A Machine-checked 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 single Agda module called Demos.HSP. This module extracts only those parts of the library needed to prove Birkhoff's variety theorem. In order to meet page limit guidelines, and to reduce strain on the reader, we omit proofs of some routine or technical lemmas that do not provide much insight into the overall development. However, a long version of this paper, which includes all code in the Demos.HSPmodule, is available on the arXiv. [reference needed]

In the course of our exposition of the proof of the HSP theorem, 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 this 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. Perhpas our presentation will be viewed as a sobering glimpse of the painstaking process of doing mathematics in the languages of dependent type theory using the Agda proof assistant. Nonetheless we hope to make a compelling case for investing in these technologies. Indeed, we are excited to share the gratifying rewards that come with some mastery of type theory and interactive theorem proving.

1.1 Prior art

There have been a number of efforts to formalize parts of universal algebra in type theory prior to ours, most notably

1. In [2], Capretta formalized the basics of universal algebra in the Calculus of Inductive Constructions using the Coq proof assistant;
2. In [4], Spitters and van der Weegen formalized the basics of universal algebra and some classical algebraic structures, also in the Calculus of Inductive Constructions using the Coq proof assistant and promoting the use of type classes;

3. In [3] Gunther, et al developed what was (prior to the agda-algebras library) the most extensive library of formalized universal algebra to date; like agda-algebras, that work is based on dependent type theory, is programmed in Agda, and goes beyond the Noether isomorphism theorems to include some basic equational logic; although the coverage is less extensive than that of agda-algebras, Gunther et al do treat *multisorted* algebras, whereas agda-algebras is currently limited to single sorted structures.
4. Lynge and Spitters [@Lynge:2019] (2019) formalize basic, mutisorted universal algebra, up to the Noether isomorphism theorems, in homotopy type theory; in this setting, the authors can avoid using setoids by postulating a strong extensionality axiom called *univalence*.

Some other projects aimed at formalizing mathematics generally, and algebra in particular, have developed into very extensive libraries that include definitions, theorems, and proofs about algebraic structures, such as groups, rings, modules, etc. However, the goals of these efforts seem to be the formalization of special classical algebraic structures, as opposed to the general theory of (universal) algebras. Moreover, the part of universal algebra and equational logic formalized in the agda-algebras library extends beyond the scope of prior efforts.

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, including the present module (Demos.HSP), begins with the following line.

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

We give only very terse descriptions of these options, and refer the reader to the accompanying links for more details.

- *without-K* disables Streicher's K axiom. See the section on axiom K in the Agda Language Reference Manual [5].
- *exact-split* makes Agda accept only those definitions that behave like so-called *judgmental* equalities. See the Pattern matching and equality section of the Agda Tools documentation [7].
- *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 the cmdoption-safe section of the Agda Tools documentation and the Safe Agda section of the Agda Language Reference [6].

The OPTIONS pragma is usually followed by the start of a module and a list of import directives. For example, the collection of imports required for the present module, Demos.HSP, is relatively modest and appears below.

```
{-# 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 ; Surjection ; flip ; Injection ; _∘_ )
                                        renaming ( Func to _⟶_ )
open import Level                       using     ( Level )
open import Relation.Binary             using     ( Setoid ; IsEquivalence ; Rel )
open import Relation.Binary.Definitions using     ( Sym ; Symmetric ; Trans ; Transitive ; Reflexive )
open import Relation.Binary.PropositionalEquality
                                        using     ( _≡_ )
open import Relation.Unary              using     ( Pred ; _⊆_ ; _∈_ )

import Function.Definitions                  as FD
import Relation.Binary.PropositionalEquality as ≡
import Relation.Binary.Reasoning.Setoid      as SetoidReasoning

open _⟶_ using ( cong ) renaming ( f to _⟨$⟩_ )
```

Note, in particular, we prefer to use Type to denote the built-in Set type, and the infix long arrow symbol, _⟶_, to denote the Func type of the standard library. We use fst and snd in place of proj_1 and proj_2 for the first and second projections out of the product type, _×_, and, when it improves readability of the code, we use the alternative notation |_| and ∥_∥ (resp.) for these projections.

2.2 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 (MLTT). [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 MLTT; in particular, no function extensionality axioms are postulated in our current formalization of Birkhoff's variety theorem.

2.3 Setoid functions

In addition to the Setoid type, much of our code employs the standard library's Func 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. As mentioned above, we renamed Func to the more visually appealing infix long arrow symbol, _⟶_, and throughout the paper we refer to inhabitants of this type as "setoid functions."

Inverses of setoid functions

We define a data type that represents the semantic concept of the *image* of a function.[1]

```
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
```

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

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

The latter (InvIsInverse^r) proves that Inv f is the range-restricted right-inverse of the setoid function f.

