The Agda Universal Algebra Library and Birkhoff's Theorem in Dependent Type Theory

William DeMeo

Department of Algebra, Charles University in Prague

Abstract

The Agda Universal Algebra Library (UALib) is a library of types and programs (theorems and proofs) we developed to formalize the foundations of universal algebra in Martin-Löf-style dependent type theory using the Agda programming language and proof assistant. This paper describes the UALib and demonstrates that Agda is accessible to working mathematicians (such as ourselves) as a tool for formally verifying nontrivial results in general algebra and related fields. The library 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 UALib project is a complete proof of Birkhoff's HSP theorem. To the best of our knowledge, this is the first time Birkhoff's theorem has been formulated and proved in dependent type theory and verified with a proof assistant.

In this paper we describe the Agda UALib and the formal proof of Birkhoff's theorem, discussing some of the technically challenging parts of the proof. In so doing, we illustrate the effectiveness of dependent type theory, Agda, and the UALib for formally verifying nontrivial theorems in universal algebra and equational logic.

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

Keywords and phrases Universal algebra, Equational logic, Martin-Löf Type Theory, Birkhoff's HSP Theorem, Formalization of mathematics, Agda, Proof assistant

Digital Object Identifier 10.4230/LIPIcs..2021.0

Related Version hosted on arXiv
Extended Version: `arxiv.org/pdf/2101.10166`

Supplementary Material
Documentation: `ualib.org`
Software: `https://gitlab.com/ualib/ualib.gitlab.io.git`

Acknowledgements The author wishes to thank Hyeyoung Shin and Siva Somayyajula for their contributions to this project and Martín Escardó for creating the Type Topology library and teaching a course on Univalent Foundations of Mathematics with Agda at the 2019 Midlands Graduate School in Computing Science. Of course, this work would not exist in its current form without the Agda 2 language by Ulf Norell.[1]

1 Introduction

To support formalization in type theory of research level mathematics in universal algebra and related fields, we present the Agda Universal Algebra Library (Agda UALib), a software library containing formal statements and proofs of the core definitions and results of universal

[1] Agda 2 is partially based on code from Agda 1 by Catarina Coquand and Makoto Takeyama, and from Agdalight by Ulf Norell and Andreas Abel.

algebra. The UALib is written in Agda [7], a programming language and proof assistant based on Martin-Löf Type Theory that not only supports dependent and inductive types, but also provides powerful *proof tactics* for proving things about the objects that inhabit these types.

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

- Capretta [2] (1999) formalized the basics of universal algebra in the Calculus of Inductive Constructions using the Coq proof assistant;
- Spitters and van der Weegen [9] (2011) formalized the basics of universal algebra and some classical algebraic structures, also in the Calculus of Inductive Constructions using the Coq proof assistant, promoting the use of type classes as a preferable alternative to setoids;
- Gunther, et al [6] (2018) developed what seems to be (prior to the UALib) the most extensive library of formal universal algebra to date; in particular, this work includes a formalization of some basic equational logic; the project (like the UALib) uses Martin-Löf Type Theory and the Agda proof assistant.

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, goals of this prior work seem to be the formalization of special classical algebraic structures, as opposed to the general theory of (universal) algebras.

Although the Agda UALib project was initiated relatively recently (in 2018), the part of universal algebra and equational logic that it formalizes extends beyond the scope of prior efforts. In particular, the UALib now includes a constructive, machine-checked proof of Birkhoff's variety theorem. Most other proofs of this theorem that we know of are informal and nonconstructive. After the completion this work, however, we learned about a constructive version of Birkhoff's theorem that was proved by Carlström in [3]. The latter is presented in an informal style and, as far as we know, was never implemented formally and machine-checked with a proof assistant. Nonetheless, a comparison of the version of Birkhoff's theorem presented in [3] to our version in Agda would be interesting, and we will make some remarks about that in §**??**.

The seminal idea for the Agda UALib project was the observation that, on the one hand, a number of fundamental constructions in universal algebra can be defined recursively, and theorems about them proved by structural induction, while, on the other hand, inductive and dependent types make possible very precise formal representations of recursively defined objects, which often admit elegant constructive proofs of properties of such objects. An important feature of such proofs in type theory is that they are total functional programs and, as such, they are computable and composable.

Finally, our own research experience has taught us that a proof assistant and programming language (like Agda), when equipped with specialized libraries and domain-specific tactics to automate proof idioms of a particular field, can be an extremely powerful and effective asset. We believe that such libraries, and the proof assistants they support, will eventually become indispensable tools in the working mathematician's toolkit.

