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.

```
-- Import 3 definitions from the agda-algebras library.
open import Algebras.Basic using ( 𝓞 ; 𝓥 ; Signature )
```

```
-- Import 16 definitions from the Agda Standard Library.
open import Function                                using ( id ; flip ; _∘_                 )
open import Level                                   using ( Level                           )
open import Relation.Binary                         using ( Rel ; Setoid ; IsEquivalence    )
open import Relation.Binary.Definitions             using ( Reflexive ; Symmetric           )
                                                    using ( Transitive ; Sym ; Trans        )
open import Relation.Binary.PropositionalEquality   using ( _≡_                             )
open import Relation.Unary                          using ( Pred ; _⊆_ ; _∈_                )

-- Import 23 definitions from the Agda Standard Library and rename 12 of them.
open import Agda.Primitive renaming ( Set    to Type    ) using ( _⊔_ ; lsuc                 )
open import Data.Product   renaming ( proj₁ to fst      ) using ( _×_ ; _,_ ; Σ ; Σ-syntax    )
                           renaming ( proj₂ to snd      )
open import Function       renaming ( Func  to _⟶_     ) using ( Injection ; Surjection      )
open       _⟶_             renaming ( f     to _⟨$⟩_   ) using ( cong                        )
open       Setoid          renaming ( refl  to reflˢ   ) using ( Carrier ; isEquivalence     )
                           renaming ( sym   to symˢ    )
                           renaming ( trans to transˢ  )
                           renaming ( _≈_   to _≈ˢ_    )
open       IsEquivalence   renaming ( refl  to reflᵉ   ) using (                            )
                           renaming ( sym   to symᵉ    )
                           renaming ( trans to transᵉ  )

-- Assign handles to 3 modules of the Agda Standard Library.
import     Function.Definitions                 as FD
import     Relation.Binary.PropositionalEquality as ≡
import     Relation.Binary.Reasoning.Setoid     as SetoidReasoning
```

Note that the above imports include some of the minor adjustments to "standard Agda" syntax (e.g., that of the Agda Standard Library) to suite our own tastes. Take special note of the following conventions used throughout the agda-algebras library and this paper: we use Type in place of Set, the infix long arrow symbol, _⟶_, instead of Func (the type of "setoid functions" discussed in §2.3 below), and the symbol _⟨$⟩_ in place of f (application of the map of a setoid function); we use fst and snd, and sometimes |_| and ∥_∥, to denote the first and second projections out of the product type _×_.

2.2 Setoids

A *setoid* is a pair (A, ≈) where A is a type and ≈ is an equivalence relation on A. Setoids seem to have gotten a bad wrap in some parts of the interactive theorem proving community because of the extra overhead that their use requires. However, we feel they are ideally suited to the task of representing the basic objects of informal mathematics (i.e., sets) in a constructive, type-theoretic way.

A set used informally typically comes equipped with an equivalence relation manifesting the notion of equality of elements of the set. When working informally, we often take the equivalence for granted or view it as self-evident; rarely do we take the time to define it explicitly. While this approach is well-suited to informal mathematics, formalization using a machine demands that we make nearly everything explicit, including notions of equality.

Actually, the agda-algebras library was first developed without setoids, relying exclusively on the Agda Standard Library's inductively defined equality type, _≡_, along with some

experimental, domain-specific types for equivalence classes, quotients, etc. One notable consequence of this design decision was that our formalization of many 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. A primary motivation for taking this approach is to make it clear that the library is fully constructive and confined to pure Martin-Löf dependent type theory (as defined, e.g., in [ref needed]). In particular, there are no appeals to function extensionality in the present work. Finally, we are confident that the current version[1] of the agda-algebras library is free of hidden assumptions or inconsistencies that could be used to "fool" the type-checker.

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 begin by defining an inductive type that represents the semantic concept of the *image* of a function.[2]

```
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 range-restricted right-inverse of f. For convenience, we define this inverse function and give it the name Inv.

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

For each b : B, given a pair (a , p) : Image f ∋ b witnessing the fact that b belongs to the image of f, the function Inv simply returns the witness a, which is a preimage of b under f. We can formally verify that Inv f is indeed the (range-restricted) right-inverse of f, as follows.

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

---

[1] [ref. with version information needed]
[2] cf. the Overture.Func.Inverses module of the agda-algebras library.

Injective and surjective setoid functions

If f is a setoid function from (A, $\approx_0$) to (B, $\approx_1$), then we call f *injective* provided $\forall$ (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 $\forall$ (b : B), $\exists$ (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) ∈ 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.

