

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  
  e : U  
  bin-op : U → U → U  
}
```

Defining Homomorphism

```
theory Foo {  
  U : type  
  e : U  
  bin-op : U → U → U  
}  
  
theory FooHom{  
  f1 , f2 : Foo  
  h : f1.U → f2.U  
  pres-e : h f1.e = f2.e  
  pres-bin-op :  
    h (f1.bin-op x y)  
    = f2.bin-op (h x) (h y)  
}
```

Defining Homomorphism

```
theory Foo {  
  U : type  
  e : U  
  bin-op : U → U → U  
}
```

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

```
theory Bar {  
  U : type  
  e : U  
  un-op : U → U  
}
```

Defining Homomorphism

```
theory Foo {  
  U : type  
  e : U  
  bin-op : U → U → U  
}
```

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

```
theory Bar {  
  U : type  
  e : U  
  un-op : U → U  
}
```

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

Real Example: Agda Library ¹

```
Homomorphic₀ : Morphism → From → To → Set _
```

```
Homomorphic₀ [f] • ° = [f •] ≈ °
```

```
Homomorphic₁ : Morphism → Fun₁ From → Op₁ To → Set _
```

```
Homomorphic₁ [f] • _ ° = ∀ x → [f • x] ≈ (° [f x])
```

```
Homomorphic₂ : Morphism → Fun₂ From → Op₂ To → Set _
```

```
Homomorphic₂ [f] _•_ °_ =
```

```
  ∀ x y → [f x • y] ≈ ([f x] ° [f y])
```

```
record IsMonoidMorphism ([f] : Morphism) :
```

```
  Set (C₁ ⊔ ℓ₁ ⊔ C₂ ⊔ ℓ₂) where
```

```
  field
```

```
    sm-homo : IsSemigroupMorphism F.semigroup T.semigroup [f]
```

```
    ε-homo  : Homomorphic₀ [f] F.ε T.ε
```

```
record IsCommutativeMonoidMorphism ([f] : Morphism) :
```

```
  Set (C₁ ⊔ ℓ₁ ⊔ C₂ ⊔ ℓ₂) where
```

```
  field
```

```
    mn-homo : IsMonoidMorphism F.monoid T.monoid [f]
```

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.unique1--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.inverse1 x) )
```

```
  [ F.ε ] ≈ ( ε-homo )
```

```
  T.ε ■
```

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 :: "_  $\Rightarrow$  _  $\Rightarrow$  ('a  $\times$  'b) monoid" (infixr " $\times\times$ " 80) where  
"G  $\times\times$  H =  
  ( $\lambda$ (carrier = carrier G  $\times$  carrier H,  
    mult = ( $\lambda$ (g, h) (g', h'). (g  $\otimes_G$  g', h  $\otimes_H$  h')),  
    one = (1_G, 1_H))"
```

lemma DirProd_monoid:

```
  assumes "monoid G" and "monoid H"  
  shows "monoid (G  $\times\times$  H)"
```

lemma DirProd_group:

```
  assumes "group G" and "group H"  
  shows "group (G  $\times\times$  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

Defining Homomorphism

```
theory Foo {  
  U : type  
  e : U  
  bin_op : U → U → U  
} generates Hom
```

```
theory Bar {  
  U : type  
  e : U  
  un_op : U → U  
} generates Hom
```

Why is that Useful?

Why is that Useful?

- Avoid Boilerplate
 - ▶ Distracting
 - ▶ Error-prone
 - ▶ Not reusable
 - ▶ Not useful

Why is that Useful?