1.1 Contributions and organization

TODO: revise this section when the paper is almost done.

Apart from the library itself, we describe the formal implementation and proof of a deep result, Garrett Birkhoff's celebrated HSP theorem [1], which was among the first major results of universal algebra. The theorem states that a *variety* (a class of algebras closed under quotients, subalgebras, and products) is an equational class (defined by the set of identities satisfied by all its members). The fact that we now have a formal proof of this is noteworthy, not only because this is the first time the theorem has been proved in dependent type theory and verified with a proof assistant, but also because the proof is constructive. As the paper [3] of Carlström makes clear, it is a highly nontrivial exercise to take a well-known informal proof of a theorem like Birkhoff's and show that it can be formalized using only constructive logic and natural deduction, without appealing to, say, the Law of the Excluded Middle or the Axiom of Choice.

Each of the sections that follow describes the most important or noteworthy components of the UALib. We cover just enough to keep the paper somewhat self-contained. Of course, space does not permit us to cover every definition and theorem required to present a complete formal proof of a theorem like Birkhoff's. We remedy this in two ways. First, throughout the paper we include pointers to places in the documentation where the omitted material can be found. Second, we include an appendix containing Agda background, discussion of the foundational assumptions of the UALib, and definitions of some important types of dependent type theory and how they are represented in Agda and in the UALib. We hope this appendix is especially useful to readers who are not already proficient users of Agda.

Finally, the official sources of information about the Agda UALib are

- `ualib.org` (the web site) includes every line of code in the library, rendered as html and accompanied by documentation, and
- `gitlab.com/ualib/ualib.gitlab.io` (the source code) freely available and licensed under the Creative Commons Attribution-ShareAlike 4.0 International License.

2 Algebras

We define the type of *operations* and, as an example, the type of *projections*.

```
Op : 𝓥 ̇ → 𝓤 ̇ → 𝓤 ⊔ 𝓥 ̇
Op I A = (I → A) → A

π : {I : 𝓥 ̇ } {A : 𝓤 ̇ } → I → Op I A
π i x = x i
```

The type Op encodes the arity of an operation as an arbitrary type $I : \mathcal{V}^{\cdot}$, which gives us a very general way to represent an operation as a function type with domain $I \to A$ (the type of "tuples") and codomain A. The last two lines of the code block above codify the i-th I-ary projection operation on A.

2.1 Signature type

We define the signature of an algebraic structure in Agda like this.

```
Signature : (𝓞 𝓥 : Universe) → (𝓞 ⊔ 𝓥) ⁺ ̇
Signature 𝓞 𝓥 = Σ F : 𝓞 ̇ , (F → 𝓥 ̇)
```

Here $\mathcal{O}$ is the universe level of operation symbol types, while $\mathcal{V}$ is the universe level of arity types. We denote the first and second projections by ∣_∣ and ∥_∥ (§A.3) so if S is a signature, then ∣ S ∣ denotes the type of *operation symbols*, and ∥ S ∥ denotes the *arity* function. If f : ∣ S ∣ is an operation symbol in the signature S, then ∥ S ∥ f is the arity of f. For example, here is the signature of *monoids*, as a member of the type Signature $\mathcal{O}$ $\mathcal{U}_0$.

```
data monoid-op : 𝓞 ̇ where
  e : monoid-op
  · : monoid-op
```

```
monoid-sig : Signature 𝓞 𝓤₀
monoid-sig = monoid-op , λ { e → 𝟘; · → 𝟚 }
```

This signature has two operation symbols, `e` and `·`, and a function `λ { e → 𝟘 ; · → 𝟚 }` which maps `e` to the empty type `𝟘` (since `e` is nullary), and `·` to the 2-element type `𝟚` (since `·` is binary).