```
kernel : {A : Type α}{B : Type β} → 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 $S = (\mathsf{F}, \rho)$ consisting of a collection F of *operation symbols* and an *arity function* $\rho : \mathsf{F} \to \mathsf{N}$ which maps each operation symbol to its arity. Here, N denotes the *arity type*. Heuristically, the arity $\rho\ \mathsf{f}$ of an operation symbol $\mathsf{f} \in \mathsf{F}$ 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 $\mathcal{O}$ $\mathcal{V}$ is a signature, then $|\ S\ |$ denotes the set of operation symbols and $\|\ S\ \|$ denotes the arity function. Thus, if $\mathsf{f} : |\ S\ |$ is an operation symbol in the signature S, then $\|\ S\ \|\ \mathsf{f}$ 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 $\mathsf{u} : \|\ S\ \|\ \mathsf{f} \to \mathsf{A}$ of arguments for f, and a tuple $\mathsf{v} : \|\ S\ \|\ \mathsf{g} \to \mathsf{A}$ of arguments for g. If we know that f is identically equal to g—that is, $\mathsf{f} \equiv \mathsf{g}$ (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 $\mathsf{f} \equiv \mathsf{g}$, 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 $\langle\_\rangle$ and define it as follows.

```
⟨_⟩ : Signature 𝓞 𝓥 → Setoid α ρᵃ → Setoid _ _

Carrier (⟨ S ⟩ ξ)                = Σ[ f ∈ | S | ] (∥ S ∥ f → ξ .Carrier)
_≈ˢ_ (⟨ S ⟩ ξ)(f , u)(g , v) = Σ[ eqv ∈ f ≡ g ] EqArgs{ξ = ξ} eqv u v

reflᵉ  (isEquivalence (⟨ S ⟩ ξ))                    = ≡.refl , λ i → reflˢ  ξ
symᵉ   (isEquivalence (⟨ S ⟩ ξ)) (≡.refl , g)       = ≡.refl , λ i → symˢ   ξ (g i)
transᵉ (isEquivalence (⟨ S ⟩ ξ)) (≡.refl , g)(≡.refl , h) = ≡.refl , λ i → transˢ 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 **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.

The Domain is a actually a setoid whose Carrier denotes the carrier of the algebra and whose equivalence relation denotes equality of elements of the domain.

Here is the definition of the Algebra type followed by an explaination of how the standard library's Func type is used to represent the interpretation of operation symbols in an algebra.

```
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 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.

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 $\mathcal{A}$: I → Algebra α ρ^{a} represents an *indexed family of algebras*. Denote by ⨅ $\mathcal{A}$ the *product of algebras* in $\mathcal{A}$ (or *product algebra*), by which we mean the algebra whose domain is the Cartesian product Π i : I , 𝔻[$\mathcal{A}$ i] of the domains of the algebras in $\mathcal{A}$, 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[$\mathcal{A}$ i] is, for each i : I, a J-tuple of elements of the domain 𝔻[$\mathcal{A}$ i], then we define the interpretation of f in ⨅ $\mathcal{A}$ by (f ̂ ⨅ $\mathcal{A}$) a := λ (i : I) → (f ̂ $\mathcal{A}$ i)(a i).

The agda-algebras library defines a function called ⨅ which formalizes the foregoing notion of *product algebra* in Martin-Löf type theory. Here we merely display this function's interface, but see the Algebras.Func.Products module for the complete definition.

```
module _ {ι : Level}{I : Type ι } where
  ⨅ : (𝒜 : I → Algebra α ρᵃ) → Algebra (α ⊔ ι) (ρᵃ ⊔ ι)
```

3.4 Homomorphisms

Basic definitions

Suppose $\mathbf{A}$ and $\mathbf{B}$ are S-algebras. A *homomorphism* (or "hom") from $\mathbf{A}$ to $\mathbf{B}$ 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
  compatible-map-op : (𝔻[ 𝐀 ] ⟶ 𝔻[ 𝐁 ]) → ∣ 𝑆 ∣ → Type _
  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) $\mathbf{B}$.

```
compatible-map : (𝔻[ 𝐀 ] ⟶ 𝔻[ 𝐁 ]) → Type _
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 $\mathbf{A}$ to B.

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

hom : Type _
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 $\mathbf{A}$ to that of $\mathbf{B}$ and whose second component is p : IsHom h, a proof 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.

The composition of homomorphisms is again a homomorphism, and similarly for epimorphisms (and monomorphisms). The proofs of these facts are relatively straightforward so we omit them. When applied below, they are called ∘-hom and ∘-epi.

Another basic but important fact about homomorphisms is the following factorization theorem: 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. The type HomFactor, defined below, formalizes this result in MLTT. Here we merely give a formal statement of this theorem.

