

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.HSP](#) module, 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. Perhaps 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;



This work and the [agda-algebras](#) library by William DeMeo and the [agda-algebras](#) team is licensed under a [Creative Commons Attribution-ShareAlike 4.0 International License](#).

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 ( Ⓞ ; ℳ ; Signature )
```

```

module Demos.HSP {S : Signature 0 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 `proj₁` and `proj₂` 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 setoids 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, \longrightarrow , 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 $\text{Image } f \ni b$ is a dependent pair (a, p) , where $a : A$ and $p : b \approx f a$ 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, $(a, p) : \text{Image } f \ni b$, and returns the witness a .

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

InvlsInverser : {f : A → B} {b : B} (q : Image f ⇒ b) → f ($) (Inv f q) ≈ b
InvlsInverser (eq _ p) = sym p
```

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

Injective and surjective setoid functions

If $f : A \rightarrow B$ is a setoid function from $A = (A, \approx_0)$ to $B = (B, \approx_1)$, 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 $f : A \rightarrow B$ (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 $\{(x, y) \in A \times A : f \langle \$ \rangle x \approx f \langle \$ \rangle y\}$, where $_ \approx _$ 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 *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 $\rho : C + F + R \rightarrow N$ 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 $S = (F, \rho)$ consisting of a collection F of *operation symbols* and a function $\rho : F \rightarrow N$ which maps each operation symbol to its arity. Here, N denotes the *arity type*. Heuristically, the arity ρf of an operation symbol $f \in F$ 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 : (⊙ ℳ : Level) → Type (lsuc (⊙ ⊔ ℳ))
Signature ⊙ ℳ = Σ[ F ∈ Type ⊙ ] (F → Type ℳ)

```

Using special syntax for the first and second projections—`|_` and `||_||` (resp.)—if $S : \text{Signature } \odot \mathcal{V}$ 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 $u : \parallel S \parallel f \rightarrow A$ of arguments for f , and a tuple $v : \parallel S \parallel g \rightarrow A$ of arguments for g . If we know that f is identically equal to g —that is, $f \equiv g$ (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 $f \equiv g$, 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} {V : Setoid} {ρa : Setoid}
  → ∀ {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.

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

  ⟨ _ ⟩ : Signature ⊔ 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* $S = (F, \rho)$ (or *S-algebra*) is denoted by $A = (A, F^A)$ and consists of

- a *nonempty* set (or type) A , called the *domain* of the algebra;
- a collection $F^A := \{ f^A \mid f \in F, f^A : (\rho f \rightarrow A) \rightarrow A \}$ 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 .

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  $\alpha$   $\rho$  : Type ( $\mathbb{O} \sqcup \mathcal{V} \sqcup \text{Isuc } (\alpha \sqcup \rho)$ ) where
  field Domain : Setoid  $\alpha$   $\rho$ 
  Interp : ( $\langle S \rangle$  Domain)  $\longrightarrow$  Domain

```

Recall, we renamed Agda’s `Func` type, preferring instead the long-arrow symbol \longrightarrow , so the `Interp` field has type `Func ($\langle S \rangle$ Domain) Domain`, a record type with two fields:

- a function `f : Carrier ($\langle S \rangle$ Domain) \rightarrow Carrier Domain` representing the operation;
- a proof `cong : f Preserves $_ \approx_1 _ \rightarrow _ \approx_2 _$` 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: $\forall u v \text{ in Carrier } (\langle S \rangle \text{ Domain}), \text{ if } u \approx_1 v, \text{ then } f \langle \$ \rangle u \approx_2 f \langle \$ \rangle v$.

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 \mathbf{A} is an algebra, then

- `D[\mathbf{A}]` denotes the setoid `Domain \mathbf{A}` ,
- `U[\mathbf{A}]` is the underlying carrier or “universe” of the algebra \mathbf{A} , and
- `f ^ \mathbf{A}` denotes the interpretation in the algebra \mathbf{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 (Isuc ℓ)`, `Type (Isuc ℓ) : Type (Isuc (Isuc ℓ))`, This means that `Type ℓ` has type `Type (Isuc ℓ)`, etc. However, this does *not* imply that `Type ℓ : Type (Isuc (Isuc 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
 $\alpha$  !=  $\mathbb{O} \sqcup \mathcal{V} \sqcup (\text{Isuc } \alpha)$  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 `$\mathbb{O} \sqcup \mathcal{V} \sqcup (\text{Isuc } \alpha)$` . 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 ($\mathbb{O} \sqcup \mathcal{V} \sqcup (\text{Isuc } \alpha)$) 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 o 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 o sym o 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-Algl (Lift-Algr 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 ρa` has a domain setoid (called `Domain`) of type `Setoid α ρa`. This domain setoid packages a “carrier set” (`Carrier`), inhabiting `Type α`, with an equality on `Carrier` of type `Rel Carrier ρa`. 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 ρa`—and constructs a new algebra with carrier set inhabiting `Type (α ⊔ ℓ0)` and having an equality of type `Rel Carrier (ρa ⊔ ℓ1)`. 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 $I : \text{Type } \iota$ a type (which, in the present context, we might refer to as the “indexing type”). Then the dependent function type $\mathcal{A} : I \rightarrow \text{Algebra } \alpha \rho^a$

represents an *indexed family of algebras*. Denote by $\prod \mathcal{A}$ the *product of algebras* in \mathcal{A} (or *product algebra*), by which we mean the algebra whose domain is the Cartesian product $\prod i : I, \mathbb{D}[\mathcal{A} i]$ 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 $\mathbf{a} : \prod i : I, J \rightarrow \mathbb{D}[\mathcal{A} i]$ is, for each $i : I$, a J -tuple of elements of the domain $\mathbb{D}[\mathcal{A} i]$, then we define the interpretation of f in $\prod \mathcal{A}$ by $(f \hat{\ } \prod \mathcal{A}) \mathbf{a} := \lambda (i : I) \rightarrow (f \hat{\ } \mathcal{A} i)(\mathbf{a} i)$.

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 → ∪[ A i ]
          ; _≈_ = λ a b → ∀ i → (Setoid._≈_ ∅[ A i ]) (a i)(b i)
          ; isEquivalence =
              record { refl = λ i → IsEquivalence.refl (isEquivalence ∅[ A i ])
                    ; sym = λ x i → IsEquivalence.sym (isEquivalence ∅[ A i ])(x i)
                    ; trans = λ x y i → IsEquivalence.trans (isEquivalence ∅[ 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 \mathbf{A} and \mathbf{B} are S -algebras. A *homomorphism* (or “hom”) from \mathbf{A} to \mathbf{B} is a setoid function $h : \mathbb{D}[\mathbf{A}] \rightarrow \mathbb{D}[\mathbf{B}]$ that is *compatible* (or *commutes*) with all basic operations; that is, for every operation symbol $f : |S|$ and all tuples $\mathbf{a} : \parallel S \parallel f \rightarrow \mathbb{D}[\mathbf{A}]$, the following equality holds: $h \langle \$ \rangle (f \hat{\ } \mathbf{A}) \mathbf{a} \approx (f \hat{\ } \mathbf{B}) \lambda x \rightarrow h \langle \$ \rangle (\mathbf{a} x)$.

To formalize this concept in Agda, we first define a type `compatible-map-op` representing the assertion that a given setoid function $h : \mathbb{D}[\mathbf{A}] \rightarrow \mathbb{D}[\mathbf{B}]$ commutes with a given basic operation f .

