom :: "[('a,'m) ring_scheme,('b,'n) ring_scheme] ⇒
                                                ('a ⇒ 'b) set"
   where
   "ring_hom R S =
      {h . h ∈ carrier R → carrier S ∧
        (∀ x y . x ∈ carrier R ∧ y ∈ carrier R →.
            h (x ⊗_R y) = h x ⊗_S h y ∧
            h (x ⊕_R y) = h x ⊕_S h y) ∧
             h 1_R = 1_S}"
definition
    FactGroup :: "[('a,'b) monoid_scheme, 'a set] ⇒ ('a set) monoid"
    (infixl "Mod" 65)
    where FactGroup G H =
    (| carrier = rcosets_G H, mult = set_mult G, one = H |)
definition
    FactRing :: "[('a,'b) ring_scheme, 'a set] ⇒ ('a set) ring"
    (infixl "Quot" 65)
    where FactRing R I =
    (| carrier = a_rcosets_R I, mult = rcoset_mult R I,
    one = (I +>_R 1_R), zero = I, add = set_add R |)
```

Figure 9: The definition of homomorphisms and quotient algebras in Isabelle library

MSL

```
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));
}
```

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_neutral : forall x, x * 1 = x
}.
```

Haksell

```
class Semigroup a => Monoid a where
    mempty :: a
    mappend :: a -> a -> a
    mappend = (<>)
    mconcat :: [a] -> a
    mconcat =
    foldr mappend mempty
```

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
      #assocLR %I1 %I2 %I3
  assocRightToLeft :
      {x,y,z} ⊢ x * (y * z)
                  = (x * y) * z
      = [x,y,z] sym assocLR
      # assocRL %I1 %I2 %I3
theory Monoid : ?NatDed
    includes ?Semigroup
    unit : tm u # e
    @_description the unit
          element of the monoid
    unit_axiom : ⊢ ∀ [x] = x * e = x
    @_description the axiom
          of the neutral element
```

Agda

```
data Monoid (A : Set)
            (Eq : Equivalence A) : Set
where
    monoid :
      (z : A)
      (_+_ : A -> A -> A)
      (left_Id : LeftIdentity Eq z _+_)
      (right_Id : RightIdentity Eq z _+_)
      (assoc : Associative Eq _+_) ->
      Monoid A Eq
```

Alternative Definition:

```
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 IsMonoid (• : Op₂) (ε : A)
             : Set (a ⊔ ℓ) where
    field
        isSemigroup : IsSemigroup •
        identity : Identity ε •