2.2 Algebra type

For a fixed signature `S : Signature 𝓞 𝓥` and universe `𝓤`, we define the type of *algebras* in the signature S (or S-*algebras*) and with *domain* (or *carrier*) `A : 𝓤 ̇` as follows[2]

```
Algebra : (𝓤 : Universe)(S : Signature 𝓞 𝓥) → 𝓞 ⊔ 𝓥 ⊔ 𝓤 ⁺ ̇

Algebra 𝓤 S = Σ A : 𝓤 ̇ , ((f : ∣ S ∣) → Op (∥ S ∥ f) A)
```

We may refer to an inhabitant of `Algebra S 𝓤` as an "∞-algebra" because its domain can be an arbitrary type, say, `A : 𝓤 ̇` and need not be truncated at some level (§A.6). In particular, A need not be a set (as defined in §A.6).[3]

Next we define a convenient shorthand for the interpretation of an operation symbol. We use this often in the sequel.

```
_̂_ : (f : ∣ S ∣)(𝑨 : Algebra 𝓤 S) → (∥ S ∥ f → ∣ 𝑨 ∣) → ∣ 𝑨 ∣

f ̂ 𝑨 = λ x → (∥ 𝑨 ∥ f) x
```

This is similar to the standard notation that one finds in the literature and seems much more natural to us than the double bar notation that we started with.

We assume that we always have at our disposal an arbitrary collection X of variable symbols such that, for every algebra $\mathbf{A}$, no matter the type of its domain, we have a surjective map h_0 : `X → ∣ 𝑨 ∣` from variables onto the domain of $\mathbf{A}$.

```
_↠_ : {𝓤 𝓧 : Universe} → 𝓧 ̇ → Algebra 𝓤 S → 𝓧 ⊔ 𝓤 ̇
X ↠ 𝑨 = Σ h : (X → ∣ 𝑨 ∣) , Epic h
```

Finally, we define the type of *product algebras* the obvious way.

```
⨅ : {𝓘 : Universe}{I : 𝓘 ̇ }(𝒜 : I → Algebra 𝓤 S ) → Algebra (𝓘 ⊔ 𝓤) S
⨅ {𝓘}{I} 𝒜 =
  ((i : I) → ∣ 𝒜 i ∣) , λ(f : ∣ S ∣)(𝒂 : ∥ S ∥ f → (j : I) → ∣ 𝒜 j ∣)(i : I) → (f ̂ 𝒜 i) λ{x → 𝒂 x i}
```

3 Quotient Types and Quotient Algebras

For a binary relation R on A, we denote a single R-class by `[ a ] R` (this denotes the class containing a). We denote the type of all classes of a relation R on A by `𝒞 { A } { R }`. These

[2] The Agda UALib includes an alternative definition of the type of algebras using records, but we don't discuss these here since they are not needed in the sequel. We refer the interested reader to [4] and the html documentation available at `https://ualib.gitlab.io/UALib.Algebras.Algebras.html`.

[3] We could pause here to define the type of "0-algebras," which are algebras whose domains are sets. This type is probably closer to what most of us think of when doing informal universal algebra. However, in the UALib we have so far only needed to know that the domain of an algebra is a set in a handful of specific instances, so it seems preferable to work with general (∞-)algebras throughout the library and then assume *uniquness of identity proofs* explicitly where, and only where, a proof relies on this assumption.

are defined as in the UALib as follows.

```
[_] : {A : 𝓤 ̇ } → A → Rel A 𝓡 → Pred A 𝓡
[ a ] R = λ x → R a x

𝒞 : {A : 𝓤 ̇}{R : Rel A 𝓡} → Pred A 𝓡 → (𝓤 ⊔ 𝓡 ⁺) ̇
𝒞 {A}{R} = λ (C : Pred A 𝓡) → Σ a ꞉ A , C ≡ ( [ a ] R)
```

There are a few ways we could define the quotient with respect to a relation. We have found the following to be the most convenient.

```
_/_ : (A : 𝓤 ̇ ) → Rel A 𝓡 → 𝓤 ⊔ (𝓡 ⁺) ̇
A / R = Σ C ꞉ Pred A 𝓡 , 𝒞{A}{R} C
```

We then have the following introduction and elimination rules for a class with a designated representative.

```
⟦_⟧ : {A : 𝓤 ̇} → A → {R : Rel A 𝓡} → A / R
⟦ a ⟧ {R} = ([ a ] R) , a , 𝓇ℯ𝒻ℓ


⌜_⌝ : {A : 𝓤 ̇}{R : Rel A 𝓡} → A / R → A
⌜ 𝒂 ⌝ = ∣ ∥ 𝒂 ∥ ∣
```

3.1 Quotient extensionality

We will need a subsingleton identity type for congruence classes over sets so that we can equate two classes, even when they are presented using different representatives. For this we assume that our relations are on sets, rather than arbitrary types. As mentioned earlier, this is equivalent to assuming that there is at most one proof that two elements of a set are the same. The following *class extensionality principle* accomplishes this for us.

```
class-extensionality' : propext 𝓡 → global-dfunext → {A : 𝓤 ̇}{a a' : A}{R : Rel A 𝓡}
   →                    (∀ a x → is-subsingleton (R a x)) → (∀ C → is-subsingleton (𝒞 C))
   →                    IsEquivalence R
                        ----------------------------
   →                    R a a' → (⟦ a ⟧ {R}) ≡ (⟦ a' ⟧ {R})
```

We omit the proof. (See [4] or `ualib.org` for details.)