```
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_ )
  open Setoid 𝔻[ C ] using () renaming ( _≈_ to _≈₃_ )
  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 ( _≈_ )
  open Setoid 𝔻[ B ] using () renaming ( _≈_ to _≈ᴮ_ )

  record _≅_ : Type (𝓞 ⊔ 𝓥 ⊔ α ⊔ ρᵃ ⊔ β ⊔ 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
```

We conclude this section on homomorphisms with what seems, for our purposes, the most useful way to represent the class of *homomorphic images* of an algebra in dependent type theory. (The first function, ov, merely provides a handy shorthand for universe levels.)

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

_IsHomImageOf_ : (B : Algebra β ρᵇ)(A : Algebra α ρᵃ) → Type _
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
```

3.5 Subalgebras

Basic definitions

Given S-algebras $\mathbf{A}$ and $\mathbf{B}$, we say that $\mathbf{A}$ is a *subalgebra* of $\mathbf{A}$ and write $\mathbf{A} \leq \mathbf{B}$ just in case $\mathbf{A}$ can be *homomorphically embedded* in $\mathbf{B}$; in other terms, $\mathbf{A} \leq \mathbf{B}$ iff there exists a monomorphism h : mon A B from $\mathbf{A}$ to $\mathbf{B}$.

The following definition codifies the binary subalgebra relation _≤_ on the class of S-algebras in MLTT.

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

Obviously the subalgebra relation is reflexive by the identity monomorphism, as well as transitive since composition of monomorphisms is a monomorphism. Here we merely give the formal statements, but omit the easy proofs, of these results.

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

≤-transitive : {A : Algebra α ρᵃ}{B : Algebra β ρᵇ}{C : Algebra γ ρᶜ}
  →            A ≤ B → B ≤ C → A ≤ C
```

If $\mathscr{A} : I \to$ Algebra $\alpha\ \rho^a$ and $\mathscr{B} : I \to$ Algebra $\beta\ \rho^b$ are families of S-algebras such that $\mathscr{B}\ i \leq \mathscr{A}\ i$ for every $i : I$, then $\prod \mathscr{B}$ is a subalgebra of $\prod \mathscr{A}$. We omit the straightforward proof and merely assign the formalization of this result the name $\prod$-$\leq$ for future reference. We conclude this brief subsection on subalgebras with two easy facts that will be useful later, when we prove the HSP theorem. The first merely converts a monomorphism into a pair in the subalgebra relation while the second is an algebraic invariance property of $\_\leq\_$. (Proofs omitted.)

```
mon→≤  : {A : Algebra α ρᵃ}{B : Algebra β ρᵇ} → mon A B → A ≤ B

≅-trans-≤ : {A : Algebra α ρᵃ}{B : Algebra β ρᵇ}{C : Algebra γ ρᶜ}
  →         A ≅ B → B ≤ C → A ≤ C
```

3.6 Terms

Basic definitions

Fix a signature S and let X denote an arbitrary nonempty collection of variable symbols. (The chosen collection of variable symbols is sometimes called the *context*.) Assume the symbols in X are distinct from the operation symbols of S, that is $X \cap \mid S \mid = \emptyset$.

A *word* in the language of S is a 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]): $T_0 := X \cup S_0$ and $T_{n+1} := T_n \cup \mathcal{T}_n$, where $\mathcal{T}_n$ 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.) An *S-term* is a term in the language of S and the collection of all *S-terms* in the context X is given by Term $X := \bigcup_n T_n$.

As even its informal definition of Term X is recursive, it should come as no surprise that the semantics of terms can be faithfully represented in type theory as an inductive type. Indeed, here is such a representation.