        identityˡ : LeftIdentity ε •
        identityˡ = proj₁ identity

        identityʳ : RightIdentity ε •
        identityʳ = proj₂ identity
```

Figure 10: The representation of Monoid theory in different formal systems.

allow proving that instances of the class satisfy them. The Agda definition have some decorative pieces like `infixl` which specifies that an operator is infix and associates left. A monoid can still be defined without those decorative pieces, or they can be specified by the user in a declarative manner. In all these representations the pieces of information that we need to write the monoid definition is the same. Therefore, having one of them should suffice to generate the others.

We understand that the different representations stem from different design choices of the different systems, but this does not deny the fact that they are all representations of the same concept of a `Monoid` theory. The possibilities and limitations of moving from one representation to another are interesting research points.

Our aim in this study is to eliminate the boilerplate associated with defining and using theory presentations by automating the generation of related constructions. We imagine the user writ-

ing an algebraic theory and getting a set of tools customized to her theory that she can directly use towards getting her task done. Our implementation is meant to be generic in terms of the representation language. The toolbox should be able to generate code for different formal systems. We start by using MathScheme and MMT as data description languages (DDL) of theory presentations, and then generalize to other representations.

Despite the fact that the algorithms to generate the algebraic structures are the same, non-trivial work needs to be done to have generic, re-usable components that generates these structures in different representation languages. For example, MSL uses the keyword `axiom` to indicate that an axiom is being declared. On the other hand, by using Curry-Howard correspondence, axioms are declared in Coq as fields of a class or record definition, in the same way that a function is declared. Therefore, it is not easy to distinguish whether a declaration corresponds to a constant, a function or an axiom. Also some languages provide special literals to define notations or precedence, while other do not. Dealing with these languages as a family and providing re-usable generators is one important aspect of our research work. We describe our contribution as:

- Evaluating the status of current popular libraries of mathematics, in terms of modularity and reusability.
- Compiling a list of constructions that can be automatically generated from the syntax of a theory presentation.
- Building a library of generic algorithms for computing these constructions, given a theory written in DDL, which abstracts over the presentation language. At this point, we only consider theory presentations in their most abstract format as a collection of sorts, function symbols and axioms written in some formal language.
- Implementing generators that can produce these constructions for a specific theory in a specific language, taking into considerations all similarities and differences between different formal systems. We start by the two systems HOL and Agda, but aim to expand the implementation to different systems as time permits.

The expected output of our work is a domain-specific language for developing a library of theories structured as theory graphs that minimizes boilerplate, maximizes reuse and generate useful information from minimum description.

In Section 2 we describe the class of theory presentations we are interested in. We give a brief introduction to MathScheme and MMT, the two frameworks in which we conduct our experiments. We present out initial list of constructions that can be automatically generated in Section 3. We give a summary of related work in Section 4. Then we describe the approach we are following in Section 5.

2 Background

In order to handle theory presentations and manipulate them, we need to arrive at a concrete definition of what they are. This section sets the ground for our work. We define theory presentations from our perspective in Section 2.1. The library that we will use in this work comes from MathScheme system. It is introduced in Section 2.2. As a preprocessing step to working on the library, we translate it to MMT, another formalization system that we introduce in Section 2.3. We benefit from the translation by expanding the team working on the library and having access to more resources, like web interface, graph viewer and a latex processor.

2.1 Theory Presentations

According to [Farmer, 2007], a general language L can be defined as a pair $(\mathcal{E}, \mathcal{F})$ such that $\mathcal{E}$ is a set of syntactic entities and $\mathcal{F} \subseteq \mathcal{E}$ is a set of formulas in L. A general logic is a set of general languages with some notion of logical consequence. A theory in some language L is the declaration of the syntactic entities and a set of formulas closed under the notion of logical consequence provided by the underlying logic. The focus of our work is to mechanize the generation of algebraic structures from theory presentations. Therefore, we need to focus on a syntactic representation that corresponds to the concept of a theory. A theory presentation for us is a specification from which a theory could be generated. Instead of containing all possible formulas, as in a theory, a theory presentation contain only the kernel from which all formulas could be generated. Since we are interested in typed languages, we consider theory presentations that allow definitions of new types, declarations of typed symbols, definitions of these symbols (in the form of lambda expressions) and axioms describing them. The semantics of a theory presentation is the corresponding abstract theory in logic.

