A Machine-checked Formal Proof of Birkhoff's Variety Theorem in Martin-Löf Type Theory

William DeMeo
https://williamdemeo.org

Jacques Carette
McMaster University

1 Introduction

The Agda Universal Algebra Library (agda-algebras) is a collection of types and programs (theorems and proofs) formalizing the foundations of universal algebra in dependent type theory using the Agda programming language and proof assistant. The agda-algebras library now includes a substantial collection of definitions, theorems, and proofs from universal algebra and equational logic and as such provides many examples that exhibit the power of inductive and dependent types for representing and reasoning about general algebraic and relational structures.

The first major milestone of the agda-algebras project is a new formal proof of *Birkhoff's variety theorem* (also known as the *HSP theorem*), the first version of which was completed in January of 2021. To the best of our knowledge, this was the first time Birkhoff's theorem had been formulated and proved in dependent type theory and verified with a proof assistant.

In this paper, we present a subset of the agda-algebras library that culminates in a complete, self-contained, formal proof of the HSP theorem. In the course of our exposition of the proof, we discuss some of the more challenging aspects of formalizing universal algebra in type theory and the issues that arise when attempting to constructively prove some of the basic results in that area. We demonstrate that dependent type theory and Agda, despite the demands they place on the user, are accessible to working mathematicians who have sufficient patience and a strong enough desire to constructively codify their work and formally verify the correctness of their results.

Our presentation may be viewed as a sobering glimpse at the painstaking process of doing mathematics in the languages of dependent types and Agda. Nonetheless we hope to make a compelling case for investing in these languages. Indeed, we are excited to share the gratifying rewards that come with some mastery of type theory and interactive theorem proving technologies.

2 Preliminaries

2.1 Logical foundations

An Agda program typically begins by setting some language options and by importing types from existing Agda libraries. The language options are specified using the OPTIONS *pragma* which affect control the way Agda behaves by controlling the deduction rules that are available to us and the logical axioms that are assumed when the program is type-checked by Agda to verify its correctness. Every Agda program in the agda-algebras library begins with the following line.

```
{-# OPTIONS –without-K –exact-split –safe #-}
```

These options control certain foundational assumptions that Agda makes when type-checking the program to verify its correctness.

- --without-K disables Streicher's K axiom ; see also the section on axiom K in the Agda Language Reference Manual.
- --exact-split makes Agda accept only those definitions that behave like so-called *judgmental* equalities. Martín Escardó explains this by saying it "makes sure that pattern matching corresponds to Martin-Löf eliminators;" see also the Pattern matching and equality section of the Agda Tools documentation.
- safe ensures that nothing is postulated outright—every non-MLTT axiom has to be an explicit assumption (e.g., an argument to a function or module); see also this section of the Agda Tools documentation and the Safe Agda section of the Agda Language Reference.

2.2 Agda Modules

The OPTIONS pragma is usually followed by the start of a module. Indeed, the HSP.lagda program that is subject of this paper begins with the following import directives, which import the parts of the Agda Standard Library that we will use in our program.

```
{-# OPTIONS –without-K –exact-split –safe #-}

open import Algebras.Basic using ( 𝓞 ; 𝓥 ; Signature )
module Demos.HSP {𝑆 : Signature 𝓞 𝓥} where
open import Agda.Primitive using ( _⊔_ ; lsuc ) renaming ( Set to Type )
open import Data.Product using ( Σ-syntax ; _×_ ; _,_ ; Σ ) renaming ( proj₁ to fst ; proj₂ to snd )
open import Function using ( id ; _∘_ ; flip ; Injection ; Surjection ) renaming ( Func to _⟶_ )
open import Level using ( Level )
open import Relation.Binary using ( Setoid ; Rel ; IsEquivalence )
open import Relation.Binary.Definitions using ( Reflexive ; Sym ; Trans ; Symmetric ; Transitive )
open import Relation.Unary using ( Pred ; _⊆_ ; _∈_ )
open import Relation.Binary.PropositionalEquality as ≡ using (_≡_)
import Function.Definitions                     as FD
import Relation.Binary.Reasoning.Setoid as SetoidReasoning
open _⟶_ using ( cong ) renaming ( f to _⟨$⟩_ )
open Setoid using ( Carrier ; isEquivalence )
private variable
  α ρᵃ β ρᵇ γ ρᶜ δ ρᵈ ρ χ ℓ : Level
  Γ Δ : Type χ
  f : fst 𝑆
```

2.3 Setoids

A *setoid* is a type packaged with an equivalence relation on the collection of inhabitants of that type. Setoids are useful for representing classical (set-theory-based) mathematics in a constructive, type-theoretic way because most mathematical structures are assumed to come equipped with some (often implicit) equivalence relation manifesting a notion of equality of elements, and therefore a type-theoretic representation of such a structure should also model its equality relation.

The agda-algebras library was first developed without the use of setoids, opting instead for specially constructed experimental quotient types. However, this approach resulted in code that was hard to comprehend and it became difficult to determine whether the resulting proofs were fully constructive. In particular, our initial proof of the Birkhoff variety theorem required postulating function extensionality, an axiom that is not provable in pure Martin-Löf type theory.[reference needed]

In contrast, our current approach using setoids makes the equality relation of a given type explicit and this transparency can make it easier to determine the correctness and constructivity of the proofs. Using setiods we need no additional axioms beyond Martin-Löf type theory; in particular, no function extensionality axioms are postulated in our current formalization of Birkhoff's variety theorem.

Since it plays such a central role in the present development, we reproduce in the appendix the definition of the Setoid type of the Agda Standard Library. In addition to Setoid, much of our code employs the standard library's Func record type which represents a function from one setoid to another and packages such a function with a proof (called cong) that the function respects the underlying setoid equalities. In the list of imports above we renamed Func to the more visually appealing long-arrow symbol ⟶, and we will refer to its inhabitants as "setoid functions" throughout the paper.

A special example of a setoid function is the identity function from a setoid to itself. We define it, along with a binary composition operation for setoid functions, _⟨∘⟩_, as follows.

```
id : {A : Setoid α ρᵃ} → A ⟶ A
id {A} = record { f = id ; cong = id }

_⟨∘⟩_ : {A : Setoid α ρᵃ} {B : Setoid β ρᵇ} {C : Setoid γ ρᶜ}
  → B ⟶ C → A ⟶ B → A ⟶ C

f ⟨∘⟩ g = record { f = (_⟨$⟩_ f) ∘ (_⟨$⟩_ g)
                 ; cong = (cong f) ∘ (cong g) }
```

2.4 Projection notation

The definition of Σ (and thus, of ×) includes the fields proj_1 and proj_2 representing the first and second projections out of the product. However, we prefer the shorter names fst and snd (hence renaming when importing above) Alternatively, we use |_| and ∥_∥, respectively, for the first and second projections out of a product. This syntax is defined as follows.

```
module _ {A : Type α }{B : A → Type β} where

  |_| : Σ[ x ∈ A ] B x → A
  |_| = fst

  ∥_∥ : (z : Σ[ a ∈ A ] B a) → B | z |
  ∥_∥ = snd
```

(Here we put the definitions inside an *anonymous module*, which starts with the module keyword followed by an underscore instead of a module name. The purpose is simply to move the postulated typing judgments—the "parameters" of the module, e.g., A : Type α—out of the way so they don't obfuscate the definitions inside the module.)

2.5 Inverses of setoid functions

We define a data type that represent the semantic concept of the *image* of a function (cf. the Overture.Func.Inverses module of the agda-algebras library).

```
module _ {𝐀 : Setoid α ρᵃ}{𝐁 : Setoid β ρᵇ} where
 open Setoid 𝐁 using ( _≈_ ; sym ) renaming ( Carrier to B )

 data Image_∋_ (f : 𝐀 ⟶ 𝐁) : B → Type (α ⊔ β ⊔ ρᵇ) where
  eq : {b : B} → ∀ a → b ≈ (f ⟨$⟩ a) → Image f ∋ b
 open Image_∋_
```

An inhabitant of Image f ∋ b is a dependent pair (a , p), where a : A and p : b ≈ f a is a proof that f maps a to b. Since the proof that b belongs to the image of f is always accompanied by a witness a : A, we can actually *compute* a (pseudo)inverse of f. For convenience, we define this inverse function, which we call Inv, and which takes an arbitrary b : B and a (witness, proof)-pair, (a , p) : Image f ∋ b, and returns the witness a.

```
 Inv : (f : 𝐀 ⟶ 𝐁){b : B} → Image f ∋ b → Carrier 𝐀
 Inv _ (eq a _) = a
```

Given a setoid function f, Inv f is the range-restricted right-inverse of f, which is easily proved as follows.

```
 InvIsInverseʳ : {f : 𝐀 ⟶ 𝐁}{b : B}(q : Image f ∋ b) → (f ⟨$⟩ (Inv f q)) ≈ b
 InvIsInverseʳ (eq _ p) = sym p
```

2.6 Injective setoid functions

We call a function f : 𝐀 ⟶ 𝐀 from one setoid 𝐀 = (A , $\approx_0$) to another 𝐁 = (B , $\approx_1$) *injective* provided ∀ a_0 a_1, if f ⟨$⟩ $a_0 \approx_1$ f ⟨$⟩ a_1, then $a_0 \approx_0 a_1$. A *surjective function* is such that for all b : B there exists a : A such that f ⟨$⟩ a $\approx_1$ b.

We codify these notions in Agda and prove some of their properties beginning inside the next module where we have set the stage by declaring two setoids 𝐀 and 𝐁, naming their equality relations, and making some definitions from the standard library available.

```
module _ {𝐀 : Setoid α ρᵃ}{𝐁 : Setoid β ρᵇ} where
 open Setoid 𝐀 using () renaming ( _≈_ to _≈₁_ ; Carrier to A )
 open Setoid 𝐁 using () renaming ( _≈_ to _≈₂_ ; Carrier to B )
 open Surjection renaming (f to _⟨$⟩_)
 open FD _≈₁_ _≈₂_
```

The Agda Standard Library defines the type Injective to represent injective functions on bare types and we use Injective to define the type IsInjective representing the property of being an injective setoid function (cf. the Overture.Func.Injective module of the agda-algebras library).

```
 IsInjective : (𝐀 ⟶ 𝐁) → Type (α ⊔ ρᵃ ⊔ ρᵇ)
 IsInjective f = Injective (_⟨$⟩_ f)
```

We define a surjective setoid function in Agda as follows (cf. the Overture.Func.Surjective module of the agda-algebras library).

```
 IsSurjective : (𝐀 ⟶ 𝐁) → Type (α ⊔ β ⊔ ρᵇ)
```

```
IsSurjective F = ∀ {y} → Image F ∋ y where open Image_∋_
```

With the next definition we represent a *right-inverse* of a surjective setoid function.

```
SurjInv : (f : 𝐀 ⟶ 𝐁) → IsSurjective f → B → A
SurjInv f fE b = Inv f (fE {b})
```

Proving that the composition of injective functions is again injective is simply a matter of composing the two assumed witnesses to injectivity.

Proving that surjectivity is preserved under composition is only slightly more involved.

```
module _ {𝐀 : Setoid α ρᵃ}{𝐁 : Setoid β ρᵇ}{𝐂 : Setoid γ ρᶜ} where

 ∘-injective : (f : 𝐀 ⟶ 𝐁)(g : 𝐁 ⟶ 𝐂)
  → IsInjective f → IsInjective g → IsInjective (g ⟨∘⟩ f)
 ∘-injective _ _ finj ginj = finj ∘ ginj


 ∘-IsSurjective : {f : 𝐀 ⟶ 𝐂}{g : 𝐂 ⟶ 𝐁} → IsSurjective f → IsSurjective g → IsSurjective (g ⟨∘⟩ f)
 ∘-IsSurjective {f}{g}fE gE {y} = Goal
  where
  mp : Image g ∋ y → Image g ⟨∘⟩ f ∋ y
  mp (eq c p) = η fE
   where
   open Setoid 𝐁 using ( trans )
   η : Image f ∋ c → Image g ⟨∘⟩ f ∋ y
   η (eq a q) = eq a (trans p (cong g q))

  Goal : Image g ⟨∘⟩ f ∋ y
  Goal = mp gE
```

2.7 Kernels

The *kernel* of a function f : A → B is defined informally by {(x , y) ∈ A × A : f x = f y}. This can be represented in Agda in a number of ways, but for our purposes it is most convenient to define the kernel as an inhabitant of a (unary) predicate over the square of the function's domain, as follows.

```
module _ {A : Type α}{B : Type β} where

 kernel : Rel B ρ → (A → B) → Pred (A × A) ρ
 kernel _≈_ f (x , y) = f x ≈ f y
```

The kernel of a function f : $A \longrightarrow B$ from a setoid A to a setoid B (with carriers A and B, respectively) is defined informally by {(x , y) ∈ A × A : f ⟨$⟩ x $\approx_2$ f ⟨$⟩ y} and may be formalized in Agda as follows.

```
module _ {A : Setoid α ρᵃ}{B : Setoid β ρᵇ} where
 open Setoid A using () renaming ( Carrier to A )

 ker : (A ⟶ B) → Pred (A × A) ρᵇ
```

```
ker g (x , y) = g ⟨$⟩ x ≈ g ⟨$⟩ y where open Setoid B using ( _≈_ )
```

3 Algebras

3.1 Basic definitions

Here we define algebras over a setoid, instead of a mere type with no equivalence on it.