```

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

  compatible-map-op : (∅[ A ] → ∅[ B ]) → |S| → Type (ℳ ⊔ α ⊔ ρb)
  compatible-map-op h f = ∀ {a} → h ($) (f ^ A) a ≈ (f ^ B) λ x → h ($) (a x)
  where open Setoid ∅[ B ] using ( _≈_ )

```

Generalizing over operation symbols gives the following type of compatible maps from (the domain of) \mathbf{A} to (the domain of) \mathbf{B} .

```

compatible-map : (∅[ A ] → ∅[ 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 \mathbf{A} to \mathbf{B} .

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

```
record IsHom (h :  $\mathbb{D}[A] \rightarrow \mathbb{D}[B]$ ) : Type (ov  $\sqcup \alpha \sqcup \rho^b$ ) where
  constructor mkhom
  field compatible : compatible-map h
```

```
hom : Type c
hom =  $\Sigma (\mathbb{D}[A] \rightarrow \mathbb{D}[B])$  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, $p : \text{IsHom } h$, 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  $\alpha \rho^a$ } {B : Algebra  $\beta \rho^b$ } {C : Algebra  $\gamma \rho^c$ }
  {g :  $\mathbb{D}[A] \rightarrow \mathbb{D}[B]$ } {h :  $\mathbb{D}[B] \rightarrow \mathbb{D}[C]$ } where

  open Setoid  $\mathbb{D}[C]$  using ( trans )

  o-is-hom : IsHom A B g  $\rightarrow$  IsHom B C h  $\rightarrow$  IsHom A C (h  $\langle \circ \rangle$  g)
  o-is-hom ghom hhom = mkhom c
  where
    c : compatible-map A C (h  $\langle \circ \rangle$  g)
    c = trans (cong h (compatible ghom)) (compatible hhom)

  o-is-epi : IsEpi A B g  $\rightarrow$  IsEpi B C h  $\rightarrow$  IsEpi A C (h  $\langle \circ \rangle$  g)
  o-is-epi gE hE = record { isHom = o-is-hom (isHom gE) (isHom hE)
    ; isSurjective = o-IsSurjective g h (isSurjective gE) (isSurjective hE) }
```

Factorization of homomorphisms

If $g : \text{hom } A B$, $h : \text{hom } A C$, h is surjective, and $\ker h \subseteq \ker g$, then there exists $\varphi : \text{hom } C B$ such that $g = \varphi \circ h$.

