Leveraging Information Contained in Theory Presentations

Jacques Carette, William M. Farmer, and Yasmine Sharoda

McMaster University

July 28, 2020

Large Math Libraries

A large library of Mathematics is:

- useful

Large Math Libraries

A large library of Mathematics is:

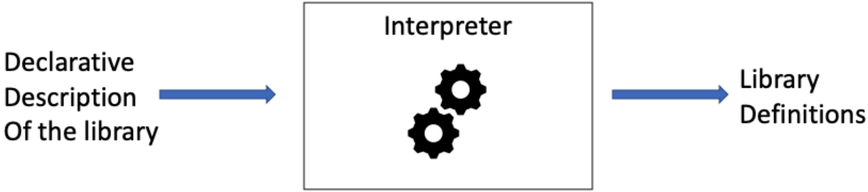
- useful
- not-easy to build
    - Different foundations
    - Organization of information
    - ...

Large Math Libraries

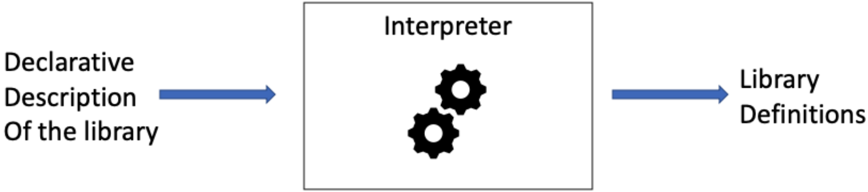
A large library of Mathematics is:

- useful
- not-easy to build
  - Different foundations
  - Organization of information
  - ...
- Labor intensive
  - Creating
  - Maintaining

Large Math Libraries: Generative Approach



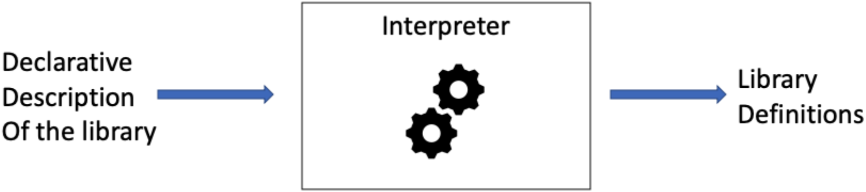
Large Math Libraries: Generative Approach



- Inspiration: Haskell

```
data List a = Nil | Cons a (List a)
          deriving (Eq, Show, Ord, Read)
```

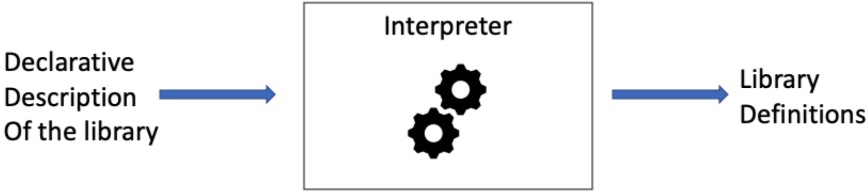
Large Math Libraries: Generative Approach



- 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)}
```

Large Math Libraries: Generative Approach



- Inspiration: Haskell

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

Large Math Libraries: Generative Approach

- Which parts of the library can be generated and which parts need to be created by the developer?

Large Math Libraries: Generative Approach

- Which parts of the library can be generated and which parts need to be created by the developer?
- What are the preconditions for generating this information?

Large Math Libraries: Generative Approach

- Which parts of the library can be generated and which parts need to be created by the developer?
- What are the preconditions for generating this information?
- How would a generative approach affect the activity of library development?

Which parts of the library can be generated?

Algebra libraries typically contain

Which parts of the library can be generated?

Algebra libraries typically contain

- Theories

```
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
```

Which parts of the library can be generated?

Algebra libraries typically contain

- Theories

```
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
```

- Related Constructions

```
record IsMonoidMorphism (⟦_⟧:Morphism) : Set(c₁ ⊔ ℓ₁ ⊔ c₂ ⊔ ℓ₂) where
field
  sm-homo : IsSemigroupMorphism F.semigroup T.semigroup ⟦_⟧
  ε-homo   : Homomorphic₀ ⟦_⟧   F.ε T.ε
```

Which parts of the library can be generated?

Algebra libraries typically contain

- Theories

```
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
```

- **Related Constructions**

```
record IsMonoidMorphism (⟦_⟧:Morphism) : Set(c₁ ⊔ ℓ₁ ⊔ c₂ ⊔ ℓ₂) where
field
  sm-homo : IsSemigroupMorphism F.semigroup T.semigroup ⟦_⟧
  ε-homo   : Homomorphic₀ ⟦_⟧   F.ε T.ε
```

- Theorems and proofs