Injective and surjective setoid functions

If f : $\mathbf{A}$ ⟶ $\mathbf{B}$ is a setoid function from $\mathbf{A}$ = (A , $\approx_0$) to $\mathbf{B}$ = (B , $\approx_1$), then we call f *injective* provided ∀ (a_0 a_1 : A), f ⟨$⟩ a_0 $\approx_1$ f ⟨$⟩ a_1 implies a_0 $\approx_0$ a_1; we call f *surjective* provided ∀ (b : B), ∃ (a : A) such that f ⟨$⟩ a $\approx_1$ b. The Agda Standard Library represents injective functions on bare types by the type Injective, and uses this to define the IsInjective type to represent the property of being an injective setoid function. Similarly, the type IsSurjective represents the property of being a surjective setoid function. SurjInv represents the *right-inverse* of a surjective function. We omit the relatively straightforward formal definitions of these types, but see the unabridged version of this paper for the complete formalization, as well as formal proofs of some of their properties.

Kernels of setoid functions

The *kernel* of a function f : A → B (where A and B are bare types) is defined informally by

$$\{(x , y) \in A \times A : f\,x = f\,y\}.$$

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.

[1] cf. the Overture.Func.Inverses module of the agda-algebras library.

```
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 *setoid* function f : $A \longrightarrow B$ is {(x , y) ∈ A × A : f ⟨$⟩ x ≈ f ⟨$⟩ y}, where _≈_ denotes equality in B. This can 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 Types for Basic Universal Algebra

In this section we develop a working vocabulary and formal types for classical, single-sorted, set-based universal algebra. We cover a number of important concepts, but we limit ourselves to those concepts required in our formal proof of Birkhoff's HSP theorem. In each case, we give a type-theoretic version of the informal definition, followed by a formal implementation of the definition in Martin-Löf dependent type theory using the Agda language.

This section is organized into the following subsections: §3.1 defines a general notion of *signature* of a structure and then defines a type that represent signatures; §§3.2–3.3 do the same for for *algebraic structures* and *product algebras*, respectively; §3.4 defines *homomorphism*, *monomorphism*, and *epimorphism*, presents types that codify these concepts and formally verifies some of their basic properties; §§3.5–3.6 do the same for *subalgebra* and *term*, respectively.

3.1 Signatures and signature types

In model theory, the *signature* $S = (C, F, R, \rho)$ of a structure consists of three (possibly empty) sets C, F, and R—called *constant*, *function*, and *relation* symbols, respectively—along with a function $\rho : C + F + R \to N$ that assigns an *arity* to each symbol. Often, but not always, N is taken to be the set of natural numbers.

As our focus here is universal algebra, we are more concerned with the restricted notion of an *algebraic signature*, that is, a signature for "purely algebraic" structures, by which is meant a pair S = (F , ρ) consisting of a collection F of *operation symbols* and a function ρ : F → N which maps each operation symbol to its arity. Here, N denotes the *arity type*. Heuristically, the arity ρ f of an operation symbol f ∈ F may be thought of as the number of arguments that f takes as "input."

The agda-algebras library represents an (algebraic) signature as an inhabitant of the following dependent pair type:

```
Signature : (𝓞 𝓥 : Level) → Type (lsuc (𝓞 ⊔ 𝓥))
Signature 𝓞 𝓥 = Σ[ F ∈ Type 𝓞 ] (F → Type 𝓥)
```

Using special syntax for the first and second projections—∣_∣ and ∥_∥ (resp.)—if S : Signature 𝓞 𝓥 is a signature, then ∣ S ∣ denotes the set of operation symbols and ∥ S ∥ denotes the arity function. Thus, if f : ∣ S ∣ is an operation symbol in the signature S, then ∥ S ∥ f is the arity of f.

We need to augment the ordinary Signature type so that it supports algebras over setoid domains. To do so, we follow Andreas Abel's lead [ref needed] and define an operator that translates an ordinary signature into a *setoid signature*, that is, a signature over a setoid domain. This raises a minor technical issue concerning the dependent types involved in the definition; some readers might find the resolution of this issue instructive, so let's discuss it.

Suppose we are given two operations f and g, a tuple u : ∥ 𝑆 ∥ f → A of arguments for f, and a tuple v : ∥ 𝑆 ∥ g → A of arguments for g. If we know that f is identically equal to g—that is, f ≡ g (intensionally)—then we should be able to check whether u and v are pointwise equal. Technically, though, u and v inhabit different types, so, before comparing them, we must first convince Agda that u and v inhabit the same type. Of course, this requires an appeal to the hypothesis f ≡ g, as we see in the definition of EqArgs below (adapted from Andreas Abel's development [ref needed]), which neatly resolves this minor technicality.

```
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 ( _≈_ )
```

Finally, we are ready to define an operator which translates an ordinary (algebraic) signature into a signature of algebras over setoids. We denote this operator by ⟨_⟩ and define it as follows.

```
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 i → 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)
```

3.2 Algebras and algebra types

Informally, an *algebraic structure in the signature* $S = (\mathsf{F}, \rho)$ (or S-*algebra*) is denoted by $\mathbf{A} = (\mathsf{A}, \mathsf{F}^{A})$ and consists of

- a *nonempty* set (or type) A, called the *domain* of the algebra;
- a collection $\mathsf{F}^A := \{ \mathsf{f}^A \mid \mathsf{f} \in \mathsf{F}, \mathsf{f}^A : (\rho\ \mathsf{f} \to \mathsf{A}) \to \mathsf{A} \}$ of *operations* on A;
- a (potentially empty) collection of *identities* satisfied by elements and operations of $\mathbf{A}$.

The agda-algebras library represents algebras as the inhabitants of a record type with two fields:

- Domain, representing the domain of the algebra;
- Interp, representing the *interpretation* in the algebra of each operation symbol in S.

We now present the definition of the Algebra type and explain how the standard library's Func type is used to represent the interpretation of operation symbols in an algebra.[2]

[2] We postpone introducing identities until they are needed (e.g., for equational logic); see §4.

