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

William DeMeo
New Jersey Institute of Technology, Newark, NJ, USA

Jacques Carette
McMaster University, Hamilton, Canada

Abstract

The Agda Universal Algebra Library is a project aimed at formalizing the foundations of universal algebra, equational logic and model theory in dependent type theory using Agda. In this paper we draw from many components of the library to present a self-contained, formal, constructive proof of Birkhoff's HSP theorem in Martin-Löf dependent type theory. This achieves one of the project's initial goals: to demonstrate the expressive power of inductive and dependent types for representing and reasoning about general algebraic and relational structures by using them to formalize a significant theorem in the field.

2012 ACM Subject Classification Theory of computation → Logic and verification; Computing methodologies → Representation of mathematical objects; Theory of computation → Type theory

Keywords and phrases Agda, constructive mathematics, dependent types, equational logic, formal verification, Martin-Löf type theory, model theory, universal algebra

Digital Object Identifier 10.4230/LIPIcs.TYPES.2021.4

Related Version *Full Version*: `https://arxiv.org/abs/2101.10166`

Supplementary Material *Software (Agda Sources)*: `https://github.com/ualib/agda-algebras` archived at `swh:1:dir:29817e5c87bb55467269dad672f7f4b4152733d7`

Funding *William DeMeo*: partially supported by ERC Consolidator Grant No. 771005.

Acknowledgements This work would not have been possible without the wonderful Agda language and the Agda Standard Library, developed and maintained by The Agda Team [21]. We thank the three anonymous referees for carefully reading the manuscript and offering many excellent suggestions which resulted in a vast improvement in the overall presentation. One referee went above and beyond and provided us with a simpler formalization of free algebras which led to simplifications of the proof of the main theorem. We are extremely grateful for this.

1 Introduction

The Agda Universal Algebra Library (agda-algebras) [8] formalizes the foundations of universal algebra in intensional Martin-Löf type theory (MLTT) using Agda [15, 18]. The library includes a collection of definitions and verified theorems originated in classical (set-theory based) universal algebra and equational logic, but adapted to MLTT.

The first major milestone of the project is a complete formalization of *Birkhoff's variety theorem* (also known as the *HSP theorem*) [4]. To the best of our knowledge, this is the first time Birkhoff's celebrated 1935 result has been formalized in MLTT.[1]

Our first attempt to formalize Birkhoff's theorem suffered from two flaws.[2] First, we assumed function extensionality in MLTT; consequently, it was unclear whether the formalization was fully constructive. Second, an inconsistency could be contrived by taking the

[1] An alternative formalization based on classical set-theory was achieved in [13].
[2] See the Birkhoff.lagda file in the ualib/ualib.gitlab.io repository (15 Jan 2021 commit 71f1738) [6].

type X, representing an arbitrary collection of variable symbols, to be the two element type (see §7.1 for details). To resolve these issues, we developed a new formalization of the HSP theorem based on *setoids* and rewrote much of the agda-algebras library to support this approach. This enabled us to avoid function extensionality altogether. Moreover, the type X of variable symbols was treated with more care using the *context* and *environment* types that Andreas Abel uses in [1] to formalize Birkhoff's completeness theorem. These design choices are discussed further in §2.2–2.3.

What follows is a self-contained formal proof of the HSP theorem in Agda. This is achieved by extracting a subset of the agda-algebras library, including only the pieces needed for the proof, into a single literate Agda file.[3] For spaces reasons, we elide some inessential parts, but strive to preserve the essential content and character of the development. Specifically, routine or overly technical components, as well as anything that does not seem to offer insight into the central ideas of the proof are omitted. (The file src/Demos/HSP.lagda mentioned above includes the full proof.)

In this paper, we highlight some of the more challenging aspects of formalizing universal algebra in type theory. To some extent, this is a sobering glimpse of the significant technical hurdles that must be overcome to do mathematics in dependent type theory. Nonetheless, we hope to demonstrate that MLTT is a relatively natural language for formalizing universal algebra. Indeed, we believe that researchers with sufficient patience and resolve can reap the substantial rewards of deeper insight and greater confidence in their results by using type theory and a proof assistant like Agda. On the other hand, this paper is probably not the best place to learn about the latter, since we assume the reader is already familiar with MLTT and Agda. In summary, our main contribution is to show that a straightforward but very general representation of algebraic structures in dependent type theory is quite practical, as we demonstrate by formalizing a major seminal result of universal algebra.

2 Preliminaries

2.1 Logical foundations

To best emulate MLTT, we use {-# OPTIONS *–without-K –exact-split –safe* #-}; *without-K* disables Streicher's K axiom; *exact-split* directs Agda to accept only definitions behaving like *judgmental* equalities; *safe* ensures that nothing is postulated outright. (See [19, 20, 22].)

We also use some definitions from Agda's standard library (ver. 1.7). As shown in Appendix §A, these are imported using the open import directive and they include some adjustments to "standard" Agda syntax. In particular, we use Type in place of Set, the infix long arrow symbol, _⟶_, in place of Func (the type of "setoid functions," discussed in §2.3), 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 consisting of a type and an equivalence relation on that type. Setoids are useful for representing a set with an explicit, "local" notion of equivalence, instead of relying on an implicit, "global" one as is more common in set theory. In reality, informal mathematical

[3] src/Demos/HSP.lagda in the agda-algebras repository: github.com/ualib/agda-algebras

practice relies on equivalence relations quite pervasively, taking great care to define only functions that preserve equivalences, while eliding the details. To be properly formal, such details must be made explicit. While there are many different workable approaches, the one that requires no additional meta-theory is based on setoids, which is why we adopt it here. While in some settings setoids are found by others to be burdensome, we have not found them to be so for universal algebra.

The agda-algebras library was first developed without setoids, relying on propositional equality instead, along with some experimental, domain-specific types for equivalence classes, quotients, etc. This required postulating function extensionality,[4] which is known to be independent from MLTT [9, 10]; this was unsatisfactory as we aimed to show that the theorems hold directly in MLTT without extra axioms. The present work makes no appeal to functional extensionality or classical axioms like Choice or Excluded Middle.

2.3 Setoid functions

A *setoid function* is a function from one setoid to another that respects the underlying equivalences. If **A** and **B** are setoids, we use **A** ⟶ **B** to denote the type of setoid functions from **A** to **B**. We define the *inverse* of such a function in terms of the image of the function's domain, as follows.

```
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 the Image f ∋ b type is a point a : Carrier **A**, along with a proof p : b ≈ f a, that f maps a to b. Since a proof of Image f ∋ b must include a concrete witness a : Carrier **A**, we can actually *compute* a range-restricted right-inverse of f. Here is the definition of Inv accompanied by a proof that it gives a right-inverse.

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

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

If f : **A** ⟶ **B** then we call f *injective* provided $\forall(a_0\ a_1 : A)$, f ⟨$⟩ $a_0 \approx^B$ f ⟨$⟩ a_1 implies $a_0 \approx^A a_1$; we call f *surjective* provided $\forall(b : B)\ \exists(a : A)$ such that f ⟨$⟩ a $\approx^B$ b. We omit the straightforward Agda definitions.

Factorization of setoid functions[5]

Any (setoid) function f : A ⟶ B factors as a surjective map toIm : A ⟶ Im f followed by an injective map fromIm : Im f ⟶ B.

[4] the axiom asserting that two point-wise equal functions are equal
[5] The code in this paragraph was suggested by an anonymous referee.

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

  Im : (f : A ⟶ B) → Setoid _ _
  Carrier (Im f) = Carrier A
  _≈ˢ_ (Im f) b1 b2 = f ⟨$⟩ b1 ≈ f ⟨$⟩ b2 where open Setoid B

  isEquivalence (Im f) = record { refl = refl ; sym = sym; trans = trans }
    where open Setoid B

  toIm : (f : A ⟶ B) → A ⟶ Im f
  toIm f = record { f = id ; cong = cong f }

  fromIm : (f : A ⟶ B) → Im f ⟶ B
  fromIm f = record { f = λ x → f ⟨$⟩ x ; cong = id }

  fromIm-inj : (f : A ⟶ B) → IsInjective (fromIm f)
  fromIm-inj _ = id

  toIm-surj : (f : A ⟶ B) → IsSurjective (toIm f)
  toIm-surj _ = eq _ (reflˢ B)
```

