Metaprogramming Agda

Jacques Carette, Musa Al-hassy, Yasmine Sharoda, Wolfram Kahl

McMaster University

April 29, 2019

The Problem

What information can we generate from this *theory presentation*?

```
record Monoid : Set₁ where
  field
    -- a type or sort
    Carrier  : Set₀

    -- some operations
    Id       : Carrier
    _⨾_      : Carrier → Carrier → Carrier

    -- some equations
    left-unit : ∀ {x}   → Id ⨾ x ≡ x
    right-unit : ∀ {x} → x ⨾ Id ≡ x
    assoc : ∀ {x y z} → (x ⨾ y) ⨾ z ≡ x ⨾ (y ⨾ z)
```

The Problem

What information can we generate from this *theory presentation*?

```
record Monoid : Set₁ where
  field
    -- a type or sort
    Carrier   : Set₀

    -- some operations
    Id        : Carrier
    _⨾_       : Carrier → Carrier → Carrier

    -- some equations
    left-unit : ∀ {x}   → Id ⨾ x ≡ x
    right-unit : ∀ {x} → x ⨾ Id ≡ x
    assoc : ∀ {x y z} → (x ⨾ y) ⨾ z ≡ x ⨾ (y ⨾ z)
```

```
record Hom′ (A B : Monoid) : Set₁ where
  open Monoid A renaming
    (Carrier to Carrier₁; Id to Id₁; _⨾_ to _⨾₁_)
  open Monoid B renaming
    (Carrier to Carrier₂; Id to Id₂; _⨾_ to _⨾₂_)
  field
    mor    : Carrier₁ → Carrier₂
    pres-Id : mor Id₁ ≡ Id₂
    pres-⨾ : ∀ x y → mor (x ⨾₁ y) ≡ (mor x) ⨾₂ (mor y)
```

The Problem

What information can we generate from this *theory presentation*?

```
record Monoid : Set₁ where
  field
    -- a type or sort
    Carrier  : Set₀

    -- some operations
    Id       : Carrier
    _⨾_      : Carrier → Carrier → Carrier

    -- some equations
    left-unit : ∀ {x}   → Id ⨾ x ≡ x
    right-unit : ∀ {x} → x ⨾ Id ≡ x
    assoc : ∀ {x y z} → (x ⨾ y) ⨾ z ≡ x ⨾ (y ⨾ z)
```

```
module Monoid₁ (M : Monoid) where
  open Monoid M public renaming
    ( Carrier to Carrier₁; Id to Id₁; _⨾_ to _⨾₁_; left-unit to left-unit₁

module Monoid₂ (M : Monoid) where
  open Monoid M public renaming
    ( Carrier to Carrier₂; Id to Id₂; _⨾_ to _⨾₂_; left-unit to left-unit₂
```

The Problem

What information can we generate from this *theory presentation*?

```
record Monoid : Set₁ where
  field
    -- a type or sort
    Carrier  : Set₀

    -- some operations
    Id       : Carrier
    _⨾_      : Carrier → Carrier → Carrier

    -- some equations
    left-unit : ∀ {x}   → Id ⨾ x ≡ x
    right-unit : ∀ {x} → x ⨾ Id ≡ x
    assoc : ∀ {x y z} → (x ⨾ y) ⨾ z ≡ x ⨾ (y ⨾ z)
```

```
record Hom (A B : Monoid) : Set₁ where
  open Monoid₁ A; open Monoid₂ B
  field
    mor    : Carrier₁ → Carrier₂
    pres-Id : mor Id₁ ≡ Id₂
    pres-⨾ : ∀ x y → mor (x ⨾₁ y) ≡ (mor x) ⨾₂ (mor y)
```