A theory presentation gives the means of writing down a theory in a formal language. It provides means to define the components of the kernel of a theory. Our notion of theory presentations is not only limited to proof assistants. Different systems incorporate some form of theory presentation under different names; for example module signatures (as in Ocaml), data definitions (as in Agda), class definitions (as in Coq, Isabelle and Lean), record definitions (as in Agda and Coq), type classes (as in Haskell), traits (as in Scala) and others holding similar structures of having types, constants, functions, predicates and lambda expressions. In [Müller et al., 2018], a correspondence is established between the following concepts from theorem provers and programming languages: theories, record types, modules, ML signatures, C++ and java classes. We understand that they have different features. Part of this work is to explore their commonalities and differences.

We use MathScheme library to run our experiments. The MathScheme system is introduced in the next section.

2.2 MathScheme

MathScheme [Carette et al., 2011b] is a mechanized mathematics system being developed at McMaster University. It offers a library of over 1000 theory presentations [Carette et al., 2011a,Zhang, 2009] defined in the MathScheme language. The theories are defined by means of theory combinators [Carette and O'Connor, 2012]. Three main combinators are used

- Extends: A theory B is an extension of a theory A if all declarations of A are contained in B, but not all declarations of B are contained in A.
- Renames: A theory B is a rename of theory A if it renames one or more of the declarations in A.
- Combines: A theory C combines two theories A and B by computing the pushout of the two theory presentations in the category of theories and theory morphisms. The resulting theory includes all the declarations of the two theories, gluing together those that can be traced back to the source.

Defining theories using those combinators result in a theory graph in which theories are the nodes and theory morphisms are the edges. MathScheme also employs the little theories approach [Farmer et al., 1992] in which theories are defined incrementally by adding minimal details at each increment. This way allows theorems to be proved using minimal axiomatizations which provides maximum generality. The system implements a flattener that outputs the expanded theory presentation as would be seen in a mathematics textbook. In this work, we consider theories in their flattened format. This gives us access to its components, which we need in order to manipulate the theories. As part of the collaboration between MathScheme and MMT projects, the library is being translated into MMT language to provide more support and expand the development team.

2.3 MMT

MMT [MMT, 2005] is a framework for representing mathematical knowledge. It is based on two design concepts; foundation independence and modularity.

Foundation independence avoids the commitment to any specific logical framework and allows the translation between them. This way, it enables interoperability and reuse. MMT realizes foundation independence by representing core mathematical knowledge as theories (logics-as-theories approach).

Modularity is realized by linking the theories via theory-morphisms [Rabe and Kohlhase, 2008]. MMT defines three types of theory morphisms; inclusions, structures and views [Michael Kohlhase, 2017]. Morphisms created via inclusion make the declarations of one theory included in the other without any change in the source theory. Structures are named inclusions that allow changing

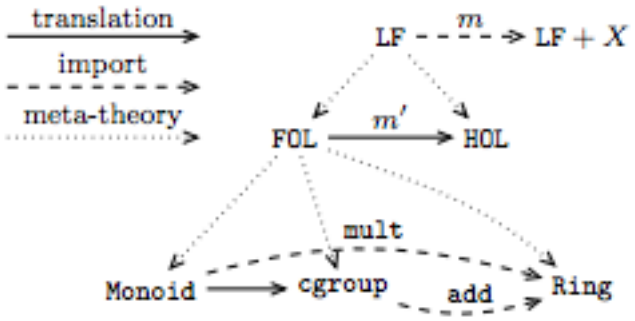

Figure 11: Part of the MMT theory graph [Kohlhase and Rabe, 2016].

notations or definitions of symbols of the source theory. Views map symbols of the source theory to those of the target one. Meta-theory relation is one form of theory morphisms. Meta-theories are useful to achieve both modularity and foundation independence. They act as the language infrastructure for their descendants to be able to present domain-specific knowledge. We write M/T to denote that theory T is defined using the meta-theory M. This implies that the semantics of T can only be given with respect to the semantics of M.

Using logics-as-theories and theory morphism approaches induce a theory graph structure on the MMT library, where nodes are theories and arcs are theory morphism relations. Figure 11 shows part of the theory graph in the MMT library. The graph shows that the LF/FOL/Monoid, LF/FOL/cgroup and LF/FOL/Ring theories all have the same foundation (meta-theory) . They are defined in FOL, but can be translated to HOL via the m' morphism.

MathScheme and MMT share many of the main ideas of representing theories. This is one reason why merging both systems is being done.

3 Operations

We present some of the algebraic constructions that can be extracted automatically from a theory presentation. These are the operations that our library will generate. Our current list is a starting list. We believe there are more constructions can be generated and will be adding them to the list as we proceed with the work.

The algorithms to generate these constructions will take as input a flattened theory presentation consisting of a set of declarations. Based on our discussion in the previous section, we define a theory presentation as a triplet (S, F, P) where S is a set of types, F is a set of function symbols and P is a set of propositions. We now start defining the operations in terms of how they make use of these three components.

- The *signature* of a theory describes the language presented by the theory and consists of sorts and function symbols. A signature is obtained from a theory by dropping the axioms. Therefore, given a theory (S, F, P), the signature of this theory is the tuple (S, F).