```
record Algebra α ρ : Type (𝓞 ⊔ 𝓥 ⊔ lsuc (α ⊔ ρ)) where
  field Domain : Setoid α ρ
        Interp : (⟨ S ⟩ Domain) ⟶ Domain
```

Recall, we renamed Agda's Func type, prefering instead the long-arrow symbol ⟶, so the Interp field has type Func (⟨ S ⟩ Domain) Domain, a record type with two fields:

- a function f : Carrier (⟨ S ⟩ Domain) → Carrier Domain representing the operation;
- a proof cong : f Preserves _≈₁_ ⟶ _≈₂_ that the operation preserves the relevant setoid equalities.

Thus, for each operation symbol in the signature S, we have a setoid function f—with domain a power of Domain and codomain Domain—along with a proof that this function respects the setoid equalities. The latter means that the operation f is accompanied by a proof of the following: ∀ u v in Carrier (⟨ S ⟩ Domain), if u ≈₁ v, then f ⟨$⟩ u ≈₂ f ⟨$⟩ v.

In the agda-algebras library is defined some syntactic sugar that helps to make our formalizations easier to read and comprehend. The following are three examples of such syntax that we use below: if **A** is an algebra, then

- 𝔻[**A**] denotes the setoid Domain **A**,
- 𝕌[**A**] is the underlying carrier or "universe" of the algebra **A**, and
- f ̂ **A** denotes the interpretation in the algebra **A** of the operation symbol f.

We omit the straightforward formal definitions of these types, but see the unabridged version of this paper for the complete formalization.

Universe levels of algebra types

The hierarchy of type universes in Agda is structured as follows: Type ℓ : Type (lsuc ℓ), Type (lsuc ℓ) : Type (lsuc (lsuc ℓ)), ... This means that Type ℓ has type Type (lsuc ℓ), etc. However, this does *not* imply that Type ℓ : Type (lsuc (lsuc ℓ)). In other words, Agda's universe hierarchy is *noncumulative*. This can be advantageous as it becomes possible to treat universe levels more generally and precisely. On the other hand, an unfortunate side-effect of this noncumulativity is that it sometimes seems unduly difficult to convince Agda that a program or proof is correct.

This aspect of the language was one of the few stumbling blocks we encountered while learning how to use Agda for formalizing universal algebra in type theory. Although some may consider this to be one of the least interesting and most annoying aspects of our work, others might find this presentation most helpful if we resist the urge to gloss over the more technical and less elegant aspects of the library. Therefore, we will show how to use the general universe lifting and lowering functions, available in the Agda Standard Library, to develop bespoke, domain-specific tools for dealing with the noncumulative universe hierarchy.

Let us be more concrete about what is at issue here by considering a typical example. Agda frequently encounters errors during the type-checking process and responds by printing an error message. Often the message has the following form.

```
HSP.lagda:498,20-23
α != 𝓞 ⊔ 𝓥 ⊔ (lsuc α) when checking that... has type...
```

Here Agda informs us that it encountered universe level α on line 498 of the HSP module, where it was expecting level 𝓞 ⊔ 𝓥 ⊔ (lsuc α). For example, we may have tried to use an algebra inhabiting the type Algebra α ρ^a whereas we should have used one inhabiting the type Algebra (𝓞 ⊔ 𝓥 ⊔ (lsuc α)) ρ^a. One resolves such problems using the general Lift record type, available in the standard library, which takes a type inhabiting some universe and embeds it into a higher universe. To apply this strategy in our domain of interest, we develop the following utility functions.

```
module _ (𝐀 : Algebra α ρᵃ) where
 open Setoid 𝔻[ 𝐀 ] using ( _≈_ ; refl ; sym ; trans ) ; open Level