First we define an operator that translates an ordinary signature into a signature over a setoid domain.

```
EqArgs : {𝑆 : Signature 𝓞 𝓥}{ξ : Setoid α ρᵃ}
  → ∀ {f g} → f ≡ g → (∥ 𝑆 ∥ f → Carrier ξ) → (∥ 𝑆 ∥ g → Carrier ξ) → Type (𝓥 ⊔ ρᵃ)

EqArgs {ξ = ξ} ≡.refl u v = ∀ i → u i ≈ v i
  where
  open Setoid ξ using ( _≈_ )

module _ where
  open Setoid using ( _≈_ )
  open IsEquivalence using ( refl ; sym ; trans )

  ⟨_⟩ : Signature 𝓞 𝓥 → Setoid α ρᵃ → Setoid _ _
  Carrier (⟨ 𝑆 ⟩ ξ) = Σ[ f ∈ ∣ 𝑆 ∣ ] ((∥ 𝑆 ∥ f) → ξ .Carrier)
  _≈_ (⟨ 𝑆 ⟩ ξ) (f , u) (g , v) = Σ[ eqv ∈ f ≡ g ] EqArgs{ξ = ξ} eqv u v
  refl  (isEquivalence (⟨ 𝑆 ⟩ ξ))                    = ≡.refl , λ _ → Setoid.refl ξ
  sym (isEquivalence (⟨ 𝑆 ⟩ ξ)) (≡.refl , g) = ≡.refl , λ i → Setoid.sym ξ (g i)
  trans (isEquivalence (⟨ 𝑆 ⟩ ξ)) (≡.refl , g)(≡.refl , h) = ≡.refl , λ i → Setoid.trans ξ (g i) (h i)
```

We represent an algebra using a record type with two fields: Domain is a setoid denoting the underlying *universe* of the algebra (informally, the set of elements of the algebra); Interp represents the *interpretation* in the algebra of each operation symbol of the given signature. The record type Func from the Agda Standard Library provides what we need for an operation on the domain setoid.

Let us present the definition of the Algebra type and then discuss the definition of the Func type that provides the interpretation of each operation symbol.

```
record Algebra α ρ : Type (𝓞 ⊔ 𝓥 ⊔ lsuc (α ⊔ ρ)) where
  field
    Domain : Setoid α ρ
    Interp : (⟨ 𝑆 ⟩ Domain) ⟶ Domain
  ≡→≈ : ∀{x}{y} → x ≡ y → (Setoid._≈_ Domain) x y
  ≡→≈ ≡.refl = Setoid.refl Domain
```

We have thus codified the concept of (universal) algebra as a record type with two fields

1. a function f : Carrier (⟨ 𝑆 ⟩ Domain) → Carrier Domain
2. a proof cong : f Preserves $\_\approx_1\_$ ⟶ $\_\approx_2\_$ that f preserves the underlying setoid equalities.

Comparing this with the definition of the Func (or _⟶_) type shown in the appendix, here A is Carrier (⟨ 𝑆 ⟩ Domain) and B is Carrier Domain. Thus Interp gives, for each operation symbol in the signature S, a setoid function f—namely, a function where the domain is a power of Domain and the codomain is Domain—along with a proof that all operations so interpreted respect the underlying setoid equality on Domain.

We define the following syntactic sugar: if $\mathbf{A}$ is an algebra, 𝔻[$\mathbf{A}$] gives the setoid Domain $\mathbf{A}$, while 𝕌[$\mathbf{A}$] exposes the underlying carrier or "universe" of the algebra $\mathbf{A}$; finally, f ^ $\mathbf{A}$ denotes the interpretation in the algebra $\mathbf{A}$ of the operation symbol f.

```
open Algebra

𝕌[_] : Algebra α ρᵃ → Type α
𝕌[ A ] = Carrier (Domain A)

𝔻[_] : Algebra α ρᵃ → Setoid α ρᵃ
𝔻[ A ] = Domain A

_^_ : (f : ∣ S ∣)(A : Algebra α ρᵃ) → (∥ S ∥ f → 𝕌[ A ]) → 𝕌[ A ]

f ^ A = λ a → (Interp A) ⟨$⟩ (f , a)
```

3.2 Universe lifting of algebra types

```
module _ (A : Algebra α ρᵃ) where
  open Algebra A using () renaming ( Domain to A ; Interp to InterpA )
  open Setoid A using (sym ; trans ) renaming ( Carrier to |A| ; _≈_ to _≈₁_ ; refl to refl₁ )
  open Level

  Lift-Algˡ : (ℓ : Level) → Algebra (α ⊔ ℓ) ρᵃ

  Domain (Lift-Algˡ ℓ) = record { Carrier = Lift ℓ |A|
                                ; _≈_ = λ x y → lower x ≈₁ lower y
                                ; isEquivalence = record { refl = refl₁
                                                         ; sym = sym
                                                         ; trans = trans }}

  Interp (Lift-Algˡ ℓ) ⟨$⟩ (f , la) = lift ((f ^ A) (lower ∘ la))
  cong (Interp (Lift-Algˡ ℓ)) (≡.refl , la=lb) = cong InterpA ((≡.refl , la=lb))


  Lift-Algʳ : (ℓ : Level) → Algebra α (ρᵃ ⊔ ℓ)

  Domain (Lift-Algʳ ℓ) =
    record { Carrier = |A|
           ; _≈_    = λ x y → Lift ℓ (x ≈₁ y)
           ; isEquivalence = record { refl = lift refl₁
                                    ; sym = λ x → lift (sym (lower x))
                                    ; trans = λ x y → lift (trans (lower x) (lower y)) }}

  Interp (Lift-Algʳ ℓ ) ⟨$⟩ (f , la) = (f ^ A) la
  cong (Interp (Lift-Algʳ ℓ)) (≡.refl , la≡lb) =
    lift (cong (Interp A) (≡.refl , λ i → lower (la≡lb i)))

Lift-Alg : (A : Algebra α ρᵃ)(ℓ₀ ℓ₁ : Level) → Algebra (α ⊔ ℓ₀) (ρᵃ ⊔ ℓ₁)
Lift-Alg A ℓ₀ ℓ₁ = Lift-Algʳ (Lift-Algˡ A ℓ₀) ℓ₁
```

3.3 Product Algebras

(cf. the Algebras.Func.Products module of the Agda Universal Algebra Library.)

```
module _ {ι : Level}{I : Type ι } where

 ⨅ : (𝒜 : I → Algebra α ρᵃ) → Algebra (α ⊔ ι) (ρᵃ ⊔ ι)

 Domain (⨅ 𝒜) =
  record { Carrier = ∀ i → 𝕌[ 𝒜 i ]
         ; _≈_ = λ a b → ∀ i → (Setoid._≈_ 𝔻[ 𝒜 i ]) (a i)(b i)
         ; isEquivalence =
             record { refl = λ i → IsEquivalence.refl    (isEquivalence 𝔻[ 𝒜 i ])
                    ; sym = λ x i → IsEquivalence.sym (isEquivalence 𝔻[ 𝒜 i ])(x i)
                    ; trans = λ x y i → IsEquivalence.trans (isEquivalence 𝔻[ 𝒜 i ])(x i)(y i) }}

 Interp (⨅ 𝒜) ⟨$⟩ (f , a) = λ i → (f ̂ (𝒜 i)) (flip a i)
 cong (Interp (⨅ 𝒜)) (≡.refl , f=g ) = λ i → cong (Interp (𝒜 i)) (≡.refl , flip f=g i )
```

4 Homomorphisms

4.1 Basic definitions

Here are some useful definitions and theorems extracted from the Homomorphisms.Func.Basic module of the Agda Universal Algebra Library.

```
module _ (𝐀 : Algebra α ρᵃ)(𝐁 : Algebra β ρᵇ) where
 open Algebra 𝐀 using () renaming (Domain to A )
 open Algebra 𝐁 using () renaming (Domain to B )
 open Setoid A using () renaming ( _≈_ to _≈₁_ )
 open Setoid B using () renaming ( _≈_ to _≈₂_ )

 compatible-map-op : (A ⟶ B) → ∣ 𝑆 ∣ → Type (𝓥 ⊔ α ⊔ ρᵇ)
 compatible-map-op h f = ∀ {a} → (h ⟨$⟩ ((f ̂ 𝐀) a)) ≈₂ ((f ̂ 𝐁) (λ x → (h ⟨$⟩ (a x))))

 compatible-map : (A ⟶ B) → Type (𝓞 ⊔ 𝓥 ⊔ α ⊔ ρᵇ)
 compatible-map h = ∀ {f} → compatible-map-op h f

 -- The property of being a homomorphism.
 record IsHom (h : A ⟶ B) : Type (𝓞 ⊔ 𝓥 ⊔ α ⊔ ρᵃ ⊔ ρᵇ) where
  field compatible : compatible-map h

 -- The type of homomorphisms.
 hom : Type (𝓞 ⊔ 𝓥 ⊔ α ⊔ ρᵃ ⊔ β ⊔ ρᵇ)
 hom = Σ (A ⟶ B) IsHom
```

4.2 Monomorphisms and epimorphisms

```
 record IsMon (h : A ⟶ B) : Type (𝓞 ⊔ 𝓥 ⊔ α ⊔ ρᵃ ⊔ β ⊔ ρᵇ) where
  field isHom : IsHom h ; isInjective : IsInjective h

  HomReduct : hom
```

```
    HomReduct = h , isHom

  mon : Type (𝓞 ⊔ 𝓥 ⊔ α ⊔ ρᵃ ⊔ β ⊔ ρᵇ)
  mon = Σ (A ⟶ B) IsMon

  record IsEpi (h : A ⟶ B) : Type (𝓞 ⊔ 𝓥 ⊔ α ⊔ ρᵃ ⊔ β ⊔ ρᵇ) where
    field isHom : IsHom h ; isSurjective : IsSurjective h

    HomReduct : hom
    HomReduct = h , isHom

  epi : Type (𝓞 ⊔ 𝓥 ⊔ α ⊔ ρᵃ ⊔ β ⊔ ρᵇ)
  epi = Σ (A ⟶ B) IsEpi

open IsHom ; open IsMon ; open IsEpi

module _ (𝐀 : Algebra α ρᵃ)(𝐁 : Algebra β ρᵇ) where

  mon→intohom : mon 𝐀 𝐁 → Σ[ h ∈ hom 𝐀 𝐁 ] IsInjective ∣ h ∣
  mon→intohom (hh , hhM) = (hh , isHom hhM) , isInjective hhM

  epi→ontohom : epi 𝐀 𝐁 → Σ[ h ∈ hom 𝐀 𝐁 ] IsSurjective ∣ h ∣
  epi→ontohom (hh , hhE) = (hh , isHom hhE) , isSurjective hhE
```

4.3 Basic properties of homomorphisms

Here are some definitions and theorems extracted from the Homomorphisms.Func.Properties module of the Agda Universal Algebra Library.

4.3.1 Composition of homomorphisms

The composition of homomorphisms is again a homomorphism. Similarly, the composition of epimorphisms is again an epimorphism.

```
module _ {𝐀 : Algebra α ρᵃ} {𝐁 : Algebra β ρᵇ} {𝐂 : Algebra γ ρᶜ}
         {g : 𝔻[ 𝐀 ] ⟶ 𝔻[ 𝐁 ]}{h : 𝔻[ 𝐁 ] ⟶ 𝔻[ 𝐂 ]} where
  open Setoid 𝔻[ 𝐂 ] using ( trans )

  ∘-is-hom : IsHom 𝐀 𝐁 g → IsHom 𝐁 𝐂 h → IsHom 𝐀 𝐂 (h ⟨∘⟩ g)
  ∘-is-hom ghom hhom = record { compatible = c }
    where
    c : compatible-map 𝐀 𝐂 (h ⟨∘⟩ g)
    c = trans (cong h (compatible ghom)) (compatible hhom)

  ∘-is-epi : IsEpi 𝐀 𝐁 g → IsEpi 𝐁 𝐂 h → IsEpi 𝐀 𝐂 (h ⟨∘⟩ g)
  ∘-is-epi gE hE =
    record { isHom = ∘-is-hom (isHom gE) (isHom hE)
           ; isSurjective = ∘-IsSurjective (isSurjective gE) (isSurjective hE) }

module _ {𝐀 : Algebra α ρᵃ} {𝐁 : Algebra β ρᵇ} {𝐂 : Algebra γ ρᶜ} where

  ∘-hom : hom 𝐀 𝐁 → hom 𝐁 𝐂 → hom 𝐀 𝐂
  ∘-hom (h , hhom) (g , ghom) = (g ⟨∘⟩ h) , ∘-is-hom hhom ghom

  ∘-epi : epi 𝐀 𝐁 → epi 𝐁 𝐂     → epi 𝐀 𝐂
```

```
∘-epi (h , hepi) (g , gepi) = (g ⟨∘⟩ h) , ∘-is-epi hepi gepi
```