- The theory B is a *sub-algebra* of A, if operations of A are restricted to a closed subset B. Given an algebra (S, F, P), the sub-algebra is defined as (S', F', P'), where $\forall s \in S \cdot \exists s' \in S' \cdot s' \subseteq s$ and all functions and axioms in F' and P' are changed accordingly to reflect the new sorts. A sub-algebra is called minimal if it contains no proper sub-algebras.

- The algebra $A \times B = (S, F, P)$ is a *product algebra* of the two algebras $A = (S_A, F_A, P_A)$ and $B = (S_B, F_B, P_B)$ if the sorts $s_i \in S = s_{a_i} \times s_{b_i}$ and all function symbols and axioms are changed accordingly.

- A *projection* from a product algebra to one of its components.

- A *congruence* relation R on an algebra A with respect to a set of operations S is an equivalence relation on A that preserves its structure. A structure-preserving relation $\rightarrow$ has the property that for every operation symbols, $\sigma \in S$, of the algebra, the following axiom holds:
  $x_1 \rightarrow y_1, \cdots, x_n \rightarrow y_n$ implies $\sigma(x_1, \cdots x_n) \rightarrow \sigma(y_1, \cdots y_n)$
  The relation induced by a homomorphism is called an induced congruence.

- The *quotient algebra* of an algebra A given by a congruence relation R (a.k.a the factorization of A by R), is the algebra whose sorts and operations are described with respect to the congruence classes created by the relation R. A congruence class $[a]$ is defined $\{b \in s_i | (a, b) \in R\}$

- The *record* definition of the theory, in which each declaration may depend only on those before it.

- The *homomorphism* of an algebra A can be obtained by defining two instances of A, $x\ y : A$, as well as a function that maps elements of the two instances together $h : x \rightarrow y$. Then adding the preservation axiom for each function symbol. The preservation axiom for a function symbol f_x that maps to f_y states that $h(f_x(a_1, \ldots, a_n))) = f_y(h(a1), \ldots, h(an))$

- The *homomorphism equality* defined as the extensional equality between the two functions.

- The *composition* of homomorphisms

- The *kernel* of a homomorphism is a congruence relation R associated with the homomorphism f from A to B by the rule $R\ x\ y$ iff $h(x) = h(y)$

- The *isomorphism* from algebra A to algebra $B$is a homomorphism $h : A \to B$ such that a homomorphism $g : B \to A$ exists and $f \circ g = id_B$ and $g \circ f = id_A$.
- The *endomorphism* is a homomorphism from algebra A to itself, $h : A \to A$.
- The *automorphism* is an isomorphism from algebra A to itself, $I : A \to A$.
- The *term language* induced by the theory. This language consists of the set of variables, constants and function symbols of the theory.
- The *ground terms* of algebra is the term language when the set of variable is empty.
- The *staged terms* of the algebra which are used for partial evaluation in a multi-staged programming setup
- The *structural induction* axiom on the term language to assert a proposition $p(x)$ is true for all terms of the language. For statement $p(x)$ to be true, we need to prove that it is true for constants and variables, then inductively for the term $N(t1, \ldots, tn)$ by proving that $p(t1)$, $p(t2)$, ... and $p(tn)$ holds. This is the definition of weak structural induction. We want to generate both weak and strong structural induction.
- The *evaluation* of a term is defined in terms of an `eval` : $T \to A$ function, where T is the type of terms. If we consider open terms, which contain variables, we need to have a variable assignment function `assign` : $X \to A$. The function is defined recursively based on the definition of the term language.
- A *simplify* function that applies axioms as term rewrite rules. For example, the unit axiom can result in reducing the term by removing the unit part of it. The function applies simplification rules until no more rules can be applied.
- The *rewrite rules* defined on equations based on axioms. Given a set E of equations with a fixed set of variables $X = X_e$, each equation $e = (L, R) \in E$ defines two substitution rules $L \Rightarrow R$ and $R \Rightarrow L$.
  The application of a substitution rule $t1 \Rightarrow t2$ to a term t is valid if there is an extension `assign` : $T_{OP}(X) \to T_{OP}(X)$ s.t.
  - $\overline{t1} =$ `assign`$(t1)$.
  - $\overline{t2} =$ `assign`$(t2)$.
  - $\overline{t1}$ is a sub-term of t.

  Therefore, the substitution yields a term t' defined as $t' = t(\overline{t1}/\overline{t2})$
  $t \Rightarrow_E^* t'$ is called a derivation from t to t' via E.

- The set of *sub-terms* of a term in the language, where a term $t1$ is a sub-term of $t2$ if $t1 \leq t2$. The $\leq$ relation is defined as:
  - $t \leq t$ for all $t \in T_{OP}(X)$
  - $t \leq N(t1, \ldots, tn)$ if $t = t_i$ for some $i \in 1, \ldots, n$ and $N(t1, \ldots tn) \in T_{OP}(X)$
  - If $t1 \leq t2$ and $t2 \leq t3$ then $t1 \leq t3$.
- The *equivalence* of two terms, where $t1 = t2$ iff $\texttt{eval}_A(t1) = \texttt{eval}_A(t2)$ for all algebras A
- The *parse tree* for a term, where the term is represented as an instance of the parse tree datatype
- The homomorphism between terms and trees.

4 Related Work