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

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

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


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

 Domain (Lift-Algʳ ℓ) =
  record { Carrier       = 𝕌[ 𝐀 ]
         ; _≈_           = λ x y → Lift ℓ (x ≈ y)
         ; isEquivalence = record { refl = lift refl
                                  ; sym = lift ∘ sym ∘ lower
                                  ; 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 , lab) = lift(cong(Interp 𝐀)(≡.refl , λ i → lower (lab 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
```

To see why these functions are useful, first recall that our definition of the algebra record type uses two universe level parameters corresponding to those of the algebra's underlying domain setoid. Concretely, an algebra of type Algebra α ρ^a has a domain setoid (called Domain) of type Setoid α ρ^a. This domain setoid packages a "carrier set" (Carrier), inhabiting Type α, with an equality on Carrier of type Rel Carrier ρ^a. Now, examining the Lift-Alg function, we see that it takes an algebra—one whose carrier set inhabits Type α and has an equality of type Rel Carrier ρ^a—and constructs a new algebra with carrier set inhabiting Type ($\alpha \sqcup \ell_0$) and having an equality of type Rel Carrier ($\rho^a \sqcup \ell_1$). Of course, this lifting operation would be useless if we couldn't establish a connection (beyond universe levels) between the input and output algebras. Fortunately, we can prove that universe lifting is an *algebraic invariant*, which is to say that the lifted algebra has the same algebraic properties as the original algebra; more precisely, the input algebra and the lifted algebra are *isomorphic*, as we prove below. (See Lift-≅.)

3.3 Product Algebras

We give an informal description of the *product* of a family of S-algebras and then define a type which formalizes this notion.

Let ι be a universe and I : Type ι a type (which, in the present context, we might refer to as the "indexing type"). Then the dependent function type $\mathscr{A}$: I → Algebra α ρ^a

represents an *indexed family of algebras*. Denote by ⨅ 𝒜 the *product of algebras* in 𝒜 (or *product algebra*), by which we mean the algebra whose domain is the Cartesian product Π i : I , 𝔻[𝒜 i] of the domains of the algebras in 𝒜, and whose operations are those arising by point-wise interpretation in the obvious way: if f is a J-ary operation symbol and if a : Π i : I , J → 𝔻[𝒜 i] is, for each i : I, a J-tuple of elements of the domain 𝔻[𝒜 i], then we define the interpretation of f in ⨅ 𝒜 by (f ^ ⨅ 𝒜) a := λ (i : I) → (f ^ 𝒜 i)(a i).

The following type definition formalizes the foregoing notion of *product algebra* in Martin-Löf type theory.[3]

```
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 )
```

3.4 Homomorphisms

Basic definitions

Suppose **A** and **B** are S-algebras. A *homomorphism* (or "hom") from **A** to **B** is a setoid function h : 𝔻[**A**] → 𝔻[**B**] that is *compatible* (or *commutes*) with all basic operations; that is, for every operation symbol f : ∣ S ∣ and all tuples a : ∥ S ∥ f → 𝔻[**A**], the following equality holds: h ⟨$⟩ (f ^ **A**) a ≈ (f ^ **B**) λ x → h ⟨$⟩ (a x).

To formalize this concept in Agda, we first define a type compatible-map-op representing the assertion that a given setoid function h : 𝔻[**A**] → 𝔻[**B**] commutes with a given basic operation f.

```
module _ (𝐀 : Algebra α ρᵃ)(𝐁 : Algebra β ρᵇ) where
 private ov = 𝓞 ⊔ 𝓥 ; a = α ⊔ ρᵃ ; b = β ⊔ ρᵇ ; c = 𝓞 ⊔ 𝓥 ⊔ α ⊔ ρᵃ ⊔ β ⊔ ρᵇ

 compatible-map-op : (𝔻[ 𝐀 ] ⟶ 𝔻[ 𝐁 ]) → ∣ 𝑆 ∣ → Type (𝓥 ⊔ α ⊔ ρᵇ)
 compatible-map-op h f = ∀ {a} → h ⟨$⟩ (f ^ 𝐀) a ≈ (f ^ 𝐁) λ x → h ⟨$⟩ (a x)
  where open Setoid 𝔻[ 𝐁 ] using ( _≈_ )
```

Generalizing over operation symbols gives the following type of compatible maps from (the domain of) A to (the domain of) **B**.

```
compatible-map : (𝔻[ 𝐀 ] ⟶ 𝔻[ 𝐁 ]) → Type (ov ⊔ α ⊔ ρᵇ)
compatible-map h = ∀ {f} → compatible-map-op h f
```

With this we define a record type IsHom representing the property of being a homomorphism, and finally the type hom of homomorphisms from **A** to B.

[3] cf. the Algebras.Func.Products module of the agda-algebras library.

```
record IsHom (h : 𝔻[ 𝐀 ] ⟶ 𝔻[ 𝐁 ]) : Type (ov ⊔ α ⊔ ρᵇ) where
  constructor mkhom
  field compatible : compatible-map h

hom : Type c
hom = Σ (𝔻[ 𝐀 ] ⟶ 𝔻[ 𝐁 ]) IsHom
```

Observe that an inhabitant of hom is a pair (h , p) whose first component is a setoid function from the domain of **A** to the domain of B and whose second component is a proof, p : IsHom h, that h is a homomorphism.

A *monomorphism* (resp. *epimorphism*) is an injective (resp. surjective) homomorphism. The agda-algebras library defines types IsMon and IsEpi to represent these properties, as well as mon and epi, the types of monomorphisms and epimorphisms, respectively. We won't reproduce the formal definitions of these types here, but see the unabridged version of this paper for the complete formalization.

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 = mkhom 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 g h (isSurjective gE) (isSurjective hE) }
```

Factorization of homomorphisms

If g : hom **A** **B**, h : hom **A** **C**, h is surjective, and ker h ⊆ ker g, then there exists φ : hom **C** **B** such that g = φ ∘ h.