3 Basic Universal Algebra

We now develop a working vocabulary in MLTT corresponding to classical, single-sorted, set-based universal algebra. We cover a number of important concepts, but limit ourselves to those required to prove Birkhoff's HSP theorem. In each case, we give a type-theoretic version of the informal definition, followed by its Agda implementation.

This section is organized into the following subsections: §3.1 defines a general type of *signatures* of algebraic structures; §3.2 does the same for structures and their products; §3.3 defines *homomorphisms*, *monomorphisms*, and *epimorphisms*, presents types that codify these concepts, and formally verifies some of their basic properties; §3.5–3.6 do the same for *subalgebras* and *terms*, respectively.

3.1 Signatures

An (algebraic) *signature* is a pair $S = (\mathsf{F}\ ,\ \rho)$ where F is a collection of *operation symbols* and $\rho : \mathsf{F} \to \mathsf{N}$ is an *arity function* which maps each operation symbol to its arity. Here, N denotes the *arity type*. Heuristically, the arity ρ f of an operation symbol f ∈ F may be thought of as the number of arguments that f takes as "input." We represent signatures as inhabitants of the following dependent pair type.

```
Signature:(𝓞𝓥:Level)→Type(lsuc(𝓞⊔𝓥))

Signature𝓞𝓥=Σ[F∈Type𝓞](F→Type𝓥)
```

Recalling our syntax for the first and second projections, if S is a signature, then $\mid S \mid$ denotes the set of operation symbols and $\| S \|$ denotes the arity function. Thus, if f : $\mid S \mid$ is an operation symbol in the signature S, then $\| S \|$ f is the arity of f.

We need to augment our Signature type so that it supports algebras over setoid domains. To do so, following Abel [1], we 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:

given operations f and g, with arguments u : ∥ S ∥ f → A and v : ∥ S ∥ g → A, respectively, and a proof of f ≡ g (*intensional* equality), we ought to be able to check whether u and v are pointwise equal. Technically, u and v appear to inhabit different types; of course, this is reconciled by the hypothesis f ≡ g, as we see in the next definition (borrowed from [1]).

```
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 ( _≈_ )
```

This makes it possible to define an operator which translates a signature for algebras over bare types into a signature for algebras over setoids. We denote this operator by ⟨_⟩.

```
⟨_⟩ : 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

An *algebraic structure* $\mathbf{A} = (\mathsf{A}, \mathsf{F}^{A})$ *in the signature* $S = (\mathsf{F}, \rho)$, or S-*algebra*, consists of

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

Our Agda implementation represents algebras as inhabitants of a record type with two fields – a Domain setoid denoting the domain of the algebra, and an Interp function denoting the interpretation in the algebra of each operation symbol in S. We postpone introducing identities until §4.

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

Thus, for each operation symbol in S we have a setoid function f whose domain is a power of Domain and whose codomain is Domain. Further, we define some syntactic sugar to make our formalizations easier to read and reason about. Specifically, if $\mathbf{A}$ is an algebra, then

- 𝔻[$\mathbf{A}$] denotes the Domain setoid of $\mathbf{A}$,
- 𝕌[$\mathbf{A}$] is the underlying carrier of (the Domain setoid of) $\mathbf{A}$, and
- f ̂ $\mathbf{A}$ denotes the interpretation of the operation symbol f in the algebra $\mathbf{A}$.

We omit the straightforward formal definitions (see [7] for details).

Universe levels of algebra types

Types belong to *universes*, which are structured in Agda as follows: Type ℓ : Type (suc ℓ), Type (suc ℓ) : Type (suc (suc ℓ)), ….[6] While this means that Type ℓ has type Type (suc ℓ), it does *not* imply that Type ℓ has type Type (suc (suc ℓ)). In other words, Agda's

[6] suc ℓ denotes the successor of ℓ in the universe hierarchy.

universes are *non-cumulative*. This can be advantageous as it becomes possible to treat size issues more generally and precisely. However, dealing with explicit universe levels can be daunting, and the standard literature (in which uniform smallness is typically assumed) offers little guidance. While in some settings, such as category theory, formalizing in Agda works smoothly with respect to universe levels (see [12]), in universal algebra the terrain is bumpier. Thus, it seems worthwhile to explain how we make use of universe lifting and lowering functions, available in the Agda Standard Library, to develop domain-specific tools for dealing with Agda's non-cumulative universe hierarchy.

The Lift operation of the standard library embeds a type into a higher universe. Specializing Lift to our situation, we define a function Lift-Alg with the following interface.

```
Lift-Alg : Algebra α ρᵃ → (ℓ₀ ℓ₁ : Level) → Algebra (α ⊔ ℓ₀) (ρᵃ ⊔ ℓ₁)
```

Lift-Alg takes an algebra parametrized by levels a and ρ^a and constructs a new algebra whose carrier inhabits Type $(\alpha \sqcup \ell_0)$ and whose equivalence inhabits Rel Carrier $(\rho^a \sqcup \ell_1)$. To be useful, this lifting operation should result in an algebra with the same semantic properties as the one we started with. We will see in §3.4 that this is indeed the case.

Product Algebras

We define the *product* of a family of algebras as follows. Let ι be a universe and I : Type ι a type (the "indexing type"). Then $\mathscr{A}$: I → Algebra α ρ^a represents an *indexed family of algebras*. Denote by $\prod \mathscr{A}$ the *product of algebras* in $\mathscr{A}$ (or *product algebra*), by which we mean the algebra whose domain is the Cartesian product Π i : I , 𝔻[$\mathscr{A}$ i] of the domains of the algebras in $\mathscr{A}$, and whose operations are those arising from pointwise interpretation in the obvious way: if f is a J-ary operation symbol and if a : Π i : I , J → 𝔻[$\mathscr{A}$ i] is, for each i : I, a J-tuple of elements of the domain 𝔻[$\mathscr{A}$ i], then we define the interpretation of f in $\prod \mathscr{A}$ by