Enhancing modularization and removing boilerplate code are the basis to many great ideas in computer science. In [Goguen and Burstall, 1984] institutions are introduced to abstract over syntax and semantics of logics and work over the language structure. Data type generic programming (DGP) [Gibbons, 2007] is another area of computer science that abstracts over the details of data types and focus on representing its structure. There are different approaches to DGP, surveyed in [Hinze et al., 2007]. One approach is to map a data type to a structure representation type, that only represents information about the structure. For example a type $\alpha : + : \beta$ represents the structure of a data type with two data constructors. This approach is followed by generic Haskell [Hinze and Jeuring, 2003] and PolyP [Jansson and Jeuring, 1997]. Another approach uses reflection, as in DrIFT [Winstanley, 1997] and template Haskell [Norell and Jansson, 2004]. DrIFT extracts type declarations and directives to fire rules that generate code at compile time. Template Haskell allows meta programming within Haskell by providing access to the abstract syntax. This way, the structure of data types can be analyzed on the meta-level, and code can be generated according to that structure.

In our work we abstract over the details of theory presentations and offer a library of algorithms to operate on their generic structure to generate related constructions. This abstraction is studied in mathematics under the field of universal algebra [Wechler, 1992]. Different formal systems have different ways of representing algebraic theories. Some systems, like Coq, may have different libraries formalizing the same theories based on different languages constructs, in case of Coq the constructs are modules and type classes. This is one kind of redundancy we aim to remove in this research work. It is worth mentioning that we believe the existence of different representations is needed, what we want to change is that no human need to rewrite the declarations, given that the information in all these representation is the same.

We now look at the literature that discusses formalizing algebraic theories and related constructions. In [Capretta, 1999], some universal algebra constructs are formalized in the Coq proof assistant, which is based on the extended calculus of constructions, but it is mentioned that a weaker type system could have been used. The work in [Spitters and Van der Weegen, 2011], uses Coq type classes to formalize some concepts of universal algebra. A formalization in Agda is presented here [Gunther et al., 2018]. The work in [Benke et al., 2003] employs techniques from generic programming to define universes of codes for parameterized term algebra and inductive families. In [Heras et al., 2015], cartesian product and sub-algebra are formalized in ACL2. We embody a broader scope in our study in the sense that we are not focusing on one system, but looking at many systems and generating knowledge in different languages. We also aim to generate many more structures than already found in literature.

Part of this work is translating the generated constructions in different languages. Both Isabelle/HOL and Coq include code generation facilities, done on two stages. In both cases the input is a proof and the output is a functional language. In case of Isabelle/HOL [Haftmann and Nipkow, 2010], the annotated proof is translated into an intermediate language called miniHaskell; a version of Haskell containing type classes, instances, type and function definitions. If the target language is not Haskell, a second step is needed to remove type classes. The semantics of this translation is given by a higher-order rewrite systems. Extraction in Coq is presented in [Letouzey, 2002]. It starts by eliminating logical parts of the proof; the parts concerned with proving properties of the system. The remaining parts, the informative part of the proof, is then translated to functions in the target language. The user annotates the logical and informative parts by assigning them the types `Prop` and `Set`, respectively. The extracted code is optimized using predefined optimizations to make it more readable and efficient. The idea of generating code from proofs is also presented in [Poernomo et al., 2005]. Our work differs from these systems, as we do not deal with proofs, and our target systems are not only functional languages, but include theorem provers. Also related is the work on CASL presented in [Mossakowski, 2002], which uses institutions to relate CASL to other specification languages. The logics underlying the languages are formalized as institutions, with institution representations describing how they relate together. A translation from one specification system to another happen through the institution representation. The translation process start by first applying semantics to the source specification to obtain its representation in the institution. Then, the institution morphism is applied to obtain the corresponding theory in the target institution, which is then written as a specification of the target language. This related to our work because the different systems we are targeting are based on different logics, but is different because it does not consider the algebraic constructions we are looking to generate.

The idea of calculating new theories from existing ones is implemented in Specware [Smith, 1999] by using colimits of diagrams to define new specifications. It also related to our work as it uses a library of generic refinements to refine a given specification, which is close to what we are doing with algebraic theories by defining a generic library of constructions, then generating them

for specific theories.

Close to the spirit to what we do is the work on Stratego\XT [Bravenboer et al., 2008], which generates parse, format checker and pretty printer, from a declarative description of a language. Also the typedefs project [typ,] that generates different representations of a datatype in different languages, given its representation in Elm language or as an s-expression in Lisp. Our work still has different scope and generates different structures than both projects.

Our work is inspired by two developments in Haskell. The deriving mechanism in Haskell generates functions specific to datatypes when the user adds the `deriving` clause followed by some directives of what functions to generate. For example, consider the following `Expr` type