Here we merely give the formal statement of this theorem, but see the unabridged version of this paper for the complete formalization or the Homomorphisms.Func.Factor module of the agda-algebras library.

```
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 )
  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
```

Isomorphisms

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

The agda-algebras library's _≅_ type codifies the definition of isomorphism, as well as some obvious consequences. Here we display only the core part of this record type, but see the unabridged version of this paper for the complete formalization or the Homomorphisms.Func.Isomorphisms module of the agda-algebras library.

```
module _ (𝐀 : Algebra α ρᵃ) (𝐁 : Algebra β ρᵇ) where
  open Setoid 𝔻[ 𝐀 ] using ( _≈_ ; sym ; trans )
  open Setoid 𝔻[ 𝐁 ] using () renaming ( _≈_ to _≈ᴮ_ ; sym to symᵇ ; trans to transᵇ)

  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) ≈ᴮ b
      from∼to : ∀ a → ∣ from ∣ ⟨$⟩ (∣ to ∣   ⟨$⟩ a) ≈ a
```

Homomorphic Images

We begin with what for our purposes is the most useful way to represent the class of *homomorphic images* of an algebra in dependent type theory.[4]

```
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.

3.5 Subalgebras

Basic definitions

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

Basic properties

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

[4] cf. the Homomorphisms.Func.HomomorphicImages module of the agda-algebras library.

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

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

Products of subalgebras

```
module _ {ι : Level} {I : Type ι}{𝒜 : I → Algebra α ρᵃ}{ℬ : I → Algebra β ρᵇ} where

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

  hfunc : 𝔻[ ⨅ ℬ ] ⟶ 𝔻[ ⨅ 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 hfunc
  hM = λ xy i → ∥ B≤A i ∥ (xy i)
```

3.6 Terms

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 $X \cap \mid S \mid = \emptyset$. By a *word* in the language of S, we mean a nonempty, finite sequence of members of $X \cup \mid S \mid$. 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 $X \cup \mid S \mid$ as follows (cf. [1, Def. 4.19]):

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

where $\mathscr{T}_n$ is the collection of all f t such that $f : \mid S \mid$ and $t : \parallel S \parallel f \to T_n$. (Recall, $\parallel S \parallel f$ 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
  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 (ℊ); hence the constructor names (ℊ for "generator" and node for "node").

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.

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

 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)
```

It is easy to show that the equality-of-terms relation _≃_ is an equivalence relation, so we omit the formal proof. (See the Terms.Func.Basic module of the agda-algebras library for details.)

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
```

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.[5]

A substitution from X to Y associates a term in X with each variable in Y. The definition of Sub given here is essentially the same as the one given by Andreas Abel, as is the recursive definition of the syntax t [σ] , which denotes a term t applied to a substitution σ.

```
Sub : Type χ → Type χ → Type (ov χ)
Sub X Y = (y : Y) → Term X

_[_] : {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 an algebra **A** in a context X is a map that assigns to each variable x : X an element in the domain of **A**, packaged together with an equality of environments, which is simply pointwise equality (relatively to the setoid equality of the underlying domain of **A**).

```
module Environment (A : Algebra α ℓ) where
 open Setoid 𝔻[ A ] using ( _≈_ ; refl ; sym ; trans )
 Env : Type χ → Setoid _ _
 Env X = record { Carrier = X → 𝕌[ A ]
                ; _≈_ = λ ρ ρ' → (x : X) → ρ x ≈ ρ' x
                ; isEquivalence = record { refl  = λ _ → refl
                                         ; sym   = λ h x → sym (h x)
                                         ; trans = λ g h x → trans (g x)(h x) }}
```

Next we define *evaluation of a term* in an environment ρ, interpreted in the algebra **A**.

```
⟦_⟧ : {X : Type χ}(t : Term X) → (Env X) ⟶ 𝔻[ 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.[6]

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

[5] See http://www.cse.chalmers.se/~abela/agda/MultiSortedAlgebra.pdf.
[6] cf. Andreas Abel's formal proof of Birkhoff's completeness theorem [reference needed].

The proof that Equal is an equivalence relation is trivial, so we omit it. (See the Varieties.Func.SoundAndComplete module of the agda-algebras library for details.)

Evaluation of a substitution gives an environment.[7]

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

Next we prove that ⟦ t [σ] ⟧ ρ ≃ ⟦ t ⟧ ⟦ σ ⟧ ρ.

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

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 Setoid 𝔻[ B ] using ( _≈_ ; refl )
  open SetoidReasoning 𝔻[ B ]
  private hfunc = ∣ hh ∣ ; h = _⟨$⟩_ hfunc
  open Environment A using ( ⟦_⟧ )
  open Environment B using () renaming ( ⟦_⟧ to ⟦_⟧ᴮ )

  comm-hom-term : (t : Term X) (a : X → 𝕌[ A ]) → h (⟦ t ⟧ ⟨$⟩ a) ≈ ⟦ t ⟧ᴮ ⟨$⟩ (h ∘ a)
  comm-hom-term (ℊ x) a = refl
  comm-hom-term (node f t) a = goal
    where
    goal : h (⟦ node f t ⟧ ⟨$⟩ a) ≈ ⟦ node f t ⟧ᴮ ⟨$⟩ (h ∘ a)
    goal = begin
              h ( ⟦ node f t ⟧   ⟨$⟩       a ) ≈⟨ compatible ∥ hh ∥ ⟩
              (f ̂ B)( λ i → h( ⟦ t i ⟧   ⟨$⟩       a) ) ≈⟨ cong(Interp B)(≡.refl , λ i → comm-hom-term (t i) a) ⟩
              (f ̂ B)( λ i →     ⟦ t i ⟧ᴮ ⟨$⟩ (h ∘ a) ) ≈⟨ refl                                                  ⟩
                 ⟦ node f t ⟧ᴮ ⟨$⟩ (h ∘ a) ∎
```

4 Model Theory and Equational Logic

(cf. the Varieties.Func.SoundAndComplete module of the agda-algebras library)

4.1 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.