4.3.2 Universe lifting of homomorphisms

First we define the identity homomorphism for setoid algebras and then we prove that the operations of lifting and lowering of a setoid algebra are homomorphisms.

```
𝒾𝒹 : {𝐀 : Algebra α ρᵃ} → hom 𝐀 𝐀
𝒾𝒹 {𝐀 = 𝐀} = 𝑖𝑑 , record { compatible = reflexive ≡.refl }
  where open Setoid ( Domain 𝐀 ) using ( reflexive )

module _ {𝐀 : Algebra α ρᵃ}{ℓ : Level} where
  open Setoid 𝔻[ 𝐀 ] using ( reflexive ) renaming ( _≈_ to _≈₁_ ; refl to refl₁ )
  open Setoid 𝔻[ Lift-Algˡ 𝐀 ℓ ] using () renaming ( _≈_ to _≈ˡ_ ; refl to reflˡ)
  open Setoid 𝔻[ Lift-Algʳ 𝐀 ℓ ] using () renaming ( _≈_ to _≈ʳ_ ; refl to reflʳ)
  open Level

  ToLiftˡ : hom 𝐀 (Lift-Algˡ 𝐀 ℓ)
  ToLiftˡ = record { f = lift ; cong = id } , record { compatible = reflexive ≡.refl }

  FromLiftˡ : hom (Lift-Algˡ 𝐀 ℓ) 𝐀
  FromLiftˡ = record { f = lower ; cong = id } , record { compatible = reflˡ }

  ToFromLiftˡ : ∀ b → (| ToLiftˡ | ⟨$⟩ (| FromLiftˡ | ⟨$⟩ b)) ≈ˡ b
  ToFromLiftˡ b = refl₁

  FromToLiftˡ : ∀ a → (| FromLiftˡ | ⟨$⟩ (| ToLiftˡ | ⟨$⟩ a)) ≈₁ a
  FromToLiftˡ a = refl₁

  ToLiftʳ : hom 𝐀 (Lift-Algʳ 𝐀 ℓ)
  ToLiftʳ = record { f = id ; cong = lift } , record { compatible = lift (reflexive ≡.refl) }

  FromLiftʳ : hom (Lift-Algʳ 𝐀 ℓ) 𝐀
  FromLiftʳ = record { f = id ; cong = lower } , record { compatible = reflˡ }

  ToFromLiftʳ : ∀ b → (| ToLiftʳ | ⟨$⟩ (| FromLiftʳ | ⟨$⟩ b)) ≈ʳ b
  ToFromLiftʳ b = lift refl₁

  FromToLiftʳ : ∀ a → (| FromLiftʳ | ⟨$⟩ (| ToLiftʳ | ⟨$⟩ a)) ≈₁ a
  FromToLiftʳ a = refl₁


module _ {𝐀 : Algebra α ρᵃ}{ℓ r : Level} where
  open Level
  open Setoid 𝔻[ 𝐀 ] using (refl)
  open Setoid 𝔻[ Lift-Alg 𝐀 ℓ r ] using ( _≈_ )

  ToLift : hom 𝐀 (Lift-Alg 𝐀 ℓ r)
  ToLift = ∘-hom ToLiftˡ ToLiftʳ

  FromLift : hom (Lift-Alg 𝐀 ℓ r) 𝐀
  FromLift = ∘-hom FromLiftʳ FromLiftˡ

  ToFromLift : ∀ b → (| ToLift | ⟨$⟩ (| FromLift | ⟨$⟩ b)) ≈ b
  ToFromLift b = lift refl

  ToLift-epi : epi 𝐀 (Lift-Alg 𝐀 ℓ r)
```

```
ToLift-epi = ∣ ToLift ∣ , (record { isHom = ∥ ToLift ∥
                                  ; isSurjective = λ {y} → eq (∣ FromLift ∣ ⟨$⟩ y) (ToFromLift y) })
```

4.4 Homomorphisms of product algebras

Suppose we have an algebra $\mathbf{A}$, a type $\mathsf{I} : \mathsf{Type}\ \mathscr{I}$, and a family $\mathscr{B} : \mathsf{I} \to \mathsf{Algebra}\ \beta\ S$ of algebras. We sometimes refer to the inhabitants of I as *indices*, and call $\mathscr{B}$ an *indexed family of algebras*. If in addition we have a family $\hbar : (\mathsf{i} : \mathsf{I}) \to \mathsf{hom}\ \mathbf{A}\ (\mathscr{B}\ \mathsf{i})$ of homomorphisms, then we can construct a homomorphism from $\mathbf{A}$ to the product $\prod \mathscr{B}$ in the natural way. Here is how we implement these notions in dependent type theory (cf. the [Homomorphisms.Func.Products][] module of the Agda Universal Algebra Library).

```
module _ {ι : Level}{I : Type ι}{A : Algebra α ρᵃ}(ℬ : I → Algebra β ρᵇ) where
 ⨅-hom-co : (∀(i : I) → hom A (ℬ i)) → hom A (⨅ ℬ)
 ⨅-hom-co 𝒽 = h , hhom
  where
  h : 𝔻[ A ] ⟶ 𝔻[ ⨅ ℬ ]
  _⟨$⟩_ h = λ a i → ∣ 𝒽 i ∣ ⟨$⟩ a
  cong h xy i = cong ∣ 𝒽 i ∣ xy
  hhom : IsHom A (⨅ ℬ) h
  compatible hhom = λ i → compatible ∥ 𝒽 i ∥
```

4.5 Factorization of homomorphisms

(cf. the Homomorphisms.Func.Factor module of the Agda Universal Algebra Library.)

If $\mathsf{g} : \mathsf{hom}\ \mathbf{A}\ \mathbf{B}$, $\mathsf{h} : \mathsf{hom}\ \mathbf{A}\ \mathbf{C}$, h is surjective, and $\mathsf{ker}\ \mathsf{h} \subseteq \mathsf{ker}\ \mathsf{g}$, then there exists $\varphi : \mathsf{hom}\ \mathbf{C}\ \mathbf{B}$ such that $\mathsf{g} = \varphi \circ \mathsf{h}$ so the following diagram commutes:

```
A --- h -->> C
 \          .
  \        .
   g     φ
    \   .
     \ .
      V
      B
```

We will prove this in case h is both surjective and injective.

```
module _ {A : Algebra α ρᵃ}(B : Algebra β ρᵇ){C : Algebra γ ρᶜ}
         (gh : hom A B)(hh : hom A C) where
 open Setoid 𝔻[ B ] using () renaming ( _≈_ to _≈₂_ ; sym to sym₂ )
 open Setoid 𝔻[ C ] using ( trans ) renaming ( _≈_ to _≈₃_ ; sym to sym₃ )
 open SetoidReasoning 𝔻[ B ]
 private
  gfunc = ∣ gh ∣ ; g = _⟨$⟩_ gfunc
  hfunc = ∣ hh ∣ ; h = _⟨$⟩_ hfunc

 HomFactor : kernel _≈₃_ h ⊆ kernel _≈₂_ g → IsSurjective hfunc
```

```
  → Σ[ φ ∈ hom 𝐂 𝐁 ] ∀ a → (g a) ≈₂ ∣ φ ∣ ⟨$⟩ (h a)

HomFactor Khg hE = (φmap , φhom) , gφh
  where
  kerpres : ∀ a₀ a₁ → h a₀ ≈₃ h a₁ → g a₀ ≈₂ g a₁
  kerpres a₀ a₁ hyp = Khg hyp

  h⁻¹ : 𝕌[ 𝐂 ] → 𝕌[ 𝐀 ]
  h⁻¹ = SurjInv hfunc hE

  η : ∀ {c} → h (h⁻¹ c) ≈₃ c
  η = InvIsInverseʳ hE

  ξ : ∀ {a} → h a ≈₃ h (h⁻¹ (h a))
  ξ = sym₃ η

  ζ : ∀{x y} → x ≈₃ y → h (h⁻¹ x) ≈₃ h (h⁻¹ y)
  ζ xy = trans η (trans xy (sym₃ η))


  φmap : 𝔻[ 𝐂 ] ⟶ 𝔻[ 𝐁 ]
  _⟨$⟩_ φmap = g ∘ h⁻¹
  cong φmap = Khg ∘ ζ

  gφh : (a : 𝕌[ 𝐀 ]) → g a ≈₂ φmap ⟨$⟩ (h a)
  gφh a = Khg ξ

  open _⟶_ φmap using () renaming (cong to φcong)

  φcomp : compatible-map 𝐂 𝐁 φmap
  φcomp {f}{c} =
    begin
      φmap ⟨$⟩ ((f ̂ 𝐂) c) ≈˘⟨ φcong (cong (Interp 𝐂) (≡.refl , (λ _ → η))) ⟩
      g (h⁻¹ ((f ̂ 𝐂)(h ∘ (h⁻¹ ∘ c)))) ≈˘⟨ φcong (compatible ∥ hh ∥) ⟩
      g (h⁻¹ (h ((f ̂ 𝐀)(h⁻¹ ∘ c)))) ≈˘⟨ gφh ((f ̂ 𝐀)(h⁻¹ ∘ c)) ⟩
      g ((f ̂ 𝐀)(h⁻¹ ∘ c))           ≈⟨ compatible ∥ gh ∥ ⟩
      (f ̂ 𝐁)(g ∘ (h⁻¹ ∘ c))          ∎

  φhom : IsHom 𝐂 𝐁 φmap
  compatible φhom = φcomp
```

4.6 Isomorphisms

(cf. the Homomorphisms.Func.Isomorphisms of the Agda Universal Algebra Library.)

Two structures are *isomorphic* provided there are homomorphisms going back and forth between them which compose to the identity map.

```
module _ (𝐀 : Algebra α ρᵃ) (𝐁 : Algebra β ρᵇ) where
 open Setoid 𝔻[ 𝐀 ] using ( sym ; trans ) renaming ( _≈_ to _≈₁_ )
 open Setoid 𝔻[ 𝐁 ] using () renaming ( _≈_ to _≈₂_ ; sym to sym₂ ; trans to trans₂)

 record _≅_ : Type (𝓞 ⊔ 𝓥 ⊔ α ⊔ β ⊔ ρᵃ ⊔ ρᵇ ) where
  constructor mkiso
  field
   to : hom 𝐀 𝐁
```

```
      from : hom B A
      to∼from : ∀ b → (| to | ⟨$⟩ (| from | ⟨$⟩ b)) ≈₂ b
      from∼to : ∀ a → (| from | ⟨$⟩ (| to | ⟨$⟩ a)) ≈₁ a

    toIsSurjective : IsSurjective | to |
    toIsSurjective {y} = eq (| from | ⟨$⟩ y) (sym₂ (to∼from y))

    toIsInjective : IsInjective | to |
    toIsInjective {x} {y} xy = Goal
      where
      ξ : | from | ⟨$⟩ (| to | ⟨$⟩ x) ≈₁ | from | ⟨$⟩ (| to | ⟨$⟩ y)
      ξ = cong | from | xy
      Goal : x ≈₁ y
      Goal = trans (sym (from∼to x)) (trans ξ (from∼to y))

    fromIsSurjective : IsSurjective | from |
    fromIsSurjective {y} = eq (| to | ⟨$⟩ y) (sym (from∼to y))

    fromIsInjective : IsInjective | from |
    fromIsInjective {x} {y} xy = Goal
      where
      ξ : | to | ⟨$⟩ (| from | ⟨$⟩ x) ≈₂ | to | ⟨$⟩ (| from | ⟨$⟩ y)
      ξ = cong | to | xy
      Goal : x ≈₂ y
      Goal = trans₂ (sym₂ (to∼from x)) (trans₂ ξ (to∼from y))

  open _≅_
```

4.6.1 Properties of isomorphisms

```
  ≅-refl : Reflexive (_≅_ {α}{ρᵃ})
  ≅-refl {α}{ρᵃ}{A} = mkiso 𝑖𝑑 𝑖𝑑 (λ b → refl) λ a → refl
    where open Setoid 𝔻[ A ] using ( refl )

  ≅-sym : Sym (_≅_{β}{ρᵇ}) (_≅_{α}{ρᵃ})
  ≅-sym φ = mkiso (from φ) (to φ) (from∼to φ) (to∼from φ)

  ≅-trans : Trans (_≅_ {α}{ρᵃ})(_≅_{β}{ρᵇ})(_≅_{α}{ρᵃ}{γ}{ρᶜ})
  ≅-trans {ρᶜ = ρᶜ}{A}{B}{C} ab bc = mkiso f g τ ν
    where
      open Setoid 𝔻[ A ] using () renaming ( _≈_ to _≈₁_ ; trans to trans₁ )
      open Setoid 𝔻[ C ] using () renaming ( _≈_ to _≈₃_ ; trans to trans₃ )
      f : hom A C
      f = ∘-hom (to ab) (to bc)
      g : hom C A
      g = ∘-hom (from bc) (from ab)
      τ : ∀ b → (| f | ⟨$⟩ (| g | ⟨$⟩ b)) ≈₃ b
      τ b = trans₃ (cong | to bc | (to∼from ab (| from bc | ⟨$⟩ b))) (to∼from bc b)
      ν : ∀ a → (| g | ⟨$⟩ (| f | ⟨$⟩ a)) ≈₁ a
      ν a = trans₁ (cong | from ab | (from∼to bc (| to ab | ⟨$⟩ a))) (from∼to ab a)
```

Fortunately, the lift operation preserves isomorphism (i.e., it's an *algebraic invariant*). As our focus is universal algebra, this is important and is what makes the lift operation a workable solution to the technical problems that arise from the noncumulativity of Agda's universe hierarchy.

```
module _ {𝐀 : Algebra α ρᵃ}{ℓ : Level} where
  Lift-≅ˡ : 𝐀 ≅ (Lift-Algˡ 𝐀 ℓ)
  Lift-≅ˡ = mkiso ToLiftˡ FromLiftˡ (ToFromLiftˡ{𝐀 = 𝐀}) (FromToLiftˡ{𝐀 = 𝐀}{ℓ})

  Lift-≅ʳ : 𝐀 ≅ (Lift-Algʳ 𝐀 ℓ)
  Lift-≅ʳ = mkiso ToLiftʳ FromLiftʳ (ToFromLiftʳ{𝐀 = 𝐀}) (FromToLiftʳ{𝐀 = 𝐀}{ℓ})

Lift-≅ : {𝐀 : Algebra α ρᵃ}{ℓ ρ : Level} → 𝐀 ≅ (Lift-Alg 𝐀 ℓ ρ)
Lift-≅ = ≅-trans Lift-≅ˡ Lift-≅ʳ
```

4.7 Homomorphic Images

(cf. the Homomorphisms.Func.HomomorphicImages module of the Agda Universal Algebra Library.)