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  $\alpha \rho^a$ } {B : Algebra  $\beta \rho^b$ } {C : Algebra  $\gamma \rho^c$ }
  (gh : hom A B) (hh : hom A C) where

  open Setoid  $\mathbb{D}[B]$  using () renaming (  $\approx$  to  $\approx_2$  ; sym to sym2 )
  open Setoid  $\mathbb{D}[C]$  using ( trans ) renaming (  $\approx$  to  $\approx_3$  ; sym to sym3 )
  private gfunc = | gh | ; g =  $\_ \langle \$ \rangle \_$  gfunc ; hfunc = | hh | ; h =  $\_ \langle \$ \rangle \_$  hfunc

  HomFactor : kernel  $\approx_3$  h  $\subseteq$  kernel  $\approx_2$  g  $\rightarrow$  IsSurjective hfunc
   $\rightarrow \Sigma [\varphi \in \text{hom } C B] \forall a \rightarrow g a \approx_2 | \varphi | \langle \$ \rangle h a$ 
```

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 ( _≈_ to _≈B_ ; sym to symb ; trans to transb )

  record _≅_ : Type (ℓ ⊔ ℳ ⊔ 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 = ℓ ⊔ ℳ ⊔ lsuc α

_IsHomImageOf_ : (B : Algebra β ρb)(A : Algebra α ρa) → Type (ℓ ⊔ ℳ ⊔ 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 $A : \text{Algebra } \alpha \rho$, the type `HomImages A` denotes the class of algebras $B : \text{Algebra } \beta \rho$ with a map $\varphi : |A| \rightarrow |B|$ such that φ is a surjective homomorphism.

3.5 Subalgebras

Basic definitions

```
_≤_ : Algebra α ρa → Algebra β ρb → Type (ℓ ⊔ ℳ ⊔ 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.

\leq -reflexive $\{A = A\} = id, id$

$mon \rightarrow \leq : \{A : Algebra \alpha \rho^a\} \{B : Algebra \beta \rho^b\} \rightarrow mon A B \rightarrow A \leq B$

$mon \rightarrow \leq \{A = A\} \{B\} x = mon \rightarrow intohom A B x$

module $_ \{A : Algebra \alpha \rho^a\} \{B : Algebra \beta \rho^b\} \{C : Algebra \gamma \rho^c\}$ **where**

\leq -trans : $A \leq B \rightarrow B \leq C \rightarrow A \leq C$

\leq -trans (f , finj) (g , ginj) = (o-hom f g) , o-IsInjective | f | | g | finj ginj

\cong -trans- \leq : $A \cong B \rightarrow B \leq C \rightarrow A \leq C$

\cong -trans- $\leq A \cong B$ (h , hinj) = (o-hom (to A \cong B) h) , (o-IsInjective | to A \cong B | | h | (toIsInjective A \cong B) hinj)

Products of subalgebras

module $_ \{l : Level\} \{I : Type \iota\} \{A : I \rightarrow Algebra \alpha \rho^a\} \{B : I \rightarrow Algebra \beta \rho^b\}$ **where**

$\prod \leq : (\forall i \rightarrow B i \leq A i) \rightarrow \prod B \leq \prod A$

$\prod \leq B \leq A = (hfunc, hhom), hM$

where

$hi : \forall i \rightarrow hom (B i) (A i)$

$hi i = | B \leq A i |$

$hfunc : \mathbb{D} [\prod B] \rightarrow \mathbb{D} [\prod A]$

$(hfunc \langle \$ \rangle x) i = | hi i | \langle \$ \rangle x i$

$cong hfunc = \lambda xy i \rightarrow cong | hi i | (xy i)$

$hhom : IsHom (\prod B) (\prod A) hfunc$

$compatible hhom = \lambda i \rightarrow compatible || hi i ||$

$hM : IsInjective hfunc$

$hM = \lambda xy i \rightarrow || B \leq A i || (xy i)$

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 $X \cap | S | = \emptyset$. By a *word* in the language of S , we mean a nonempty, finite sequence of members of $X \cup | S |$. We denote the concatenation of such sequences by simple juxtaposition. Let S_0 denote the set of nullary operation symbols of S . We define by induction on n the sets T_n of *words* over $X \cup | S |$ as follows (cf. [1, Def. 4.19]):

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

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

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

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

data $Term (X : Type \chi) : Type (ov \chi)$ **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, $\text{ov } \chi$ is our shorthand notation for the universe level $\mathbb{O} \sqcup \mathbb{V} \sqcup \text{lsuc } \chi$.

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 \equiv 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 $\text{Term } X$ is nonempty (equivalently, if X or $| S |$ is nonempty), then we can define an algebraic structure, denoted by $\mathbf{T } X$ 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 $f^\wedge (\mathbf{T } X)$ the operation on $\text{Term } X$ that maps a tuple $t : || S || f \rightarrow | \mathbf{T } X |$ to the formal term $f t$.
- Define $\mathbf{T } X$ to be the algebra with universe $| \mathbf{T } X | := \text{Term } X$ and operations $f^\wedge (\mathbf{T } X)$, 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 \mathbf{A} in a context X is a map that assigns to each variable $x : X$ an element in the domain of \mathbf{A} , packaged together with an equality of environments, which is simply pointwise equality (relatively to the setoid equality of the underlying domain of \mathbf{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 \mathbf{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) (gnt 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 $\llbracket t \llbracket \sigma \rrbracket \rrbracket \rho \simeq \llbracket t \rrbracket \llbracket \sigma \rrbracket \rho$.

```

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 $X : \text{Type } \chi$, 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

open Eq public
```

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

If \mathbf{A} is an S -algebra we say that \mathbf{A} *satisfies* $p \approx \cdot q$ provided for all environments $\rho : X \rightarrow | \mathbf{A} |$ (assigning values in the domain of \mathbf{A} to variable symbols in X) we have $\llbracket p \rrbracket \langle \$ \rangle \rho \approx \llbracket q \rrbracket \langle \$ \rangle \rho$. In this situation, we write $\mathbf{A} \models (p \approx \cdot q)$ and say that \mathbf{A} *models* the identity $p \approx \cdot q$.