[7] cf. Andreas Abel's formal proof of Birkhoff's completeness theorem [reference needed].

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.[8]

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

infix 8 _≐_
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 **A**.

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

If 𝒦 is a class of algebras, all of the same signature, we write 𝒦 |⊨ (p ≐ q) if, for every **A** ∈ 𝒦, we have **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 𝒦 ⊨ p ≈ q, we have settled on 𝒦 |⊨ (p ≐ q) to denote this relation. To reiterate, if 𝒦 is a class of S-algebras, we write 𝒦 |⊨ (p ≐ q) provided every **A** ∈ 𝒦 satisfies **A** ⊨ (p ≐ q).

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

_⊨_≈_ : Algebra α ρᵃ → Term Γ → Term Γ → 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

_|⊨_≈_ : Pred (Algebra α ρᵃ) ℓ → Term Γ → Term Γ → Type _
𝒦 |⊨ p ≈ q = ∀ A → 𝒦 A → A ⊨ p ≈ q
```

We denote by **A** ⊧ ℰ the assertion that the algebra **A** models every equation in a collection 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
```

4.2 Equational theories and models

If 𝒦 denotes a class of structures, then Th 𝒦 represents the set of identities modeled by the members of 𝒦.

[8] cf. Andreas Abel's formal proof of Birkhoff's completeness theorem [reference needed].

```
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
```

4.3 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 }
```

4.4 Soundness

In any model $\mathbf{A}$ of the equations $\mathscr{E}$ derived equality is actual equality.[9]

```
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 SetoidReasoning 𝔻[ 𝐀 ]
 open Environment 𝐀
 open IsEquivalence using ( refl ; sym ; trans )

 sound : ∀ {p q} → ℰ ⊢ Γ ▹ p ≈ q → 𝐀 ⊨ p ≈ q
 sound (hyp i) = V i
 sound (app es) ρ = cong (Interp 𝐀) (≡.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 ⟧s ρ ≈⟨  sound Epq (⟦ σ ⟧s ρ) ⟩
   ⟦ q       ⟧ ⟨$⟩ ⟦ σ ⟧s ρ ≈˘⟨ substitution q σ ρ    ⟩
   ⟦ q [ σ ] ⟧ ⟨$⟩          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)
```

[9] cf. Andreas Abel's Agda formalization of Birkhoff's completeness theorem [ref needed].

4.5 The Closure Operators H, S, P and V

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

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

A straight-forward verification confirms that H, S, and P are *closure operators* (expansive, monotone, and idempotent). A class $\mathscr{K}$ of S-algebras is said to be *closed under the taking of homomorphic images* provided H $\mathscr{K}$ ⊆ $\mathscr{K}$. Similarly, $\mathscr{K}$ is *closed under the taking of subalgebras* (resp., *arbitrary products*) provided S $\mathscr{K}$ ⊆ $\mathscr{K}$ (resp., P $\mathscr{K}$ ⊆ $\mathscr{K}$). 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 $\mathscr{K}$ (resp., S $\mathscr{K}$; resp., P $\mathscr{K}$) 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. Thus, if $\mathscr{K}$ is a class of S-algebras‘, then V $\mathscr{K}$:= H (S (P $\mathscr{K}$)), and $\mathscr{K}$ is a variety iff V $\mathscr{K}$ ⊆ $\mathscr{K}$.

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.

```
module _ {α ρᵃ β ρᵇ : Level} where
 private a = α ⊔ ρᵃ ; b = β ⊔ ρᵇ

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

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

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

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))
```

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)
```

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

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
  private hh = ∣ B≤A ∣ ; h = _⟨$⟩_ ∣ hh ∣
  open Setoid 𝔻[ 𝑨 ]   using ( _≈_ )
  open Setoid 𝔻[ 𝑩 ]   using () renaming ( _≈_ to _≈ᴮ_ )
  open Environment 𝑨 using () renaming ( ⟦_⟧ to ⟦_⟧ᴬ )
  open Environment 𝑩 using ( ⟦_⟧ )
  open SetoidReasoning 𝔻[ 𝑨 ]