```
data Term (X : Type χ ) : Type (ov χ) where
  ℊ : X → Term X
  node : (f : ∣ S ∣)(t : ∥ S ∥ f → Term X) → Term X
```

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 ($\mathcal{g}$); hence the constructor names ($\mathcal{g}$ for "generator" and node for "node"). We will enrich this type with an inductive type $\_\simeq\_$ representing equality of terms, and then we will package up Term, $\_\simeq\_$, and a proof that

_≃_ is an equivalence relation into a setoid of S-terms. Ultimately we will use this term setoid as the domain of an algebra—the (absolutely free) *term algebra* in the signature S.

First, the equality-of-terms type is defined as follows.

```
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)
```

Next, we would show that equality of terms so defined is an equivalence relation, but the proof of this fact is trivial, so we omit it and merely give the fact a name; call it ≃-isEquiv.

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 : $|\, S \,|$, we denote by f ̂ **T** X the operation on Term X that maps each tuple of terms, say, t : $\| S \|$ f → Term X, to the formal term f t. We let **T** X denote the term algebra in S over X; it has universe Term X and operations f ̂ **T** X, one for each symbol f in $|\, S \,|$. Finally, we formalize this notion of term algebra in Agda as follows.

```
TermSetoid : (X : Type χ) → Setoid _ _
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
```

Environments and the interpretation of terms therein

In this section, we formalize the notions *environment* and *interpretation of terms* in an algebra, evaluated in an environment. The approach to formalizing these notions, as well as the Agda code presented in this subsection, is based on similar code developed by Andreas Abel to formalize Birkhoff's completeness theorem.[3]

Fix a signature S, a context of variable symbols X, and an S-algebra **A**. An *environment* for these data is a function ρ : X → 𝕌[**A**] which assigns a value in the universe to each variable symbol in the context. We represent the notion of environment in Agda using a function, Env, which takes an algebra **A** and a context X and returns a setoid whose Carrier has type X → 𝕌[**A**] and whose equivalence relation is pointwise equality of functions in X → 𝕌[**A**] (relative to the setoid equality of 𝔻[**A**]).

Before defining the Env function (which will depend on a specific algebra) we first define a substitution from one context, say, X, to another Y, which assigns a term in X to each symbol in Y. The definition of Sub (which does not depend on a specific algebra) is a slight

[3] See http://www.cse.chalmers.se/~abela/agda/MultiSortedAlgebra.pdf.

modification of the one given by Andreas Abel (*op. cit.*), 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 _
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 ])
```

Now we are ready to define the aforementioned environment function Env as well as the recursive function ⟦_⟧ which defines the *interpretation* of a term in a given algebra, *evaluated* in a given environment. Since the next few definitions are relative to a certain fixed algebra, we put them inside a submodule called Environment so that later, when we load the environment, we can associate its definitions with different algebras.

```
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 τ → (x : X) → ρ x ≈ τ 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) ⟶ 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 )
```

Two terms interpreted in 𝐀 are proclaimed *equal* if they are equal for all environments. This equivalence of termsis formalized in Agda as follows.

```
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 )
```

The proof that Equal is an equivalence relation is trivial, so we omit it.

A substitution from one context X to another Y is used to transport an environment from X to Y and the function ⟦_⟧ defined below carries out this transportation of environments.

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

Finally, we have a substitution lemma says that ⟦ t [σ] ⟧ ρ, a term applied to a substitution and evaluated in the evironment ρ, is the same as the term evaluated in the transported environment ⟦ 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)
```

This concludes the definition of the Environment module (based on Abel's Agda proof of the completeness theorem; *op. cit.*).

Later we will need two important facts about term operations. The first, called comm-hom-term, asserts that every term commutes with every homomorphism. The second, interp-prod, shows how to express the interpretation of a term in a product algebra. We omit the formal definitions and proofs of these types, but see the Types.Func.Properties module of the agda-algebras library for details.

4 Model Theory and Equational Logic

4.1 Basic definitions

Term identities and the $\models$ relation

Given a signature S and a context of variable symbols X, a *term equation* or *identity* (in this signature and context) is an ordered pair (p , q) of S-terms. (Informally, such an equation is often denoted by p ≈ q.)

For instance, if the context is the type X : Type χ, then a term equation is a pair inhabiting the Cartesian product type Term X × Term X.

If $\mathbf{A}$ is an S-algebra we say that $\mathbf{A}$ *satisfies* p ≈ q if for all environments ρ : X → 𝔻[$\mathbf{A}$] (assigning values in the domain of $\mathbf{A}$ to variable symbols in X) we have ⟦ p ⟧ ⟨$⟩ ρ ≈ ⟦ q ⟧ ⟨$⟩ ρ. In other words, when they are interpreted in the algebra $\mathbf{A}$, the terms p and q are equal (no matter what values in $\mathbf{A}$ are assigned to variable symbols in X). In this situation, we write $\mathbf{A}$ ⊨ p ≈ q and say that $\mathbf{A}$ *models* the identity p ≈ q. If $\mathscr{K}$ is a class of algebras, all of the same signature, we write $\mathscr{K}$ ⊫ p ≈ q and say that $\mathscr{K}$ *models* the identity p ≈ q provided for every $\mathbf{A}$ ∈ $\mathscr{K}$, we have $\mathbf{A}$ ⊨ p ≈ q.