The Problem

What information can we generate from this *theory presentation*?

```
record Monoid : Set₁ where
  field
    -- a type or sort
    Carrier  : Set₀

    -- some operations
    Id       : Carrier
    _⨾_      : Carrier → Carrier → Carrier

    -- some equations
    left-unit : ∀ {x}   → Id ⨾ x ≡ x
    right-unit : ∀ {x} → x ⨾ Id ≡ x
    assoc : ∀ {x y z} → (x ⨾ y) ⨾ z ≡ x ⨾ (y ⨾ z)
```

```
record Hom (A B : Monoid) : Set₁ where
  open Monoid₁ A; open Monoid₂ B
  field
    mor    : Carrier₁ → Carrier₂
    pres-Id : mor Id₁ ≡ Id₂
    pres-⨾ : ∀ x y → mor (x ⨾₁ y) ≡ (mor x) ⨾₂ (mor y)
```

```
-- ‘‘Apply’’ a homomorphism onto an element
infixr 20 _$_
_$_ : {A B : Monoid} → Hom A B →
      (Monoid.Carrier A → Monoid.Carrier B)
_$_ = Hom.mor
```

The Problem

What information can we generate from this *theory presentation*?

```
record Monoid : Set₁ where
  field
    -- a type or sort
    Carrier   : Set₀


    -- some operations
    Id        : Carrier
    _⨾_       : Carrier → Carrier → Carrier


    -- some equations
    left-unit : ∀ {x}   → Id ⨾ x ≡ x
    right-unit : ∀ {x} → x ⨾ Id ≡ x
    assoc : ∀ {x y z} → (x ⨾ y) ⨾ z ≡ x ⨾ (y ⨾ z)
```

```
record Hom (A B : Monoid) : Set₁ where
  open Monoid₁ A; open Monoid₂ B
  field
    mor    : Carrier₁ → Carrier₂
    pres-Id : mor Id₁ ≡ Id₂
    pres-⨾ : ∀ x y → mor (x ⨾₁ y) ≡ (mor x) ⨾₂ (mor y)
```

```
record Signature : Set₁ where
  field
    Carrier : Set₀
    Id      : Carrier
    _⨾_     : Carrier → Carrier → Carrier
```

The Problem

What information can we generate from this *theory presentation*?

```
record Monoid : Set₁ where
  field
    -- a type or sort
    Carrier  : Set₀

    -- some operations
    Id       : Carrier
    _⨾_      : Carrier → Carrier → Carrier

    -- some equations
    left-unit : ∀ {x}   → Id ⨾ x ≡ x
    right-unit : ∀ {x} → x ⨾ Id ≡ x
    assoc : ∀ {x y z} → (x ⨾ y) ⨾ z ≡ x ⨾ (y ⨾ z)
```

```
record Hom (A B : Monoid) : Set₁ where
  open Monoid₁ A; open Monoid₂ B
  field
    mor    : Carrier₁ → Carrier₂
    pres-Id : mor Id₁ ≡ Id₂
    pres-⨾ : ∀ x y → mor (x ⨾₁ y) ≡ (mor x) ⨾₂ (mor y)
```

```
_∼_ : {A B : Set} (f g : A → B) → Set
f ∼ g = ∀ a → f a ≡ g a
```

```
record Hom-Equality {A B : Monoid} (F G : Hom A B) : Set where
  field
    equal : Hom.mor F ∼ Hom.mor G
```

```
_≈_ = Hom-Equality
```

The Problem

What information can we generate from this *theory presentation*?

```
record Monoid : Set₁ where
  field
    -- a type or sort
    Carrier  : Set₀


    -- some operations
    Id       : Carrier
    _⨾_      : Carrier → Carrier → Carrier


    -- some equations
    left-unit : ∀ {x}   → Id ⨾ x ≡ x
    right-unit : ∀ {x} → x ⨾ Id ≡ x
    assoc : ∀ {x y z} → (x ⨾ y) ⨾ z ≡ x ⨾ (y ⨾ z)
```

```
record Hom (A B : Monoid) : Set₁ where
  open Monoid₁ A; open Monoid₂ B
  field
    mor    : Carrier₁ → Carrier₂
    pres-Id : mor Id₁ ≡ Id₂
    pres-⨾ : ∀ x y → mor (x ⨾₁ y) ≡ (mor x) ⨾₂ (mor y)
```

```
_∼_ : {A B : Set} (f g : A → B) → Set
f ∼ g = ∀ a → f a ≡ g a
```

```
record Hom-Equality {A B : Monoid} (F G : Hom A B) : Set where
  field
    equal : Hom.mor F ∼ Hom.mor G
```

```
_≈_ = Hom-Equality
```

```
Hom-Equality′ : ∀ {A B : Monoid} (F G : Hom A B) → Set
Hom-Equality′ F G = Hom.mor F ∼ Hom.mor G
```

