Generative Tools for Library Building

Yasmine Sharoda, Jacques Carette, William M. Farmer

McMaster University

A large library of Mathematics is useful but hard to build

A large library of Mathematics is **useful**

- QED Manifesto, 1994:
  - One library to formalize all of Mathematics
- Universal Digital Math Library, 2004:
  - Heterogeneous, Interconnecting libraries.

A large library of Mathematics is **hard to build**

- Foundation
- Organization Structures
- ...
- Huge amount of knowledge $\Rightarrow$ Labor Intensive

A large library of Mathematics is **hard to build**

- Foundation
- Organization Structures
- ...
- Huge amount of knowledge $\Rightarrow$ Labor Intensive

Current Libraries of Mathematics are full of *redundancy*

Monoid: One theory, Multiple Representations

Lean

```
class monoid (M : Type u)
 extends semigroup M,
        has_one M :=
  (one_mul : ∀ a : M, 1 * a = a)
  (mul_one : ∀ a : M, a * 1 = a)
```

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
```

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;
}
```

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 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
```

Monoid: Multiple theories, Same Constructions

```
class monoid (M : Type u)
      extends semigroup M, has_one M :=
  (one_mul : ∀ a : M, 1 * a = a)
  (mul_one : ∀ a : M, a * 1 = a)
```

```
class add_monoid (M : Type u)
      extends add_semigroup M, has_zero M :=
  (zero_add : ∀ a : M, 0 + a = a)
  (add_zero : ∀ a : M, a + 0 = a)
```

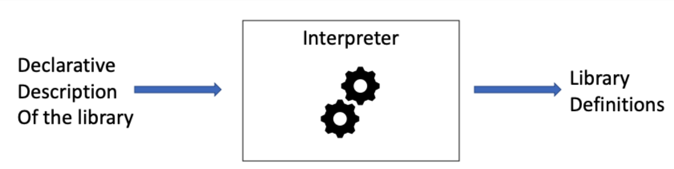
```
structure monoid_hom (M : Type*) (N : Type*)
          [monoid M] [monoid N]
  extends one_hom M N, mul_hom M N
```

```
structure add_monoid_hom (M : Type*) (N : Type*)
          [add_monoid M] [add_monoid N]
  extends zero_hom M N, add_hom M N
```

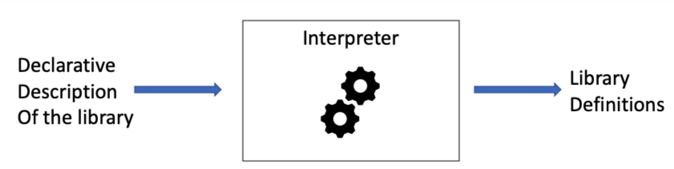
```
instance [monoid M] [monoid N] : monoid (M × N) :=
{ one_mul := assume a, prod.rec_on a $
    λ a b, mk.inj_iff.mpr ⟨one_mul _, one_mul _⟩,
  mul_one := assume a, prod.rec_on a $
    λ a b, mk.inj_iff.mpr ⟨mul_one _, mul_one _⟩,
  .. prod.semigroup, .. prod.has_one }
```

```
Products: 15 definitions.
```

Generative Tools



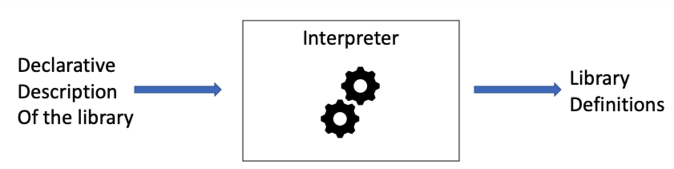
Generative Tools



- Inspiration: Haskell

```
data List a = Nil | Cons a (List a)
          deriving (Eq, Show, Ord, Read,
              -- by enabling some extensions
                    Functor, Generic, Data,
                    Foldable,Traversable, Lift)}
```

Generative Tools



- Inspiration: Haskell