```
_⊨_≈_ : Algebra α ρᵃ → Term Γ → Term Γ → Type _
𝐀 ⊨ p ≈ q = Equal p q where open Environment 𝐀

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

We represent a collection of identities as a predicate over pairs of terms—for example, ℰ : Pred(Term X × Term X) _—and we denote by $\mathbf{A}$ ⊨ ℰ the assertion that the algebra $\mathbf{A}$ models every equation p ≈ q

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

Equational theories and classes

In (informal) equational logic, if $\mathscr{K}$ is a class of structures and ℰ a set of term identities, then the set of term equations modeled by $\mathscr{K}$ is denoted Th $\mathscr{K}$ and called the *equational*

theory of $\mathscr{K}$, while the class of structures modeling $\mathscr{C}$ is denoted by Mod $\mathscr{C}$ and is called the *equational class axiomatized* by $\mathscr{C}$. These notions may be formalize in type theory as follows.

```
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 𝒞 𝐀 = ∀ {p q} → (p , q) ∈ 𝒞 → Equal p q where open Environment 𝐀
```

The entailment relation

We represent entailment in type theory by defining an inductive type that is similar to the one Andreas Abel defined for formalizing Birkhoff's completeness theorem (*op. cit.*).

```
data _⊢_▷_≈_ (𝒞 : {Y : Type χ} → Pred(Term Y × Term Y) (ov χ)) :
             (X : Type χ)(p q : Term X) → Type (ov χ) where

  hyp        : ∀{Y}{p q : Term Y} → (p , q) ∈ 𝒞 → 𝒞 ⊢ _ ▷ p ≈ q
  app        : ∀{Y}{ps qs : ∥ S ∥ f → Term Y}
                  → (∀ i → 𝒞 ⊢ Y ▷ ps i ≈ qs i) → 𝒞 ⊢ Y ▷ (node f ps) ≈ (node f qs)
  sub        : ∀{p q}    → 𝒞 ⊢ Γ ▷ p ≈ q → (σ : Sub Δ Γ) → 𝒞 ⊢ Δ ▷ (p [ σ ]) ≈ (q [ σ ])
  reflexive  : ∀{p}      → 𝒞 ⊢ Γ ▷ p ≈ p
  symmetric  : ∀{p q}    → 𝒞 ⊢ Γ ▷ p ≈ q → 𝒞 ⊢ Γ ▷ q ≈ p
  transitive : ∀{p q r}  → 𝒞 ⊢ Γ ▷ p ≈ q → 𝒞 ⊢ Γ ▷ q ≈ r → 𝒞 ⊢ Γ ▷ p ≈ r
```

Entailment is *sound* in the following sense: if $\mathscr{C}$ entails $p \approx q$ and $\mathbf{A} \models \mathscr{C}$, then $p \approx q$ holds in $\mathbf{A}$. In other terms, the derivation $\mathscr{C} \vdash X \triangleright p \approx q$ implies that $p \approx q$ holds in every model of $\mathscr{C}$. We will apply this result—called sound and borrowed from Andreas Abel's proof of Birkhoff's completeness theorem (*op. cit.*)—only once below (in §**??**), so we omit its straightforward formalization.

4.2 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 $\mathscr{K} \subseteq \mathscr{K}$. Similarly, $\mathscr{K}$ is *closed under the taking of subalgebras* (resp., *arbitrary products*) provided S $\mathscr{K} \subseteq \mathscr{K}$ (resp., P $\mathscr{K} \subseteq \mathscr{K}$). 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 $\mathscr{K} \subseteq \mathscr{K}$.