We begin with what seems, for our purposes, the most useful way to represent the class of *homomorphic images* of an algebra in dependent type theory.

```
ov : Level → Level
ov α = 𝓞 ⊔ 𝓥 ⊔ lsuc α

_IsHomImageOf_ : (𝐁 : Algebra β ρᵇ)(𝐀 : Algebra α ρᵃ) → Type (𝓞 ⊔ 𝓥 ⊔ α ⊔ β ⊔ ρᵃ ⊔ ρᵇ)
𝐁 IsHomImageOf 𝐀 = Σ[ φ ∈ hom 𝐀 𝐁 ] IsSurjective ∣ φ ∣

HomImages : Algebra α ρᵃ → Type (α ⊔ ρᵃ ⊔ ov (β ⊔ ρᵇ))
HomImages {β = β}{ρᵇ = ρᵇ} 𝐀 = Σ[ 𝐁 ∈ Algebra β ρᵇ ] 𝐁 IsHomImageOf 𝐀
```

These types should be self-explanatory, but just to be sure, let's describe the Sigma type appearing in the second definition. Given an S-algebra $\mathbf{A}$: Algebra α ρ, the type HomImages $\mathbf{A}$ denotes the class of algebras $\mathbf{B}$: Algebra β ρ with a map $\varphi : \mid \mathbf{A} \mid \to \mid \mathbf{B} \mid$ such that φ is a surjective homomorphism.

```
module _ {𝐀 : Algebra α ρᵃ}{𝐁 : Algebra β ρᵇ} where
  Lift-HomImage-lemma : ∀{γ} → (Lift-Alg 𝐀 γ γ) IsHomImageOf 𝐁 → 𝐀 IsHomImageOf 𝐁
  Lift-HomImage-lemma {γ} φ = ∘-hom ∣ φ ∣ (from Lift-≅) ,
                              ∘-IsSurjective ∥ φ ∥ (fromIsSurjective (Lift-≅{𝐀 = 𝐀}))

module _ {𝐀 𝐀' : Algebra α ρᵃ}{𝐁 : Algebra β ρᵇ} where
  HomImage-≅ : 𝐀 IsHomImageOf 𝐀' → 𝐀 ≅ 𝐁 → 𝐁 IsHomImageOf 𝐀'
  HomImage-≅ φ A≅B = ∘-hom ∣ φ ∣ (to A≅B) , ∘-IsSurjective ∥ φ ∥ (toIsSurjective A≅B)
```

5 Subalgebras

5.1 Basic definitions

```
_≤_ : Algebra α ρᵃ → Algebra β ρᵇ → Type (𝓞 ⊔ 𝓥 ⊔ α ⊔ ρᵃ ⊔ β ⊔ ρᵇ)
𝐀 ≤ 𝐁 = Σ[ h ∈ hom 𝐀 𝐁 ] IsInjective ∣ h ∣
```

5.2 Basic properties

```
≤-reflexive : {𝐀 : Algebra α ρᵃ} → 𝐀 ≤ 𝐀
≤-reflexive {𝐀 = 𝐀} = 𝒾𝒹 , id

mon→≤ : {𝐀 : Algebra α ρᵃ}{𝐁 : Algebra β ρᵇ} → mon 𝐀 𝐁 → 𝐀 ≤ 𝐁
mon→≤ {𝐀 = 𝐀}{𝐁} x = mon→intohom 𝐀 𝐁 x

module _ {𝐀 : Algebra α ρᵃ}{𝐁 : Algebra β ρᵇ}{𝐂 : Algebra γ ρᶜ} where
  open Injection using () renaming ( function to fun )

  ≤-trans : 𝐀 ≤ 𝐁 → 𝐁 ≤ 𝐂 → 𝐀 ≤ 𝐂
  ≤-trans ( f , finj ) ( g , ginj ) = (∘-hom f g ) , ∘-injective ∣ f ∣ ∣ g ∣ finj ginj

  ≅-trans-≤ : 𝐀 ≅ 𝐁 → 𝐁 ≤ 𝐂 → 𝐀 ≤ 𝐂
  ≅-trans-≤ A≅B (h , hinj) = (∘-hom (to A≅B) h) , (∘-injective ∣ to A≅B ∣ ∣ h ∣ (toIsInjective A≅B) hinj)
```

5.3 Products of subalgebras

```
module _ {ι : Level} {I : Type ι}{𝒜 : I → Algebra α ρᵃ}{ℬ : I → Algebra β ρᵇ} where
  open Algebra (⨅ 𝒜) using () renaming ( Domain to ⨅A )
  open Algebra (⨅ ℬ) using () renaming ( Domain to ⨅B )
  open Setoid ⨅A using ( refl )

  ⨅-≤ : (∀ i → ℬ i ≤ 𝒜 i) → ⨅ ℬ ≤ ⨅ 𝒜
  ⨅-≤ B≤A = h , hM
    where
    h : hom (⨅ ℬ) (⨅ 𝒜)
    h = hfunc , hhom
      where
      hi : ∀ i → hom (ℬ i) (𝒜 i)
      hi i = ∣ B≤A i ∣

      hfunc : ⨅B ⟶ ⨅A
      (hfunc ⟨$⟩ x) i = ∣ hi i ∣ ⟨$⟩ (x i)
      cong hfunc = λ xy i → cong ∣ hi i ∣ (xy i)
      hhom : IsHom (⨅ ℬ) (⨅ 𝒜) hfunc
      compatible hhom = λ i → compatible ∥ hi i ∥

    hM : IsInjective ∣ h ∣
    hM = λ xy i → ∥ B≤A i ∥ (xy i)
```

6 Terms

6.1 Basic definitions

Fix a signature S and let X denote an arbitrary nonempty collection of variable symbols. Assume the symbols in X are distinct from the operation symbols of S, that is X ∩ | S | = ∅.

By a *word* in the language of S, we mean a nonempty, finite sequence of members of X ∪ | S |. We denote the concatenation of such sequences by simple juxtaposition.

Let S_0 denote the set of nullary operation symbols of S. We define by induction on n the sets T_n of *words* over X ∪ | S | as follows (cf. Bergman (2012) Def. 4.19):

$T_0 :=$ X ∪ S_0 and $T_{n+1} := T_n \cup \mathscr{T}_n$

where $\mathscr{T}_n$ is the collection of all f t such that f : | S | and t : ∥ S ∥ f → T_n. (Recall, ∥ S ∥ f is the arity of the operation symbol f.)

We define the collection of *terms* in the signature S over X by Term X := $\bigcup_n T_n$. By an S-*term* we mean a term in the language of S.

The definition of Term X is recursive, indicating that an inductive type could be used to represent the semantic notion of terms in type theory. Indeed, such a representation is given by the following inductive type.

```
data Term (X : Type χ ) : Type (ov χ) where
  ℊ : X → Term X -- (ℊ for "generator")
  node : (f : ∣ S ∣)(t : ∥ S ∥ f → Term X) → Term X
open Term
```

This is a very basic inductive type that represents each term as a tree with an operation symbol at each node and a variable symbol at each leaf (generator).

Notation. As usual, the type X represents an arbitrary collection of variable symbols. Recall, ov χ is our shorthand notation for the universe level 𝓞 ⊔ 𝓥 ⊔ lsuc χ.

6.2 Equality of terms

We take a different approach here, using Setoids instead of quotient types. That is, we will define the collection of terms in a signature as a setoid with a particular equality-of-terms relation, which we must define. Ultimately we will use this to define the (absolutely free) term algebra as a Algebra whose carrier is the setoid of terms.

```
module _ {X : Type χ } where

  -- Equality of terms as an inductive datatype
  data _≐_ : Term X → Term X → Type (ov χ) where
    rfl : {x y : X} → x ≡ y → (ℊ x) ≐ (ℊ y)
    gnl : ∀ {f}{s t : ∥ S ∥ f → Term X} → (∀ i → (s i) ≐ (t i)) → (node f s) ≐ (node f t)

  -- Equality of terms is an equivalence relation
  open Level
  ≐-isRefl : Reflexive _≐_
  ≐-isRefl {ℊ _} = rfl ≡.refl
  ≐-isRefl {node _ _} = gnl (λ _ → ≐-isRefl)

  ≐-isSym : Symmetric _≐_
  ≐-isSym (rfl x) = rfl (≡.sym x)
  ≐-isSym (gnl x) = gnl (λ i → ≐-isSym (x i))
```

```
≐-isTrans : Transitive _≐_
≐-isTrans (rfl x) (rfl y) = rfl (≡.trans x y)
≐-isTrans (gnl x) (gnl y) = gnl (λ i → ≐-isTrans (x i) (y i))

≐-isEquiv : IsEquivalence _≐_
≐-isEquiv = record { refl = ≐-isRefl ; sym = ≐-isSym ; trans = ≐-isTrans }
```

6.3 The term algebra

For a given signature S, if the type Term X is nonempty (equivalently, if X or | S | is nonempty), then we can define an algebraic structure, denoted by **T** X and called the *term algebra in the signature* S *over* X. Terms are viewed as acting on other terms, so both the domain and basic operations of the algebra are the terms themselves.

- For each operation symbol f : | S |, denote by f ̂ (**T** X) the operation on Term X that maps a tuple t : ∥ S ∥ f → | **T** X | to the formal term f t.
- Define **T** X to be the algebra with universe | **T** X | := Term X and operations f ̂ (**T** X), one for each symbol f in | S |.

In Agda the term algebra can be defined as simply as one might hope.

```
TermSetoid : (X : Type χ) → Setoid (ov χ) (ov χ)
TermSetoid X = record { Carrier = Term X ; _≈_ = _≐_ ; isEquivalence = ≐-isEquiv }

T : (X : Type χ) → Algebra (ov χ) (ov χ)
Algebra.Domain (T X) = TermSetoid X
Algebra.Interp (T X) ⟨$⟩ (f , ts) = node f ts
cong (Algebra.Interp (T X)) (≡.refl , ss≐ts) = gnl ss≐ts
```

6.4 Interpretation of terms

The approach to terms and their interpretation in this module was inspired by Andreas Abel's formal proof of Birkhoff's completeness theorem.

A substitution from X to Y associates a term in X with each variable in Y.

```
-- Parallel substitutions.
Sub : Type χ → Type χ → Type (ov χ)
Sub X Y = (y : Y) → Term X

-- Application of a substitution.
_[_] : {X Y : Type χ}(t : Term Y) (σ : Sub X Y) → Term X
(ℊ x) [ σ ] = σ x
(node f ts) [ σ ] = node f (λ i → ts i [ σ ])
```

An environment for Γ maps each variable x : Γ to an element of A, and equality of environments is defined pointwise.

```
module Environment (A : Algebra α ℓ) where
  open Algebra A using ( Interp ) renaming ( Domain to A )
  open Setoid 𝔻[ A ] using ( refl ; sym ; trans ) renaming ( _≈_ to _≈ₐ_ ; Carrier to |A| )
```

```
Env : Type χ → Setoid _ _
Env X = record { Carrier = X → |A|
               ; _≈_ = λ ρ ρ' → (x : X) → ρ x ≈a ρ' x
               ; isEquivalence =
                   record { refl = λ _ → refl
                          ; sym = λ h x → sym (h x)
                          ; trans = λ g h x → trans (g x) (h x) }}

⟦_⟧ : {X : Type χ}(t : Term X) → (Env X) ⟶ A
⟦ g x ⟧ ⟨$⟩ ρ = ρ x
⟦ node f args ⟧ ⟨$⟩ ρ = (Interp 𝐀) ⟨$⟩ (f , λ i → ⟦ args i ⟧ ⟨$⟩ ρ)
cong ⟦ g x ⟧ u≈v = u≈v x
cong ⟦ node f args ⟧ x≈y = cong (Interp 𝐀)(≡.refl , λ i → cong ⟦ args i ⟧ x≈y )
```

An equality between two terms holds in a model if the two terms are equal under all valuations of their free variables (cf. Andreas Abel's formal proof of Birkhoff's completeness theorem).

```
Equal : ∀ {X : Type χ} (s t : Term X) → Type _
Equal {X = X} s t = ∀ (ρ : Carrier (Env X)) → ⟦ s ⟧ ⟨$⟩ ρ ≈a ⟦ t ⟧ ⟨$⟩ ρ

≐→Equal : {X : Type χ}(s t : Term X) → s ≐ t → Equal s t
≐→Equal .(g _) .(g _) (rfl ≡.refl) = λ _ → refl
≐→Equal (node _ s)(node _ t)(gnl x) =
  λ ρ → cong (Interp 𝐀)(≡.refl , λ i → ≐→Equal(s i)(t i)(x i)ρ )
```

Equal is an equivalence relation.

```
EqualIsEquiv : {Γ : Type χ} → IsEquivalence (Equal {X = Γ})
IsEquivalence.refl  EqualIsEquiv = λ _ → refl
IsEquivalence.sym EqualIsEquiv = λ x=y ρ → sym (x=y ρ)
IsEquivalence.trans EqualIsEquiv = λ ij jk ρ → trans (ij ρ) (jk ρ)
```

Evaluation of a substitution gives an environment (cf. Andreas Abel's formal proof of Birkhoff's completeness theorem)

```
⟦_⟧s : {X Y : Type χ} → Sub X Y → Carrier(Env X) → Carrier (Env Y)
⟦ σ ⟧s ρ x = ⟦ σ x ⟧ ⟨$⟩ ρ
```