3.2 Compatibility

We say that a (unary) operation $f : X \to X$ is *compatible* with (or *respects*, or *preserves*) the binary relation R on X just in case $\forall\ x, y :$ X, we have $R\ x\ y \to R\ (f\ x)\ (f\ y)$. Now suppose $\boldsymbol{\mathcal{U}}$, $\boldsymbol{\mathcal{V}}$, and $\boldsymbol{\mathcal{W}}$ are universes and assume the following typing judgments: $\gamma : \boldsymbol{\mathcal{V}}\ ̇$, $X : \boldsymbol{\mathcal{U}}\ ̇$. We lift the definition of compatibility from unary to γ-ary operations in the following obvious way: for all $u\ v : \gamma \to X$ (γ-tuples of X), we say that the pair $u\ v$ is R-related, and we write lift-rel $R\ u\ v$ provided $\forall\ i : \gamma$, $R\ (u\ i)\ (v\ i)$. The function lift-rel is defined as follows.

```
lift-rel : Rel Z 𝓦 → (γ → Z) → (γ → Z) → 𝓥 ⊔ 𝓦 ̇
lift-rel R f g = ∀ x → R (f x) (g x)
```

If $f : (\gamma \to X) \to X$ is a γ-ary operation on X, we say that f is *compatible* with R provided for all $u\ v : \gamma \to X$, lift-rel $R\ u\ v$ implies $R\ (f\ u)\ (f\ v)$.

```
compatible-op : {𝑨 : Algebra 𝓤 𝑆} → ∣ 𝑆 ∣ → Rel ∣ 𝑨 ∣ 𝓦 → 𝓤 ⊔ 𝓥 ⊔ 𝓦 ̇
compatible-op {𝑨} f R = ∀{𝒂}{𝒃} → (lift-rel R) 𝒂 𝒃 → R ((f ̂ 𝑨) 𝒂) ((f ̂ 𝑨) 𝒃)
```

Finally, to represent that all basic operations of an algebra are compatible with a given relation, we define

```
compatible : (𝑨 : Algebra 𝓤 𝑆) → Rel ∣ 𝑨 ∣ 𝓦 → 𝓞 ⊔ 𝓤 ⊔ 𝓥 ⊔ 𝓦 ̇
compatible 𝑨 R = ∀ f → compatible-op{𝑨} f R
```

We'll see this definition of compatibility at work very soon when we define congruence relations in the next section.

3.3 Congruence relations

This UALib.Relations.Congruences module of the Agda UALib defines three alternative representations of congruence relations of an algebra—first as a function, then as a predicate on binary relations, and finally as a record type.

```
Con : {𝓤 : Universe}(A : Algebra 𝓤 S) → 𝓞 ⊔ 𝓥 ⊔ 𝓤 ⁺ ̇
Con {𝓤} A = Σ θ ꞉ ( Rel ∣ A ∣ 𝓤 ) , IsEquivalence θ × compatible A θ

con : {𝓤 : Universe}(A : Algebra 𝓤 S) → Pred (Rel ∣ A ∣ 𝓤) (𝓞 ⊔ 𝓥 ⊔ 𝓤)
con A = λ θ → IsEquivalence θ × compatible A θ

record Congruence {𝓤 𝓦 : Universe} (A : Algebra 𝓤 S) : 𝓞 ⊔ 𝓥 ⊔ 𝓤 ⊔ 𝓦 ⁺ ̇ where
  constructor mkcon
  field
    ⟨_⟩ : Rel ∣ A ∣ 𝓦
    Compatible : compatible A ⟨_⟩
    IsEquiv : IsEquivalence ⟨_⟩

open Congruence

compatible-equivalence : {𝓤 𝓦 : Universe}{𝑨 : Algebra 𝓤 S} → Rel ∣ 𝑨 ∣ 𝓦 → 𝓞 ⊔ 𝓥 ⊔ 𝓦 ⊔ 𝓤 ̇
compatible-equivalence {𝓤}{𝓦} {𝑨} R = compatible 𝑨 R × IsEquivalence R
```