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

  H : ∀ ℓ → Pred(Algebra α ρᵃ) (a ⊔ ov ℓ) → Pred(Algebra β ρᵇ) _
  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) _
  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) _
  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

  V : ∀ ℓ ι → Pred(Algebra α ρᵃ) (a ⊔ ov ℓ) → Pred(Algebra δ ρᵈ) _
  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 , ≤-transitive 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 ⟧    ⟨$⟩ ρ              ≈˘⟨ cong ⟦ p ⟧ (f∼g ∘ ρ)          ⟩
      ⟦ p ⟧    ⟨$⟩ (f ∘ (g ∘ ρ)) ≈˘⟨ comm-hom-term fh p (g ∘ ρ) ⟩
      f(⟦ p ⟧ᴬ ⟨$⟩ (g ∘ ρ))       ≈⟨  cong | fh | (Apq (g ∘ ρ))     ⟩
      f(⟦ q ⟧ᴬ ⟨$⟩ (g ∘ ρ))       ≈⟨  comm-hom-term fh q (g ∘ ρ) ⟩
      ⟦ q ⟧    ⟨$⟩ (f ∘ (g ∘ ρ)) ≈⟨  cong ⟦ q ⟧ (f∼g ∘ ρ)          ⟩
      ⟦ q ⟧    ⟨$⟩ ρ              ■
    where
    private f = _⟨$⟩_ | fh | ; g = _⟨$⟩_ | gh |
    open Environment A using () renaming ( ⟦_⟧ to ⟦_⟧ᴬ )
```

```
open Environment 𝐁 using ( ⟦_⟧ )
open SetoidReasoning 𝔻[ 𝐁 ]
```

Identities modeled by an algebra $\mathbf{A}$ are also modeled by every subalgebra of $\mathbf{A}$. We will refer to this fact as ⊧-S-invar. We omit its proof since it is similar to the proof of ⊧-I-invar. Next, an identity satisfied by all algebras in an indexed collection is also satisfied by the product of algebras in that collection. We omit the formal proof of this fact, and refer to it as ⊧-P-invar below.

Identity preservation

The classes H $\mathscr{K}$, S $\mathscr{K}$, P $\mathscr{K}$, and V $\mathscr{K}$ all satisfy the same set of equations. We will only use a subset of the inclusions used to prove this fact. For complete proofs, see the Varieties.Func.Preservation module of the agda-algebras library. Specifically, we will cite the following facts, whose formal proofs we omit.

```
H-id1 : 𝒦 ⊫ p ≈ q → (H {β = α}{ρ = ρᵃ} ℓ 𝒦) ⊫ p ≈ q

S-id1 : 𝒦 ⊫ p ≈ q → (S {β = α}{ρ = ρᵃ} ℓ 𝒦) ⊫ p ≈ q

S-id2 : S ℓ 𝒦 ⊫ p ≈ q → 𝒦 ⊫ p ≈ q

P-id1 : ∀{ι} → 𝒦 ⊫ p ≈ q → P {β = α}{ρ = ρᵃ} ℓ ι 𝒦 ⊫ p ≈ q

V-id1 : 𝒦 ⊫ p ≈ q → V ℓ ι 𝒦 ⊫ p ≈ q
```

5 Free Algebras

5.1 The absolutely free algebra T X

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

- Every function from X to | $\mathbf{A}$ | lifts to a homomorphism from $\mathbf{T}$ X to $\mathbf{A}$.
- 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 | $\mathbf{A}$ | lifts to a map from | $\mathbf{T}$ X | to | $\mathbf{A}$ | 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
  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 = fl cong
    where
    open Setoid 𝔻[ 𝐀 ] using ( _≈_ ) renaming ( reflexive to reflexiveᴬ )
    fl cong : ∀ {s t} → s ≃ t → free-lift s ≈ free-lift t
    fl cong (_≃_.rfl x) = reflexiveᴬ (≡.cong h x)
    fl cong (_≃_.gnl x) = cong (Interp 𝐀) (≡.refl , (λ i → fl cong (x i)))
```