```
data Expr = I Int
| Var String
| Add Expr Expr
| Mul Expr Expr
| App String Expr
deriving (Show,Eq)
```

Haskell generates the functions `show`, `(==)` and `(!=)` just because the user added the clause `deriving (Show,Eq)` Another related example can be found in the lenses package in Haskell. The lenses package provide getters and setters for the datatype under the names `view` and `over`, respectively. Consider, for example, the following datatypes, taken from [Control.Lense Tutorial,]

```
data Atom = Atom { _element :: String, _point :: Point } deriving (Show)
data Point = Point { _x :: Double, _y :: Double } deriving (Show)
```

by calling the function `makeLenses` as follows

```
makeLenses ''Atom
makeLenses ''Point
```

The following four functions are generated

```
element :: Lens' Atom String
point   :: Lens' Atom Point
x       :: Lens' Point Double
y       :: Lens' Point Double
```

A function is generated for every field defined in the type whose name starts with underscore. The function has the same name without the underscore. Lenses can be composed, so we can write something like

```
shiftAtomX = over (point . x) (+ 1)
```

without having to pattern match over the point type to have access to the field with the name `x`. These two small examples give an intuition of how a lot of code can be generated by adding simple clauses to the code we write. Generating this code removes boilerplate, increases productivity and enhances readability of the code. Our work differs from these example in that we aim to generate code in many output languages.

By examining the literature presented in this section, we conclude that related research presents different pieces of solutions to our problem. In the next section, we present our approach to solving the problem.

5 Approach

This research explores theory presentations and their related constructions in different languages (including Agda, Coq, Haskell, Lean, Isabelle, Scala and Idris). In our work theory presentations are data that can be syntactically manipulated to generate new pieces of knowledge. We first use a data description language (DDL) to generate as much information as possible on an abstract level. For this purpose we use MMT as our abstraction language. We make use of the fact that MMT is platform independent to study the minimal logic in which the different generated constructions can be defined. Afterwards, we start exploring the specific language features of every language of interest, determining the commonalities and variabilities among them, and how different features would affect the generated constructions. We then implement the generators to the different target languages. In the sequel of this section, we discuss the approach in details.

5.1 Building Library of Generic algorithms

As we discussed in Section 1, many algebraic constructions can be automatically generated by syntactic manipulation of theory presentations. In Section 3, we discuss some of these constructions and argue that they can be derived without necessarily knowing the specific details of the theory in hand, by just identifying its basic components (carrier sets, function symbols, ...). To be able to abstract over the language details, we choose to derive our construction in MMT language. MMT is a platform independent system built with the idea of minimizing the commitment to a specific logic or formal system. This makes it a perfect fit for our idea of abstracting over formal systems details.

Manipulating MMT theories requires dealing with OMDoc terms, the object language of MMT. For example, manipulating a theory to generate homomorphism requires:

- Reading the theory as an OMDoc term, an `OMA` term. An `OMA` term represents the application of a term to a given list of terms.

- Creating two instances of the theory, by copying the list of declarations and prefixing their names. A declaration is represented as an `OML` term, which encodes an MMT declaration with its name and optional type, definition and notation.
- Creating the preservation axioms for every function symbol. Function symbols are instances of `Funtype`. That's how they can be distinguished from sorts and axioms. The axioms are generated as `OML` terms.

The result of this step will be a library that contains datatypes describing the constructions to be generated and generic algorithms to map between these datatypes using Scala rules, which are introduced in [Rabe, 2018]

To be able to test our constructions, we started translating MathScheme library into MMT. This provides a large enough library for testing.

5.2 Exploring Representation Languages

One major observation on which we build this work is that different formal languages have different ways of presenting mathematical structures that have standard definitions, and embody the same information. In Section 1 we gave an example of representing monoids in different formal systems. It can also be seen that the same formal system may represent the theory using different language constructs, like in the case of Agda using records and classes. These variabilities are mainly due to the different strength of different logics as well as design decisions of the language builders as of what features to include in the language. Based on these features, some of the generated structures may need to be tuned, or may need further information to proceed. In this case, we need to determine what information is needed from the user.

The output of this step is a detailed comparative study of the features of target formal systems and what constraints this pose on the representation of different constructions. The only tool we are aware of for measuring commonalities and variabilities are feature models, which we plan to use in our research.

5.3 Implementing Generators

We now move to the step of actually generating theories in the target languages. Our system enables the user to have access to algebraic constructions like homomorphism, term languages, simplifiers and others for a specific theory in one of the languages we consider in this work. For example, using our library and generator, the user should be able to do something like