If \mathcal{K} is a class of algebras, all of the same signature, we write $\mathcal{K} \models (p \approx \cdot q)$ if, for every $\mathbf{A} \in \mathcal{K}$, we have $\mathbf{A} \models (p \approx \cdot q)$.

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 \models and \approx . As a reasonable alternative to what we would normally express informally as $\mathcal{K} \models p \approx q$, we have settled on $\mathcal{K} \models (p \approx \cdot q)$ to denote this relation. To reiterate, if \mathcal{K} is a class of S -algebras, we write $\mathcal{K} \models (p \approx \cdot q)$ provided every $\mathbf{A} \in \mathcal{K}$ satisfies $\mathbf{A} \models (p \approx \cdot q)$.

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

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

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

```
_|=_ : (A : Algebra α ρa) → {ℓ : Level}{I : Type ℓ} → (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 \mathcal{K} denotes a class of structures, then $\text{Th } \mathcal{K}$ represents the set of identities modeled by the members of \mathcal{K} .

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

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

4.3 The entailment relation

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

```

module _ {χ ι : Level} where

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

  ⊢refl : ∀ {p} → ℳ ⊢ Γ▷ p ≈ p
  ⊢sym : ∀ {p q : Term Γ} → ℳ ⊢ Γ▷ p ≈ q → ℳ ⊢ Γ▷ q ≈ p
  ⊢trans : ∀ {p q r : Term Γ} → ℳ ⊢ Γ▷ p ≈ q → ℳ ⊢ Γ▷ q ≈ r → ℳ ⊢ Γ▷ p ≈ r

  ⊢≈IsEquiv : {X : Type χ} {l : Type ι} {ℳ : l → Eq} → IsEquivalence (ℳ ⊢ X▷_≈_)
  ⊢▷IsEquiv = record { refl = ⊢refl ; sym = ⊢sym ; trans = ⊢trans }

```

4.4 Soundness

In any model **A** of the equations \mathcal{E} derived equality is actual equality.⁹

```

module Soundness {χ α ι : Level} {l : Type ι} {ℳ : l → Eq {χ}}
  (A : Algebra α ρα) – We assume an algebra A
  (V : A ⊢ ℳ) – that models all equations in ℳ.
  where

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

  sound : ∀ {p q} → ℳ ⊢ 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 EqualsEquiv {x = p}
  sound (⊢sym {p = p}{q} Epq) = sym EqualsEquiv {x = p}{q} (sound Epq)
  sound (⊢trans {p = p}{q}{r} Epq Eqr) = trans EqualsEquiv {i = p}{q}{r} (sound Epq)(sound Eqr)

```

4.5 The Closure Operators H, S, P and V

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

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

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

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

A *variety* is a class of S -algebras that is closed under the taking of homomorphic images, subalgebras, and arbitrary products. To represent varieties we define types for the closure operators \mathbf{H} , \mathbf{S} , and \mathbf{P} that are composable. Separately, we define a type \mathbf{V} which represents closure under all three operators, \mathbf{H} , \mathbf{S} , and \mathbf{P} . Thus, if \mathcal{K} is a class of S -algebras, then $\mathbf{V} \mathcal{K} := \mathbf{H} (\mathbf{S} (\mathbf{P} \mathcal{K}))$, and \mathcal{K} is a variety iff $\mathbf{V} \mathcal{K} \subseteq \mathcal{K}$.

We now define the type \mathbf{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 \mathbf{S} to represent classes of algebras that closed under the taking of subalgebras, and the type \mathbf{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 _ ℓ B = Σ[ A ∈ Algebra α ρa ] A ∈ ℓ × B IsHomImageOf A

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

  P : ∀ ℓ ι → Pred(Algebra α ρa) (a ⊔ ov ℓ) → Pred(Algebra β ρb) (b ⊔ ov(a ⊔ ℓ ⊔ ι))
  P _ ι ℓ B = Σ[ I ∈ Type ι ] (Σ[ A ∈ (I → Algebra α ρa) ] (∀ i → A i ∈ ℓ) × (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 ℓ = H{γ}{ρc}{δ}{ρd} (a ⊔ b ⊔ ℓ ⊔ ι) (S{β}{ρb} (a ⊔ ℓ ⊔ ι) (P ℓ ι ℓ))

```

Idempotence of S

\mathbf{S} is a closure operator. The facts that \mathbf{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 \mathbf{S} , so we prove that property as follows.

```

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

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

```

Algebraic invariance of \models

The binary relation \models 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 \models

Identities modeled by an algebra \mathbf{A} are also modeled by every subalgebra of \mathbf{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 \models

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

```

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

  |P-invar : (∀ i → A i ⊨ (p ≈ · q)) → ∏ A i ⊨ (p ≈ · q)
  |P-invar A pq 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 → a p q 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 \mathcal{K} is a subalgebra of a product of algebras in \mathcal{K} . We denote this inclusion by **PS**⊆**SP**, which we state and prove as follows.