  ξ : ∀ b → h (⟦ p ⟧ ⟨$⟩ b) ≈ h (⟦ q ⟧ ⟨$⟩ b)
  ξ b = begin
         h (⟦ p ⟧ ⟨$⟩      b) ≈⟨  comm-hom-term hh p b ⟩
         ⟦ p ⟧ᴬ   ⟨$⟩ (h ∘ b) ≈⟨  Apq (h ∘ b)             ⟩
         ⟦ q ⟧ᴬ   ⟨$⟩ (h ∘ b) ≈˘⟨ comm-hom-term hh q b ⟩
         h (⟦ q ⟧ ⟨$⟩      b) ∎

  goal : ⟦ p ⟧ ⟨$⟩ b ≈ᴮ ⟦ q ⟧ ⟨$⟩ b
  goal = ∥ B≤A ∥ (ξ b)
```

Product invariance of ⊧

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 =
  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               ∎
  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)
```

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 𝒦 is a subalgebra of a product of algebras in 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)
```

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-algebras library.)

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 =
  begin
   ⟦ p ⟧       ⟨$⟩             ρ    ≈˘⟨ cong ⟦ p ⟧ ζ                   ⟩
   ⟦ p ⟧       ⟨$⟩ (φ      ∘ φ⁻¹ ∘ ρ)  ≈˘⟨ comm-hom-term φh p (φ⁻¹ ∘ ρ) ⟩
   φ ( ⟦ p ⟧ᴬ ⟨$⟩ (      φ⁻¹ ∘ ρ) ) ≈⟨  cong ∣ φh ∣ (IH (φ⁻¹ ∘ ρ))     ⟩
   φ ( ⟦ q ⟧ᴬ ⟨$⟩ (      φ⁻¹ ∘ ρ) ) ≈⟨  comm-hom-term φh q (φ⁻¹ ∘ ρ) ⟩
   ⟦ q ⟧       ⟨$⟩ (φ      ∘ φ⁻¹ ∘ ρ)  ≈⟨  cong ⟦ q ⟧ ζ                   ⟩
   ⟦ q ⟧       ⟨$⟩             ρ    ∎
    where
    open Environment 𝐀 using () renaming ( ⟦_⟧ to ⟦_⟧ᴬ)
    open Environment 𝐁 using ( ⟦_⟧ )
    open Setoid 𝔻[ 𝐁 ]  using () renaming ( _≈_ to _≈ᴮ_ )
    open SetoidReasoning 𝔻[ 𝐁 ]

    IH : 𝐀 ⊨ p ≈ q
    IH = σ 𝐀 kA

    φh : hom 𝐀 𝐁
    φh = ∣ BimgOfA ∣
    private φ = (_⟨$⟩_ ∣ φh ∣)

    φE : IsSurjective ∣ φh ∣
    φE = ∥ BimgOfA ∥

    φ⁻¹ : 𝕌[ 𝐁 ] → 𝕌[ 𝐀 ]
    φ⁻¹ = SurjInv ∣ φh ∣ φE

    ζ : ∀ x → (φ ∘ φ⁻¹ ∘ ρ) x ≈ᴮ ρ x
    ζ = λ _ → InvIsInverseʳ φE
```

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))
```

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
```

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)
```

Th 𝒦 ⊆ Th (V 𝒦)

From V-id1 it follows that if 𝒦 is a class of algebras, then the set of identities modeled by the algebras in 𝒦 is contained in the set of identities modeled by the algebras in V 𝒦. In other terms, Th 𝒦 ⊆ Th (V 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
```

5 Free Algebras and the HSP Theorem

5.1 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.

- Every function from X to ∣ **A** ∣ lifts to a homomorphism from **T** X to **A**.
- 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 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 (Interp 𝐀) (≡.refl , (λ i → flcong (x i)))
```

Naturally, at the base step of the induction, when the term has the form ℊ 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 (Interp 𝐀) (≡.refl , (λ i → (cong free-lift-func){a i} ≃-isRefl))

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

module _ {X : Type χ}{𝐀 : Algebra α ρᵃ} where
 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 (Interp 𝐀) (≡.refl , (free-lift-interp η) ∘ t)
```

5.2 The relatively free algebra 𝔽[X]

We now define the algebra 𝔽[X], which represents the relatively free algebra. Here, as above, X plays the role of an arbitrary nonempty collection of variables. (It would suffice to takeX 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 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
```

5.3 Basic properties of free algebras

```
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[_] )
```

The natural epimorphism from 𝐓 X to 𝔽[X]

We now define the natural epimorphism from 𝐓 X onto the relatively free algebra 𝔽[X] and prove that the kernel of this morphism is the congruence of 𝐓 X defined by the identities modeled by (S 𝒦, hence by) 𝒦.

```
epi𝔽[_] : (X : Type χ) → epi (𝐓 X) 𝔽[ X ]
epi𝔽[ X ] = h , hepi
  where
  open Algebra (𝐓 X) using () renaming ( Domain to TX     )
  open Algebra 𝔽[ X ] using () renaming ( Domain to F      )
  open Setoid TX      using () renaming ( _≈_    to _≈₀_ ; refl to refl₀ )
  open Setoid F       using () renaming ( _≈_    to _≈₁_ ; refl to refl₁ )
  open _≃_

  c : ∀ {x y} → x ≈₀ y → x ≈₁ y
  c (rfl {x}{y} ≡.refl) = refl₁
  c (gnl {f}{s}{t} x) = cong (Interp 𝔽[ X ]) (≡.refl , c ∘ x)

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

  hepi : IsEpi (𝐓 X) 𝔽[ X ] h
  compatible (isHom hepi) = cong h refl₀
  isSurjective hepi {y} = eq y refl₁

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 ]))
```

As promised, we prove that the kernel of the natural epimorphism is the congruence defined by the identities modelled by $\mathscr{K}$.