```
(f ̂ ∏ 𝒜) a := λ (i : I) → (f ̂ 𝒜 i)(a i).
```

Here is the formal definition of the product algebra type in Agda.

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

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

  Domain (∏ 𝒜) = record { Carrier = ∀ i → 𝕌[ 𝒜 i ]
                        ; _≈_ = λ a b → ∀ i → (_≈ˢ_ 𝔻[ 𝒜 i ]) (a i)(b i)
                        ; isEquivalence =
                            record { refl = λ i → reflᵉ (isEquivalence 𝔻[ 𝒜 i ])
                                   ; sym = λ x i → symᵉ (isEquivalence 𝔻[ 𝒜 i ])(x i)
                                   ; trans = λ x y i → transᵉ (isEquivalence 𝔻[ 𝒜 i ])(x i)(y i) }}

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

Evidently, the carrier of the product algebra type is indeed the (dependent) product of the carriers in the indexed family. The rest of the definitions are the "pointwise" versions of the underlying ones.

3.3 Structure preserving maps and isomorphism

Throughout the rest of the paper, unless stated otherwise, **A** and **B** will denote S-algebras inhabiting the types Algebra α ρ^a and Algebra β ρ^b, respectively.

A *homomorphism* (or "hom") from **A** to **B** is a setoid function h : 𝔻[**A**] ⟶ 𝔻[**B**] that is *compatible* with all basic operations; that is, for every operation symbol f : ∣ S ∣ and all tuples a : ∥ S ∥ f → 𝕌[**A**], we have h ⟨$⟩ (f ̂ **A**) a ≈ (f ̂ **B**) λ x → h ⟨$⟩ (a x).

It is convenient to first formalize "compatible" (compatible-map-op), representing the assertion that a given setoid function h : 𝔻[**A**] ⟶ 𝔻[**B**] commutes with a given operation symbol f, and then generalize over operation symbols to yield the type (compatible-map) of compatible maps from (the domain of) **A** to (the domain of) **B**.

```
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 ( _≈_ )

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

Using these we define the property (IsHom) of being a homomorphism, and finally the type (hom) of homomorphisms from **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
```

Thus, an inhabitant of hom is a pair (h , p) consisting of a setoid function h, from the domain of **A** to that of **B**, along with a proof p that h is a homomorphism.

A *monomorphism* (resp. *epimorphism*) is an injective (resp. surjective) homomorphism. The agda-algebras library defines predicates IsMon and IsEpi for these, as well as mon and epi for the corresponding types.

```
  record IsMon (h : 𝔻[ 𝐀 ] ⟶ 𝔻[ 𝐁 ]) : Type (𝓞 ⊔ 𝓥 ⊔ α ⊔ ρᵃ ⊔ ρᵇ) where
    field isHom : IsHom h
          isInjective : IsInjective h
    HomReduct : hom
    HomReduct = h , isHom

  mon : Type _
  mon = Σ (𝔻[ 𝐀 ] ⟶ 𝔻[ 𝐁 ]) IsMon
```

As with hom, the type mon is a dependent product type; each inhabitant is a pair consisting of a setoid function, say, h, along with a proof that h is a monomorphism.

```
  record IsEpi (h : 𝔻[ 𝐀 ] ⟶ 𝔻[ 𝐁 ]) : Type (𝓞 ⊔ 𝓥 ⊔ α ⊔ β ⊔ ρᵇ) where
    field isHom : IsHom h
          isSurjective : IsSurjective h
    HomReduct : hom
    HomReduct = h , isHom

  epi : Type _
  epi = Σ (𝔻[ 𝐀 ] ⟶ 𝔻[ 𝐁 ]) IsEpi
```

Composition of homomorphisms

The composition of homomorphisms is again a homomorphism, and similarly for epimorphisms and monomorphisms. The proofs of these facts are straightforward so we omit them, but give them the names ∘-hom and ∘-epi so we can refer to them below.

Two structures are *isomorphic* provided there are homomorphisms from each to the other that compose to the identity. We define the following record type to represent this concept.

```
module _ (𝐀 : Algebra α ρᵃ) (𝐁 : Algebra β ρᵇ) where
  open Setoid 𝔻[ 𝐀 ] using () renaming ( _≈_ to _≈ᴬ_ )
  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 a
```

The agda-algebras library also includes formal proof that the to and from maps are bijections and that _≅_ is an equivalence relation, but we suppress these details.

Homomorphic images

We have found that a useful way to encode the concept of *homomorphic image* is to produce a witness, that is, a surjective hom. Thus we define the type of surjective homs and also record the fact that an algebra is its own homomorphic image via the identity hom.[7]

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

IdHomImage : {𝐀 : Algebra α ρᵃ} → 𝐀 IsHomImageOf 𝐀
IdHomImage {α = α}{𝐀 = 𝐀} = 𝒾𝒹 , λ {y} → Image_∋_.eq y refl
  where open Setoid 𝔻[ 𝐀 ] using ( refl )
```

Factorization of homomorphisms

Another theorem in the agda-algebras library, called HomFactor, formalizes the following factorization result: 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. A special case of this result that we use below is the fact that the setoid function factorization we saw above lifts to factorization of homomorphisms. Moreover, we associate a homomorphism h with its image – which is (the domain of) a subalgebra of the codomain of h – using the function HomIm defined below.[8]

```
module _ {𝐀 : Algebra α ρᵃ}{𝐁 : Algebra β ρᵇ} where

  HomIm : (h : hom 𝐀 𝐁) → Algebra _ _
  Domain (HomIm h) = Im ∣ h ∣
  Interp (HomIm h) ⟨$⟩ (f , la) = (f ̂ 𝐀) la
  cong (Interp (HomIm h)) {x1 , x2} {.x1 , y2} (≡.refl , e) =
```

[7] Here and elsewhere we use the shorthand ov $\alpha := \mathcal{O} \sqcup \mathcal{V} \sqcup \alpha$, for any level α.
[8] The definition of HomIm was provided by an anonymous referee.

```
begin
    ∣ h ∣    ⟨$⟩ (Interp 𝐀          ⟨$⟩ (x1 , x2)) ≈⟨ h-compatible                ⟩
  Interp 𝐁 ⟨$⟩ (x1 , λ x → ∣ h ∣ ⟨$⟩ x2 x)        ≈⟨ cong (Interp 𝐁) (≡.refl , e) ⟩
  Interp 𝐁 ⟨$⟩ (x1 , λ x → ∣ h ∣ ⟨$⟩ y2 x)        ≈˘⟨ h-compatible               ⟩
    ∣ h ∣    ⟨$⟩ (Interp 𝐀          ⟨$⟩ (x1 , y2)) ∎
  where open Setoid 𝔻[ 𝐁 ] ; open SetoidReasoning 𝔻[ 𝐁 ]
        open IsHom ∥ h ∥ renaming (compatible to h-compatible)

toHomIm : (h : hom 𝐀 𝐁) → hom 𝐀 (HomIm h)
toHomIm h = toIm ∣ h ∣ , mkhom (reflˢ 𝔻[ 𝐁 ])

fromHomIm : (h : hom 𝐀 𝐁) → hom (HomIm h) 𝐁
fromHomIm h = fromIm ∣ h ∣ , mkhom (IsHom.compatible ∥ h ∥)
```

3.4 Lift-Alg is an algebraic invariant

The Lift-Alg operation neatly resolves the technical problem of universe non-cumulativity because isomorphism classes of algebras are closed under Lift-Alg.

