# Leveraging Information in Theory Presentations

Yasmine Sharoda

Supervisors: Jacques Carette and William Farmer

Department of Computing and Software, McMaster University

July 8,2019

# Defining Homomorphism

```
theory Foo {
  U : type
  s1 : U
  s2 : U → U → U
}
```

# Defining Homomorphism

```
theory Foo {
  U : type
  s1 : U
  s2 : U → U → U
}

theory FooHom{
  f1 , f2 : Foo
  h : f1.U → f2.U
  pres-s1 : h f1.s1 = f2.s1
  pres-s2 : h (f1.s2 x y)
          = f2.s2 (h x) (h
              y)
}
```

# Defining Homomorphism

```
theory Foo {
  U : type
  s1 : U
  s2 : U → U → U
}
```

```
theory Bar {
  U : type
  s1 : U
  s2 : U → U
}
```

```
theory FooHom{
  f1 , f2 : Foo
  h : f1.U → f2.U
  pres-s1 : h f1.s1 = f2.s1
  pres-s2 : h (f1.s2 x y)
          = f2.s2 (h x) (h
              y)
}
```

# Defining Homomorphism

```
theory Foo {
  U : type
  s1 : U
  s2 : U → U → U
}
```

```
theory Bar {
  U : type
  s1 : U
  s2 : U → U
}
```

```
theory FooHom{
  f1 , f2 : Foo
  h : f1.U → f2.U
  pres-s1 : h f1.s1 = f2.s1
  pres-s2 : h (f1.s2 x y)
          = f2.s2 (h x) (h
              y)
}
```

```
theory BarHom {
  b1, b2 : Bar
  h : b1.U → b2.U
  pres-s1 : h s1 = s2
  pres-s2 : h (b1.s2 x)
          = b2.s2 (h x)
}
```

# Defining Homomorphism

```
theory Foo {
  U : type
  s1 : U
  s2 : U → U → U
} generates Hom
```

```
theory Bar {
  U : type
  s1 : U
  s2 : U → U
} generates Hom
```

# Real Example: Agda Library [1]

```
Homomorphic₀ : Morphism → From → To → Set _
Homomorphic₀ ⟦_⟧ • ∘ = ⟦ • ⟧ ≈ ∘

Homomorphic₁ : Morphism → Fun₁ From → Op₁ To → Set _
Homomorphic₁ ⟦_⟧ •_ ∘_ = ∀ x → ⟦ • x ⟧ ≈ (∘ ⟦ x ⟧)

Homomorphic₂ : Morphism → Fun₂ From → Op₂ To → Set _
Homomorphic₂ ⟦_⟧ _•_ _∘_ =
  ∀ x y → ⟦ x • y ⟧ ≈ (⟦ x ⟧ ∘ ⟦ y ⟧)

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

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

1 source: https://github.com/agda/agda-stdlib/blob/master/src/Algebra/Morphism.agda

# Real Example: Agda Library [1]

```
Homomorphic₀ : Morphism → From → To → Set _
Homomorphic₀ ⟦_⟧ • ∘ = ⟦ • ⟧ ≈ ∘

Homomorphic₁ : Morphism → Fun₁ From → Op₁ To → Set _
Homomorphic₁ ⟦_⟧ •_ ∘_ = ∀ x → ⟦ • x ⟧ ≈ (∘ ⟦ x ⟧)

Homomorphic₂ : Morphism → Fun₂ From → Op₂ To → Set _
Homomorphic₂ ⟦_⟧ _•_ _∘_ =
  ∀ x y → ⟦ x • y ⟧ ≈ (⟦ x ⟧ ∘ ⟦ y ⟧)

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

  open IsMonoidMorphism mn-homo public

  ⁻¹-homo : Homomorphic₁ ⟦_⟧ F._⁻¹ T._⁻¹
  ⁻¹-homo x = let open EqR T.setoid in T.uniqueˡ-⁻¹ ⟦ x F.⁻¹ ⟧ ⟦ x ⟧ $ begin
    ⟦ x F.⁻¹ ⟧ T.• ⟦ x ⟧ ≈⟨ T.sym (•-homo (x F.⁻¹) x) ⟩
    ⟦ x F.⁻¹ F.• x ⟧      ≈⟨ ⟦⟧-cong (F.inverseˡ x) ⟩
    ⟦ F.ε ⟧               ≈⟨ ε-homo ⟩
    T.ε ∎
```

1

# Example: Isabelle [2]

```
locale submonoid = ✐‹contributor ‹Martin Baillon››
  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"
```

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

# Example: Isabelle [2]