```
comm+idˡ ⇒idʳ : LeftIdentity e _•_ → RightIdentity e _•_
comm+idˡ⇒idʳ idˡ x = begin
  x • e ≈ ⟨ comm x e ⟩
  e • x ≈ ⟨ idˡ x ⟩
  x
```

What are the preconditions for generating them?

Universal Algebra

- theory: `(S,F,E)`

What are the preconditions for generating them?

Universal Algebra

- theory: `(S,F,E)`
- homomorphism between (S_1,F_1,E_1) and (S_2,F_2,E_2)

What are the preconditions for generating them?

Universal Algebra

- theory: `(S,F,E)`
- homomorphism between (S_1, F_1, E_1) and (S_2, F_2, E_2)
  - A function $\texttt{hom} : S_1 \rightarrow S_2$

What are the preconditions for generating them?

Universal Algebra

- theory: `(S,F,E)`
- homomorphism between (S_1, F_1, E_1) and (S_2, F_2, E_2)
  - A function $\mathtt{hom} : S_1 \to S_2$
  - For every $\mathtt{op} \in F$,

$$\mathtt{hom}\ (\mathtt{op}_1\ x_1\ \cdots\ x_n) = \mathtt{op}_2\ (\mathtt{hom}\ x_1)\ \cdots\ (\mathtt{hom}\ x_2)$$

How would this affect the library building process?

How would this affect the library building process?

One theory, Multiple Representations

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);
}
```

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
```

How would this affect the library building process?

Multiple theories, One Construction

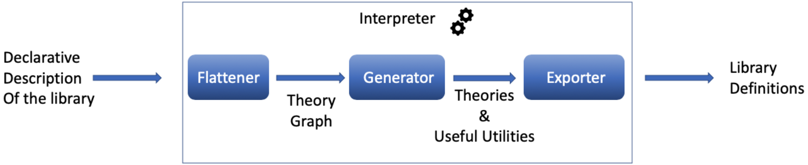
```
module _ {c₁ ℓ₁ c₂ ℓ₂}
(From : Monoid c₁ ℓ₁)
(To   : Monoid c₂ ℓ₂) where

 private
  module F = Monoid From
  module T = Monoid To
 open Definitions F.Carrier T.Carrier 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
```

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

 private
  module F = CommutativeMonoid From
  module T = CommutativeMonoid To
 open Definitions F.Carrier T.Carrier T._≈_

record IsCommutativeMonoidMorphism (⟦_⟧:Morphism)
: Set(c₁ ⊔ ℓ₁ ⊔ c₂ ⊔ ℓ₂) where
field
  mn-homo :
    IsMonoidMorphism F.monoid T.monoid ⟦_⟧

  open IsMonoidMorphism mn-homo public
```

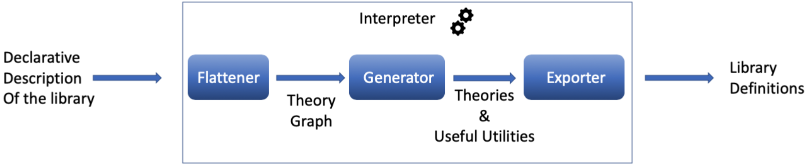
How would this affect the library building process?

Signature, Product Algebra, Basic Term Language, Homomorphism, Closed Term Language, Open Term Language, Evaluator, Simplification rules, Staged terms, Finally tagless representations, induction principle, Relational Interpretation, Monomorphism, Isomorphism, Endomorphism, Congruence relation, Quotient algebra, Trivial subtheory, Flipped theory, Monoid action, Monoid Cosets, composition of morphisms, kernel of homomorphisms, parse trees.

Approach


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

Approach



Internalize Universal Algebra abstractions into a prover

Tog: A small Language and TypeChecker

- dependently typed language
    - Martin Löf type theory.
- experimental language, in the style of Agda

Tog: A small 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)
```

Tog: Internal Representation

- One universe: `Set`
- Functions: `Fun Expr Expr`.
- Axioms: `Pi Telescope Expr`.
    - They use the underlying propositional equality: `Eq Expr Expr`.
- Theories: Σ-types.
    - parameters: `Binding`.
        - `HBind [Arg] Expr`
        - `Bind  [Arg] Expr`
    - declarations: `Constr Name Expr`.

Equational Theory

```
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!

Example: Product Algebra

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

```
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)
```

Constructions for Free!

Example: Homomorphism

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

```
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)
```

More Constructions

Signature, Product Algebra, Basic Term Language, Homomorphism, Closed Term Language, Open Term Language, Evaluator, Simplification rules, Staged terms, Finally tagless representations, induction principle, Relational Interpretation, Monomorphism, Isomorphism, Endomorphism, Congruence relation, Quotient algebra, Trivial subtheory, Flipped theory, Monoid action, Monoid Cosets, composition of morphisms, kernel of homomorphisms, parse trees.

More Constructions

Signature, **Product Algebra**, **Basic Term Language**, **Homomorphism**, **Closed Term Language**, **Open Term Language**, **Evaluator**, **Simplification rules**, **Staged terms**, **Finally tagless representations**, **induction principle**, **Relational Interpretation**, Monomorphism, Isomorphism, Endomorphism, Congruence relation, Quotient algebra, Trivial subtheory, Flipped theory, Monoid action, Monoid Cosets, composition of morphisms, kernel of homomorphisms, parse trees.

Generated

```
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 MonoidSig (AS : Set) : Set where
 constructor MonoidSigSigC
 field
  eS : AS
  opS : AS -> AS -> AS


data MonoidLang : Set where
  eL : MonoidLang
  opL : MonoidLang -> MonoidLang -> MonoidLang
```

```
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)

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)
```

Testing: MathScheme Library

- Built using 3 combinators:
  - Extension.

    `CommMagma = extend Magma {comm : ...}`
  - Rename.

    `AddMagma = rename Magma {op to +}`
  - Combine.

    `Monoid = combine Semigroup {} Unital {} over PointedMagma`

Testing: MathScheme Library

- Theory Graph

$$
\begin{tikzcd}
\texttt{Carrier} \arrow[r, hook] \arrow[d, hook] \arrow[dr, hook] & \texttt{Pointed} \arrow[d, hook] & \\
\texttt{Magma} \arrow[r, hook] \arrow[d, hook] & \texttt{PointedMagma} \arrow[r, hook] \arrow[d, hook] \arrow[dr, hook] & \texttt{RightUnital} \arrow[d, hook] \\
\texttt{Semigroup} \arrow[d, hook] & \texttt{LeftUnital} \arrow[r, hook] & \texttt{Unital} \arrow[dll, hook] \\
\texttt{Monoid} & &
\end{tikzcd}
$$

Results

	Input	Time of Writing	Now
Definitions	227	1132	5047
LOC	316	14811	106468

Conclusion

Algebra libraries formalizes the same standard mathematical information again and again

- Algebra library in every formal system
- At least 4 libraries of Algebra in Coq.

Conclusion

Algebra libraries formalizes the same standard mathematical information again and again

- Algebra library in every formal system
- At least 4 libraries of Algebra in Coq.

> *"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."*[1]

[1]Garillot, François, et al. "Packaging mathematical structures." International Conference on Theorem Proving in Higher Order Logics. Springer, Berlin, Heidelberg, 2009.

Conclusion and Future Work

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

Conclusion and Future Work

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

Conclusion and Future Work

- Support the process of building libraries
  - Goal: Eliminate Redundancy.
  - Technique: Generative Programming.
- Abstract over design decisions.
- Generate uniform constructions.
- Future Work:
  - Generating more definitions.
  - Enrich the theory graph structure.

Conclusion and Future Work

- Support the process of building libraries
  - Goal: Eliminate Redundancy.
  - Technique: Generative Programming.
- Abstract over design decisions.
- Generate uniform constructions.
- Future Work:
  - Generating more definitions.
  - Enrich the theory graph structure.
  - Exporting to existing, full-featured systems.

Conclusion and Future Work

- Support the process of building libraries
  - Goal: Eliminate Redundancy.
  - Technique: Generative Programming.
- Abstract over design decisions.
- Generate uniform constructions.
- Future Work:
  - Generating more definitions.
  - Enrich the theory graph structure.
  - Exporting to existing, full-featured systems.
  - Generalizing to higher order logics.

Conclusion and Future Work

- Support the process of building libraries
  - Goal: Eliminate Redundancy.
  - Technique: Generative Programming.
- Abstract over design decisions.
- Generate uniform constructions.
- Future Work:
  - Generating more definitions.
  - Enrich the theory graph structure.
  - Exporting to existing, full-featured systems.
  - Generalizing to higher order logics.
  - Scripting language for referencing theories and constructions within the library.

Conclusion

```
Monoid = combine Semigroup {} Unital {}  over PointedMagma
         generate (Homomorphism, ProductTheory, TermLang)
         using ···
```