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

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

3.5 Subalgebras

We say that $\mathbf{A}$ is a *subalgebra* of $\mathbf{B}$ 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 an injective hom from $\mathbf{A}$ to $\mathbf{B}$.

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

The subalgebra relation is reflexive, by the identity monomorphism (and transitive by composition of monomorphisms, hence, a *preorder*, though we won't need this fact here).

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

We conclude this subsection with a simple utility function that converts a monomorphism into a proof of a subalgebra relationship.

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

3.6 Terms

Fix a signature S and let X denote an arbitrary nonempty collection of variable symbols. Such a collection is called a *context*. Assume the symbols in X are distinct from the operation symbols of S, that is X ∩ ∣ S ∣ = ∅. A *word* in the language of S is a finite sequence of members of X ∪ ∣ S ∣. We denote the concatenation of such sequences by simple juxtaposition.

Let S_0 denote the set of nullary operation symbols of S. We define by induction on n the sets T_n of *words* over $\mathsf{X} \cup |\ S\ |$ as follows: $T_0 := \mathsf{X} \cup \mathsf{S}_0$ and $T_{n+1} := T_n \cup \mathscr{T}_n$, where $\mathscr{T}_n$ is the collection of all f t such that f : $|\ S\ |$ and t : $\|\ S\ \|$ f $\to T_n$. An S-*term* is a term in the language of S and the collection of all S-*terms* in the context X is Term X := $\bigcup_n T_n$.

In type theory, this translates to two cases: variable injection and applying an operation symbol to a tuple of terms. This 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") in the following inductively defined type.

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

The term algebra

We enrich the Term type to a setoid of S-terms, which will ultimately be the domain of an algebra, called the *term algebra in the signature* S. This requires an equivalence on 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)
```

Below we denote by ≃-isEquiv the easy (omitted) proof that _≃_ is an equivalence relation.

For a given signature S and context X, we define the algebra $\mathbf{T}$ X, known as the *term algebra in* S *over* X. The domain of $\mathbf{T}$ X is Term X and, for each operation symbol f : $|\ S\ |$, we define f $\hat{}$ $\mathbf{T}$ X to be the operation which maps each tuple t : $\|\ S\ \|$ f → Term X of terms to the formal term f t.

```
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 interpretation of terms

Fix a signature S and a context X. An *environment* for $\mathbf{A}$ and X is a setoid whose carrier is a mapping from the variable symbols X to the domain $\mathbb{U}[\ \mathbf{A}\ ]$ and whose equivalence relation is pointwise equality. Our formalization of this concept is the same as that of [1], which Abel uses to formalize Birkhoff's completeness theorem.

```
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) }}
```

The *interpretation* of a term *evaluated* in a particular environment is defined as follows.

```
⟦_⟧ : {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 are proclaimed *equal* if they are equal for all environments.

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

Proof that Equal is an equivalence relation, and that the implication s ≃ t → Equal s t holds for all terms s and t, is also found in [1]. We denote the latter by ≃→Equal in the sequel.

Compatibility of terms

We need to formalize two more concepts involving terms. The first (comm-hom-term) is the assertion that every term commutes with every homomorphism, and the second (interp-prod) is the interpretation of a term in a product algebra.

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

  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 = 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) ⟩
    ⟦ node f t ⟧ᴮ ⟨$⟩ (h ∘ a)        ∎ where open SetoidReasoning 𝔻[ 𝐁 ]

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

  interp-prod : (p : Term X) → ∀ ρ → (⟦ ⨅ 𝒜 ⟧ p) ⟨$⟩ ρ ≈ λ i → (⟦ 𝒜 i ⟧ p) ⟨$⟩ λ x → (ρ x) i
  interp-prod (ℊ x)        = λ ρ i → ≃→Equal (𝒜 i) (ℊ x) (ℊ x) ≃-isRefl λ _ → (ρ x) i
  interp-prod (node f t) = λ ρ   → cong (Interp (⨅ 𝒜)) ( ≡.refl , λ j k → interp-prod (t j) ρ k )
```

4 Equational Logic

4.1 Term identities, equational theories, and the ⊨ relation

An *S-term equation* (or *S-term identity*) is an ordered pair (p , q) of S-terms, also denoted by p ≈ q. We define an *equational theory* (or *algebraic theory*) to be a pair T = (S , $\mathscr{E}$) consisting of a signature S and a collection $\mathscr{E}$ of S-term equations.[9]

[9] Some authors reserve the term *theory* for a *deductively closed* set of equations, that is, a set of equations that is closed under entailment.

We say that the algebra $\mathbf{A}$ *models* the identity p ≈ q and we write $\mathbf{A}$ ⊧ p ≈ q if for all ρ : X → 𝔻[$\mathbf{A}$] we have ⟦ p ⟧ ⟨\$⟩ ρ ≈ ⟦ q ⟧ ⟨\$⟩ ρ. In other words, when interpreted in the algebra $\mathbf{A}$, the terms p and q are equal no matter what values are assigned to variable symbols occurring in p and q. If $\mathcal{K}$ is a class of algebras of a given signature, then we write $\mathcal{K}$ |⊧ p ≈ q and say that $\mathcal{K}$ *models* the identity p ≈ q provided $\mathbf{A}$ ⊧ p ≈ q for every $\mathbf{A} \in \mathcal{K}$.

```
module _ {X : Type χ} where
  _⊧_≈_ : Algebra α ρᵃ → Term X → Term X → Type _
  A ⊧ p ≈ q = Equal p q where open Environment A

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

We represent a set of term identities as a predicate over pairs of terms, and we denote by $\mathbf{A}$ ⊨ ℰ the assertion that $\mathbf{A}$ models p ≈ q for all (p , q) ∈ ℰ.

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

An important property of the binary relation ⊧ is *algebraic invariance* (i.e., invariance under isomorphism). We formalize this result as follows.

```
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 ]
```

If $\mathcal{K}$ is a class of S-algebras, the set of identities modeled by $\mathcal{K}$, denoted Th $\mathcal{K}$, is called the *equational theory* of $\mathcal{K}$. If ℰ is a set of S-term identities, the class of algebras modeling ℰ, denoted Mod ℰ, is called the *equational class axiomatized* by ℰ. We codify these notions in the next two definitions.

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

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

4.2 The Closure Operators H, S, P and V

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

- H $\mathcal{K}$:= the class of all homomorphic images of members of $\mathcal{K}$;
- S $\mathcal{K}$:= the class of all subalgebras of members of $\mathcal{K}$;
- P $\mathcal{K}$:= the class of all products of members of $\mathcal{K}$.

H, S, and P are *closure operators* (expansive, monotone, and idempotent). A class $\mathscr{K}$ of S-algebras is said to be *closed under the taking of homomorphic images* provided H $\mathscr{K}$ ⊆ $\mathscr{K}$. Similarly, $\mathscr{K}$ is *closed under the taking of subalgebras* (resp., *arbitrary products*) provided S $\mathscr{K}$ ⊆ $\mathscr{K}$ (resp., P $\mathscr{K}$ ⊆ $\mathscr{K}$). The operators H, S, and P can be composed with one another repeatedly, forming yet more closure operators. We represent these three closure operators in type theory as follows.