```

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

PS⊆SP : P (a ⊔ ℓ) oal (S {β = α} {ρa} ℓ K) ⊆ S oal (P ℓ oal K)
PS⊆SP {B} (I , ( a , sA , B ≅ [A] )) = Goal
where
  B : I → Algebra α ρa
  B i = [ sA i ]
  kB : (i : I) → B i ∈ K
  kB i = fst [ sA i ]
  [A] ≤ [B] : [ a ] ≤ [ B ]
  [A] ≤ [B] = [ a ] ≤ 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** \mathcal{K} , **S** \mathcal{K} , **P** \mathcal{K} , and **V** \mathcal{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

```

$$\begin{aligned}
& \llbracket p \rrbracket \quad \langle \$ \rangle \quad \rho \approx \langle \text{cong } \llbracket p \rrbracket \zeta \rangle \\
& \llbracket p \rrbracket \quad \langle \$ \rangle (\varphi \circ \varphi^{-1} \circ \rho) \approx \langle \text{comm-hom-term } \varphi h p (\varphi^{-1} \circ \rho) \rangle \\
& \varphi (\llbracket p \rrbracket^A \langle \$ \rangle (\varphi^{-1} \circ \rho)) \approx \langle \text{cong } | \varphi h | (IH (\varphi^{-1} \circ \rho)) \rangle \\
& \varphi (\llbracket q \rrbracket^A \langle \$ \rangle (\varphi^{-1} \circ \rho)) \approx \langle \text{comm-hom-term } \varphi h q (\varphi^{-1} \circ \rho) \rangle \\
& \llbracket q \rrbracket \quad \langle \$ \rangle (\varphi \circ \varphi^{-1} \circ \rho) \approx \langle \text{cong } \llbracket q \rrbracket \zeta \rangle \\
& \llbracket q \rrbracket \quad \langle \$ \rangle \quad \rho \quad \blacksquare
\end{aligned}$$

where

open Environment A using () renaming ($\llbracket _ \rrbracket$ to $\llbracket _ \rrbracket^A$)
 open Environment B using ($\llbracket _ \rrbracket$)
 open Setoid $\mathbb{D}[B]$ using ($_ \approx _$)
 open SetoidReasoning $\mathbb{D}[B]$

IH : A \models (p \approx · q)
 IH = σ A kA

φh : hom A B
 φh = | BimgOfA |
 private φ = ($_ \langle \$ \rangle _$ | φh |)

φE : IsSurjective | φh |
 φE = || BimgOfA ||

φ^{-1} : $\mathbb{U}[B] \rightarrow \mathbb{U}[A]$
 φ^{-1} = SurjInv | φh | φE

ζ : $\forall x \rightarrow (\varphi \circ \varphi^{-1} \circ \rho) x \approx \rho x$
 ζ = $\lambda _ \rightarrow \text{InvlInverse}^r \varphi E$

S preserves identities

$$\begin{aligned}
& \text{S-id1} : \mathcal{K} \models (p \approx \cdot q) \rightarrow (\text{S } \{\beta = \alpha\} \{\rho^a\} \ell \mathcal{K}) \models (p \approx \cdot q) \\
& \text{S-id1 } \sigma B (A, kA, B \leq A) = \models \text{S-invar} \{p = p\} \{q\} (\sigma A kA) B \leq A
\end{aligned}$$

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

$$\begin{aligned}
& \text{S-id2} : \text{S } \ell \mathcal{K} \models (p \approx \cdot q) \rightarrow \mathcal{K} \models (p \approx \cdot q) \\
& \text{S-id2 } \text{Spq } A kA = \text{Spq } A (A, (kA, \leq\text{-reflexive}))
\end{aligned}$$

P preserves identities

$$\begin{aligned}
& \text{P-id1} : \forall \{\iota\} \rightarrow \mathcal{K} \models (p \approx \cdot q) \rightarrow \text{P } \{\beta = \alpha\} \{\rho^a\} \ell \iota \mathcal{K} \models (p \approx \cdot q) \\
& \text{P-id1 } \sigma A (I, \mathcal{A}, kA, A \cong \sqcap A) = \models \text{I-invar } A p q IH (\cong\text{-sym } A \cong \sqcap A) \\
& \text{where} \\
& ih : \forall i \rightarrow \mathcal{A} i \models (p \approx \cdot q) \\
& ih i = \sigma (\mathcal{A} i) (kA i) \\
& IH : \sqcap \mathcal{A} \models (p \approx \cdot q) \\
& IH = \models \text{P-invar } \mathcal{A} \{p\} \{q\} ih
\end{aligned}$$

V preserves identities

Finally, we prove the analogous preservation lemmas for the closure operator \mathbf{V} .

```

module _ {X : Type} {χ : {ι : Level}} {K : Pred (Algebra α ρa) (α ⊔ ρa ⊔ ov ℓ)} {p q : Term X} where
  private
    alℓ = 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 = alℓ} {K = S alℓ (P {β = α} {ρa} ℓ ι K)} {p = p} {q} spK⊨pq B (A , (spA , BimgA))
  where
    spA : A ∈ S alℓ (P {β = α} {ρa} ℓ ι K)
    spA = ⊔A , (p⊔A , A≤⊔A)
    spK⊨pq : S alℓ (P ℓ ι K) ⊨ (p ≈ · q)
    spK⊨pq = S-id1 {ℓ = alℓ} {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 $\mathbf{V} K$. In other terms, $\text{Th } K \subseteq \text{Th } (\mathbf{V} K)$. 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 $\mathbf{T} X$

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 A , the following hold.