```
data Point = Point { _x :: Double, _y :: Double }
makeLenses ''Point
```

Research Questions

- What is the right abstraction for theory presentations of algebraic structures?
- What pieces of information are needed for the system to generate particular constructions?
- Is there enough information that can be generated from theory presentations?
- How would this affect the activity of library building?

Research Questions

- What is the right abstraction for theory presentations of algebraic structures?
- What pieces of information are needed for the system to generate particular constructions?
- Is there enough information that can be generated from theory presentations?
- How would this affect the activity of library building?

Answers are given by **Universal Algebra**

Universal Algebra

A theory:

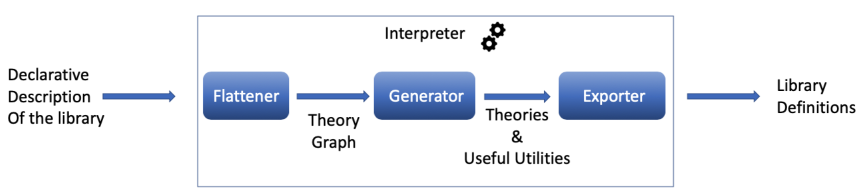
$$\Gamma = (\mathbb{S}, \mathbb{F}, \mathbb{E})$$

A Homomorphism between two Γ-Algebra:

1. hom : $\mathbb{S}_1 \rightarrow \mathbb{S}_2$
2. For every `op` $\in \mathbb{F}$.

   $\texttt{hom (op}_1\ \texttt{x}_1\ \texttt{..}\ \texttt{x}_n\texttt{) = op}_2\ \texttt{(hom x}_1\texttt{) .. (hom x}_n\texttt{)}$

Approach


Interpreter
Declarative
Description
Of the library
Flattener
Theory
Graph
Generator
Theories
&
Useful Utilities
Exporter
Library
Definitions

Requirements

1. A small language to represent theories without unnecessary details.
2. A large library of theories.
3. Meta programs to manipulate these theories
4. A type checker for the theories and constructions

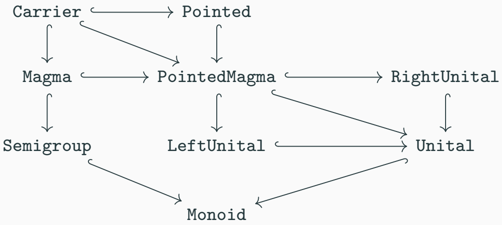
Tog: Language and TypeChecker

- Dependently typed language
    - Martin Löf type theory.
- Experimental language, in the style of Agda

```
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)
```

The Flattener

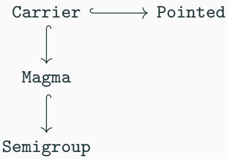
Build library as a theory graph



The Flattener

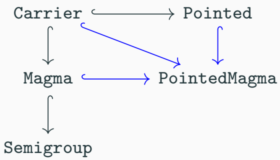
Build library as a theory graph: Combinators

```
Pointed = extend Carrier {e : A}
Magma   = extend Carrier {op : A -> A -> A}
Semigroup  = extend Magma {assoc: ...}
```



The Flattener

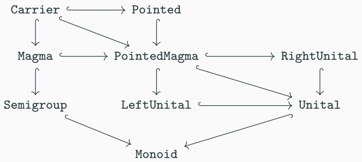
Build library as a theory graph: Combinators




```
Pointed = extend Carrier {e : A}
Magma   = extend Carrier {op : A -> A -> A}
Semigroup    =
    extend Magma {assoc: ...}
PointedMagma =
    combine Pointed {} Magma {} over Carrier
```

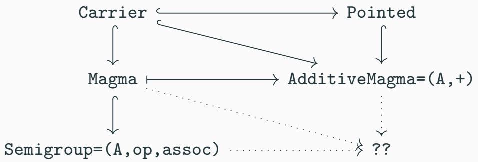
The Flattener

Build library as a theory graph: Combinators

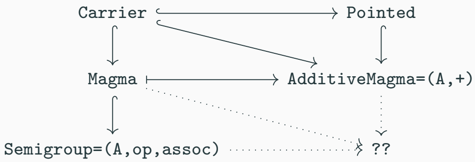



```
Pointed = extend Carrier {e : A}
Magma   = extend Carrier {op : A -> A -> A}
Semigroup    = extend Magma {assoc: ...}
PointedMagma = combine Pointed {} Magma {} over Carrier
LeftUnital   = extend PointedMagma { lunit_e : ... }
RightUnital  = extend PointedMagma { runit_e : ... }
Unital = combine LeftUnital {} RightUnital {}
         over PointedMagma
Monoid = combine Unital {} Semigroup {} over Magma
```

The Flattener: Combinators



The Flattener: Combinators




```
AdditiveSemigroup =
  combine AdditiveMagma {} Semigroup {op to +}
  over Magma
```

The Generator