Combinators for (presentations of) theories

Extension:

```
CommutativeMonoid := Monoid extended by {
  axiom commutative_* : forall x,y,z:U. x*y=y*x}
```

Renaming:

```
AdditiveMonoid := Monoid[ * |-> +, e |-> 0 ]
```

Combination:

```
AdditiveCommutativeMonoid :=
  combine AdditiveMonoid, CommutativeMonoid over Monoid
```

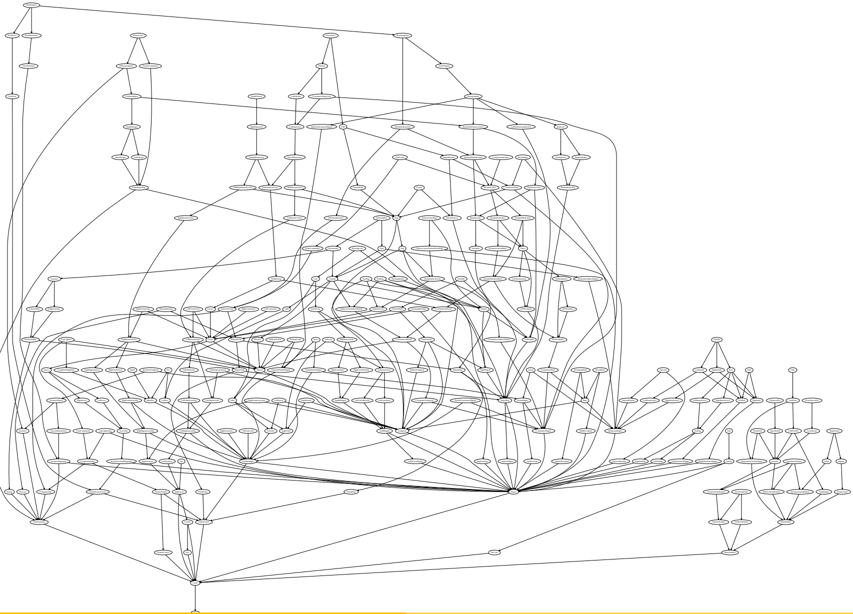
Library fragment 1

```
MoufangLoop := combine Loop, MoufangIdentity over Magma
LeftShelfSig := Magma[ * |-> |> ]
LeftShelf := LeftDistributiveMagma [ * |-> |> ]
RightShelfSig := Magma[ * |-> <| ]
RightShelf := RightDistributiveMagma[ * |-> <| ]
RackSig := combine LeftShelfSig, RightShelfSig over Carrier
Shelf := combine LeftShelf, RightShelf over RackSig
LeftBinaryInverse := RackSig extended by {
    axiom leftInverse_|>_<| : forall x,y:U. (x |> y) <| x = y }
RightBinaryInverse := RackSig extended by {
    axiom rightInverse_|>_<| : forall x,y:U. x |> (y <| x) = y }
Rack := combine RightShelf, LeftShelf, LeftBinaryInverse,
    RightBinaryInverse over RackSig
LeftIdempotence := IdempotentMagma[ * |-> |> ]
RightIdempotence := IdempotentMagma[ * |-> <| ]
LeftSpindle := combine LeftShelf, LeftIdempotence over LeftShelfSig
RightSpindle := combine RightShelf, RightIdempotence over RightShelfSig
Quandle := combine Rack, LeftSpindle, RightSpindle over Shelf
```

Library fragment 2

```
NearSemiring := combine AdditiveSemigroup, Semigroup, RightRingoid over
NearSemifield := combine NearSemiring, Group over Semigroup
Semifield := combine NearSemifield, LeftRingoid over RingoidSig
NearRing := combine AdditiveGroup, Semigroup, RightRingoid over Ringoi
Rng := combine AbelianAdditiveGroup, Semigroup, Ringoid over RingoidSi
Semiring := combine AdditiveCommutativeMonoid, Monoid1, Ringoid, Left0
SemiRng := combine AdditiveCommutativeMonoid, Semigroup, Ringoid over
Dioid := combine Semiring, IdempotentAdditiveMagma over AdditiveMagma
Ring := combine Rng, Semiring over SemiRng
CommutativeRing := combine Ring, CommutativeMagma over Magma
BooleanRing := combine CommutativeRing, IdempotentMagma over Magma
```