- Every function from X to $|A|$ lifts to a homomorphism from $\mathbf{T} X$ 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 $|\mathbf{T} X|$ to $|A|$ in a natural way, by induction on the structure of the given term.

```

module _ {X : Type} {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 $\text{node } f\ t$, 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 $f^\wedge A$ 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 $\mathbb{F}[X]$

We now define the algebra $\mathbb{F}[X]$, which represents the relatively free algebra. Here, as above, X plays the role of an arbitrary nonempty collection of variables. (It would suffice to take X to be the cardinality of the largest algebra in \mathcal{K} , but since we don't know that cardinality, we leave X arbitrary for now.)

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

  FreeDomain : Type χ → Setoid _ _
  FreeDomain X = record { Carrier      = Term X
                        ; ≡≡          = ℳ ⊢ X ▷≡
                        ; isEquivalence = ⊢≡IsEquiv }
```

The interpretation of an operation is simply the operation itself. This works since $\mathcal{M} \vdash X \triangleright_{\approx} X$ 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 $\mathbb{F}[X] = \text{FreeDomain } X$

Interp $\mathbb{F}[X] = \text{FreeInterp}$

5.3 Basic properties of free algebras

```

module FreeHom ( $\chi : \text{Level}$ ) { $\mathcal{K} : \text{Pred}(\text{Algebra } \alpha \rho^a) (\alpha \sqcup \rho^a \sqcup \text{ov } \ell)$ } where
  private  $\iota = \text{ov}(\chi \sqcup \alpha \sqcup \rho^a \sqcup \ell)$ 
  open Eq

   $\mathcal{J} : \text{Type } \iota$  — indexes the collection of equations modeled by  $\mathcal{K}$ 
   $\mathcal{J} = \Sigma[\text{eq} \in \text{Eq}\{\chi\}] \mathcal{K} \models ((\text{lhs eq}) \approx \cdot (\text{rhs eq}))$ 

   $\mathcal{E} : \mathcal{J} \rightarrow \text{Eq}$ 
   $\mathcal{E}(\text{eqv}, p) = \text{eqv}$ 

   $\mathcal{E} \vdash \_ \triangleright \text{Th} \mathcal{K} : (X : \text{Type } \chi) \rightarrow \forall \{p q\} \rightarrow \mathcal{E} \vdash X \triangleright p \approx q \rightarrow \mathcal{K} \models (p \approx \cdot q)$ 
   $\mathcal{E} \vdash [X] \triangleright \text{Th} \mathcal{K} \times \mathbf{A} \text{ kA} = \text{sound } (\lambda i \rho \rightarrow \parallel i \parallel \mathbf{A} \text{ kA } \rho) \times \text{where open Soundness } \mathcal{E} \mathbf{A}$ 
  open FreeAlgebra { $\iota = \iota$ } { $l = \mathcal{J}$ }  $\mathcal{E}$  using ( $\mathbb{F}[\_]$ )

```

The natural epimorphism from $\mathbf{T} X$ to $\mathbb{F}[X]$

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

```

epiF $[\_]$  : ( $X : \text{Type } \chi$ )  $\rightarrow$  epi ( $\mathbf{T} X$ )  $\mathbb{F}[X]$ 
epiF $[X]$  = h , hepi
  where
    open Algebra ( $\mathbf{T} X$ ) using () renaming ( Domain to TX )
    open Algebra  $\mathbb{F}[X]$  using () renaming ( Domain to F )
    open Setoid TX using () renaming (  $\_ \approx \_$  to  $\_ \approx_0 \_$ ; refl to refl $_0$  )
    open Setoid F using () renaming (  $\_ \approx \_$  to  $\_ \approx_1 \_$ ; refl to refl $_1$  )
    open  $\_ \doteq \_$ 

    c :  $\forall \{x y\} \rightarrow x \approx_0 y \rightarrow x \approx_1 y$ 
    c (rfl {x}{y}  $\equiv$ .refl) = refl $_1$ 
    c (gnl {f}{s}{t} x) = cong (Interp  $\mathbb{F}[X]$ ) ( $\equiv$ .refl , c o x)

    h : TX  $\longrightarrow$  F
    h = record { f = id ; cong = c }

    hepi : IsEpi ( $\mathbf{T} X$ )  $\mathbb{F}[X]$  h
    compatible (isHom hepi) = cong h refl $_0$ 
    isSurjective hepi {y} = eq y refl $_1$ 

    homF $[\_]$  : ( $X : \text{Type } \chi$ )  $\rightarrow$  hom ( $\mathbf{T} X$ )  $\mathbb{F}[X]$ 
    homF $[X]$  = IsEpi.HomReduct  $\parallel$  epiF $[X]$   $\parallel$ 

    homF $[\_]$ -is-epic : ( $X : \text{Type } \chi$ )  $\rightarrow$  IsSurjective  $\mid$  homF $[X]$   $\mid$ 
    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 \mathcal{K} .