```
data EqTheory = EqTheory  {
  name        :: Name_    ,
  sort        :: Constr   ,  -- the carrier 𝕊
  funcTypes   :: [Constr],  -- function symbols 𝔽
  axioms      :: [Constr],  -- equations 𝔼
  waist       :: Int        -- the number of parameters
}
```

Constructions for Free!

Homomorphisms

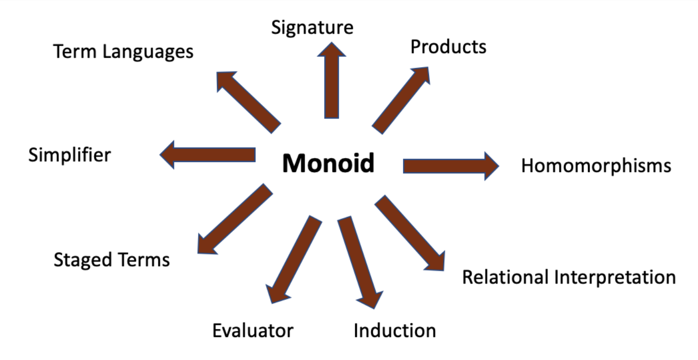
```haskell
homomorphism :: Eq.EqTheory -> Decl
homomorphism thry =
  let nm = "Hom"
      i1@(n1,b1,e1) = Eq.eqInstance thry (Just 1)
      i2@(n2,b2,e2) = Eq.eqInstance thry (Just 2)
      fnc    = homFunc thry i1 i2 (thry ^. Eq.sort)
      axioms = map (presAxiom thry i1 i2 fnc) (thry ^. Eq.funcTypes)
  in Record (mkName nm)
   (mkParams $ b1 ++ b2 ++
              map (\(n,e) -> Bind [mkArg n] e) [(n1,e1),(n2,e2)])
   (RecordDeclDef setType (mkName $ nm ++ "C") (mkField $ fnc : axioms))
```

Constructions for Free!

Homomorphisms

```
homomorphism :: Eq.EqTheory -> Decl
homomorphism thry =
  let nm = "Hom"
     i1@(n1,b1,e1) = Eq.eqInstance thry (Just 1)
     i2@(n2,b2,e2) = Eq.eqInstance thry (Just 2)
     fnc     = homFunc thry i1 i2 (thry ^. Eq.sort)
     axioms  = map (presAxiom thry i1 i2 fnc) (thry ^. Eq.funcTypes)
  in Record (mkName nm)
   (mkParams $ b1 ++ b2 ++
             map (\(n,e) -> Bind [mkArg n] e) [(n1,e1),(n2,e2)])
   (RecordDeclDef setType (mkName $ nm ++ "C") (mkField $ fnc : axioms))
```

Constructions for Free!



Monomorphism, Isomorphism, Endomorphism, Congruence relation, Quotient algebra, Trivial subtheory, Flipped theory, Monoid action, Monoid Cosets, composition of morphisms, kernel of homomorphisms, parse trees.

The Exporter

```
class Export a where
  export :: Config -> a -> Doc
```

The Exporter

```
class Export a where
  export :: Config -> a -> Doc
```

Useful functions:

- `replace :: String -> String`
    - replacing `"Nat"` with $\mathbb{N}$

The Exporter

```
class Export a where
  export :: Config -> a -> Doc
```

Useful functions:

- `replace :: String -> String`
    - replacing `"Nat"` with $\mathbb{N}$
- `callFunc :: Expr -> Expr`
    - replacing `(lookup x vars)` with `(nth vars x)`

The Exporter

```
class Export a where
  export :: Config -> a -> Doc
```

Useful functions:

- `replace :: String -> String`
    - replacing `"Nat"` with ℕ
- `callFunc :: Expr -> Expr`
    - replacing `(lookup x vars)` with `(nth vars x)`
- `preprocessDecls :: [Decl] -> [Decl]`

```
inductive ClMonoidTerm  (A : Type) : Type
  | singleton : A → ClMonoidTerm
  | op : ClMonoidTerm → ClMonoidTerm → ClMonoidTerm
  | e  : ClMonoidTerm
```

Results

Starting with **227** theory expressions:

- **5092** library definitions.
- **32,459** lines of code.
- Exported to **Lean**, **Agda** (flat and predicate style theories).

Conclusion

- Support the process of building libraries
  - Goal: Eliminate Redundancy.
  - Technique: Generative Programming.
- Abstract over design decisions.
- Generate uniform constructions.

Future Work

- Generating more definitions
  - possibly outside universal algebra
- Enrich the theory graph structure.
- Exporting to more formal systems.
  - Studying them as program families.
- Generalizing to higher order logics.

Future Work

```
Monoid = combine Unital and Semigroup over Magma
         generate Homomorphism, OpenTerms, Simplifier
         using (waist=1,eq="=")
         export_to lean
```

References

1 Jacques Carette, Russell O'Connor, and Yasmine Sharoda. *Building on the diamonds between theories: Theory presentation combinators.* arXiv preprint arXiv:1812.08079, 2019.

2 Jacques Carette, William M. Farmer, and Yasmine Sharoda. *Leveraging the information contained in theory presentations.* In Proceedings of the 13th International Conference on Intelligent Computer Mathematics, CICM 2020.

3 Florian Rabe and Yasmine Sharoda. *Diagram Combinators in MMT.* In Proceedings of the 12th International Conference on Intelligent Computer Mathematics, CICM 2019

4 Musa Al-hassy, Jacques Carette, and Wolfram Kahl. *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