```
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))
```

Identities modeled by an algebra **A** are also modeled by every homomorphic image of **A** and by every subalgebra of **A**. We refer to these facts as ⊧-H-invar and ⊧-S-invar; their definitions are similar to that of ⊧-I-invar. An identity satisfied by all algebras in an indexed collection is also satisfied by the product of algebras in the collection. We refer to this fact as ⊧-P-invar.

A *variety* is a class of S-algebras that is closed under the taking of homomorphic images, subalgebras, and arbitrary products. If we define V $\mathscr{K}$:= H (S (P $\mathscr{K}$)), then $\mathscr{K}$ is a variety iff V $\mathscr{K}$ ⊆ $\mathscr{K}$. The class V $\mathscr{K}$ is called the *varietal closure* of $\mathscr{K}$. Here is how we define V in type theory. (The explicit universe level declarations that appear in the definition are needed for disambiguation.)

```
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))
```

The classes H $\mathscr{K}$, S $\mathscr{K}$, P $\mathscr{K}$, and V $\mathscr{K}$ all satisfy the same term identities. We will only use a subset of the inclusions needed to prove this assertion.[10] First, the closure operator H preserves the identities modeled by the given class; this follows almost immediately from the invariance lemma ⊧-H-invar.

```
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 , BimgA) = ⊧-H-invar{p = p}{q} (σ 𝐀 kA) BimgA
```

The analogous preservation result for S is a consequence of the invariance lemma ⊧-S-invar; the converse, which we call S-id2, has an equally straightforward proof.

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

[10] The others are included in the Setoid.Varieties.Preservation module of the agda-algebras library.

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

The agda-algebras library includes analogous pairs of implications for P, H, and V, called P-id1, P-id2, H-id1, etc. whose formalizations we suppress.

5 Free Algebras

5.1 The absolutely free algebra

The term algebra **T** X is the *absolutely free* S-algebra over X. That is, for every S-algebra **A**, the following hold.

- Every function from X to 𝕌[**A**] lifts to a homomorphism from **T** X to **A**.
- That homomorphism is unique.

Here we formalize the first of these properties by defining the lifting function free-lift and its setoid analog free-lift-func, and then proving the latter is a homomorphism.[11]

```
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 = flcong where
    open Setoid 𝔻[ 𝐀 ] using ( _≈_ ) renaming ( reflexive to reflexiveᴬ )
    flcong : ∀ {s t} → s ≃ t → free-lift s ≈ free-lift t
    flcong (_≃_.rfl x) = reflexiveᴬ (≡.cong h x)
    flcong (_≃_.gnl x) = cong (Interp 𝐀) (≡.refl , λ i → flcong (x i))

  lift-hom : hom (𝐓 X) 𝐀
  lift-hom = free-lift-func ,
    mkhom λ{_}{a} → cong (Interp 𝐀) (≡.refl , λ i → (cong free-lift-func){a i} ≃-isRefl)
```

It turns out that the interpretation of a term p in an environment η is the same as the free lift of η evaluated at p. We apply this fact a number of times in the sequel.

```
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

Given an arbitrary class 𝒦 of S-algebras, we cannot expect that **T** X belongs to 𝒦. Indeed, there may be no free algebra in 𝒦. Nonetheless, it is always possible to construct an algebra that is free for 𝒦 and belongs to the class S (P 𝒦). Such an algebra is called a *relatively*

[11] For the proof of uniqueness, see the Setoid.Terms.Properties module of the agda-algebras library.

free algebra over X (relative to $\mathcal{K}$). There are several informal approaches to defining this algebra. We now describe the approach on which our formal construction is based and then we present the formalization.

Let 𝔽[X] denote the relatively free algebra over X. We represent 𝔽[X] as the quotient $\mathbf{T}$ X / ≈ where x ≈ y if and only if h x = h y for every homomorphism h from $\mathbf{T}$ X into a member of $\mathcal{K}$. More precisely, if $\mathbf{A} \in \mathcal{K}$ and h : hom ($\mathbf{T}$ X) $\mathbf{A}$, then h factors as $\mathbf{T}$ X $\overset{h}{\twoheadrightarrow}$ HomIm h $\overset{\subseteq}{\rightarrowtail}$ $\mathbf{A}$ and $\mathbf{T}$ X / ker h ≅ HomIm h ≤ $\mathbf{A}$; that is, $\mathbf{T}$ X / ker h is (isomorphic to) an algebra in S $\mathcal{K}$. Letting ≈ := ⋂ {θ ∈ Con $\mathbf{T}$ X | $\mathbf{T}$ X / θ ∈ S $\mathcal{K}$ }, observe that 𝔽[X] := $\mathbf{T}$ X / ≈ is a subdirect product of the algebras {$\mathbf{T}$ X / ker h } as h ranges over all homomorphisms from $\mathbf{T}$ X to algebras in $\mathcal{K}$. Thus, 𝔽[X] ∈ P (S $\mathcal{K}$) ⊆ S (P $\mathcal{K}$). As we have seen, every map ρ : X → 𝕌[$\mathbf{A}$] extends uniquely to a homomorphism h : hom ($\mathbf{T}$ X) $\mathbf{A}$ and h factors through the natural projection $\mathbf{T}$ X → 𝔽[X] (since ≈ ⊆ ker h) yielding a unique homomorphism from 𝔽[X] to $\mathbf{A}$ extending ρ.

In Agda we construct 𝔽[X] as a homomorphic image of $\mathbf{T}$ X in the following way. First, given X we define $\mathbf{C}$ as the product of pairs ($\mathbf{A}$, ρ) of algebras $\mathbf{A} \in \mathcal{K}$ along with environments ρ : X → 𝕌[$\mathbf{A}$]. To do so, we contrive an index type for the product; each index is a triple ($\mathbf{A}$, p, ρ) where $\mathbf{A}$ is an algebra, p is proof of $\mathbf{A} \in \mathcal{K}$, and ρ : X → 𝕌[$\mathbf{A}$] is an arbitrary environment.

```
module FreeAlgebra (𝒦 : Pred (Algebra α ρᵃ) ℓ) where
  private c = α ⊔ ρᵃ ; ι = ov c ⊔ ℓ
  ℑ : {χ : Level} → Type χ → Type (ι ⊔ χ)
  ℑ X = Σ[ 𝐀 ∈ Algebra α ρᵃ ] 𝐀 ∈ 𝒦 × (X → 𝕌[ 𝐀 ])

  𝐂 : {χ : Level} → Type χ → Algebra (ι ⊔ χ)(ι ⊔ χ)
  𝐂 X = ⨅ {I = ℑ X} ∣_∣
```

We then define 𝔽[X] to be the image of a homomorphism from $\mathbf{T}$ X to $\mathbf{C}$ as follows.

```
homC : (X : Type χ) → hom (𝐓 X) (𝐂 X)
homC X = ⨅-hom-co _ (λ i → lift-hom (snd ∥ i ∥))

𝔽[_] : {χ : Level} → Type χ → Algebra (ov χ) (ι ⊔ χ)
𝔽[ X ] = HomIm (homC X)
```

Observe that if the identity p ≈ q holds in all $\mathbf{A} \in \mathcal{K}$ (for all environments), then p ≈ q holds in 𝔽[X]; equivalently, the pair (p , q) belongs to the kernel of the natural homomorphism from $\mathbf{T}$ X onto 𝔽[X]. This natural epimorphism is defined as follows.