3.4 Quotient algebras

An important construction in universal algebra is the quotient of an algebra $\mathbf{A}$ with respect to a congruence relation θ of $\mathbf{A}$. This quotient is typically denoted by $\mathbf{A} \,/\, \theta$ and Agda allows us to define and express quotients using this standard notation.

```
_╱_ : {𝓤 𝓡 : Universe}(A : Algebra 𝓤 S) → Congruence{𝓤}{𝓡} A → Algebra (𝓤 ⊔ 𝓡 ⁺) S
A ╱ θ = (( ∣ A ∣ / ⟨ θ ⟩ ) , -- carrier
           (λ f args -- operations
             → ([ (f ̂ A) (λ i₁ → ∣ ∥ args i₁ ∥ ∣) ] ⟨ θ ⟩) ,
               ((f ̂ A) (λ i₁ → ∣ ∥ args i₁ ∥ ∣) , refl _ )
           )
         )
```

4 Homomorphisms, terms, and subalgebras

4.1 Homomorphisms

The definition of homomorphism we use is a standard, *extensional* one; that is, the homomorphism condition holds pointwise. This will become clearer once we have the formal definitions in hand. Generally speaking, though, we say that two functions $f\ g : X \to Y$ are extensionally equal iff they are pointwise equal, that is, for all $x : X$ we have $f\ x \equiv g\ x$.

To define *homomorphism*, we first say what it means for an operation f, interpreted in the algebras $\mathbf{A}$ and $\mathbf{B}$, to commute with a function $g : A \to B$.

```
compatible-op-map : {𝓠 𝓤 : Universe}(A : Algebra 𝓠 S)(B : Algebra 𝓤 S)
                    (f : ∣ S ∣)(g : ∣ A ∣ → ∣ B ∣) → 𝓥 ⊔ 𝓤 ⊔ 𝓠 ̇
compatible-op-map A B f g = ∀ 𝒂 → g ((f ̂ A) 𝒂) ≡ (f ̂ B) (g ∘ 𝒂)
```

Note the appearance of the shorthand $\forall\ \boldsymbol{a}$ in the definition of compatible-op-map. We can get away with this in place of $\boldsymbol{a} : \| S \|\ f \to |\ \mathbf{A}\ |$ since Agda is able to infer that the $\boldsymbol{a}$ here must be a tuple on $|\ \mathbf{A}\ |$ of "length" $\| S \|\ f$ (the arity of f).

Next we will define the type hom $\mathbf{A}\ \mathbf{B}$ of *homomorphisms* from $\mathbf{A}$ to $\mathbf{B}$ in terms of the property is-homomorphism.

```
is-homomorphism : {𝓠 𝓤 : Universe}(A : Algebra 𝓠 S)(B : Algebra 𝓤 S)
 →                (∣ A ∣ → ∣ B ∣) → 𝓞 ⊔ 𝓥 ⊔ 𝓠 ⊔ 𝓤 ̇
is-homomorphism A B g = ∀ (f : ∣ S ∣) → compatible-op-map A B f g

hom : {𝓠 𝓤 : Universe} → Algebra 𝓠 S → Algebra 𝓤 S → 𝓞 ⊔ 𝓥 ⊔ 𝓠 ⊔ 𝓤 ̇
hom A B = Σ g ꞉ (∣ A ∣ → ∣ B ∣) , is-homomorphism A B g
```

We define an inductive datatype called Term which, not surprisingly, represents the type of *terms* in a given signature. Here the type $X : \mathfrak{X}\ ̇$ represents an arbitrary collection of variable symbols.

```
data Term {𝓧 : Universe}{X : 𝓧 ̇} : 𝓞 ⊔ 𝓥 ⊔ 𝓧 ⁺ ̇ where
  generator : X → Term{𝓧}{X}
  node : (f : ∣ S ∣)(args : ∥ S ∥ f → Term{𝓧}{X}) → Term
```

4.2 Terms

Terms can be viewed as acting on other terms and we can form an algebraic structure whose domain and basic operations are both the collection of term operations. We call this the *term algebra* and denote it by $\mathbf{T}\ X$.

```
T : {𝓧 : Universe}(X : 𝓧 ̇) → Algebra (𝓞 ⊔ 𝓥 ⊔ 𝓧 ⁺) S
T {𝓧} X = Term{𝓧}{X} , node
```