- Avoid Boilerplate
 - ▶ Distracting
 - ▶ Error-prone
 - ▶ Not reusable
 - ▶ Not useful
- More rich libraries with less labor

Algebraic Theories

Algebraic Theories

- Well-typed dependent record. It contains
 - ▶ sorts
 - ▶ typed symbols
 - ▶ axioms

$$\frac{}{\emptyset \text{ ctx}} \quad \frac{\Gamma \text{ ctx} \quad x \notin |\Gamma| \quad \Gamma \vdash \sigma : \kappa : \square}{(\Gamma ; x : \sigma) \text{ ctx}}$$

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
}.
```

Haskell

```
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
      •
    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));
}
```

Haskell

```
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
```

Agda

```
data Monoid (A : Set)
  (Eq : Equivalence A) :
  Set

where
  monoid :
  (z : A)
  (l_ : A -> A -> A)
  (left_Id : LeftIdentity Eq z
    l_)
  (right_Id : RightIdentity Eq
    z l_)
  (assoc : Associative Eq l_)
```

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_0*_1y" := (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
}
```

Abstract Over Details of Theory Presentations

```

  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
```

```
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 (ε : Op₂) (ε : A)
  : Set (a ⊔ ℓ) where
  field
    isSemigroup : isSemigroup
    •
    identity : Identity ε •
    •
    identity' : LeftIdentity ε
    •
    identity' = proj₁ identity
    •
    identity' : RightIdentity
    ε •
    identity' = proj₂ identity
```

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

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 o 2  
theory 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

Translating Definitions: Domain Analysis

- Domain Scoping

MathScheme

MMT

Agda

Lean

Coq

Haskell

Isabelle

Idris

- Data Collection

- ▶ Foundations
- ▶ Module System
- ▶ Supported Features

Translating Definitions: Domain Analysis

- Domain Scoping

MathScheme

MMT

Agda

Lean

Coq

Haskell

Isabelle

Idris

- Data Collection

- ▶ Foundations
- ▶ Module System
- ▶ Supported Features

- Data Analysis

- ▶ Commonalties, Differences and Dependencies

Translating Definitions: Domain Analysis

- Domain Scoping

MathScheme

MMT

Agda

Lean

Coq

Haskell

Isabelle

Idris

- Data Collection

- ▶ Foundations
- ▶ Module System
- ▶ Supported Features

- Data Analysis

- ▶ Commonalties, Differences and Dependencies

Result: Feature Model

Feature Model

Consists of

- Feature Diagram: Hierarchies and Dependencies
 - ▶ Mandatory, Alternative or Optional

Feature Model

Consists of

- Feature Diagram: Hierarchies and Dependencies
 - ▶ Mandatory, Alternative or Optional
- Feature Definition

Feature Model

Consists of

- Feature Diagram: Hierarchies and Dependencies
 - ▶ Mandatory, Alternative or Optional
- Feature Definition
- Composition Rules
 - ▶ Valid/Invalid Feature Combinations

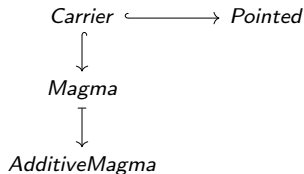
Feature Model

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

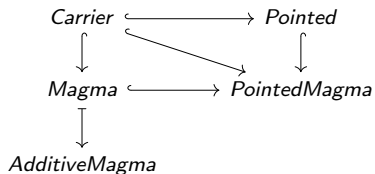
Evaluation: MathScheme Library

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



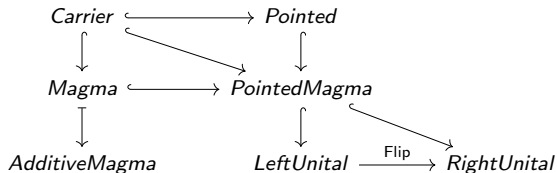
Evaluation: MathScheme Library

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



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 AdditiveMagma := Magma rename {∘ := +}
LeftUnital := PointedMagma extended_by {left_unital : ... }
view Flip : Magma -> Magma :=
  U = U
  * = [a , b : U] b * a
RightUnital := Mixin LeftUnital {} Flip {}
```



Timeline

- (May - Sept 2019)
 - ▶ Finalize MathScheme Library Translation
- (October - Dec 2019)
 - ▶ Automatic generation of constructions from abstract representation of a theory
- (Jan - April 2020)
 - ▶ Feature Modelling of Formal Systems.
 - ▶ Implementing the code generators

Conclusion

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