6.5 Substitution lemma

We prove that $[\![t[\sigma]]\!]\rho \simeq [\![t]\!][\![\sigma]\!]\rho$ (cf. Andreas Abel's formal proof of Birkhoff's completeness theorem).

```
substitution : {X Y : Type χ} → (t : Term Y) (σ : Sub X Y) (ρ : Carrier( Env X ) )
  → ⟦ t [ σ ] ⟧ ⟨$⟩ ρ ≈a ⟦ t ⟧ ⟨$⟩ (⟦ σ ⟧s ρ)

substitution (g x) σ ρ = refl
substitution (node f ts) σ ρ = cong (Interp 𝐀)(≡.refl , λ i → substitution (ts i) σ ρ)
```

6.6 Compatibility of terms

We now prove two important facts about term operations. The first of these, which is used very often in the sequel, asserts that every term commutes with every homomorphism.

```
module _ {X : Type χ}{𝐀 : Algebra α ρᵃ}{𝐁 : Algebra β ρᵇ}(hh : hom 𝐀 𝐁) where
  open Algebra 𝐁 using () renaming ( Interp to Interp₂ )
  open Setoid 𝔻[ 𝐁 ] using ( _≈_ ; refl )
  open SetoidReasoning 𝔻[ 𝐁 ]
  private hfunc = ∣ hh ∣ ; h = _⟨$⟩_ hfunc

  open Environment 𝐀 using () renaming ( ⟦_⟧ to ⟦_⟧₁ )
  open Environment 𝐁 using () renaming ( ⟦_⟧ to ⟦_⟧₂ )

  comm-hom-term : (t : Term X) (a : X → 𝕌[ 𝐀 ])
                  -----------------------------------------
   →              h (⟦ t ⟧₁ ⟨$⟩ a) ≈ ⟦ t ⟧₂ ⟨$⟩ (h ∘ a)

  comm-hom-term (ℊ x) a = refl
  comm-hom-term (node f t) a = goal
    where
    goal : h (⟦ node f t ⟧₁ ⟨$⟩ a) ≈ (⟦ node f t ⟧₂ ⟨$⟩ (h ∘ a))
    goal =
      begin
       h (⟦ node f t ⟧₁ ⟨$⟩ a)              ≈⟨ (compatible ∥ hh ∥) ⟩
       (f ̂ 𝐁)(λ i → h (⟦ t i ⟧₁ ⟨$⟩ a)) ≈⟨ cong Interp₂ (≡.refl , λ i → comm-hom-term (t i) a) ⟩
       (f ̂ 𝐁)(λ i → ⟦ t i ⟧₂ ⟨$⟩ (h ∘ a)) ≈⟨ refl ⟩
       (⟦ node f t ⟧₂ ⟨$⟩ (h ∘ a))
      ∎
```

6.7 Interpretation of terms in product algebras

```
module _ {X : Type χ}{ι : Level} {I : Type ι} (𝒜 : I → Algebra α ρᵃ) where
  open Algebra (⨅ 𝒜) using () renaming ( Interp to ⨅Interp )
  open Setoid 𝔻[ ⨅ 𝒜 ] using ( _≈_ )
  open Environment (⨅ 𝒜) using () renaming ( ⟦_⟧ to ⟦_⟧₁ )
  open Environment using ( ⟦_⟧ ; ≐→Equal )

  interp-prod : (p : Term X) → ∀ ρ → ⟦ p ⟧₁ ⟨$⟩ ρ ≈ (λ i → (⟦ 𝒜 i ⟧ p) ⟨$⟩ (λ x → (ρ x) i))
  interp-prod (ℊ x) = λ ρ i → ≐→Equal (𝒜 i) (ℊ x) (ℊ x) ≐-isRefl λ x' → (ρ x) i
  interp-prod (node f t) = λ ρ i → cong ⨅Interp (≡.refl , (λ j k → interp-prod (t j) ρ k)) i
```

7 Model Theory and Equational Logic

(cf. the Varieties.Func.SoundAndComplete module of the Agda Universal Algebra Library)

7.1 Basic definitions

Let S be a signature. By an *identity* or *equation* in S we mean an ordered pair of terms in a given context. For instance, if the context happens to be the type X : Type χ, then an equation will be a pair of inhabitants of the domain of term algebra $\mathbf{T}$ X.

We define an equation in Agda using the following record type with fields denoting the left-hand and right-hand sides of the equation, along with an equation "context" representing the underlying collection of variable symbols (cf. Andreas Abel's formal proof of Birkhoff's completeness theorem).

```
record Eq : Type (ov χ) where
 constructor _≈̇_
 field
  {cxt} : Type χ
  lhs : Term cxt
  rhs : Term cxt

open Eq public
```

We now define a type representing the notion of an equation p ≈̇ q holding (when p and q are interpreted) in algebra $\mathbf{A}$.

If $\mathbf{A}$ is an S-algebra we say that $\mathbf{A}$ *satisfies* p ≈ q provided for all environments ρ : X → | $\mathbf{A}$ | (assigning values in the domain of $\mathbf{A}$ to variable symbols in X) we have ⟦ p ⟧⟨\$⟩ ρ ≈ ⟦ q ⟧ ⟨\$⟩ ρ. In this situation, we write $\mathbf{A}$ ⊨ (p ≈̇ q) and say that $\mathbf{A}$ *models* the identity p ≈ q.

If $\mathcal{K}$ is a class of algebras, all of the same signature, we write $\mathcal{K}$ |⊨ (p ≈̇ q) if, for every $\mathbf{A}$ ∈ $\mathcal{K}$, we have $\mathbf{A}$ ⊨ (p ≈̇ q)'.

Because a class of structures has a different type than a single structure, we must use a slightly different syntax to avoid overloading the relations ⊨ and ≈. As a reasonable alternative to what we would normally express informally as $\mathcal{K}$ ⊨ p ≈ q, we have settled on $\mathcal{K}$ |⊨ (p ≈̇ q) to denote this relation. To reiterate, if $\mathcal{K}$ is a class of S-algebras, we write $\mathcal{K}$ |⊨ (p ≈̇ q) provided every $\mathbf{A}$ ∈ $\mathcal{K}$ satisfies $\mathbf{A}$ ⊨ (p ≈̇ q).

```
_⊨_ : (𝐀 : Algebra α ρᵃ)(term-identity : Eq{χ}) → Type _
𝐀 ⊨ (p ≈̇ q) = Equal p q where open Environment 𝐀

_|⊨_ : Pred (Algebra α ρᵃ) ℓ → Eq{χ} → Type (ℓ ⊔ χ ⊔ ov(α ⊔ ρᵃ))
𝒦 |⊨ equ = ∀ 𝐀 → 𝒦 𝐀 → 𝐀 ⊨ equ
```

We denote by $\mathbf{A}$ ⊧ $\mathcal{E}$ the assertion that the algebra $\mathbf{A}$ models every equation in a collection $\mathcal{E}$ of equations.

```
_⊧_ : (𝐀 : Algebra α ρᵃ) → {ι : Level}{I : Type ι} → (I → Eq{χ}) → Type _
𝐀 ⊧ ℰ = ∀ i → Equal (lhs (ℰ i))(rhs (ℰ i)) where open Environment 𝐀
```

7.2 Equational theories and models

If $\mathcal{K}$ denotes a class of structures, then Th $\mathcal{K}$ represents the set of identities modeled by the members of $\mathcal{K}$.

```
Th : {X : Type χ} → Pred (Algebra α ρᵃ) ℓ → Pred(Term X × Term X) _
Th 𝒦 = λ (p , q) → 𝒦 |⊨ (p ≈̇ q)

Mod : {X : Type χ} → Pred(Term X × Term X) ℓ → Pred (Algebra α ρᵃ) _
Mod ℰ 𝐀 = ∀ {p q} → (p , q) ∈ ℰ → Equal p q where open Environment 𝐀
```

7.3 The entailment relation

Based on Andreas Abel's Agda formalization of Birkhoff's completeness theorem.)

```
module _ {χ ι : Level} where

  data _⊢_▹_≈_ {I : Type ι}(ℰ : I → Eq) : (X : Type χ)(p q : Term X) → Type (ι ⊔ ov χ) where
    hyp : ∀ i → let p ≈˙ q = ℰ i in ℰ ⊢ _ ▹ p ≈ q
    app : ∀ {ps qs} → (∀ i → ℰ ⊢ Γ ▹ ps i ≈ qs i) → ℰ ⊢ Γ ▹ (node f ps) ≈ (node f qs)
    sub : ∀ {p q} → ℰ ⊢ Δ ▹ p ≈ q → ∀ (σ : Sub Γ Δ) → ℰ ⊢ Γ ▹ (p [ σ ]) ≈ (q [ σ ])

    ⊢refl  : ∀ {p}                → ℰ ⊢ Γ ▹ p ≈ p
    ⊢sym   : ∀ {p q : Term Γ} → ℰ ⊢ Γ ▹ p ≈ q → ℰ ⊢ Γ ▹ q ≈ p
    ⊢trans : ∀ {p q r : Term Γ} → ℰ ⊢ Γ ▹ p ≈ q → ℰ ⊢ Γ ▹ q ≈ r → ℰ ⊢ Γ ▹ p ≈ r

  ⊢▹≈IsEquiv : {X : Type χ}{I : Type ι}{ℰ : I → Eq} → IsEquivalence (ℰ ⊢ X ▹_≈_)
  ⊢▹≈IsEquiv = record { refl = ⊢refl ; sym = ⊢sym ; trans = ⊢trans }
```

7.4 Soundness

In any model **A** that satisfies the equations ℰ, derived equality is actual equality (cf. Andreas Abel's Agda formalization of Birkhoff's completeness theorem.)

```
module Soundness {χ α ι : Level}{I : Type ι} (ℰ : I → Eq{χ})
                 (𝐀 : Algebra α ρᵃ) -- We assume an algebra 𝐀
                 (V : 𝐀 ⊨ ℰ)       -- that models all equations in ℰ.
                 where

 open Algebra 𝐀 using () renaming (Domain to A ; Interp to InterpA)
 open SetoidReasoning A
 open Environment 𝐀 renaming ( ⟦_⟧s to ⟪_⟫ )
 open IsEquivalence using ( refl ; sym ; trans )

 sound : ∀ {p q} → ℰ ⊢ Γ ▹ p ≈ q → 𝐀 ⊨ (p ≈˙ q)
 sound (hyp i)                    = V i
 sound (app {f = f} es) ρ         = cong InterpA (≡.refl , λ i → sound (es i) ρ)
 sound (sub {p = p} {q} Epq σ) ρ =
  begin
   ⟦ p [ σ ] ⟧ ⟨$⟩ ρ ≈⟨ substitution p σ ρ ⟩
   ⟦ p ⟧ ⟨$⟩ ⟪ σ ⟫ ρ ≈⟨ sound Epq (⟪ σ ⟫ ρ) ⟩
   ⟦ q ⟧ ⟨$⟩ ⟪ σ ⟫ ρ ≈˘⟨ substitution  q σ ρ ⟩
   ⟦ q [ σ ] ⟧ ⟨$⟩ ρ ∎

 sound (⊢refl {p = p})              = refl  EqualIsEquiv {x = p}
 sound (⊢sym {p = p} {q} Epq) = sym EqualIsEquiv {x = p}{q} (sound Epq)
 sound (⊢trans{p = p}{q}{r} Epq Eqr) = trans EqualIsEquiv {i = p}{q}{r}(sound Epq)(sound Eqr)
```

8 The Closure Operators H, S, P and V

Fix a signature S, let $\mathscr{K}$ be a class of S-algebras, and define

- H $\mathcal{K}$ = algebras isomorphic to a homomorphic image of a member of $\mathcal{K}$;
- S $\mathcal{K}$ = algebras isomorphic to a subalgebra of a member of $\mathcal{K}$;
- P $\mathcal{K}$ = algebras isomorphic to a product of members of $\mathcal{K}$.

A straight-forward verification confirms that H, S, and P are *closure operators* (expansive, monotone, and idempotent). A class $\mathcal{K}$ of S-algebras is said to be *closed under the taking of homomorphic images* provided H $\mathcal{K} \subseteq \mathcal{K}$. Similarly, $\mathcal{K}$ is *closed under the taking of subalgebras* (resp., *arbitrary products*) provided S $\mathcal{K} \subseteq \mathcal{K}$ (resp., P $\mathcal{K} \subseteq \mathcal{K}$). The operators H, S, and P can be composed with one another repeatedly, forming yet more closure operators.

An algebra is a homomorphic image (resp., subalgebra; resp., product) of every algebra to which it is isomorphic. Thus, the class H $\mathcal{K}$ (resp., S $\mathcal{K}$; resp., P $\mathcal{K}$) is closed under isomorphism.

A *variety* is a class of S-algebras that is closed under the taking of homomorphic images, subalgebras, and arbitrary products. To represent varieties we define types for the closure operators H, S, and P that are composable. Separately, we define a type V which represents closure under all three operators, H, S, and P.