```
definition
  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)"
```

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

# Other Structures?

Signature, Sub-algebra, Product-algebra, Projection of product algebra, Congruence-relation on an algebra, Quotient algebra, Record definition of a theory, Homomorphism, Homomorphism-equality, Composition of morphisms, Kernel of Homomorphism, Isomorphism, Endomorphism, Automorphism, Closed term language, Open term language, Staged terms, Structural induction, Evaluation function for terms, Simplification of terms, Rewrite rules, Sub-terms of a term language, Equivalence of terms, Parse trees, Homomorphism between terms and trees.

# Other Structures?

Signature, Sub-algebra, Product-algebra, Projection of product algebra, Congruence-relation on an algebra, Quotient algebra, Record definition of a theory, Homomorphism, Homomorphism-equality, Composition of morphisms, Kernel of Homomorphism, Isomorphism, Endomorphism, Automorphism, Closed term language, Open term language, Staged terms, Structural induction, Evaluation function for terms, Simplification of terms, Rewrite rules, Sub-terms of a term language, Equivalence of terms, Parse trees, Homomorphism between terms and trees.

Automatically Generate Useful Constructions

# Why is that Useful?

- Avoid Boilerplate
  - Distracts user from original task
  - Error-prone and not reusable
  - Does not communicate information about the structure
- Reduce the amount of labor associated with building rich libraries

# Algebraic Theories

- well-typed dependent record. It contains
  - sorts
  - typed symbols
  - axioms

$$\frac{}{\varnothing \ \mathtt{ctx}} \qquad \frac{\Gamma \ \mathtt{ctx} \qquad x \notin |\Gamma| \qquad \Gamma \vdash \sigma : \kappa : \Box}{(\Gamma \ ; \ x : \sigma) \ \mathtt{ctx}}$$

# Algebraic Theories

- well-typed dependent record. It contains
  - sorts
  - typed symbols
  - axioms

$$\frac{}{\varnothing\ \texttt{ctx}} \qquad \frac{\Gamma\ \texttt{ctx} \qquad x \notin |\Gamma| \qquad \Gamma \vdash \sigma : \kappa : \Box}{(\Gamma\ ;\ x : \sigma)\ \texttt{ctx}}$$

- How are they presented in formal systems?

# Monoid Theory

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
```

# Monoid Theory

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 :
    (dot : 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
       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 _+_)

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
```

Abstract Over Details of Theory Presentations

1

# Research Objectives

- Abstracting over representation language
  - Generate Useful Algebraic Structures
- Specializing to different languages
  - Different Syntax
  - Different Foundation
  - Example: Isabelle/HOL and Agda

# Abstraction: Use MMT

We use MMT as a framework to abstract over syntax of theory presentations

- Platform Independent
- Simple Module System
- Example:

```
theory Magma : FOL =
  U : type
  op : U → U → U # 1 ∘ 2
theoy AdditiveMagma : FOL =
  U : type
  + : U → U → U
view additive : Magma → AdditiveMagma =
   U := U
   op := +
```

# Translating Definitions: Domain Analysis

- Domain Scoping

| | | | |
|---|---|---|---|
| MathScheme | MMT | Agda | Lean |
| Coq | Haskell | Isabelle | Idris |

- Data Collection
  - Foundations
  - Module System
  - Supported Features
- Data Analysis
  - Commonlities, Differences and Dependencies

Result: Feature Model

# Feature Model

Consists of

- Feature Diagram: Hierarchies and Dependencies
  - Mandatory, Alternative or Optional
- Feature Definition
- Composition Rules
  - Valid/Invalid Feature Combinations
- Rationale
  - Reasons for choosing / not choosing a feature

# Evaluation: MathScheme Library

```
theory Carrier := Empty extended_by {U : type}
theory Magma := Carrier extended_by {_∘_ : U → U → U}
theory Pointed := Carrier extended_by {e : U}
theory AddtivieMagma := Magma rename {∘ := +}
```

$$\begin{tikzcd}
\mathit{Carrier} \arrow[r, hook] \arrow[d, hook] & \mathit{Pointed} \\
\mathit{Magma} \arrow[d, mapsto] & \\
\mathit{AdditiveMagma} &
\end{tikzcd}$$

# Evaluation: MathScheme Library

```
theory Carrier := Empty extended_by {U : type}
theory Magma := Carrier extended_by {_∘_ : U → U → U}
theory Pointed := Carrier extended_by {e : U}
theory AddtivieMagma := Magma rename {∘ := +}
theory PointedMagma := Combine Pointed {} Magma {}
```

$$
\begin{tikzcd}
\mathit{Carrier} \arrow[r, hook] \arrow[d, hook] \arrow[dr, hook] & \mathit{Pointed} \arrow[d, hook] \\
\mathit{Magma} \arrow[r, hook] \arrow[d, mapsto] & \mathit{PointedMagma} \\
\mathit{AdditiveMagma} &
\end{tikzcd}
$$

# Evaluation: MathScheme Library

```
theory Carrier := Empty extended_by {U : type}
theory Magma := Carrier extended_by {_∘_ : U → U → U}
theory Pointed := Carrier extended_by {e : U}
theory PointedMagma := Combine Pointed {} Magma {}
theory AddtivieMagma := Magma rename {∘ := +}
LeftUnital := PointedMagma extended_by {left_unital : ... }
view Flip : Magma -> Magma :=
  U = U
  * = [a , b : U] b * a
RightUnital := Mixin LeftUnital {} Flip {}
```

$$
\begin{tikzcd}
\mathit{Carrier} \arrow[r, hook] \arrow[d, hook] \arrow[dr, hook] & \mathit{Pointed} \arrow[d, hook] & \\
\mathit{Magma} \arrow[r, hook] \arrow[d, mapsto] & \mathit{PointedMagma} \arrow[d, hook] \arrow[dr, hook] & \\
\mathit{AdditiveMagma} & \mathit{LeftUnital} \arrow[r, "\mathsf{Flip}"] & \mathit{RightUnital}
\end{tikzcd}
$$

# Conclusion

- A declarative language to build highly structured libraries with less human effort
- A toolbox for making formalization tasks much easier