```

class-models-kernel :  $\forall \{X p q\} \rightarrow (p , q) \in \text{ker} \mid \text{homF}[X] \mid \rightarrow \mathcal{K} \models (p \approx \cdot q)$ 

```

```

class-models-kernel {X = X}{p}{q} pKq =  $\mathcal{E} \vdash [X] \triangleright \text{Th } \mathcal{K}$  pKq

kernel-in-theory : {X : Type  $\chi$ } → ker | homF [ X ] | ⊆ Th (V  $\ell \iota \mathcal{K}$ )
kernel-in-theory {X = X} {p , q} pKq vkA x = classIds-⊆-Vlds { $\ell = \ell$ }{p = p}{q}
                                     (class-models-kernel pKq) vkA x

module _ {X : Type  $\chi$ } {A : Algebra  $\alpha \rho^a$ }{sA : A ∈ S { $\beta = \alpha$ }{ $\rho^a$ }  $\ell \mathcal{K}$ } 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 { $\ell = \ell$ }{p = p}{q} ( $\mathcal{E} \vdash [X] \triangleright \text{Th } \mathcal{K}$  x) A sA

 $\mathcal{K} \models \mathcal{E} \vdash$  : {X : Type  $\chi$ } → ∀{p q} →  $\mathcal{K} \models (p \approx \cdot q) \rightarrow \mathcal{E} \vdash X \triangleright p \approx q$ 
 $\mathcal{K} \models \mathcal{E} \vdash$  {p = p} {q} pKq = hyp ((p  $\approx \cdot$  q) , pKq) where open  $\_ \vdash \_ \approx \_$  using (hyp)

```

The universal property

```

module _ {A : Algebra ( $\alpha \sqcup \rho^a \sqcup \ell$ ) ( $\alpha \sqcup \rho^a \sqcup \ell$ )} { $\mathcal{K}$  : Pred (Algebra  $\alpha \rho^a$ ) ( $\alpha \sqcup \rho^a \sqcup \text{ov } \ell$ )} where
  private  $\iota = \text{ov}(\alpha \sqcup \rho^a \sqcup \ell)$ 

  open FreeHom { $\ell = \ell$ }{ $\alpha \sqcup \rho^a \sqcup \ell$ } { $\mathcal{K}$ }
  open FreeAlgebra { $\iota = \iota$ }{I =  $\mathcal{J}$ }  $\mathcal{E}$  using (F[])
  open Setoid  $\mathbb{D}[A]$  using (trans ; sym ; refl) renaming (Carrier to A)

  F-ModTh-epi : A ∈ Mod (Th (V  $\ell \iota \mathcal{K}$ ))
    → epi F [ A ] A
  F-ModTh-epi A ∈ ModThK =  $\varphi$  , isEpi
  where
     $\varphi$  :  $\mathbb{D}[F[A]] \rightarrow \mathbb{D}[A]$ 
     $\_ \langle \$ \rangle \_ = \text{free-lift}\{A = A\} \text{ id}$ 
    cong  $\varphi$  {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  $\varphi$ 
    compatible (isHom isEpi) = cong (Interp A) ( $\equiv$ .refl , ( $\lambda \_ \rightarrow \text{refl}$ ))
    isSurjective isEpi {y} = eq ( $\mathcal{G}$  y) refl

  F-ModTh-epi-lift : A ∈ Mod (Th (V  $\ell \iota \mathcal{K}$ )) → epi F [ A ] (Lift-Alg A  $\iota$ )
  F-ModTh-epi-lift A ∈ ModThK = o-epi (F-ModTh-epi ( $\lambda$  {p q} → A ∈ ModThK {p = p}{q})) ToLift-epi

```

5.4 Products of classes of algebras

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

```

module _ ( $\mathcal{K}$  : Pred (Algebra  $\alpha \rho^a$ ) ( $\alpha \sqcup \rho^a \sqcup \text{ov } \ell$ )) {X : Type ( $\alpha \sqcup \rho^a \sqcup \ell$ )} where
  private  $\iota = \text{ov}(\alpha \sqcup \rho^a \sqcup \ell)$ 
  open FreeHom { $\ell = \ell$ } ( $\alpha \sqcup \rho^a \sqcup \ell$ ) { $\mathcal{K}$ }
  open FreeAlgebra { $\iota = \iota$ }{I =  $\mathcal{J}$ }  $\mathcal{E}$  using (F[])
  open Environment using (Env)

   $\mathcal{J}^+ : \text{Type } \iota$ 
   $\mathcal{J}^+ = \Sigma [A \in (\text{Algebra } \alpha \rho^a)] (A \in S \ell \mathcal{K}) \times (\text{Carrier } (\text{Env } A X))$ 

```

```

 $\mathfrak{A}^+ : \mathcal{T}^+ \rightarrow \mathbf{Algebra} \alpha \rho^a$ 
 $\mathfrak{A}^+ i = | i |$ 

 $\mathfrak{C} : \mathbf{Algebra} \iota \iota$ 
 $\mathfrak{C} = \prod \mathfrak{A}^+$ 

```

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

```

skEqual : (i :  $\mathcal{T}^+$ )  $\rightarrow \forall \{p q\} \rightarrow \mathbf{Type} \rho^a$ 
skEqual i {p}{q} =  $\llbracket p \rrbracket \langle \$ \rangle \text{snd } || i || \approx \llbracket q \rrbracket \langle \$ \rangle \text{snd } || i ||$ 
  where
    open Setoid  $\mathbb{D}[\mathfrak{A}^+ i]$  using (  $\_ \approx \_$  )
    open Environment ( $\mathfrak{A}^+ i$ ) using (  $\llbracket \_ \rrbracket$  )