A fraction of the Algebraic Zoo



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
```

Isabelle

```
class semigroup =
 fixes mult :: α => α => α
      (infixl ⊗ 70)
 assumes assoc :: (x ⊗ y) ⊗ z
         = x ⊗ (y ⊗ z)
class monoidl = semigroup +
 fixes neutral :: α (1)
 assumes neutl : 1 ⊗ x = x
class monoid = monoidl +
 assumes x ⊗ 1 = x
```

Lean

```
universe u
variables{ α : Type u }
class monoid (α : Type u) extends
  semigroup α, has_one α :=
  (one_mul : ∀ a : α, 1 * a = a)
  (mul_one : ∀ a : α, a * 1 = a)
```

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

```
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
```

Generating more structures

```
record Isomorphism (A B : Monoid) : Set₁ where
  open Monoid; open Hom
  field
    A⇒B : Hom A B
    g : Carrier B → Carrier A
    f∘g≡id : (mor A⇒B ∘ g) ~ id
    g∘f≡id : (g ∘ mor A⇒B) ~ id


  inv-is-Hom : Hom B A
  inv-is-Hom = record
    { mor = g
    ; pres-Id = trans (sym (cong g (pres-Id A⇒B))) (g∘f≡id (Id A))
    ; pres-⨾ = λ x y → trans (cong g (sym (cong₂ (_⨾_ B) (f∘g≡id x) (f∘g≡id y))))
                 (trans (cong g (sym (pres-⨾ A⇒B (g x) (g y)))) (g∘f≡id _))
    }
```

Generating more structures

```
record Isomorphism (A B : Monoid) : Set₁ where
  open Monoid; open Hom
  field
    A⇒B : Hom A B
    g : Carrier B → Carrier A
    f∘g≡id : (mor A⇒B ∘ g) ~ id
    g∘f≡id : (g ∘ mor A⇒B) ~ id

  inv-is-Hom : Hom B A
  inv-is-Hom = record
    { mor = g
    ; pres-Id = trans (sym (cong g (pres-Id A⇒B))) (g∘f≡id (Id A))
    ; pres-⨾ = λ x y → trans (cong g (sym (cong₂ (_⨾_ B) (f∘g≡id x) (f∘g≡id y))))
                (trans (cong g (sym (pres-⨾ A⇒B (g x) (g y)))) (g∘f≡id _))
    }

Endomorphism : Monoid → Set₁
Endomorphism A = Hom A A

Automorphism : Monoid → Set₁
Automorphism A = Isomorphism A A
```

And more

```
record Kernel {A B : Monoid} (F : Hom A B) : Set₁ where
  open Monoid A
  field
    x : Carrier
    y : Carrier
    cond : F $ x ≡ F $ y
```

And more

```
record _×M_ (A B : Monoid) : Set₂ where
  field
    -- There is an object:
    ProdM : Monoid
    -- Along with two maps to the orginal arguments:
    Proj1 : Hom ProdM A
    Proj2 : Hom ProdM B
```

And more

```
record _×M_ (A B : Monoid) : Set₂ where
  field
    -- There is an object:
    ProdM : Monoid
    -- Along with two maps to the orginal arguments:
    Proj1 : Hom ProdM A
    Proj2 : Hom ProdM B


Make-Cartesian-Product : (A : Monoid) → (B : Monoid) → A ×M B
Make-Cartesian-Product A B =
  let open Monoid₁ A ; open Monoid₂ B in record
  { ProdM = record
                { Carrier   = Carrier₁ × Carrier₂
                ; Id        = Id₁ , Id₂
                ; _⨾_       = zip _⨾₁_ _⨾₂_
                ; left-unit = cong₂ _,_ left-unit₁ left-unit₂
                ; right-unit = cong₂ _,_ right-unit₁ right-unit₂
                ; assoc     = cong₂ _,_ assoc₁ assoc₂
                }
  ; Proj1 = record { mor = proj₁ ; pres-Id = refl ; pres-⨾ = λ _ _ → refl }
  ; Proj2 = record { mor = proj₂ ; pres-Id = refl ; pres-⨾ = λ _ _ → refl }
  }
```

And even more