Naturally, at the base step of the induction, when the term has the form $\mathcal{g}$ 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* over X. The domain of the free algebra is a setoid whose Carrier is the type Term X of S-terms in X. The interpretation of an operation in the free algebra is simply the operation itself.

```
module FreeAlgebra {χ : Level}(𝒞 : {Y : Type χ} → Pred (Term Y × Term Y) _) where

  FreeDomain : Type χ → Setoid _ _
  FreeDomain X =
    record { Carrier       = Term X
           ; _≈_           = 𝒞 ⊢ X ▷ _≈_
           ; isEquivalence = record { refl = reflexive ; sym = symmetric ; trans = transitive } }

  𝔽[_] : Type χ → Algebra (ov χ) _
  Domain 𝔽[ X ] = FreeDomain X
  Interp 𝔽[ X ] = FreeInterp
    where
    FreeInterp : ∀ {X} → ⟨ S ⟩ (FreeDomain X) ⟶ FreeDomain X
    FreeInterp ⟨$⟩ (f , ts)       = node f ts
    cong FreeInterp (≡.refl , h) = app h
```

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.

```
module FreeHom {𝒦 : Pred(Algebra α ρᵃ) (α ⊔ ρᵃ ⊔ ov ℓ)} where
  private c = α ⊔ ρᵃ ⊔ ℓ ; ι = ov c

  open FreeAlgebra {χ = c} (Th 𝒦) using ( 𝔽[_] )

  epi𝔽[_] : (X : Type c) → epi (𝐓 X) 𝔽[ X ]
  epi𝔽[ X ] = h , hepi
    where
    open Setoid 𝔻[ 𝐓 X ]   using ()       renaming ( _≈_ to _≈₀_ ; refl to reflᵀ )
    open Setoid 𝔻[ 𝔽[ X ] ] using ( refl ) renaming ( _≈_ to _≈₁_ )

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

    h : 𝔻[ 𝐓 X ] ⟶ 𝔻[ 𝔽[ X ] ]
    h = record { f = id ; cong = con }

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

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

As promised, we prove that the kernel of the natural epimorphism is the congruence defined by the identities modelled by 𝒦.

```
  kernel-in-theory : {X : Type c} → ker ∣ hom𝔽[ X ] ∣ ⊆ Th (V ℓ ι 𝒦)
  kernel-in-theory {X = X} {p , q} pKq 𝐀 vkA = V-id1{ℓ = ℓ}{p = p}{q} (ζ pKq) 𝐀 vkA
    where
    ζ : ∀{p q} → (Th 𝒦) ⊢ X ▷ p ≈ q → 𝒦 ⊫ p ≈ q
    ζ x 𝐀 kA = sound (λ y ρ → y 𝐀 kA ρ) x where open Soundness (Th 𝒦) 𝐀
```

The universal property

```
module _ {𝐀 : Algebra (α ⊔ ρᵃ ⊔ ℓ) (α ⊔ ρᵃ ⊔ ℓ)} {𝒦 : Pred(Algebra α ρᵃ) (α ⊔ ρᵃ ⊔ ov ℓ)} where
  private c = α ⊔ ρᵃ ⊔ ℓ ; ι = ov c
  open FreeHom {ℓ = ℓ} {𝒦}
  open FreeAlgebra {χ = c}(Th 𝒦) using ( 𝔽[_] )
  open Setoid 𝔻[ 𝐀 ]                using ( refl ; sym ; trans ) 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
```

6 Birkhoff's Variety Theorem

Informal statement of the theorem

Formal statement and structure of the proof

Products of classes of algebras

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

```
module _ (𝒦 : Pred(Algebra α ρᵃ) (α ⊔ ρᵃ ⊔ ov ℓ)){X : Type (α ⊔ ρᵃ ⊔ ℓ)} where
  private c = α ⊔ ρᵃ ⊔ ℓ ; ι = ov c

  open FreeHom {ℓ = ℓ} {𝒦}
  open FreeAlgebra {χ = c}(Th 𝒦) 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 $\approx$ q for any given i = ($\mathbf{A}$, sA , ρ), where $\mathbf{A}$ is an algebra, sA is a proof that $\mathbf{A}$ belongs to S $\mathscr{K}$, and ρ is an environment map (assigning values in the domain of $\mathbf{A}$ to variable symbols in X).

```
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 ( ⟦_⟧ )
```