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

  epiF[_] : (X : Type c) → epi (𝐓 X) 𝔽[ X ]
  epiF[ X ] = ∣ toHomIm (homC X) ∣ , record { isHom = ∥ toHomIm (homC X) ∥
                                           ; isSurjective = toIm-surj ∣ homC X ∣ }

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

Before formalizing the HSP theorem in the next section, we need to prove the following important property of the relatively free algebra: For every algebra $\mathbf{A}$, if $\mathbf{A}$ ⊨ Th (V $\mathcal{K}$), then there exists an epimorphism from 𝔽[A] onto $\mathbf{A}$, where A denotes the carrier of $\mathbf{A}$.

```
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 FreeAlgebra 𝒦 using ( 𝔽[_] ; 𝐂 )
  open Setoid 𝔻[ 𝐀 ] using ( refl ; sym ; trans ) renaming ( Carrier to A ; _≈_ to _≈ᴬ_ )

  F-ModTh-epi : 𝐀 ∈ Mod (Th 𝒦) → epi 𝔽[ A ] 𝐀
  F-ModTh-epi A∈ModThK = φ , isEpi where

    φ : 𝔻[ 𝔽[ A ] ] ⟶ 𝔻[ 𝐀 ]
    _⟨$⟩_ φ            = free-lift{𝐀 = 𝐀} id
    cong φ {p} {q} pq = Goal
      where
      lift-pq : (p , q) ∈ Th 𝒦
      lift-pq 𝐁 x ρ = begin
        ⟦ p ⟧ ⟨$⟩ ρ  ≈⟨ free-lift-interp {𝐀 = 𝐁} ρ p ⟩
        free-lift ρ p ≈⟨ pq (𝐁 , x , ρ)              ⟩
        free-lift ρ q ≈˘⟨ free-lift-interp{𝐀 = 𝐁} ρ q ⟩
        ⟦ q ⟧ ⟨$⟩ ρ ∎
          where open SetoidReasoning 𝔻[ 𝐁 ] ; open Environment 𝐁 using ( ⟦_⟧ )

      Goal : free-lift id p ≈ᴬ free-lift id q
      Goal = begin
        free-lift id p ≈˘⟨ free-lift-interp {𝐀 = 𝐀} id p      ⟩
        ⟦ p ⟧ ⟨$⟩ id  ≈⟨ A∈ModThK {p = p} {q} lift-pq id ⟩
        ⟦ q ⟧ ⟨$⟩ id  ≈⟨ free-lift-interp {𝐀 = 𝐀} id q      ⟩
        free-lift id q ∎
          where open SetoidReasoning 𝔻[ 𝐀 ] ; open Environment 𝐀 using ( ⟦_⟧ )

    isEpi : IsEpi 𝔽[ A ] 𝐀 φ
    isEpi = record { isHom = mkhom refl ; isSurjective = eq (g _) refl }

  F-ModThV-epi : 𝐀 ∈ Mod (Th (V ℓ ι 𝒦)) → epi 𝔽[ A ] 𝐀
  F-ModThV-epi A∈ModThVK = F-ModTh-epi λ {p}{q} → Goal {p}{q}
    where
    Goal : 𝐀 ∈ Mod (Th 𝒦)
    Goal {p}{q} x ρ = A∈ModThVK{p}{q} (V-id1 ℓ {p = p}{q} x) ρ
```

6 Birkhoff's Variety Theorem

Let $\mathscr{K}$ be a class of algebras and recall that $\mathscr{K}$ is a *variety* provided it is closed under homomorphisms, subalgebras and products; equivalently, V $\mathscr{K} \subseteq \mathscr{K}$. (Observe that $\mathscr{K} \subseteq$ V $\mathscr{K}$ holds for all $\mathscr{K}$ since V is a closure operator.) We call $\mathscr{K}$ an *equational class* if it is the class of all models of some set of identities.

Birkhoff's variety theorem, also known as the HSP theorem, asserts that $\mathscr{K}$ is an equational class if and only if it is a variety. In this section, we present the statement and proof of this theorem – first in a style similar to what one finds in textbooks (e.g., [3, Theorem 4.41]), and then formally in the language of MLTT.

6.1 Informal proof

(⇒) *Every equational class is a variety.* Indeed, suppose $\mathscr{K}$ is an equational class axiomatized by term identities $\mathscr{E}$; that is, $\mathbf{A} \in \mathscr{K}$ iff $\mathbf{A} \models \mathscr{E}$. Since the classes H $\mathscr{K}$, S $\mathscr{K}$, P $\mathscr{K}$ and $\mathscr{K}$ all satisfy the same set of equations, we have V $\mathscr{K}$ $|\!\models$ p ≈ q for all (p , q) $\in \mathscr{E}$, so V $\mathscr{K} \subseteq \mathscr{K}$.

($\Leftarrow$) *Every variety is an equational class.*[12] Let $\mathcal{K}$ be an arbitrary variety. We will describe a set of equations that axiomatizes $\mathcal{K}$. A natural choice is to take Th $\mathcal{K}$ and try to prove that $\mathcal{K}$ = Mod (Th $\mathcal{K}$). Clearly, $\mathcal{K}$ ⊆ Mod (Th $\mathcal{K}$). To prove the converse inclusion, let $\mathbf{A}$ ∈ Mod (Th $\mathcal{K}$). It suffices to find an algebra $\mathbf{F}$ ∈ S (P $\mathcal{K}$) such that $\mathbf{A}$ is a homomorphic image of $\mathbf{F}$, as this will show that $\mathbf{A}$ ∈ H (S (P $\mathcal{K}$)) = $\mathcal{K}$.

Let X be such that there exists a surjective environment ρ : X → 𝕌[$\mathbf{A}$].[13] By the lift-hom lemma, there is an epimorphism h : $\mathbf{T}$ X → 𝕌[$\mathbf{A}$] that extends ρ. Put 𝔽[X] := $\mathbf{T}$ X/≈ and let g : $\mathbf{T}$ X → 𝔽[X] be the natural epimorphism with kernel ≈. We claim ker g ⊆ ker h. If the claim is true, then there is a map f : 𝔽[X] → $\mathbf{A}$ such that f ∘ g = h, and since h is surjective so is f. Therefore, $\mathbf{A}$ ∈ H (𝔽 X) ⊆ Mod (Th $\mathcal{K}$) completing the proof.

It remains to prove the claim ker g ⊆ ker h. Let u, v be terms and assume g u = g v. Since $\mathbf{T}$ X is generated by X, there are terms p, q such that u = ⟦ $\mathbf{T}$ X ⟧ p and v = ⟦ $\mathbf{T}$ X ⟧ q. Therefore, ⟦ 𝔽[X] ⟧ p = g (⟦ $\mathbf{T}$ X ⟧ p) = g u = g v = g (⟦ $\mathbf{T}$ X ⟧ q) = ⟦ 𝔽[X] ⟧ q, so $\mathcal{K}$ |⊨ p ≈ q; thus, (p , q) ∈ Th $\mathcal{K}$. Since $\mathbf{A}$ ∈ Mod (Th $\mathcal{K}$), we obtain $\mathbf{A}$ ⊨ p ≈ q, which implies that h u = (⟦ $\mathbf{A}$ ⟧ p) ⟨\$⟩ ρ = (⟦ $\mathbf{A}$ ⟧ q) ⟨\$⟩ ρ = h v, as desired.