AllEqual $\subseteq$ kerF :  $\forall \{p q\} \rightarrow (\forall i \rightarrow \text{skEqual } i \{p\}\{q\}) \rightarrow (p, q) \in \text{ker} \mid \text{homF}[\mathbf{X}] \mid$ 
AllEqual $\subseteq$ kerF {p}{q} x = Goal
  where
    open Setoid  $\mathbb{D}[\mathbb{F}[\mathbf{X}]]$  using (  $\_ \approx \_$  )
     $\mathcal{K} \models \text{pq} : \mathbf{S} \{ \beta = \alpha \} \{ \rho^a \} \ell \mathcal{K} \models (p \approx \cdot q)$ 
     $\mathcal{K} \models \text{pq } \mathbf{A} \text{ sA } \rho = x (\mathbf{A}, s\mathbf{A}, \rho)$ 
    Goal :  $p \approx q$ 
    Goal =  $\mathcal{K} \models \rightarrow \mathcal{K} \vdash (\text{S-id2} \{ \ell = \ell \} \{ p = p \} \{ q \} \mathcal{K} \models \text{pq})$ 

hom $\mathfrak{C}$  :  $\text{hom}(\mathbf{T} \mathbf{X}) \mathfrak{C}$ 
hom $\mathfrak{C}$  =  $\prod \text{-hom-co } \mathfrak{A}^+ h$ 
  where
    h :  $\forall i \rightarrow \text{hom}(\mathbf{T} \mathbf{X}) (\mathfrak{A}^+ i)$ 
    h i =  $\text{lift-hom}(\text{snd } || i ||)$ 

kerF $\subseteq$ ker $\mathfrak{C}$  :  $\text{ker} \mid \text{homF}[\mathbf{X}] \mid \subseteq \text{ker} \mid \text{hom}\mathfrak{C} \mid$ 
kerF $\subseteq$ ker $\mathfrak{C}$  {p, q} pKq ( $\mathbf{A}, s\mathbf{A}, \rho$ ) = Goal
  where
    open Setoid  $\mathbb{D}[\mathbf{A}]$  using (  $\_ \approx \_ ; \text{sym} ; \text{trans}$  )
    open Environment  $\mathbf{A}$  using (  $\llbracket \_ \rrbracket$  )
    fl :  $\forall t \rightarrow \llbracket t \rrbracket \langle \$ \rangle \rho \approx \text{free-lift } \rho t$ 
    fl t =  $\text{free-lift-interp } \{ \mathbf{A} = \mathbf{A} \} \rho t$ 
    subgoal :  $\llbracket p \rrbracket \langle \$ \rangle \rho \approx \llbracket q \rrbracket \langle \$ \rangle \rho$ 
    subgoal =  $\text{kerF} \subseteq \text{Equal} \{ \mathbf{A} = \mathbf{A} \} \{ s\mathbf{A} \} pKq \rho$ 
    Goal :  $(\text{free-lift} \{ \mathbf{A} = \mathbf{A} \} \rho p) \approx (\text{free-lift} \{ \mathbf{A} = \mathbf{A} \} \rho q)$ 
    Goal =  $\text{trans}(\text{sym}(\text{fl } p))(\text{trans subgoal}(\text{fl } q))$ 

homF $\mathfrak{C}$  :  $\text{hom } \mathbb{F}[\mathbf{X}] \mathfrak{C}$ 
homF $\mathfrak{C}$  =  $\mid \text{HomFactor } \mathfrak{C} \text{ hom}\mathfrak{C} \text{ homF}[\mathbf{X}] \text{ kerF} \subseteq \text{ker}\mathfrak{C} \text{ homF}[\mathbf{X}] \text{-is-epic} \mid$ 

ker $\mathfrak{C} \subseteq \text{kerF}$  :  $\forall \{p q\} \rightarrow (p, q) \in \text{ker} \mid \text{hom}\mathfrak{C} \mid \rightarrow (p, q) \in \text{ker} \mid \text{homF}[\mathbf{X}] \mid$ 

```

```

kerC ⊆ 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 C : mon F[ X ] C
monF C = | homF C | , isMon
where
isMon : IsMon F[ X ] C | homF C |
isHom isMon = || homF C ||
isInjective isMon {p} {q} φpq = kerC ⊆ kerF φpq

```

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

```

F ≤ C : F[ X ] ≤ C
F ≤ C = mon → ≤ monF C

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

```

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 = J} 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 VIA
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
epiFA : epi F[ A ] (Lift-Alg A ι)

```

```

epiFIA = F-ModTh-epi-lift{ℓ = ℓ} (λ {p q} → ModThA{p = p}{q})
IAimgFA : Lift-Alg A ℓ ℓ IsHomImageOf F[ A ]
IAimgFA = epi→ontohom F[ A ] (Lift-Alg A ℓ ℓ) epiFIA
VIA : Lift-Alg A ℓ ℓ ∈ V ℓ ℓ K
VIA = F[ A ], spFA , IAimgFA

```

The converse inclusion, $V K \subseteq \text{Mod}(\text{Th}(V K))$, 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 ℓ K → A ∈ Mod{X = U[ A ]} (Th (V ℓ ℓ 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 $V K \subseteq K$. On the other hand, we proved that V is expansive in the `Varieties.Func.Closure` module:

```
V-expa : K ⊆ V K
```

so $K (= V K = \text{H S P } K)$ 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>.