```
record MonoidOn (Carrier : Set₀) : Set₀ where
  field
    Id         : Carrier
    _⨾_        : Carrier → Carrier → Carrier
    left-unit  : ∀ {x} → Id ⨾ x ≡ x
    right-unit : ∀ {x} → x ⨾ Id ≡ x
    assoc      : ∀ {x y z} → (x ⨾ y) ⨾ z ≡ x ⨾ (y ⨾ z)
```

And even more

```
record MonoidOn (Carrier : Set₀) : Set₀ where
  field
    Id         : Carrier
    _⨾_        : Carrier → Carrier → Carrier
    left-unit  : ∀ {x} → Id ⨾ x ≡ x
    right-unit : ∀ {x} → x ⨾ Id ≡ x
    assoc      : ∀ {x y z} → (x ⨾ y) ⨾ z ≡ x ⨾ (y ⨾ z)

module EasilyFormulated (S : Set) (A B : MonoidOn S) where
  open MonoidOn A renaming (Id to Id₁; _⨾_ to _⨾₁_; right-unit to right-unit₁)
  open MonoidOn B renaming (Id to Id₂; _⨾_ to _⨾₂_; left-unit to left-unit₂)


  claim : ∀ x → Id₂ ⨾₂ (x ⨾₁ Id₁) ≡ x
  claim x = trans left-unit₂ right-unit₁
```

And even more

```
record MonoidOn (Carrier : Set₀) : Set₀ where
  field
    Id         : Carrier
    _⨾_        : Carrier → Carrier → Carrier
    left-unit  : ∀ {x} → Id ⨾ x ≡ x
    right-unit : ∀ {x} → x ⨾ Id ≡ x
    assoc      : ∀ {x y z} → (x ⨾ y) ⨾ z ≡ x ⨾ (y ⨾ z)

module EasilyFormulated (S : Set) (A B : MonoidOn S) where
  open MonoidOn A renaming (Id to Id₁; _⨾_ to _⨾₁_; right-unit to right-unit₁)
  open MonoidOn B renaming (Id to Id₂; _⨾_ to _⨾₂_; left-unit to left-unit₂)

  claim : ∀ x → Id₂ ⨾₂ (x ⨾₁ Id₁) ≡ x
  claim x = trans left-unit₂ right-unit₁

module AkwardFormulation
  (A B : Monoid) (ceq : Monoid.Carrier A ≡ Monoid.Carrier B) where
  open Monoid₁ A; open Monoid₂ B

  coe : Carrier₁ → Carrier₂
  coe = subst id ceq

  claim : ∀ x → Id₂ ⨾₂ coe (x ⨾₁ Id₁) ≡ coe x
  claim x = trans left-unit₂ (cong coe right-unit₁)
```

more more

```
record IsMonoid {Carrier : Set}
                (_⨾_ : Carrier → Carrier → Carrier)
                (Id : Carrier) : Set where
  field
    left-unit  : ∀ {x} → Id ⨾ x ≡ x
    right-unit : ∀ {x} → x ⨾ Id ≡ x
    assoc      : ∀ {x y z} → (x ⨾ y) ⨾ z ≡ x ⨾ (y ⨾ z)
```

On to term algebras: closed

```
module Closed where
  data CTerm : Set where
    Id : CTerm
    _⨾_ : CTerm → CTerm → CTerm
```

On to term algebras: closed

```
module Closed where
  data CTerm : Set where
    Id : CTerm
    _⨾_ : CTerm → CTerm → CTerm


infix 999 _⟦_⟧
_⟦_⟧ : (ℳ : Monoid) → CTerm → Monoid.Carrier ℳ
ℳ ⟦ Id ⟧ = Monoid.Id ℳ
ℳ ⟦ x ⨾ y ⟧ = ℳ ⟦ x ⟧ ⨾₁ ℳ ⟦ y ⟧ where open Monoid₁ ℳ
```

On to term algebras: closed