8.1 Basic definitions

We now define the type H to represent classes of algebras that include all homomorphic images of algebras in the class—i.e., classes that are closed under the taking of homomorphic images—the type S to represent classes of algebras that closed under the taking of subalgebras, and the type P to represent classes of algebras closed under the taking of arbitrary products.

```
H : ∀ ℓ → Pred(Algebra α ρᵃ) (α ⊔ ρᵃ ⊔ ov ℓ) → Pred(Algebra β ρᵇ) (β ⊔ ρᵇ ⊔ ov(α ⊔ ρᵃ ⊔ ℓ))
H {α}{ρᵃ} _ 𝒦 𝐁 = Σ[ 𝐀 ∈ Algebra α ρᵃ ] 𝐀 ∈ 𝒦 × 𝐁 IsHomImageOf 𝐀

S : ∀ ℓ → Pred(Algebra α ρᵃ) (α ⊔ ρᵃ ⊔ ov ℓ) → Pred(Algebra β ρᵇ) (β ⊔ ρᵇ ⊔ ov(α ⊔ ρᵃ ⊔ ℓ))
S {α}{ρᵃ} _ 𝒦 𝐁 = Σ[ 𝐀 ∈ Algebra α ρᵃ ] 𝐀 ∈ 𝒦 × 𝐁 ≤ 𝐀

P : ∀ ℓ ι → Pred(Algebra α ρᵃ) (α ⊔ ρᵃ ⊔ ov ℓ) → Pred(Algebra β ρᵇ) (β ⊔ ρᵇ ⊔ ov(α ⊔ ρᵃ ⊔ ℓ ⊔ ι))
P {α}{ρᵃ} _ ι 𝒦 𝐁 = Σ[ I ∈ Type ι ] (Σ[ 𝒜 ∈ (I → Algebra α ρᵃ) ] (∀ i → 𝒜 i ∈ 𝒦) × (𝐁 ≅ ⨅ 𝒜))
```

A class $\mathcal{K}$ of S-algebras is called a *variety* if it is closed under each of the closure operators H, S, and P defined above. The corresponding closure operator is often denoted $\mathbb{V}$ or $\mathcal{V}$, but we will denote it by V.

```
module _ {α ρᵃ β ρᵇ γ ρᶜ δ ρᵈ : Level} where
 private a = α ⊔ ρᵃ ; b = β ⊔ ρᵇ ; c = γ ⊔ ρᶜ ; d = δ ⊔ ρᵈ

 V : ∀ ℓ ι → Pred(Algebra α ρᵃ) (a ⊔ ov ℓ) → Pred(Algebra δ ρᵈ) (d ⊔ ov(a ⊔ b ⊔ c ⊔ ℓ ⊔ ι))
 V ℓ ι 𝒦 = H{γ}{ρᶜ}{δ}{ρᵈ} (a ⊔ b ⊔ ℓ ⊔ ι) (S{β}{ρᵇ} (a ⊔ ℓ ⊔ ι) (P ℓ ι 𝒦))

module _ {α ρᵃ ℓ : Level}(𝒦 : Pred(Algebra α ρᵃ) (α ⊔ ρᵃ ⊔ ov ℓ))
         (𝐀 : Algebra (α ⊔ ρᵃ ⊔ ℓ) (α ⊔ ρᵃ ⊔ ℓ)) where
 private ι = ov(α ⊔ ρᵃ ⊔ ℓ)

 V-≅-lc : Lift-Alg 𝐀 ι ι ∈ V{β = ι}{ι} ℓ ι 𝒦 → 𝐀 ∈ V{γ = ι}{ι} ℓ ι 𝒦
 V-≅-lc (𝐀' , spA' , lAimgA') = 𝐀' , (spA' , AimgA')
  where
  AimgA' : 𝐀 IsHomImageOf 𝐀'
  AimgA' = Lift-HomImage-lemma lAimgA'
```

8.2 Properties

8.2.1 Idempotence of S

S is a closure operator. The facts that S is monotone and expansive won't be needed, so we omit the proof of these facts. However, we will make use of idempotence of S, so we prove that property as follows.

```
S-idem : {𝒦 : Pred (Algebra α ρᵃ)(α ⊔ ρᵃ ⊔ ov ℓ)}
 → S{β = γ}{ρᶜ} (α ⊔ ρᵃ ⊔ ℓ) (S{β = β}{ρᵇ} ℓ 𝒦) ⊆ S{β = γ}{ρᶜ} ℓ 𝒦

S-idem (𝐀 , (𝐁 , sB , A≤B) , x≤A) = 𝐁 , (sB , ≤-trans x≤A A≤B)
```

8.2.2 Algebraic invariance of ⊧

The binary relation ⊧ would be practically useless if it were not an *algebraic invariant* (i.e., invariant under isomorphism). Let us now verify that the models relation we defined above has this essential property.

```
module _ {X : Type χ}{𝐀 : Algebra α ρᵃ}(𝐁 : Algebra β ρᵇ)(p q : Term X) where

 ⊧-I-invar : 𝐀 ⊧ (p ≈̇ q) → 𝐀 ≅ 𝐁 → 𝐁 ⊧ (p ≈̇ q)
 ⊧-I-invar Apq (mkiso fh gh f∼g g∼f) ρ =
  begin
   ⟦ p ⟧₂ ⟨$⟩ ρ ≈˘⟨ cong ⟦ p ⟧₂ (λ x → f∼g (ρ x)) ⟩
   ⟦ p ⟧₂ ⟨$⟩ (ff ∘ (g ∘ ρ)) ≈˘⟨ comm-hom-term fh p (g ∘ ρ) ⟩
   ff (⟦ p ⟧₁ ⟨$⟩ (g ∘ ρ)) ≈⟨ cong ∣ fh ∣ (Apq (g ∘ ρ)) ⟩
   ff (⟦ q ⟧₁ ⟨$⟩ (g ∘ ρ)) ≈⟨ comm-hom-term fh q (g ∘ ρ) ⟩
   ⟦ q ⟧₂ ⟨$⟩ (ff ∘ (g ∘ ρ)) ≈⟨ cong ⟦ q ⟧₂ (λ x → f∼g (ρ x)) ⟩
   ⟦ q ⟧₂ ⟨$⟩ ρ ∎
  where
  open Environment 𝐀 using () renaming ( ⟦_⟧ to ⟦_⟧₁ )
  open Environment 𝐁 using () renaming ( ⟦_⟧ to ⟦_⟧₂ )
  open SetoidReasoning 𝔻[ 𝐁 ]
  private ff = _⟨$⟩_ ∣ fh ∣ ; g = _⟨$⟩_ ∣ gh ∣
```

8.2.3 Subalgebraic invariance of ⊧

Identities modeled by an algebra **A** are also modeled by every subalgebra of **A**, which fact can be formalized as follows.

```
module _ {X : Type χ}{𝐀 : Algebra α ρᵃ}{𝐁 : Algebra β ρᵇ}{p q : Term X} where

 ⊧-S-invar : 𝐀 ⊧ (p ≈̇ q) → 𝐁 ≤ 𝐀 → 𝐁 ⊧ (p ≈̇ q)
 ⊧-S-invar Apq B≤A b = goal
  where
  open Environment 𝐀 using () renaming ( ⟦_⟧ to ⟦_⟧₁ )
  open Setoid 𝔻[ 𝐀 ] using ( _≈_ )
  open SetoidReasoning 𝔻[ 𝐀 ]
```

```
    open Environment B using () renaming ( ⟦_⟧ to ⟦_⟧₂ )
    open Setoid 𝔻[ B ] using () renaming ( _≈_ to _≈₂_ )
    hh : hom B A
    hh = ∣ B≤A ∣
    h = _⟨$⟩_ ∣ hh ∣
    ξ : ∀ b → h (⟦ p ⟧₂ ⟨$⟩ b) ≈ h (⟦ q ⟧₂ ⟨$⟩ b)
    ξ b = begin
          h (⟦ p ⟧₂ ⟨$⟩ b) ≈⟨ comm-hom-term hh p b ⟩
          ⟦ p ⟧₁ ⟨$⟩ (h ∘ b) ≈⟨ Apq (h ∘ b) ⟩
          ⟦ q ⟧₁ ⟨$⟩ (h ∘ b) ≈˘⟨ comm-hom-term hh q b ⟩
          h (⟦ q ⟧₂ ⟨$⟩ b) ∎
    goal : ⟦ p ⟧₂ ⟨$⟩ b ≈₂ ⟦ q ⟧₂ ⟨$⟩ b
    goal = ∥ B≤A ∥ (ξ b)
```

8.2.4 Product invariance of $\models$

An identity satisfied by all algebras in an indexed collection is also satisfied by the product of algebras in that collection.

```
module _ {X : Type χ}{I : Type ℓ}(𝒜 : I → Algebra α ρᵃ){p q : Term X} where

  ⊫-P-invar : (∀ i → 𝒜 i ⊫ (p ≈̇ q)) → ⨅ 𝒜 ⊫ (p ≈̇ q)
  ⊫-P-invar 𝒜pq a = goal
    where
    open Environment (⨅ 𝒜) using () renaming ( ⟦_⟧ to ⟦_⟧₁ )
    open Environment using ( ⟦_⟧ )
    open Setoid 𝔻[ ⨅ 𝒜 ] using ( _≈_ )
    open SetoidReasoning 𝔻[ ⨅ 𝒜 ]
    ξ : (λ i → (⟦ 𝒜 i ⟧ p) ⟨$⟩ (λ x → (a x) i)) ≈ (λ i → (⟦ 𝒜 i ⟧ q) ⟨$⟩ (λ x → (a x) i))
    ξ = λ i → 𝒜pq i (λ x → (a x) i)
    goal : ⟦ p ⟧₁ ⟨$⟩ a ≈ ⟦ q ⟧₁ ⟨$⟩ a
    goal = begin
           ⟦ p ⟧₁ ⟨$⟩ a ≈⟨ interp-prod 𝒜 p a ⟩
           (λ i → (⟦ 𝒜 i ⟧ p) ⟨$⟩ (λ x → (a x) i)) ≈⟨ ξ ⟩
           (λ i → (⟦ 𝒜 i ⟧ q) ⟨$⟩ (λ x → (a x) i)) ≈˘⟨ interp-prod 𝒜 q a ⟩
           ⟦ q ⟧₁ ⟨$⟩ a ∎
```

8.2.5 PS ⊆ SP

Another important fact we will need about the operators S and P is that a product of subalgebras of algebras in a class $\mathcal{K}$ is a subalgebra of a product of algebras in $\mathcal{K}$. We denote this inclusion by PS⊆SP, which we state and prove as follows.

```
module _ {𝒦 : Pred(Algebra α ρᵃ) (α ⊔ ρᵃ ⊔ ov ℓ)} where
  private
    a = α ⊔ ρᵃ
    oaℓ = ov (a ⊔ ℓ)

  PS⊆SP : P (a ⊔ ℓ) oaℓ (S{β = α}{ρᵃ} ℓ 𝒦) ⊆ S oaℓ (P ℓ oaℓ 𝒦)
```

```
PS⊆SP {𝐁} (I , ( 𝒜 , sA , B≅⨅A )) = Goal
  where
  ℬ : I → Algebra α ρᵃ
  ℬ i = ∣ sA i ∣
  kB : (i : I) → ℬ i ∈ 𝒦
  kB i = fst ∥ sA i ∥
  ⨅A≤⨅B : ⨅ 𝒜 ≤ ⨅ ℬ
  ⨅A≤⨅B = ⨅-≤ λ i → snd ∥ sA i ∥
  Goal : 𝐁 ∈ S{β = oaℓ}{oaℓ}oaℓ (P {β = oaℓ}{oaℓ} ℓ oaℓ 𝒦)
  Goal = ⨅ ℬ , (I , (ℬ , (kB , ≅-refl))) , (≅-trans-≤ B≅⨅A ⨅A≤⨅B)
```

8.3 Identity preservation

The classes H 𝒦, S 𝒦, P 𝒦, and V 𝒦 all satisfy the same set of equations. We will only use a subset of the inclusions used to prove this fact. (For a complete proof, see the Varieties.Func.Preservation module of the Agda Universal Algebra Library.)

8.3.1 H preserves identities

First we prove that the closure operator H is compatible with identities that hold in the given class.

```
module _ {X : Type χ}{𝒦 : Pred(Algebra α ρᵃ) (α ⊔ ρᵃ ⊔ ov ℓ)}{p q : Term X} where

 H-id1 : 𝒦 ⊫ (p ≈̇ q) → (H {β = α}{ρᵃ}ℓ 𝒦) ⊫ (p ≈̇ q)
 H-id1 σ 𝐁 (𝐀 , kA , BimgOfA) ρ = B⊧pq
  where
  IH : 𝐀 ⊧ (p ≈̇ q)
  IH = σ 𝐀 kA
  open Environment 𝐀 using () renaming ( ⟦_⟧ to ⟦_⟧₁)
  open Environment 𝐁 using ( ⟦_⟧ )
  open Setoid 𝔻[ 𝐁 ] using ( _≈_ )
  open SetoidReasoning 𝔻[ 𝐁 ]

  φ : hom 𝐀 𝐁
  φ = ∣ BimgOfA ∣
  φE : IsSurjective ∣ φ ∣
  φE = ∥ BimgOfA ∥
  φ⁻¹ : 𝕌[ 𝐁 ] → 𝕌[ 𝐀 ]
  φ⁻¹ = SurjInv ∣ φ ∣ φE

  ζ : ∀ x → (∣ φ ∣ ⟨$⟩ (φ⁻¹ ∘ ρ) x ) ≈ ρ x
  ζ = λ _ → InvIsInverseʳ φE

  B⊧pq : (⟦ p ⟧ ⟨$⟩ ρ) ≈ (⟦ q ⟧ ⟨$⟩ ρ)
  B⊧pq = begin
         ⟦ p ⟧ ⟨$⟩ ρ                          ≈˘⟨ cong ⟦ p ⟧ ζ ⟩
         ⟦ p ⟧ ⟨$⟩ (λ x → (∣ φ ∣ ⟨$⟩ (φ⁻¹ ∘ ρ) x)) ≈˘⟨ comm-hom-term φ p (φ⁻¹ ∘ ρ) ⟩
         ∣ φ ∣ ⟨$⟩ (⟦ p ⟧₁ ⟨$⟩ (φ⁻¹ ∘ ρ)) ≈⟨ cong ∣ φ ∣ (IH (φ⁻¹ ∘ ρ)) ⟩
         ∣ φ ∣ ⟨$⟩ (⟦ q ⟧₁ ⟨$⟩ (φ⁻¹ ∘ ρ)) ≈⟨ comm-hom-term φ q (φ⁻¹ ∘ ρ) ⟩
         ⟦ q ⟧ ⟨$⟩ (λ x → (∣ φ ∣ ⟨$⟩ (φ⁻¹ ∘ ρ) x)) ≈⟨ cong ⟦ q ⟧ ζ ⟩
```

```
        ⟦ q ⟧ ⟨$⟩ ρ                    ■
```