```
BooleanAlgebra := Theory {
    U : type;
    * : (U, U) -> U;
```

```
    + : (U, U) -> U;
    --> : (U, U) -> U;
    0 : U;
    1 : U;
    compl : U -> U;
    axiom unipotent_-->_1 : forall x : U. (x --> x) = (1);
    ....
}
generate (termLang, simplify)
```

Therefore, by adding some annotation of what algebraic constructions need to be generated, the user will have access to these constructions without the need define them. This way, the user can run something like

```
simplify ((x * y) + 1)
```

which results in `(x*y)`. This is possible because the definitions of the term language with variables and the simplification rules generated by the system. This is similar in spirit to using deriving and lenses as we have shown in the examples in Section 4.

5.4 Timeline

- Translating MathScheme Library: May - Sept 2019
- Generating Related Structures in MMT: October - Dec 2019
- Exporting to different languages: Jan - April 2020
- Writing Thesis: April - Sept 2020

6 Conclusion

Defining a theory presentation is the first step towards defining functions that compute related values as well as theorems for proving properties. In the process of defining and using these functions and theorems, users need to define some structures that can be automatically generated. The main aim of this work is to discover these structures and make them available for free for the users of the target systems. This way we provide support for the common tasks associated with using theory presentations. We envision the user defining a theory and getting a set of tools for free, by specifying which tools they need. Having these tools available for a large library of theories contribute to making formalization of software more accessible and enjoyable.

References

[typ,] Typedefs. https://typedefs.com/introduction/. Accessed: May 17, 2019.

[MMT, 2005] (2005). MMT project. http://uniformal.github.io/doc/. Accessed: 2017-09-15.

[Benke et al., 2003] Benke, M., Dybjer, P., and Jansson, P. (2003). Universes for generic programs and proofs in dependent type theory. *Nord. J. Comput.*, 10(4):265–289.

[Bravenboer et al., 2008] Bravenboer, M., Kalleberg, K. T., Vermaas, R., and Visser, E. (2008). Stratego/xt 0.17. a language and toolset for program transformation. *Technical Report Series TUD-SERG-2008-011*.

[Capretta, 1999] Capretta, V. (1999). Universal algebra in type theory. In *International Conference on Theorem Proving in Higher Order Logics*, pages 131–148. Springer.

[Carette et al., 2011a] Carette, J., Farmer, W. M., Jeremic, F., Maccio, V., O'Connor, R., and Tran, Q. M. (2011a). The mathscheme library: Some preliminary experiments. *arXiv preprint arXiv:1106.1862*.

[Carette et al., 2011b] Carette, J., Farmer, W. M., and O'Connor, R. (2011b). Mathscheme: Project description. In *Intelligent Computer Mathematics: 18th Symposium, Calculemus 2011, and 10th International Conference, MKM 2011, Bertinoro, Italy, July 18-23, 2011. Proceedings*.

[Carette and O'Connor, 2012] Carette, J. and O'Connor, R. (2012). Theory presentation combinators. In *International Conference on Intelligent Computer Mathematics*, pages 202–215. Springer.

[Control.Lense Tutorial,] Control.Lense Tutorial. Tutorial on control.lens. http://hackage.haskell.org/package/lens-tutorial-1.0.3/docs/Control-Lens-Tutorial.html. Accessed: 2018-09-18.

[Farmer, 2004] Farmer, W. M. (2004). Mkm: a new interdisciplinary field of research. *ACM SIGSAM Bulletin*, 38(2):47–52.

[Farmer, 2007] Farmer, W. M. (2007). Biform theories in chiron. In *Towards Mechanized Mathematical Assistants*, pages 66–79. Springer.

[Farmer et al., 1992] Farmer, W. M., Guttman, J. D., and Thayer, F. J. (1992). Little theories. In *International Conference on Automated Deduction*, pages 567–581. Springer.

[Gibbons, 2007] Gibbons, J. (2007). Datatype-generic programming. In Backhouse, R., Gibbons, J., Hinze, R., and Jeuring, J., editors, *Datatype-Generic Programming*, pages 1–71, Berlin, Heidelberg. Springer Berlin Heidelberg.