```
module Closed where
  data CTerm : Set where
    Id : CTerm
    _⨾_ : CTerm → CTerm → CTerm

infix 999 _⟦_⟧
_⟦_⟧ : (𝓜 : Monoid) → CTerm → Monoid.Carrier 𝓜
𝓜 ⟦ Id ⟧ = Monoid.Id 𝓜
𝓜 ⟦ x ⨾ y ⟧ = 𝓜 ⟦ x ⟧ ⨾₁ 𝓜 ⟦ y ⟧ where open Monoid₁ 𝓜

length : CTerm → ℕ
length Id = 1
length (x ⨾ y) = 1 + length x + length y
--
data _≈_ : CTerm → CTerm → Set where
  ≈-Id : Id ≈ Id
  ≈-⨾ : ∀ {a a′ b b′} → a ≈ a′ → b ≈ b′ → (a ⨾ b) ≈ (a′ ⨾ b′)
```

On to term algebras: open

```
module Open where
  data OTerm (𝒱 : DecSetoid lzero lzero) : Set where
    Var : DecSetoid.Carrier 𝒱 → OTerm 𝒱
    Id : OTerm 𝒱
    _⨾_ : OTerm 𝒱 → OTerm 𝒱 → OTerm 𝒱
```

On to term algebras: open

```
module Open where
  data OTerm (𝒱 : DecSetoid lzero lzero) : Set where
    Var : DecSetoid.Carrier 𝒱 → OTerm 𝒱
    Id : OTerm 𝒱
    _⨾_ : OTerm 𝒱 → OTerm 𝒱 → OTerm 𝒱


module Interpret {𝒱 : DecSetoid lzero lzero} (A : Monoid) where
  open DecSetoid 𝒱 renaming (Carrier to V); open Monoid₁ A; open OTerm
  --
  ⟦_⟧_ : OTerm 𝒱 → (V → Carrier₁) → Carrier₁
  ⟦ Var x ⟧ σ = σ x
  ⟦ Id ⟧ σ = Id₁
  ⟦ l ⨾ r ⟧ σ = (⟦ l ⟧ σ) ⨾₁ (⟦ r ⟧ σ)
  --
  length : OTerm 𝒱 → ℕ
  length (Var _) = 1
  length Id = 1
  length (x ⨾ y) = 1 + length x + length y
```

On to term algebras: open

```
module Open where
  data OTerm (𝒱 : DecSetoid lzero lzero) : Set where
    Var : DecSetoid.Carrier 𝒱 → OTerm 𝒱
    Id : OTerm 𝒱
    _⨾_ : OTerm 𝒱 → OTerm 𝒱 → OTerm 𝒱

module Interpret {𝒱 : DecSetoid lzero lzero} (A : Monoid) where
  open DecSetoid 𝒱 renaming (Carrier to V); open Monoid₁ A; open OTerm
  --
  ⟦_⟧_ : OTerm 𝒱 → (V → Carrier₁) → Carrier₁
  ⟦ Var x ⟧ σ = σ x
  ⟦ Id ⟧ σ = Id₁
  ⟦ l ⨾ r ⟧ σ = (⟦ l ⟧ σ) ⨾₁ (⟦ r ⟧ σ)
  --
  length : OTerm 𝒱 → ℕ
  length (Var _) = 1
  length Id = 1
  length (x ⨾ y) = 1 + length x + length y
```

Formulas, predicates, quantifiers, etc

Induction

```
induction : (P : OTerm 𝒱 → Set)
  {- Base Cases -}
  → (∀ x → P (Var x))
  → P Id
  {- Inductive step -}
  → (∀ x y → P (x ⨾ y))
  {- Conclusion -}
  → ∀ (y : OTerm 𝒱) → P y
induction P vars empty ind (Var x) = vars x
induction P vars empty ind Id = empty
induction P vars empty ind (l ⨾ r) = ind l r
```

Towards Partial Evaluation