```
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)
```

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 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 (Interp 𝑨) (≡.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
```

5.4 Products of classes of algebras

We want to pair each (𝑨 , p) (where p : 𝑨 ∈ S 𝒦) with an environment ρ : X → ∣ 𝑨 ∣ so that we can quantify over all algebras *and* all assignments of values in the domain ∣ 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 = (𝑨 , sA , ρ) (where 𝑨 is an algebra, sA : 𝑨 ∈ S 𝒦 is a proof that 𝑨 belongs to S 𝒦, and ρ is a mapping from X to the domain of 𝑨). Then we prove AllEqual⊆ker𝔽 which asserts that if the identity p ≈ q holds in all 𝑨 ∈ S 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 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 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
 S𝒦⊫pq : S{β = α}{ρᵃ} ℓ 𝒦 ⊫ p ≈ q
 S𝒦⊫pq 𝑨 sA ρ = x (𝑨 , sA , ρ)
 open Setoid 𝔻[ 𝔽[ X ] ] using ( _≈_ )
 Goal : p ≈ 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 ∥)

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))
```

```
homF𝔆 : hom 𝔽[ X ] 𝔆
homF𝔆 = ∣ HomFactor 𝔆 hom𝔆 hom𝔽[ X ] ker𝔽⊆ker𝔆 hom𝔽[ X ]-is-epic ∣

ker𝔆⊆ker𝔽 : ∀{p q} → (p , q) ∈ ker ∣ hom𝔆 ∣ → (p , q) ∈ ker ∣ hom𝔽[ X ] ∣
ker𝔆⊆ker𝔽 {p}{q} pKq = E⊢pq
  where
  pqEqual : ∀ i → skEqual i {p}{q}
  pqEqual i = goal
    where
    open Environment (𝔄⁺ i) using ( ⟦_⟧ )
    open Setoid 𝔻[ 𝔄⁺ i ]      using ( _≈_ ; sym ; trans )
    goal : ⟦ p ⟧ ⟨$⟩ snd ∥ i ∥ ≈ ⟦ q ⟧ ⟨$⟩ 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

mon𝔽𝔆 : mon 𝔽[ X ] 𝔆
mon𝔽𝔆 = ∣ hom𝔽𝔆 ∣ , isMon
  where
  isMon : IsMon 𝔽[ X ] 𝔆 ∣ hom𝔽𝔆 ∣
  isHom isMon = ∥ hom𝔽𝔆 ∥
  isInjective isMon {p} {q} φpq = ker𝔆⊆ker𝔽 φpq
```

Now that we have proved the existence of a monomorphism from 𝔽[X] to 𝔆 we can prove that 𝔽[X] is a subalgebra of 𝔆, so belongs to S (P 𝒦).

```
𝔽≤𝔆 : 𝔽[ X ] ≤ 𝔆
𝔽≤𝔆 = mon→≤ mon𝔽𝔆

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

5.5 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
  Birkhoff-converse : 𝐀 ∈ V{α}{ρᵃ}{α}{ρᵃ}{α}{ρᵃ} ℓ ι 𝒦 → 𝐀 ∈ Mod{X = 𝕌[ 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 that V is expansive in the Varieties.Func.Closure module:

V-expa : 𝒦 ⊆ V 𝒦

so 𝒦 (= V 𝒦 = H S P 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.

References

1 Clifford Bergman. *Universal Algebra: fundamentals and selected topics*, volume 301 of *Pure and Applied Mathematics (Boca Raton)*. CRC Press, Boca Raton, FL, 2012.

2 Venanzio Capretta. Universal algebra in type theory. In *Theorem proving in higher order logics (Nice, 1999)*, volume 1690 of *Lecture Notes in Comput. Sci.*, pages 131–148. Springer, Berlin, 1999. doi:10.1007/3-540-48256-3_10.

3 Emmanuel Gunther, Alejandro Gadea, and Miguel Pagano. Formalization of universal algebra in Agda. *Electronic Notes in Theoretical Computer Science*, 338:147 – 166, 2018. The 12th Workshop on Logical and Semantic Frameworks, with Applications (LSFA 2017). doi:https://doi.org/10.1016/j.entcs.2018.10.010.

4 Bas Spitters and Eelis Van der Weegen. Type classes for mathematics in type theory. *CoRR*, abs/1102.1323, 2011. arXiv:1102.1323.

5 The Agda Team. Agda Language Reference section on Axiom K, 2021. URL: https://agda.readthedocs.io/en/v2.6.1/language/without-k.html.

6 The Agda Team. Agda Language Reference section on Safe Agda, 2021. URL: https://agda.readthedocs.io/en/v2.6.1/language/safe-agda.html#safe-agda.

7 The Agda Team. Agda Tools Documentation section on Pattern matching and equality, 2021. URL: https://agda.readthedocs.io/en/v2.6.1/tools/command-line-options.html#pattern-matching-and-equality.