8.3.2 S preserves identities

```
S-id1 : 𝒦 |⊨ (p ≈̇ q) → (S {β = α}{ρᵃ} ℓ 𝒦) |⊨ (p ≈̇ q)
S-id1 σ B (A , kA , B≤A) = ⊨-S-invar{p = p}{q} (σ A kA) B≤A
```

The obvious converse is barely worth the bits needed to formalize it, but we will use it below, so let's prove it now.

```
S-id2 : S ℓ 𝒦 |⊨ (p ≈̇ q) → 𝒦 |⊨ (p ≈̇ q)
S-id2 Spq A kA = Spq A (A , (kA , ≤-reflexive))
```

8.3.3 P preserves identities

```
P-id1 : ∀{ι} → 𝒦 |⊨ (p ≈̇ q) → P {β = α}{ρᵃ}ℓ ι 𝒦 |⊨ (p ≈̇ q)
P-id1 σ A (I , 𝒜 , kA , A≅⨅A) = ⊨-I-invar A p q IH (≅-sym A≅⨅A)
  where
  ih : ∀ i → 𝒜 i ⊨ (p ≈̇ q)
  ih i = σ (𝒜 i) (kA i)
  IH : ⨅ 𝒜 ⊨ (p ≈̇ q)
  IH = ⊨-P-invar 𝒜 {p}{q} ih
```

8.3.4 V preserves identities

Finally, we prove the analogous preservation lemmas for the closure operator V.

```
module _ {X : Type χ}{ι : Level}{𝒦 : Pred(Algebra α ρᵃ)(α ⊔ ρᵃ ⊔ ov ℓ)}{p q : Term X} where
 private
  aℓι = α ⊔ ρᵃ ⊔ ℓ ⊔ ι

 V-id1 : 𝒦 |⊨ (p ≈̇ q) → V ℓ ι 𝒦 |⊨ (p ≈̇ q)
 V-id1 σ B (A , (⨅A , p⨅A , A≤⨅A) , BimgA) =
  H-id1{ℓ = aℓι}{𝒦 = S aℓι (P {β = α}{ρᵃ}ℓ ι 𝒦)}{p = p}{q} spK⊨pq B (A , (spA , BimgA))
   where
   spA : A ∈ S aℓι (P {β = α}{ρᵃ}ℓ ι 𝒦)
   spA = ⨅A , (p⨅A , A≤⨅A)
   spK⊨pq : S aℓι (P ℓ ι 𝒦) |⊨ (p ≈̇ q)
   spK⊨pq = S-id1{ℓ = aℓι}{p = p}{q} (P-id1{ℓ = ℓ} {𝒦 = 𝒦}{p = p}{q} σ)
```

8.3.5 Th $\mathcal{K}$ ⊆ Th (V $\mathcal{K}$)

From V-id1 it follows that if $\mathcal{K}$ is a class of algebras, then the set of identities modeled by the algebras in $\mathcal{K}$ is contained in the set of identities modeled by the algebras in V $\mathcal{K}$. In other terms, Th $\mathcal{K} \subseteq$ Th (V $\mathcal{K}$). We formalize this observation as follows.

```
classIds-⊆-VIds : 𝒦 |⊨ (p ≈̇ q) → (p , q) ∈ Th (V ℓ ι 𝒦)
classIds-⊆-VIds pKq 𝐀 = V-id1 pKq 𝐀
```

9 Free Algebras

9.1 The absolutely free algebra T X

The term algebra **T** X is *absolutely free* (or *universal*, or *initial*) for algebras in the signature S. That is, for every S-algebra **A**, the following hold.

1. Every function from X to | **A** | lifts to a homomorphism from **T** X to **A**.
2. The homomorphism that exists by item 1 is unique.

We now prove this in Agda, starting with the fact that every map from X to | **A** | lifts to a map from | **T** X | to | **A** | in a natural way, by induction on the structure of the given term.

```
module _ {X : Type χ}{𝐀 : Algebra α ρᵃ}(h : X → 𝕌[ 𝐀 ]) where
  open Algebra 𝐀 using () renaming ( Interp to InterpA)
  open Setoid 𝔻[ 𝐀 ] using ( _≈_ ; reflexive ; refl ; trans )

  free-lift : 𝕌[ 𝐓 X ] → 𝕌[ 𝐀 ]
  free-lift (ℊ x) = h x
  free-lift (node f t) = (f ̂ 𝐀) (λ i → free-lift (t i))

  free-lift-func : 𝔻[ 𝐓 X ] ⟶ 𝔻[ 𝐀 ]
  free-lift-func ⟨$⟩ x = free-lift x
  cong free-lift-func = flcong
    where
    flcong : ∀ {s t} → s ≐ t → free-lift s ≈ free-lift t
    flcong (_≐_.rfl x) = reflexive (≡.cong h x)
    flcong (_≐_.gnl x) = cong InterpA (≡.refl , (λ i → flcong (x i)))
```

Naturally, at the base step of the induction, when the term has the form generator x, the free lift of h agrees with h. For the inductive step, when the given term has the form node f t, the free lift is defined as follows: Assuming (the induction hypothesis) that we know the image of each subterm t i under the free lift of h, define the free lift at the full term by applying f ̂ **A** to the images of the subterms.

The free lift so defined is a homomorphism by construction. Indeed, here is the trivial proof.

```
  lift-hom : hom (𝐓 X) 𝐀
  lift-hom = free-lift-func , hhom
    where
    hfunc : 𝔻[ 𝐓 X ] ⟶ 𝔻[ 𝐀 ]
    hfunc = free-lift-func

    hcomp : compatible-map (𝐓 X) 𝐀 free-lift-func
    hcomp {f}{a} = cong InterpA (≡.refl , (λ i → (cong free-lift-func){a i} ≐-isRefl))

    hhom : IsHom (𝐓 X) 𝐀 hfunc
    hhom = record { compatible = λ{f}{a} → hcomp{f}{a} }
```

```
module _ {X : Type χ}{𝑨 : Algebra α ρᵃ} where
 open Algebra 𝑨 using () renaming ( Interp to InterpA )
 open Setoid 𝔻[ 𝑨 ] using ( _≈_ ; refl )
 open Environment 𝑨 using ( ⟦_⟧ )

 free-lift-interp : (η : X → 𝕌[ 𝑨 ])(p : Term X) → ⟦ p ⟧ ⟨$⟩ η ≈ (free-lift {𝑨 = 𝑨} η) p

 free-lift-interp η (ℊ x) = refl
 free-lift-interp η (node f t) = cong InterpA (≡.refl , (free-lift-interp η) ∘ t)
```

9.2 The relatively free algebra 𝔽

We now define the algebra 𝔽[X], which plays the role of the relatively free algebra, along with the natural epimorphism epi𝔽 : epi (𝑻 X) 𝔽[X] from 𝑻 X to 𝔽[X].

```
module FreeAlgebra {χ : Level}{ι : Level}{I : Type ι}(ℰ : I → Eq) where
 open Algebra

 FreeDomain : Type χ → Setoid _ _
 FreeDomain X = record { Carrier = Term X
                       ; _≈_     = ℰ ⊢ X ▹_≈_
                       ; isEquivalence = ⊢▹≈IsEquiv }
```

The interpretation of an operation is simply the operation itself. This works since ℰ ⊢ X ▹_≈_ is a congruence.

```
FreeInterp : ∀ {X} → ⟨ 𝑆 ⟩ (FreeDomain X) ⟶ FreeDomain X
FreeInterp ⟨$⟩ (f , ts) = node f ts
cong FreeInterp (≡.refl , h) = app h

𝔽[_] : Type χ → Algebra (ov χ) (ι ⊔ ov χ)
Domain 𝔽[ X ] = FreeDomain X
Interp 𝔽[ X ] = FreeInterp
```

9.3 Basic properties of free algebras

In the code below, X will play the role of an arbitrary collection of variables; it would suffice to take X to be the cardinality of the largest algebra in 𝒦, but since we don't know that cardinality, we leave X aribtrary for now.

```
module FreeHom (χ : Level) {𝒦 : Pred(Algebra α ρᵃ) (α ⊔ ρᵃ ⊔ ov ℓ)} where
 private ι = ov(χ ⊔ α ⊔ ρᵃ ⊔ ℓ)
 open Eq

 ℐ : Type ι -- indexes the collection of equations modeled by 𝒦
 ℐ = Σ[ eq ∈ Eq{χ} ] 𝒦 ⊫ ((lhs eq) ≈̇ (rhs eq))

 ℰ : ℐ → Eq
 ℰ (eqv , p) = eqv

 ℰ⊢[_]▹Th𝒦 : (X : Type χ) → ∀{p q} → ℰ ⊢ X ▹ p ≈ q → 𝒦 ⊫ (p ≈̇ q)
 ℰ⊢[ X ]▹Th𝒦 x 𝑨 kA = sound (λ i ρ → ∥ i ∥ 𝑨 kA ρ) x
```

```
  where open Soundness 𝓔 𝐀

open FreeAlgebra {ι = ι}{I = ℐ} 𝓔 using ( 𝔽[_] )
```

9.3.1 The natural epimorphism from T X to 𝔽[X]

Next we define an epimorphism from **T** X onto the relatively free algebra 𝔽[X]. Of course, the kernel of this epimorphism will be the congruence of **T** X defined by identities modeled by (S 𝒦, hence) 𝒦.

```
epi𝔽[_] : (X : Type χ) → epi (𝐓 X) 𝔽[ X ]
epi𝔽[ X ] = h , hepi
 where
 open Algebra 𝔽[ X ] using () renaming ( Domain to F ; Interp to InterpF )
 open Setoid F using () renaming ( _≈_ to _≈F≈_ ; refl to reflF )
 open Algebra (𝐓 X) using () renaming (Domain to TX)
 open Setoid TX using () renaming ( _≈_ to _≈T≈_ ; refl to reflT )
 open _≐_

 c : ∀ {x y} → x ≈T≈ y → x ≈F≈ y
 c (rfl {x}{y} ≡.refl) = reflF
 c (gnl {f}{s}{t} x) = cong InterpF (≡.refl , c ∘ x)

 h : TX ⟶ F
 h = record { f = id ; cong = c }

 hepi : IsEpi (𝐓 X) 𝔽[ X ] h
 compatible (isHom hepi) = cong h reflT
 isSurjective hepi {y} = eq y reflF

hom𝔽[_] : (X : Type χ) → hom (𝐓 X) 𝔽[ X ]
hom𝔽[ X ] = IsEpi.HomReduct ∥ epi𝔽[ X ] ∥

hom𝔽[_]-is-epic : (X : Type χ) → IsSurjective ∣ hom𝔽[ X ] ∣
hom𝔽[ X ]-is-epic = IsEpi.isSurjective (snd (epi𝔽[ X ]))
```

9.3.2 The kernel of the natural epimorphism

```
class-models-kernel : ∀{X p q} → (p , q) ∈ ker ∣ hom𝔽[ X ] ∣ → 𝒦 ⊫ (p ≈̇ q)
class-models-kernel {X = X}{p}{q} pKq = 𝓔⊢[ X ]▹Th𝒦 pKq

kernel-in-theory : {X : Type χ} → ker ∣ hom𝔽[ X ] ∣ ⊆ Th (V ℓ ι 𝒦)
kernel-in-theory {X = X} {p , q} pKq vkA x = classIds-⊆-VIds {ℓ = ℓ} {p = p}{q}
                                   (class-models-kernel pKq) vkA x

module _ {X : Type χ} {𝐀 : Algebra α ρᵃ}{sA : 𝐀 ∈ S {β = α}{ρᵃ} ℓ 𝒦} where
 open Environment 𝐀 using ( Equal )
 ker𝔽⊆Equal : ∀{p q} → (p , q) ∈ ker ∣ hom𝔽[ X ] ∣ → Equal p q
 ker𝔽⊆Equal{p = p}{q} x = S-id1{ℓ = ℓ}{p = p}{q} (𝓔⊢[ X ]▹Th𝒦 x) 𝐀 sA

𝒦⊫→𝓔⊢ : {X : Type χ} → ∀{p q} → 𝒦 ⊫ (p ≈̇ q) → 𝓔 ⊢ X ▹ p ≈ q
```

```
𝒦|=→ℰ⊢ {p = p} {q} pKq = hyp ((p ≈ · q) , pKq) where open _⊢_▹_≈_ using (hyp)
```