6.2 Formal proof

($\Rightarrow$) *Every equational class is a variety.* We need an arbitrary equational class, which we obtain by starting with an arbitrary collection $\mathcal{E}$ of equations and then defining $\mathcal{K}$ = Mod $\mathcal{E}$, the class axiomatized by $\mathcal{E}$. We prove that $\mathcal{K}$ is a variety by showing that $\mathcal{K}$ = V $\mathcal{K}$. The inclusion $\mathcal{K}$ ⊆ V $\mathcal{K}$, which holds for all classes $\mathcal{K}$, is called the *expansive* property of V.

```
module _ (𝒦 : Pred(Algebra α ρᵃ) (α ⊔ ρᵃ ⊔ ov ℓ)) where
  V-expa : 𝒦 ⊆ V ℓ (ov (α ⊔ ρᵃ ⊔ ℓ)) 𝒦
  V-expa {x = 𝐀}kA = 𝐀 , (𝐀 , (⊤ , (λ _ → 𝐀), (λ _ → kA), Goal), ≤-reflexive), IdHomImage
    where
    open Setoid 𝔻[ 𝐀 ]              using ( refl )
    open Setoid 𝔻[ ⨅ (λ _ → 𝐀) ] using () renaming ( refl to refl⨅ )
    to⨅    : 𝔻[ 𝐀 ]              ⟶ 𝔻[ ⨅ (λ _ → 𝐀) ]
    to⨅    = record { f = λ x _ → x ; cong = λ xy _ → xy }
    from⨅ : 𝔻[ ⨅ (λ _ → 𝐀) ] ⟶ 𝔻[ 𝐀 ]
    from⨅ = record { f = λ x → x tt ; cong = λ xy → xy tt }
    Goal   : 𝐀 ≅ ⨅ (λ x → 𝐀)
    Goal   = mkiso (to⨅ , mkhom refl⨅) (from⨅ , mkhom refl) (λ _ _ → refl) (λ _ → refl)
```

Observe how $\mathbf{A}$ is expressed as (isomorphic to) a product with just one factor (itself), that is, the product ⨅ (λ x → $\mathbf{A}$) indexed over the one-element type ⊤.

For the inclusion V $\mathcal{K}$ ⊆ $\mathcal{K}$, recall lemma V-id1 which asserts that $\mathcal{K}$ |⊨ p ≈ q implies V ℓ ι $\mathcal{K}$ |⊨ p ≈ q; whence, if $\mathcal{K}$ is an equational class, then V $\mathcal{K}$ ⊆ $\mathcal{K}$, as we now confirm.

```
module _ {ℓ : Level}{X : Type ℓ}{ℰ : {Y : Type ℓ} → Pred (Term Y × Term Y) (ov ℓ)} where
  private 𝒦 = Mod{α = ℓ}{ℓ}{X} ℰ -- an arbitrary equational class

  EqCl⇒Var : V ℓ (ov ℓ) 𝒦 ⊆ 𝒦
  EqCl⇒Var {𝐀} vA {p} {q} pℰq ρ = V-id1 ℓ {𝒦} {p} {q} (λ _ x τ → x pℰq τ) 𝐀 vA ρ
```

By V-expa and Eqcl⇒Var, every equational class is a variety.

[12] The proof we present here is based on [3, Theorem 4.41].

[13] Informally, this is done by assuming X has cardinality at least max(| 𝕌[$\mathbf{A}$] |, ω). Later we will see how to construct an X with the required property in type theory.

(⇐) *Every variety is an equational class.* To fix an arbitrary variety, start with an arbitrary class $\mathcal{K}$ of S-algebras and take the *varietal closure*, V $\mathcal{K}$. We prove that V $\mathcal{K}$ is precisely the collection of algebras that model Th (V $\mathcal{K}$); that is, V $\mathcal{K}$ = Mod (Th (V $\mathcal{K}$)). The inclusion V $\mathcal{K}$ ⊆ Mod (Th (V $\mathcal{K}$)) is a consequence of the fact that Mod Th is a closure operator.

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

  ModTh-closure : V{β = β}{ρᵇ}{γ}{ρᶜ}{δ}{ρᵈ} ℓ ι 𝒦 ⊆ Mod{X = X} (Th (V ℓ ι 𝒦))
  ModTh-closure {x = 𝐀} vA {p} {q} x ρ = x 𝐀 vA ρ
```

Our proof of the inclusion Mod (Th (V $\mathcal{K}$)) ⊆ V $\mathcal{K}$ is carried out in two steps.

1. Prove 𝔽[X] ≤ **C** X.
2. Prove that every algebra in Mod (Th (V $\mathcal{K}$)) is a homomorphic image of 𝔽[X].

From 1 we have 𝔽[X] ∈ S (P $\mathcal{K}$)), since **C** X is a product of algebras in $\mathcal{K}$. From this and 2 will follow Mod (Th (V $\mathcal{K}$)) ⊆ H (S (P $\mathcal{K}$)) (= V $\mathcal{K}$), as desired.

- 1. To prove 𝔽[X] ≤ **C** X, we construct a homomorphism from 𝔽[X] to **C** X and then show it is injective, so 𝔽[X] is (isomorphic to) a subalgebra of **C** X.

```
open FreeHom {ℓ = ℓ}{𝒦}
open FreeAlgebra 𝒦 using (homC ; 𝔽[_] ; 𝐂 )
homFC : hom 𝔽[ X ] (𝐂 X)
homFC = fromHomIm (homC X)

monFC : mon 𝔽[ X ] (𝐂 X)
monFC = ∣ homFC ∣ , record { isHom = ∥ homFC ∥
                           ; isInjective = λ {x}{y}→ fromIm-inj ∣ homC X ∣ {x}{y} }
F≤C : 𝔽[ X ] ≤ 𝐂 X
F≤C = mon→≤ monFC

open FreeAlgebra 𝒦 using ( ℑ )

SPF : 𝔽[ X ] ∈ S ι (P ℓ ι 𝒦)
SPF = 𝐂 X , ((ℑ X) , (∣_∣ , ((λ i → fst ∥ i ∥) , ≅-refl))) , F≤C
```

- 2. Every algebra in Mod (Th (V $\mathcal{K}$)) is a homomorphic image of 𝔽[X]. Indeed,

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

  Var⇒EqCl : ∀ 𝐀 → 𝐀 ∈ Mod (Th (V ℓ ι 𝒦)) → 𝐀 ∈ V ℓ ι 𝒦
  Var⇒EqCl 𝐀 ModThA = 𝔽[ 𝕌[ 𝐀 ] ] , (SPF{ℓ = ℓ} 𝒦 , Aim)
   where
   open FreeAlgebra 𝒦 using ( 𝔽[_] )
   epiFlA : epi 𝔽[ 𝕌[ 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) epiFlA

   Aim : 𝐀 IsHomImageOf 𝔽[ 𝕌[ 𝐀 ] ]
   Aim = ∘-hom ∣ φ ∣(from Lift-≅), ∘-IsSurjective _ _ ∥ φ ∥(fromIsSurjective(Lift-≅{𝐀 = 𝐀}))
```

  By ModTh-closure and Var⇒EqCl, we have V $\mathcal{K}$ = Mod (Th (V $\mathcal{K}$)) for every class $\mathcal{K}$ of S-algebras. Thus, every variety is an equational class.