[Goguen and Burstall, 1984] Goguen, J. A. and Burstall, R. M. (1984). Introducing institutions. In Clarke, E. and Kozen, D., editors, *Logics of Programs*, pages 221–256, Berlin, Heidelberg. Springer Berlin Heidelberg.

[Gunther et al., 2018] Gunther, E., Gadea, A., and Pagano, M. (2018). Formalization of universal algebra in agda. *Electronic Notes in Theoretical Computer Science*, 338:147–166.

[Haftmann and Nipkow, 2010] Haftmann, F. and Nipkow, T. (2010). Code generation via higher-order rewrite systems. In *International Symposium on Functional and Logic Programming*, pages 103–117. Springer.

[Heras et al., 2015] Heras, J., Martín-Mateos, F. J., and Pascual, V. (2015). Modelling algebraic structures and morphisms in acl2. *Applicable Algebra in Engineering, Communication and Computing*, pages 277–303.

[Hinze and Jeuring, 2003] Hinze, R. and Jeuring, J. (2003). *Chapter 1. Generic Haskell: Practice and Theory*, pages 1–56. Springer Berlin Heidelberg, Berlin, Heidelberg.

[Hinze et al., 2007] Hinze, R., Jeuring, J., and Löh, A. (2007). Comparing approaches to generic programming in haskell. In *Datatype-Generic Programming*, pages 72–149. Springer.

[Jansson and Jeuring, 1997] Jansson, P. and Jeuring, J. (1997). Polyp—a polytypic programming language extension. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '97, pages 470–482, New York, NY, USA. ACM.

[Kohlhase and Rabe, 2016] Kohlhase, M. and Rabe, F. (2016). Qed reloaded: Towards a pluralistic formal library of mathematical knowledge. *Journal of Formalized Reasoning*, 9(1):201–234.

[Kohlhase et al., 2010] Kohlhase, M., Rabe, F., and Zholudev, V. (2010). Towards mkm in the large: Modular representation and scalable software architecture. In *International Conference on Intelligent Computer Mathematics*, pages 370–384. Springer.

[Letouzey, 2002] Letouzey, P. (2002). A new extraction for coq. In *International Workshop on Types for Proofs and Programs*, pages 200–219. Springer.

[Michael Kohlhase, 2017] Michael Kohlhase, D. M. (2017). Mmt tutorial. https://gl.mathhub.info/Teaching/KRMT/tree/master/source/tutorial. Accessed: 2017-10-02.

[Mossakowski, 2002] Mossakowski, T. (2002). Relating casl with other specification languages: the institution level. *Theoretical Computer Science*, 286(2):367–475.

[Müller et al., 2018] Müller, D., Rabe, F., and Kohlhase, M. (2018). Theories as types. In *Automated Reasoning - 9th International Joint Conference, IJCAR 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings*, pages 575–590.

[Norell and Jansson, 2004] Norell, U. and Jansson, P. (2004). Prototyping generic programming in template haskell. In Kozen, D., editor, *Mathematics of Program Construction*, Berlin, Heidelberg. Springer Berlin Heidelberg.

[Piroi et al., 2007] Piroi, F., Buchberger, B., Rosenkranz, C., and Jebelean, T. (2007). Organisational tools for mkm in theorema. Technical report, Technical report.

[Poernomo et al., 2005] Poernomo, I., Crossley, J. N., and Wirsing, M. (2005). *Adapting Proofs-as-Programs: The Curry–Howard Protocol*. Springer Science & Business Media.

[Rabe, 2018] Rabe, F. (2018). A Modular Type Reconstruction Algorithm. *ACM Transactions on Computational Logic*, 19(4):1–43.

[Rabe and Kohlhase, 2008] Rabe, F. and Kohlhase, M. (2008). An exchange format for modular knowledge. In *LPAR Workshops*.

[Smith, 1999] Smith, D. R. (1999). Mechanizing the development of software. *NATO ASI Series F Computer and Systems Sciences*, 173:251–292.

[Spitters and Van der Weegen, 2011] Spitters, B. and Van der Weegen, E. (2011). Type classes for mathematics in type theory. *Mathematical Structures in Computer Science*, 21(4):795–825.

[Wechler, 1992] Wechler, W. (1992). Universal algebra for computer scientists, volume 25 of eatcs monographs on theoretical computer science.

[Winstanley, 1997] Winstanley, N. (1997). A type-sensitive preprocessor for haskell. In *Glasgow Workshop on Functional Programming, Ullapool*. Citeseer.

[Zhang, 2009] Zhang, H. (2009). A language and a library of algebraic theory-types. Master's thesis, McMaster University.