9.3.3 The universal property

```
module _ {𝐀 : Algebra (α ⊔ ρᵃ ⊔ ℓ) (α ⊔ ρᵃ ⊔ ℓ)}
         {𝒦 : Pred(Algebra α ρᵃ) (α ⊔ ρᵃ ⊔ ov ℓ)} where
 private ι = ov(α ⊔ ρᵃ ⊔ ℓ)
 open FreeHom {ℓ = ℓ}(α ⊔ ρᵃ ⊔ ℓ) {𝒦}
 open FreeAlgebra {ι = ι}{I = ℐ} ℰ using ( 𝔽[_] )
 open Algebra 𝐀 using() renaming ( Interp to InterpA )
 open Setoid 𝔻[ 𝐀 ] using ( trans ; sym ; refl ) renaming ( Carrier to A )

 𝔽-ModTh-epi : 𝐀 ∈ Mod (Th (V ℓ ι 𝒦))
  → epi 𝔽[ A ] 𝐀
 𝔽-ModTh-epi A∈ModThK = φ , isEpi
  where
   φ : 𝔻[ 𝔽[ A ] ] ⟶ 𝔻[ 𝐀 ]
   _⟨$⟩_ φ = free-lift{𝐀 = 𝐀} id
   cong φ {p} {q} pq = trans (sym (free-lift-interp{𝐀 = 𝐀} id p))
                          (trans (A∈ModThK{p = p}{q} (kernel-in-theory pq) id)
                          (free-lift-interp{𝐀 = 𝐀} id q))

   isEpi : IsEpi 𝔽[ A ] 𝐀 φ
   compatible (isHom isEpi) = cong InterpA (≡.refl , (λ _ → refl))
   isSurjective isEpi {y} = eq (ℊ y) refl

 𝔽-ModTh-epi-lift : 𝐀 ∈ Mod (Th (V ℓ ι 𝒦)) → epi 𝔽[ A ] (Lift-Alg 𝐀 ι ι)
 𝔽-ModTh-epi-lift A∈ModThK = ∘-epi (𝔽-ModTh-epi (λ {p q} → A∈ModThK{p = p}{q})) ToLift-epi
```

10 Products of classes of algebras

We want to pair each ($\mathbf{A}$, p) (where p : $\mathbf{A} \in$ S $\mathcal{K}$) with an environment ρ : X → | $\mathbf{A}$ | so that we can quantify over all algebras *and* all assignments of values in the domain | $\mathbf{A}$ | to variables in X.

```
module _ (𝒦 : Pred(Algebra α ρᵃ) (α ⊔ ρᵃ ⊔ ov ℓ)){X : Type (α ⊔ ρᵃ ⊔ ℓ)} where
 private ι = ov(α ⊔ ρᵃ ⊔ ℓ)
 open FreeHom {ℓ = ℓ} (α ⊔ ρᵃ ⊔ ℓ){𝒦}
 open FreeAlgebra {ι = ι}{I = ℐ} ℰ using ( 𝔽[_] )
 open Environment using ( Env )

 ℑ⁺ : Type ι
 ℑ⁺ = Σ[ 𝐀 ∈ (Algebra α ρᵃ) ] (𝐀 ∈ S ℓ 𝒦) × (Carrier (Env 𝐀 X))

 𝔄⁺ : ℑ⁺ → Algebra α ρᵃ
 𝔄⁺ i = | i |

 ℭ : Algebra ι ι
 ℭ = ⨅ 𝔄⁺
```

Next we define a useful type, skEqual, which we use to represent a term identity p ≈ q for any given i = ($\mathbf{A}$, sA , ρ) (where $\mathbf{A}$ is an algebra, sA : $\mathbf{A}$ ∈ S $\mathscr{K}$ is a proof that $\mathbf{A}$ belongs to S $\mathscr{K}$, and ρ is a mapping from X to the domain of $\mathbf{A}$). Then we prove AllEqual⊆ker𝔽 which asserts that if the identity p ≈ q holds in all $\mathbf{A}$ ∈ S $\mathscr{K}$ (for all environments), then p ≈ q holds in the relatively free algebra 𝔽[X]; equivalently, the pair (p , q) belongs to the kernel of the natural homomorphism from $\mathbf{T}$ X onto 𝔽[X]. We will use this fact below to prove that there is a monomorphism from 𝔽[X] into ℭ, and thus 𝔽[X] is a subalgebra of ℭ, so belongs to S (P $\mathscr{K}$).

```
skEqual : (i : ℑ⁺) → ∀{p q} → Type ρᵃ
skEqual i {p}{q} = ⟦ p ⟧ ⟨$⟩ snd ∥ i ∥ ≈ ⟦ q ⟧ ⟨$⟩ snd ∥ i ∥
  where
  open Setoid 𝔻[ 𝔄⁺ i ] using ( _≈_ )
  open Environment (𝔄⁺ i) using ( ⟦_⟧ )

AllEqual⊆ker𝔽 : ∀ {p q} → (∀ i → skEqual i {p}{q}) → (p , q) ∈ ker ∣ hom𝔽[ X ] ∣
AllEqual⊆ker𝔽 {p} {q} x = Goal
  where
  open Algebra 𝔽[ X ] using () renaming ( Domain to F ; Interp to InterpF )
  open Setoid F using () renaming ( _≈_ to _≈F≈_ ; refl to reflF )
  S𝒦|=pq : S{β = α}{ρᵃ} ℓ 𝒦 |= (p ≈̇ q)
  S𝒦|=pq 𝐀 sA ρ = x (𝐀 , sA , ρ)
  Goal : p ≈F≈ q
  Goal = 𝒦|=→ℰ⊢ (S-id2{ℓ = ℓ}{p = p}{q} S𝒦|=pq)

homℭ : hom (𝐓 X) ℭ
homℭ = ⨅-hom-co 𝔄⁺ h
  where
  h : ∀ i → hom (𝐓 X) (𝔄⁺ i)
  h i = lift-hom (snd ∥ i ∥)

open Algebra 𝔽[ X ] using () renaming ( Domain to F ; Interp to InterpF )
open Setoid F using () renaming (refl to reflF ; _≈_ to _≈F≈_ ; Carrier to ∣F∣)

ker𝔽⊆kerℭ : ker ∣ hom𝔽[ X ] ∣ ⊆ ker ∣ homℭ ∣
ker𝔽⊆kerℭ {p , q} pKq (𝐀 , sA , ρ) = Goal
  where
  open Setoid 𝔻[ 𝐀 ] using ( _≈_ ; sym ; trans )
  open Environment 𝐀 using ( ⟦_⟧ )
  fl : ∀ t → ⟦ t ⟧ ⟨$⟩ ρ ≈ free-lift ρ t
  fl t = free-lift-interp {𝐀 = 𝐀} ρ t
  subgoal : ⟦ p ⟧ ⟨$⟩ ρ ≈ ⟦ q ⟧ ⟨$⟩ ρ
  subgoal = ker𝔽⊆Equal{𝐀 = 𝐀}{sA} pKq ρ
  Goal : (free-lift{𝐀 = 𝐀} ρ p) ≈ (free-lift{𝐀 = 𝐀} ρ q)
  Goal = trans (sym (fl p)) (trans subgoal (fl q))


hom𝔽ℭ : hom 𝔽[ X ] ℭ
hom𝔽ℭ = ∣ HomFactor ℭ homℭ hom𝔽[ X ] ker𝔽⊆kerℭ hom𝔽[ X ]-is-epic ∣

open Environment ℭ

kerℭ⊆ker𝔽 : ∀{p q} → (p , q) ∈ ker ∣ homℭ ∣ → (p , q) ∈ ker ∣ hom𝔽[ X ] ∣
kerℭ⊆ker𝔽 {p}{q} pKq = E⊢pq
```

```
  where
  pqEqual : ∀ i → skEqual i {p}{q}
  pqEqual i = goal
   where
   open Environment (𝔄⁺ i) using () renaming ( ⟦_⟧ to ⟦_⟧ᵢ )
   open Setoid 𝔻[ 𝔄⁺ i ] using ( _≈_ ; sym ; trans )
   goal : ⟦ p ⟧ᵢ ⟨$⟩ snd ∥ i ∥ ≈ ⟦ q ⟧ᵢ ⟨$⟩ snd ∥ i ∥
   goal = trans (free-lift-interp{𝐀 = ∣ i ∣}(snd ∥ i ∥) p)
              (trans (pKq i)(sym (free-lift-interp{𝐀 = ∣ i ∣} (snd ∥ i ∥) q)))
  E⊢pq : ℰ ⊢ X ▹ p ≈ q
  E⊢pq = AllEqual⊆ker𝔽 pqEqual

 mon𝔽ℭ : mon 𝔽[ X ] ℭ
 mon𝔽ℭ = ∣ hom𝔽ℭ ∣ , isMon
  where
  isMon : IsMon 𝔽[ X ] ℭ ∣ hom𝔽ℭ ∣
  isHom isMon = ∥ hom𝔽ℭ ∥
  isInjective isMon {p} {q} φpq = kerℭ⊆ker𝔽 φpq
```

Now that we have proved the existence of a monomorphism from 𝔽[X] to ℭ we are in a position to prove that 𝔽[X] is a subalgebra of ℭ, so belongs to S (P 𝒦). In fact, we will show that 𝔽[X] is a subalgebra of the *lift* of ℭ, denoted ℓℭ.

```
 𝔽≤ℭ : 𝔽[ X ] ≤ ℭ
 𝔽≤ℭ = mon→≤ mon𝔽ℭ

 SP𝔽 : 𝔽[ X ] ∈ S ι (P ℓ ι 𝒦)
 SP𝔽 = S-idem SSP𝔽
  where
  PSℭ : ℭ ∈ P (α ⊔ ρᵃ ⊔ ℓ) ι (S ℓ 𝒦)
  PSℭ = ℑ⁺ , (𝔄⁺ , ((λ i → fst ∥ i ∥) , ≅-refl))
  SPℭ : ℭ ∈ S ι (P ℓ ι 𝒦)
  SPℭ = PS⊆SP {ℓ = ℓ} PSℭ
  SSP𝔽 : 𝔽[ X ] ∈ S ι (S ι (P ℓ ι 𝒦))
  SSP𝔽 = ℭ , (SPℭ , 𝔽≤ℭ)
```

11 The HSP Theorem

Finally, we are in a position to prove Birkhoff's celebrated variety theorem.

```
 module _ {𝒦 : Pred(Algebra α ρᵃ) (α ⊔ ρᵃ ⊔ ov ℓ)} where
  private ι = ov(α ⊔ ρᵃ ⊔ ℓ)
  open FreeHom {ℓ = ℓ}(α ⊔ ρᵃ ⊔ ℓ){𝒦}
  open FreeAlgebra {ι = ι}{I = ℐ} ℰ using ( 𝔽[_] )

  Birkhoff : ∀ 𝐀 → 𝐀 ∈ Mod (Th (V ℓ ι 𝒦)) → 𝐀 ∈ V ℓ ι 𝒦
  Birkhoff 𝐀 ModThA = V-≅-lc{α}{ρᵃ}{ℓ} 𝒦 𝐀 VlA
   where
   open Setoid 𝔻[ 𝐀 ] using () renaming ( Carrier to A )
   sp𝔽A : 𝔽[ A ] ∈ S{ι} ι (P ℓ ι 𝒦)
```

```
spFA = SPF{ℓ = ℓ} 𝒦
epiFlA : epi F[ A ] (Lift-Alg 𝐀 ι ι)
epiFlA = F-ModTh-epi-lift{ℓ = ℓ} (λ {p q} → ModThA{p = p}{q})
lAimgFA : Lift-Alg 𝐀 ι ι IsHomImageOf F[ A ]
lAimgFA = epi→ontohom F[ A ] (Lift-Alg 𝐀 ι ι) epiFlA
VlA : Lift-Alg 𝐀 ι ι ∈ V ℓ ι 𝒦
VlA = F[ A ] , spFA , lAimgFA
```

The converse inclusion, V 𝒦 ⊆ Mod (Th (V 𝒦)), is a simple consequence of the fact that Mod Th is a closure operator. Nonetheless, completeness demands that we formalize this inclusion as well, however trivial the proof.

```
module _ {𝐀 : Algebra α ρᵃ} where
 open Setoid 𝔻[ 𝐀 ] using () renaming ( Carrier to A )

 Birkhoff-converse : 𝐀 ∈ V{α}{ρᵃ}{α}{ρᵃ}{α}{ρᵃ} ℓ ι 𝒦 → 𝐀 ∈ Mod{X = A} (Th (V ℓ ι 𝒦))
 Birkhoff-converse vA pThq = pThq 𝐀 vA
```

We have thus proved that every variety is an equational class.

Readers familiar with the classical formulation of the Birkhoff HSP theorem as an "if and only if" assertion might worry that the proof is still incomplete. However, recall that in the Varieties.Func.Preservation module we proved the following identity preservation lemma:

V-id1 : 𝒦 |⊨ p ≈ q → V 𝒦 |⊨ p ≈ q

Thus, if 𝒦 is an equational class—that is, if 𝒦 is the class of algebras satisfying all identities in some set—then V 𝒦 ⊆ 𝒦. On the other hand, we proved thatV' is expansive in the Varieties.Func.Closure module:

V-expa : 𝒦 ⊆ V 𝒦

so 𝒦 (= V 𝒦 = HSP 𝒦) is a variety.

Taken together, V-id1 and V-expa constitute formal proof that every equational class is a variety.

This completes the formal proof of Birkhoff's variety theorem.

12 Appendix

The Setoid type is defined in the Agda Standard Library as follows.

```
record Setoid c ℓ : Set (suc (c ⊔ ℓ)) where
  field
    Carrier       : Set c
    _≈_            : Rel Carrier ℓ
    isEquivalence : IsEquivalence _≈_
```

The Func type is defined in the Agda Standard Library as follows.

```
record Func : Set (a ⊔ b ⊔ ℓ₁ ⊔ ℓ₂) where
  field
    f    : A → B
    cong : f Preserves _≈₁_ ⟶ _≈₂_

  isCongruent : IsCongruent f
```

```
isCongruent = record
  { cong           = cong
  ; isEquivalence₁ = isEquivalence From
  ; isEquivalence₂ = isEquivalence To
  }

open IsCongruent isCongruent public
  using (module Eq₁; module Eq₂)
```

Here, A and B are setoids with respective equality relations $\approx_1$ and $\approx_2$.