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

7 Conclusion

7.1 Discussion

How do we differ from the classical, set-theoretic approach? Most noticeable is our avoidance of all *size* issues. By using universe levels and level polymorphism, we always make sure we are in a *large enough* universe. So we can easily talk about "all algebras such that ..." because these are always taken from a bounded (but arbitrary) universe.

Our use of setoids introduces nothing new: all the equivalence relations we use were already present in the classical proofs. The only "new" material is that we have to prove that functions respect those equivalences.

Our first attempt to formalize Birkhoff's theorem was not sufficiently careful in its handling of variable symbols X. Specifically, this type was unconstrained; it is meant to represent the informal notion of a "sufficiently large" collection of variable symbols. Consequently, we postulated surjections from X onto the domains of all algebras in the class under consideration. But then, given a signature S and a one-element S-algebra $\mathbf{A}$, by choosing X to be the empty type $\perp$, our surjectivity postulate gives a map from $\perp$ onto the singleton domain of $\mathbf{A}$. (For details, see the Demos.ContraX module which constructs the counterexample in Agda.)

7.2 Related work

There have been a number of efforts to formalize parts of universal algebra in type theory besides ours. The Coq proof assistant, based on the Calculus of Inductive Constructions, was used by Capretta, in [5], and Spitters and Van der Weegen, in [17], to formalized the basics of universal algebra and some classical algebraic structures. In [11] Gunther et al developed what seemed (prior to the agda-algebras library) the most extensive library of formalized universal algebra to date. Like agda-algebras, [11] is based on dependent type theory, is programmed in Agda, and goes beyond the basic isomorphism theorems to include some equational logic. Although their coverage is less extensive than that of agda-algebras, Gunther et al do treat *multi-sorted* algebras, whereas agda-algebras is currently limited to single-sorted structures.

As noted by Abel [1], Amato et al, in [2], have formalized multi-sorted algebras with finitary operators in UniMath. The restriction to finitary operations was due to limitations of the UniMath type theory, which does not have W-types nor user-defined inductive types. Abel also notes that Lynge and Spitters, in [14], formalize multi-sorted algebras with finitary operators in *Homotopy type theory* ([16]) using Coq [23]. HoTT's higher inductive types enable them to define quotients as types, without the need for setoids. Lynge and Spitters prove three isomorphism theorems concerning subalgebras and quotient algebras, but do not formalize universal algebras nor varieties. Finally, in [1], Abel gives a new formal proof of the soundness and completeness theorem for multi-sorted algebraic structures.

References

1 Andreas Abel. Birkhoff's Completeness Theorem for multi-sorted algebras formalized in Agda. *CoRR*, abs/2111.07936, 2021. `arXiv:2111.07936`.

2 Gianluca Amato, Marco Maggesi, and Cosimo Perini Brogi. Universal Algebra in UniMath. *CoRR*, abs/2102.05952, 2021. `arXiv:2102.05952`.

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

4 G Birkhoff. On the structure of abstract algebras. *Proceedings of the Cambridge Philosophical Society*, 31(4):433–454, October 1935.

5 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`.

6 William DeMeo. The Agda Universal Algebra Library. GitHub.com, 2020. Ver. 1.0.0. Source code: gitlab.com/ualib/ualib.gitlab.io.

7 William DeMeo and Jacques Carette. A Machine-checked Proof of Birkhoff's Variety Theorem in Martin-Löf Type Theory. *CoRR*, abs/2101.10166, 2021. Source code: github.com/ualib/agda-algebras/. `doi:10.48550/ARXIV.2101.10166`.

8 William DeMeo and Jacques Carette. The Agda Universal Algebra Library (agda-algebras). GitHub.com, 2021. Ver. 2.0.1. Source code: agda-algebras-v.2.0.1.zip. Documentation: ualib.org. GitHub repo: github.com/ualib/agda-algebras. `doi:10.5281/zenodo.5765793`.

9 Martín Hötzel Escardó. Introduction to Univalent Foundations of mathematics with Agda. `https://www.cs.bham.ac.uk/~mhe/HoTT-UF-in-Agda-Lecture-Notes/`, May 2019. Accessed on 30 Nov 2021.

10 Martín Hötzel Escardó. Introduction to Univalent Foundations of mathematics with Agda. *CoRR*, abs/1911.00580, 2019. `arXiv:1911.00580`.

11 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:10.1016/j.entcs.2018.10.010`.

12 Jason Z. S. Hu and Jacques Carette. Formalizing Category Theory in Agda. In *Proceedings of the 10th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2021, pages 327–342, New York, NY, USA, 2021. Association for Computing Machinery. `doi:10.1145/3437992.3439922`.

13 Artur Korniłowicz. Birkhoff theorem for many sorted algebras, 1999.

14 Andreas Lynge and Bas Spitters. Universal Algebra in HoTT. In *Proceedings of the 25th International Conference on Types for Proofs and Programs (TYPES 2019)*, 2019. URL: `http://www.ii.uib.no/~bezem/abstracts/TYPES_2019_paper_7`.

15 Ulf Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Department of Computer Science and Engineering, Chalmers University of Technology, SE-412 96 Göteborg, Sweden, September 2007.

16 The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. Lulu and The Univalent Foundations Program, Institute for Advanced Study, 2013. URL: `https://homotopytypetheory.org/book`.

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

18 The Agda Team. *Agda Language Reference*, 2021. URL: `https://agda.readthedocs.io/en/v2.6.1/language/index.html`.

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

20 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`.

21 The Agda Team. *The Agda Standard Library*, 2021. URL: `https://github.com/agda/agda-stdlib`.

22 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`.

23 The Coq Development Team. *The Coq proof assistant reference manual*. LogiCal Project, 2004. Version 8.0. URL: `http://coq.inria.fr`.

A Imports from the Agda Standard Library

We import a number of definitions from Agda's standard library (ver. 1.7), as shown below. Notice that these include some adjustments to "standard" Agda syntax; in particular, we use Type in place of Set, the infix long arrow symbol, _⟶_, in place 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 _×_.

```
-- Import 16 definitions from the Agda Standard Library.
open import Data.Unit.Polymorphic using ( ⊤ ; tt )
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 ; 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 ( )
open _⟶_ renaming ( f to _⟨$⟩_ ) using ( cong )
open IsEquivalence renaming ( refl to reflᵉ )
                   renaming ( sym to symᵉ )
                   renaming ( trans to transᵉ ) using ( )
open Setoid renaming ( refl to reflˢ )
            renaming ( sym to symˢ )
            renaming ( trans to transˢ )
            renaming ( _≈_ to _≈ˢ_ ) using ( Carrier ; isEquivalence )

-- 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

private variable α ρᵃ β ρᵇ γ ρᶜ δ ρᵈ ρ χ ℓ : Level ; Γ Δ : Type χ
```