```
module Example (B : Monoid) where
  import Data.Char as C
  CharSetoid : DecSetoid lzero lzero
  CharSetoid = StrictTotalOrder.decSetoid C.strictTotalOrder
  open Interpret {CharSetoid} B
  OT = OTerm CharSetoid
  --
  left-unit-term : Formula
  left-unit-term = Id ⨾ Var 'x' ≃ Var 'x'
  assoc-term : Formula
  assoc-term = Var 'x' ⨾ (Var 'y' ⨾ Var 'z') ≃ (Var 'x' ⨾ Var 'y') ⨾ Var 'z'
```

```
reduces : Formula → Set
reduces F = length (lhs F) > length (rhs F)
```

Towards Partial Evaluation

```
module Example (B : Monoid) where
  import Data.Char as C
  CharSetoid : DecSetoid lzero lzero
  CharSetoid = StrictTotalOrder.decSetoid C.strictTotalOrder
  open Interpret {CharSetoid} B
  OT = OTerm CharSetoid
  --
  left-unit-term : Formula
  left-unit-term = Id ⨾ Var 'x' ≃ Var 'x'
  assoc-term : Formula
  assoc-term = Var 'x' ⨾ (Var 'y' ⨾ Var 'z') ≃ (Var 'x' ⨾ Var 'y') ⨾ Var 'z'
```

```
reduces : Formula → Set
reduces F = length (lhs F) > length (rhs F)
```

```
simp : OT → OT
simp (Var x)                = Var x
simp Id                     = Id
simp (Id ⨾ y)               = simp y {- Identity law -}
simp (Var x ⨾ y)            = Var x ⨾ simp y
simp (x@(_ ⨾ _) ⨾ Var y)    = simp x ⨾ Var y
simp (x@(_ ⨾ _) ⨾ Id)       = simp x {- Identity law -}
simp (x@(_ ⨾ _) ⨾ y@(_ ⨾ _)) = simp x ⨾ simp y
```

Towards Partial Evaluation

```
module Example (B : Monoid) where
  import Data.Char as C
  CharSetoid : DecSetoid lzero lzero
  CharSetoid = StrictTotalOrder.decSetoid C.strictTotalOrder
  open Interpret {CharSetoid} B
  OT = OTerm CharSetoid
  --
  left-unit-term : Formula
  left-unit-term = Id ⨾ Var 'x' ≃ Var 'x'
  assoc-term : Formula
  assoc-term = Var 'x' ⨾ (Var 'y' ⨾ Var 'z') ≃ (Var 'x' ⨾ Var 'y') ⨾ Var 'z'


reduces : Formula → Set
reduces F = length (lhs F) > length (rhs F)


open Monoid₂ B
coherence : ∀ x σ → ⟦ x ⟧ σ ≡ ⟦ simp x ⟧ σ
coherence (Var x) σ                   = refl
coherence Id σ                        = refl
coherence (Var x ⨾ x₁) σ              = cong (λ z → (σ x) ⨾₂ z) (coherence x₁ σ)
coherence (Id ⨾ x₁) σ                 = trans left-unit₂ (coherence x₁ σ)
coherence (x@(_ ⨾ _) ⨾ Var x₁) σ = cong (λ z → z ⨾₂ σ x₁) (coherence x σ)
coherence (x@(_ ⨾ _) ⨾ Id) σ      = trans right-unit₂ (coherence x σ)
coherence (x@(_ ⨾ _) ⨾ y@(_ ⨾ _)) σ = cong₂ _⨾₂_ (coherence x σ) (coherence y σ)
```

Universal Algebra...

Most of these work for Generalized Algebraic Theories (à la Cartmell):

- Signature
- Term Algebra
  - "generic functions" (à la *Scrap your Boilerplate*)
  - Structural induction
- Term Algebra parametrized by a "theory" of variables
  - predicate for ground terms
  - "simplifier" for open terms (correct but usually incomplete)
- Homomorphism; homomorphism composition; isomorphism
- kernel of homomorphism
- Theory of congruence relations over a theory
- Induced congruence of a homomorphism
- Interpreter from Term Algebra to any instance of a theory
- Partial evaluator
- Sub-theory, Product Theory, Co-product Theory