Later we prove that if the identity p $\approx$ q holds in all $\mathbf{A} \in$ S $\mathscr{K}$ (for all environments), then p $\approx$ q holds in the relatively free algebra $\mathbb{F}$[X]; equivalently, the pair (p , q) belongs to the kernel of the natural homomorphism from $\mathbf{T}$ X onto $\mathbb{F}$[X]. We will use that fact to prove that the kernel of the natural hom from $\mathbf{T}$ X to $\mathfrak{C}$ is contained in the kernel of the natural hom from $\mathbf{T}$ X onto $\mathbb{F}$[X], whence we construct a monomorphism from $\mathbb{F}$[X] into $\mathfrak{C}$, and thus $\mathbb{F}$[X] is a subalgebra of $\mathfrak{C}$, so belongs to S (P $\mathscr{K}$).

```
homℭ : hom (𝐓 X) ℭ
```

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

homF𝔠 : hom F[ X ] ℭ
homF𝔠 = | HomFactor ℭ homℭ homF[ X ] kerF⊆kerℭ (isSurjective ∥ epiF[ X ] ∥) |
```

If (p , q) belongs to the kernel of homℭ, then Th 𝒦 includes the identity p ≈ q—that is, Th 𝒦 ⊢ X ▷ p ≈ q. Equivalently, if the kernel of homℭ is contained in that of homF[X]. We omit the formal proof of this lemma and merely display its formal statement, which is the following. We conclude that the homomorphism from F[X] to ℭ is injective, whence it follows that F[X] is (isomorphic to) a subalgebra of ℭ.

```
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

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

Using the last result we will prove that F[X] belongs to S (P 𝒦). This requires one more technical lemma concerning the classes S and P; specifically, P (S 𝒦) ⊆ S (P 𝒦) holds for every class 𝒦. The Varieties.Func.Preservation.lagda module contains the formal statement and proof of this result, called PS⊆SP, which we omit.

We conclude this subsection with the proof that F[X] belongs to S (P 𝒦).

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

Finally, we are ready to present the formal statement and proof of Birkhoff's celebrated variety theorem.

```
module _ {𝒦 : Pred(Algebra α ρᵃ) (α ⊔ ρᵃ ⊔ ov ℓ)} where
  private c = α ⊔ ρᵃ ⊔ ℓ ; ι = ov c
  open FreeAlgebra {χ = c}(Th 𝒦) using ( F[_] )

  Birkhoff : ∀ 𝐀 → 𝐀 ∈ Mod (Th (V ℓ ι 𝒦)) → 𝐀 ∈ V ℓ ι 𝒦
  Birkhoff 𝐀 ModThA = F[ ∣ 𝐀 ∣ ] , (spFA , AimgF)
    where
    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})
```

```
φ : Lift-Alg A ι ι IsHomImageOf 𝔽[ 𝕌[ A ] ]
φ = epi→ontohom 𝔽[ 𝕌[ A ] ] (Lift-Alg A ι ι) epi𝔽IA
AimgF : A IsHomImageOf 𝔽[ 𝕌[ A ] ]
AimgF = ∘-hom ∣ φ ∣ (from Lift-≅) ,
        ∘-IsSurjective _ _ ∥ φ ∥ (fromIsSurjective (Lift-≅{𝓤}{A = A}))
```

The converse inclusion, $\mathsf{V}\ \mathscr{K} \subseteq \mathsf{Mod}\ (\mathsf{Th}\ (\mathsf{V}\ \mathscr{K}))$, 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.

```
Birkhoff-converse : {A : Algebra α ρᵃ}
  →                 A ∈ V{β = α}{ρᵃ}{α}{ρᵃ} ℓ ι 𝒦
  →                 A ∈ Mod{X = 𝕌[ A ]} (Th (V ℓ ι 𝒦))

Birkhoff-converse {A = A} 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 we already proved the identity preservation lemma $\mathsf{V\text{-}id1} : \mathscr{K} \models \mathsf{p} \doteq \mathsf{q} \to \mathsf{V}\ \mathscr{K} \models \mathsf{p} \doteq \mathsf{q}$. Thus, if $\mathscr{K}$ is an equational class—that is, if $\mathscr{K}$ is the class of algebras satisfying all identities in some set—then $\mathsf{V}\ \mathscr{K} \subseteq \mathscr{K}$. On the other hand, we also proved that V is expansive, that is, $\mathsf{V\text{-}expa} : \mathscr{K} \subseteq \mathsf{V}\ \mathscr{K}$, so $\mathscr{K}$ $(= \mathsf{V}\ \mathscr{K} = \mathsf{H\ S\ P}\ \mathscr{K})$ is a variety. Thus, 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.