The term algebra is *absolutely free*, or *universal*, for algebras in the signature S. That is, for every S-algebra $\mathbf{A}$, every map $h : X \to |\ \mathbf{A}\ |$ lifts to a homomorphism from $\mathbf{T}\ X$ to $\mathbf{A}$, and the induced homomorphism is unique. This is proved by induction on the structure of terms, as follows.

```
free-lift : {𝓧 𝓤 : Universe}{X : 𝓧 ̇}(A : Algebra 𝓤 S)(h : X → ∣ A ∣) → ∣ T X ∣ → ∣ A ∣
free-lift _ h (generator x) = h x
```

```
free-lift 𝑨 h (node f args) = (f ̂ 𝑨) λ i → free-lift 𝑨 h (args i)

lift-hom : {𝓧 𝓤 : Universe}{X : 𝓧 ̇}(𝑨 : Algebra 𝓤 𝑆)(h : X → ∣ 𝑨 ∣) → hom (𝑻 X) 𝑨
lift-hom 𝑨 h = free-lift 𝑨 h , λ f a → ap (_ ̂ 𝑨) 𝓇ℯ𝒻𝓁

free-unique : {𝓧 𝓤 : Universe}{X : 𝓧 ̇} → funext 𝓥 𝓤
  →         (𝑨 : Algebra 𝓤 𝑆)(g h : hom (𝑻 X) 𝑨)
  →         (∀ x → ∣ g ∣ (generator x) ≡ ∣ h ∣ (generator x))
  →         (t : Term{𝓧}{X})
            ---------------
  →         ∣ g ∣ t ≡ ∣ h ∣ t

free-unique _ _ _ _ p (generator x) = p x
free-unique fe 𝑨 g h p (node f args) =
  ∣ g ∣ (node f args)            ≡⟨ ∥ g ∥ f args ⟩
  (f ̂ 𝑨)(λ i → ∣ g ∣ (args i)) ≡⟨ ap (_ ̂ 𝑨) γ ⟩
  (f ̂ 𝑨)(λ i → ∣ h ∣ (args i)) ≡⟨ (∥ h ∥ f args)⁻¹ ⟩
  ∣ h ∣ (node f args) ∎
  where γ = fe λ i → free-unique fe 𝑨 g h p (args i)
```

4.3 Subalgebras

4.3.1 Subuniverses

The UALib.Subalgebras.Subuniverses module defines, unsurprisingly, the Subuniverses type. Perhaps counterintuitively, we begin by defining the collection of all subuniverses of an algebra. Therefore, the type will be a predicate of predicates on the domain of the given algebra.

```
Subuniverses : {𝓠 𝓤 : Universe}(𝑨 : Algebra 𝓠 𝑆) → Pred (Pred ∣ 𝑨 ∣ 𝓤) (𝓞 ⊔ 𝓥 ⊔ 𝓠 ⊔ 𝓤)
Subuniverses 𝑨 B = (f : ∣ 𝑆 ∣)(a : ∥ 𝑆 ∥ f → ∣ 𝑨 ∣) → Im a ⊆ B → (f ̂ 𝑨) a ∈ B
```

An important concept in universal algebra is the subuniverse generated by a subset of the domain of an algebra. We define the following inductive type to represent this concept.

```
data Sg {𝓤 𝓦 : Universe}(𝑨 : Algebra 𝓤 𝑆)(X : Pred ∣ 𝑨 ∣ 𝓦) :
  Pred ∣ 𝑨 ∣ (𝓞 ⊔ 𝓥 ⊔ 𝓦 ⊔ 𝓤) where
    var : ∀ {v} → v ∈ X → v ∈ Sg 𝑨 X
    app : (f : ∣ 𝑆 ∣)(𝒂 : ∥ 𝑆 ∥ f → ∣ 𝑨 ∣) → Im 𝒂 ⊆ Sg 𝑨 X → (f ̂ 𝑨) 𝒂 ∈ Sg 𝑨 X
```

The proof that Sg X is a subuniverse is as trivial as they come.

```
sgIsSub : {𝓤 𝓦 : Universe}{𝑨 : Algebra 𝓤 𝑆}{X : Pred ∣ 𝑨 ∣ 𝓦} → Sg 𝑨 X ∈ Subuniverses 𝑨
sgIsSub = app
```

And the proof that Sg X is the smallest subuniverse containing X is not much harder and proceeds by induction on the shape of elements of Sg X.

```
sgIsSmallest : {𝓤 𝓦 𝓡 : Universe}(𝑨 : Algebra 𝓤 𝑆){X : Pred ