

Leveraging Information Contained in Theory Presentations

Yasmine Sharoda

Supervisors:

Jacques Carette and William M. Farmer

Large Libraries of Mathematics

- QED Manifesto, 1994:
 - One library to formalize all of Mathematics

Large Libraries of Mathematics

- QED Manifesto, 1994:
 - One library to formalize all of Mathematics
- Building a library requires:
 - Foundation
 - Organizational Structures
 - ...
 - Huge amount of knowledge \Rightarrow **Labour Intensive**

Large Libraries of Mathematics

- QED Manifesto, 1994:
 - One library to formalize all of Mathematics
- Building a library requires:
 - Foundation
 - Organizational Structures
 - ...
 - Huge amount of knowledge \Rightarrow **Labour 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 Monoid : ?NatDed =
  includes ?Semigroup
  unit : tm u # e
  unit_axiom : ⊢ ∀ [x] = x * e = x
```

```
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
```

Haskell

```
class Semigroup 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_neutral : forall x, x * 1 = x;
}
```

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

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

open IsSemigroup isSemigroup public

identityl : LeftIdentity ε •
identityl = proj1 identity
identityr : RightIdentity ε •
identityr = proj2 identity
```

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 Monoid : ?NatDed =
  includes ?Semigroup
  unit : tm u # e
  unit_axiom : ⊢ ∀ [x] = x * e = x
```

```
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
```

Haskell

```
class Semigroup 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_neutral : forall x, x * 1 = x;
}
```

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

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

open IsSemigroup isSemigroup public

identityl : LeftIdentity ε •
identityl = proj₁ identity
identityr : RightIdentity ε •
identityr = proj₂ identity
```

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 Monoid : ?NatDed =
  includes ?Semigroup
  unit : tm u # e
  unit_axiom : ⊢ ∀ [x] = x * e = x
```

```
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
```

Haskell

```
class Semigroup 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_neutral : forall x, x * 1 = x;
}
```

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
record IsMonoid (ε : Op2) (ε : A)
  : Set (a ⊔ ℓ) where
  field
    isSemigroup : IsSemigroup ε
    identity : Identity ε
open IsSemigroup isSemigroup public
identityl = LeftIdentity ε
identityl = proj1 identity
identityr : RightIdentity ε
identityr = proj2 identity
```

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 Monoid : ?NatDed =
  includes ?Semigroup
  unit : tm u # e
  unit_axiom : ⊢ ∀ [x] = x * e = x
```

```
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
```

Haskell

```
class Semigroup 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_neutral : forall x, x * 1 = x;
}
```

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

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

open IsSemigroup isSemigroup public

identityl : LeftIdentity ε •
identityl = proj₁ identity
identityr : RightIdentity ε •
identityr = proj₂ identity
```


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 Monoid : ?NatDed =
  includes ?Semigroup
  unit : tm u # e
  unit_axiom : ⊢ ∀ [x] = x * e = x
```

```
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
```

Haskell

```
class Semigroup 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_neutral : forall x, x * 1 = x;
}
```

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

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

open IsSemigroup isSemigroup public

identityl : LeftIdentity ε •
identityl = proj₁ identity
identityr : RightIdentity ε •
identityr = proj₂ identity
```

Can we abstract over these design decisions?

Monoid: One theory, Many Constructions

```
theory Homomorphism {
  M1, M2 : Monoid
  hom : M1.A → M2.A
  pres-e : hom (M1.e) = M2.e
  pres-op : (x y : M1.A) →
    hom (M1.op x y) = M2.op (hom x) (hom y)
}
theory Isomorphism {
  M1, M2 : Monoid
  f : Homomorphism M1 M2
  g : M2.A → M1.A
  id1 : {x : M1.A} → (g o f.hom) x = x
  id2 : {x : M2.A} → (f.hom o g) x = x
}
theory Endomorphism {
  M : Monoid
  Homomorphism M M
}
theory Automorphism {
  M1, M2 : Monoid
  Isomorphism M1 M2
}
```

```
theory Product {
  M1, M2 : Monoid
  e : M1.A × M2.A
  op : M1.A × M2.A → M1.A × M2.A → M1.A × M2.A
  lunit : {x : M1.A × M2.A} → op e x = x
  runit : {x : M1.A × M2.A} → op x e = x
  assoc : {x y z : M1.A × M2.A} →
    op x (op y z) = op (op x y) z
}
theory Submonoid {
  M : Monoid
  subset : Set → Set
  AS : subset M.A
  eS : AS
  opS : AS → AS → AS
}
type Expr :=
  e : Expr
  op : Expr → Expr → Expr
type OpenExpr :=
  vars : {n : Nat} → Fin n → OpenExpr
  e : OpenExpr
  op : OpenExpr → OpenExpr → OpenExpr
```

signature, trivial sub-theory, monomorphisms, epimorphisms, kernel of a homomorphism, composition of morphisms, quotient algebra, staged term language, induction principle, evaluation of terms, simplification of terms, equivalence of terms, printers, ...

Monoid: Multiple Theories, Same Constructions

```
theory Monoid {  
  A : type  
  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  
}
```

```
theory MonoidHom {  
  M1, M2 : Monoid  
  hom : M1.A → M2.A  
  pres-e : hom (M1.e) = M2.e  
  pres-op : (x y : M1.A) →  
    hom (M1.op x y) = M2.op (hom x) (hom y)  
}
```

```
type MonoidExpr :=  
  e : MonoidExpr  
  op : MonoidExpr → MonoidExpr → MonoidExpr
```

```
theory Group {  
  A : type  
  e : A  
  op : A → A → A  
  inv : A → A  
  lunit : {x : A} → op e x = x  
  runit : {x : A} → op x e = x  
  linverse : {x : A} → op x (inv x) == e  
  rinverse : {x : A} → op (inv x) x == e  
  assoc : {x y z : A} → op x (op y z) = op (op x y) z  
}
```

```
theory GroupHom {  
  G1, G2 : Group  
  hom : G1.A → G2.A  
  pres-e : hom (G1.e) = G2.e  
  pres-op : (x y : G1.A) →  
    hom (G1.op x y) = G2.op (hom x) (hom y)  
  pres-inv : (x : G1.A) →  
    hom (G1.inv x) = G2.inv (hom x)  
}
```

```
type GroupExpr :=  
  e : GroupExpr  
  inv : GroupExpr → GroupExpr  
  op : GroupExpr → GroupExpr → GroupExpr
```

Monoid: Multiple Theories, Same Constructions

```
theory Monoid {  
  A : type  
  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  
}
```

```
theory MonoidHom {  
  M1, M2 : Monoid  
  hom : M1.A → M2.A  
  pres-e : hom (M1.e) = M2.e  
  pres-op : (x y : M1.A) →  
    hom (M1.op x y) = M2.op (hom x) (hom y)  
}
```

```
type MonoidExpr :=  
  e : MonoidExpr  
  op : MonoidExpr → MonoidExpr → MonoidExpr
```

```
theory Group {  
  A : type  
  e : A  
  op : A → A → A  
  inv : A → A  
  lunit : {x : A} → op e x = x  
  runit : {x : A} → op x e = x  
  linverse : {x : A} → op x (inv x) == e  
  rinverse : {x : A} → op (inv x) x == e  
  assoc : {x y z : A} → op x (op y z) = op (op x y) z  
}
```

```
theory GroupHom {  
  G1, G2 : Group  
  hom : G1.A → G2.A  
  pres-e : hom (G1.e) = G2.e  
  pres-op : (x y : G1.A) →  
    hom (G1.op x y) = G2.op (hom x) (hom y)  
  pres-inv : (x : G1.A) →  
    hom (G1.inv x) = G2.inv (hom x)  
}
```

```
type GroupExpr :=  
  e : GroupExpr  
  inv : GroupExpr → GroupExpr  
  op : GroupExpr → GroupExpr → GroupExpr
```

Can we make use of this uniformity?

A theory:

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

- \mathcal{S} : a sort
- \mathcal{F} : set of function symbols
- \mathcal{E} : set of axioms

A theory:

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

- \mathcal{S} : a sort
- \mathcal{F} : set of function symbols
- \mathcal{E} : set of axioms

- A homomorphism between two Γ -algebras:

- $\text{hom} : \mathcal{S}_1 \rightarrow \mathcal{S}_2$

- For every $\text{op} \in \mathcal{F}$:

$$\text{hom}(\text{op}_1 \ x_1 \ \dots \ x_n) = \text{op}_2(\text{hom } x_1) \ \dots \ (\text{hom } x_n)$$

- The closed term language L induced by Γ is the set of:

- All constants of Γ

- For every $\text{op} \in \mathcal{F}$, with $\text{arity} > 0$:

$\text{t}_{\text{op}} \ t_1 \ \dots \ t_n$, such that $t_1 \ \dots \ t_n$ are closed terms of L .

Redundancies in Libraries

Agda

Construction	Number of Occurrences
Signatures	7
Homomorphisms	7
Monomorphisms	7
Isomorphisms	7
Products	10
Products of Signatures	3
Term Language	3
Evaluation Function	3
Total	47

Lean

Construction	Number of Occurrences
Homomorphisms (Bundled)	3
Homomorphisms (Unbundled)	8
Products	22
Subtheory	5
Total	38

- > 20 algebraic structures in each library.

Redundancies in Libraries

Agda

Construction	Number of Occurrences
Signatures	7
Homomorphisms	7
Monomorphisms	7
Isomorphisms	7
Products	10
Products of Signatures	3
Term Language	3
Evaluation Function	3
Total	47

Lean

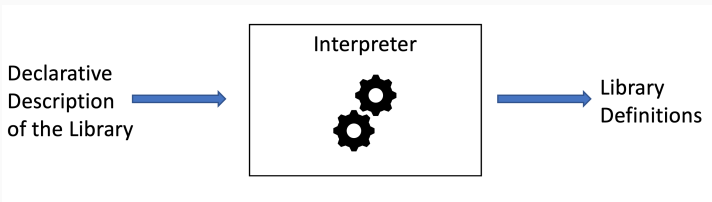
Construction	Number of Occurrences
Homomorphisms (Bundled)	3
Homomorphisms (Unbundled)	8
Products	22
Subtheory	5
Total	38

- > 20 algebraic structures in each library.
- > 300 algebraic structures collected by Peter Jipsen.¹

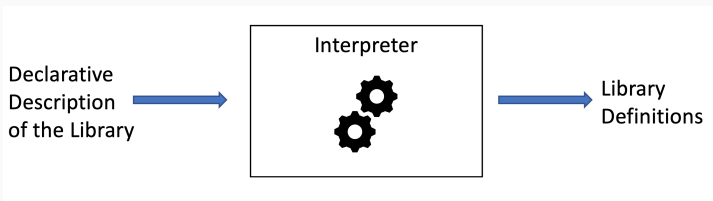
¹source: <http://math.chapman.edu/~jipsen/structures/doku.php>

**Can the abstractions and uniformity
provided by universal algebra be captured by
meta-programs that generate parts of algebra
libraries?**

Generative Approach to Library Building



Generative Approach to Library Building



- 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)

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

Requirements

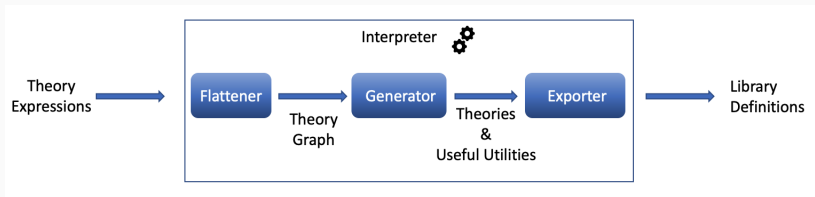
1. A small **language** to represent theories.
2. Some **meta programs** to manipulate these theories.
3. A **type checker** for the theories and constructions.
4. A large **library** of theories.

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)
```

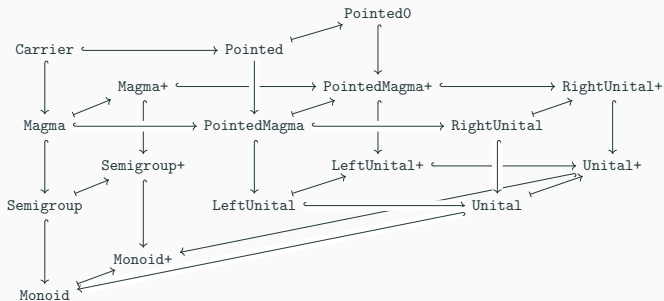
Approach: Three-Phase Interpreter



1. The Flattener



Theory Graph



1. The Flattener: Combinators

Theory Expressions

1. Extension

`Semigroup = extend Magma {assoc: ...}`

`Magma` \hookrightarrow `Semigroup`

1. The Flattener: Combinators

Theory Expressions

1. Extension

`Semigroup = extend Magma {assoc: ...}`

2. Rename

`AdditiveMagma = rename Magma {op to +}`



1. The Flattener: Combinators

Theory Expressions

1. Extension

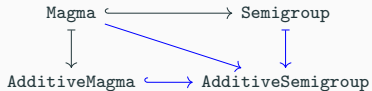
```
Semigroup = extend Magma {assoc: ...}
```

2. Rename

```
AdditiveMagma = rename Magma {op to +}
```

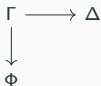
3. Combine

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



1. The Flattener: Computing Pushouts

Pushouts are a 5-ary operations:



- 3 theories.
- 2 arrows.

1. The Flattener: Computing Pushouts

Pushouts are a 5-ary operations:

$$\begin{array}{ccc} \Gamma & \longrightarrow & \Delta \\ \downarrow & & \\ \Phi & & \end{array}$$

- 3 theories.
- 2 arrows.

```
combine AdditiveMonoid {} Group { ... }  
combine AdditiveMonoid {} MultMonoid {}
```

$$\begin{array}{ccc} ?? & \longrightarrow & \text{Group} \\ \downarrow & & \\ \text{AdditiveMonoid} & & \end{array}$$

$$\begin{array}{ccc} ?? & \longrightarrow & \text{MultMonoid} \\ \downarrow & & \\ \text{AdditiveMonoid} & & \end{array}$$

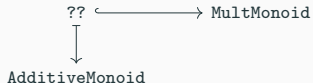
1. The Flattener: Computing Pushouts

Pushouts are a 5-ary operations:

$$\begin{array}{ccc} \Gamma & \longrightarrow & \Delta \\ \downarrow & & \\ \Phi & & \end{array}$$

- 3 theories.
- 2 arrows.

```
combine AdditiveMonoid {} Group { ... }  
combine AdditiveMonoid {} MultMonoid {}
```

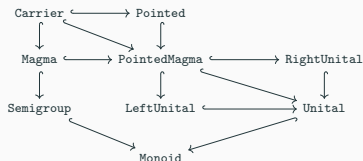


```
combine AdditiveMonoid {} Group { ... } over Monoid  
combine AdditiveMonoid {} MultMonoid {} over Carrier
```

1. The Flattener



```
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
```



Empty, Carrier, Pointed, UnaryOperation, PointedUnarySystem, FixedPoint, Magma, AdditiveMagma, MultMagma, PointedMagma, Involution, UnaryDistributes, UnaryAntiDistribution, IdempotentUnary, InvolutiveMagma, LeftInverseMagma, RightInverseMagma, IdempotentMagma, IdempotentAdditiveMagma, Pointed0Magma, PointedPlusMagma, AdditivePointedMagma, Pointed1Magma, PointedTimesMagma, MultPointedMagma, CommutativeMagma, CommutativeAdditiveMagma, CommutativePointedMagma, AntiAbsorbent, SteinerMagma, Squag, PointedSteinerMagma, Sloop, LeftDistributiveMagma, RightDistributiveMagma, Unital, LeftBiMagma, RightBiMagma, QuasiGroup, MoufangLaw, MoufangQuasiGroup, Loop, MoufangIdentity, MoufangLoop, Shelf, LeftBinaryInverse, RightBinaryInverse, BinaryInverse, Rack, Spindle, Quandle, RightSelfInverse, Semigroup, AdditiveSemigroup, CommutativeSemigroup, MultCommutativeSemigroup, CancellativeSemigroup, InvolutivePointedSemigroup, Band, MiddleAbsorption, MiddleCommute, RectangularBand, NormalBand, RightMonoid, LeftMonoid, PointedSemigroup, AdditivePointedSemigroup, AdditiveUnital, MultPointedSemigroup, Monoid, AdditiveMonoid, DoubleMonoid, CommutativeMonoid, CancellativeMonoid, CancellativeCommutativeMonoid, Zero, AdditiveCommutativeMonoid, BooleanGroup, InverseUnaryOperation, Inverse, PseudoInverse, PseudoInvolution, RegularSemigroup, QuasiInverse, Group, AdditiveGroup, CommutativeGroup, MultGroup, AbelianGroup, AbelianAdditiveGroup, RingoidSig, LeftRingoid, RightRingoid, Ringoid, NonassociativeRing, AndDeMorgan, OrDeMorgan, DualDeMorgan, AssociativeLeftRingoid, LeftPreSemiring, AssociativeRightRingoid, RightPreSemiring, PreSemiring, AssocPlusRingoid, AssocTimesRingoid, NearSemiring, NearRing, SemiRng, Rng, SemiRngWithUnit, Semiring, Ring, CommutativeRing, BooleanRing, IdempotentSemiRng, IdempotentSemiring, InvolutiveFixes, InvolutiveFixedPoint, InvolutiveRingoid, InvolutiveRing, JacobianIdentity, AntiCommutativeRing, LieRing, MeetSemilattice, MultMeetSemilattice, BoundedMeetSemilattice, JoinSemilattice, BoundedJoinSemilattice, DualSemilattices, LeftAbsorption, LeftAbsorptionOp, Absorption, Lattice, Modularity, ModularLattice, DistributiveLattice, BoundedJoinLattice, BoundedMeetLattice, BoundedLattice, BoundedModularLattice, BoundedDistributiveLattice,

2. The Generator



2. The Generator



- Uni-sorted equational theory: $\Gamma = (S, \mathcal{F}, \mathcal{E})$

```
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  
}
```

2. The Generator



- Uni-sorted equational theory: $\Gamma = (S, \mathcal{F}, \mathcal{E})$

```
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  
}
```

- Instance of a theory:

```
type EqInstance = (Name_, [Binding], Expr)
```

Example:

$$\{A : \text{Set}\} \rightarrow M : \text{Monoid } A$$

Constructions for Free!

```
record Monoid (A : Set)
  : Set where
e  : A
op : A → A → A
lunit : ...
runit : ...
assoc : ...
```

```
record Hom {A1 A2 : Set}
  (M1 : Monoid A1) (M2 : Monoid A2)
  : Set where
hom : A1 → A2
pres-e : hom (e M1) = e M2
pres-op : {x1 x2 : A1} →
  hom ((op M1) x1 x2) = (op M2) (hom x1) (hom x2)
```

```
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
```

```
    axioms = map (presAxiom thry i1 i2 fnc) (thry ^. Eq.funcTypes)
```

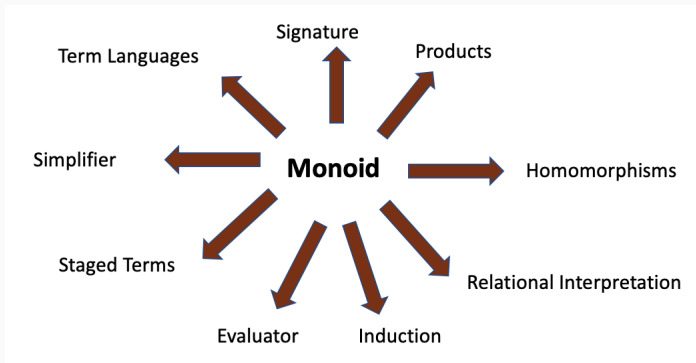
```
  in Record (mkName nm)
```

```
    (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!

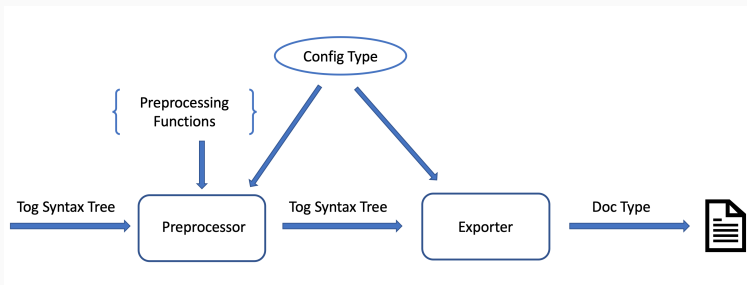


monomorphism, epimorphism, endomorphism, isomorphism, automorphism, kernel of a morphism, composition of morphisms, quotient algebra, sub-theory, trivial sub-theory, parse trees, equivalence of terms, ...

3. The Exporter



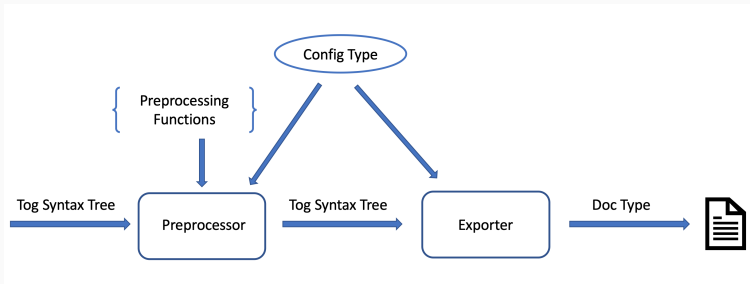
3. The Exporter



Preprocessor

- Universes
- Prelude definitions
- Non-linear pattern matching
- Functions as constructors
- Names misalignment

3. The Exporter



Exporter

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

Results

Starting with 227 theory expressions:

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

Results

Starting with 227 theory expressions:

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

Flattener	5.17 s
Generator	2.7 s
Exporter	9.1 us
Type-checking	28 mins

- Generalizing the approach to **generalized algebraic theories**.
- Proof assistants as **program families**.
 - better understanding how design decisions affect theory presentations
 - staged exporter to multiple proof assistants
- a **DSL** for library development.

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

Summary of Contributions:

- Highlighted the redundancy in libraries formalizing the algebraic hierarchy.
- Built a library of 227 theories describing the algebraic hierarchy using theory combinators.
- Compiled a list of structures that can be generated from theory presentations.
- Generated some of these constructions in Tog, a small implementation of a dependently typed language.
- Exported the library to Agda and Lean.

Sagemath:

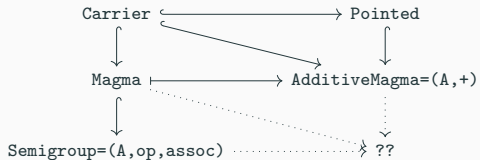
```
class Monoid_class(Parent):
    def __init__(self, names):
        from sage.categories.monoids import Monoids
        category = Monoids().FinitelyGeneratedAsMagma()
        Parent.__init__(self, base=self, names=names, category=category)

-- sage.categories.monoids
class Monoids(CategoryWithAxiom)
    _base_category_class_and_axiom = (Semigroups, "Unital")
```

Isabelle:

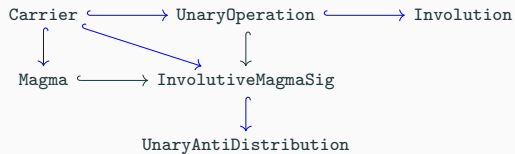
```
locale monoid =
  fixes G (structure)
  assumes m_closed [intro, simp]:
     $\llbracket x \in \text{carrier } G; y \in \text{carrier } G \rrbracket \Rightarrow x \oplus y \in \text{carrier } G$ 
  and m_assoc:
     $\llbracket x \in \text{carrier } G; y \in \text{carrier } G; z \in \text{carrier } G \rrbracket$ 
     $\Rightarrow (x \oplus y) \oplus z = x \oplus (y \oplus z)$ 
  and one_closed [intro, simp]:  $1 \in \text{carrier } G$ 
  and l_one [simp]:  $x \in \text{carrier } G \Rightarrow 1 \oplus x = x$ 
  and r_one [simp]:  $x \in \text{carrier } G \Rightarrow x \oplus 1 = x$ 
```

Combine renames



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

Distinguished Arrows



What is in Tog that needs to be changes

What are the differences between Agda and Lean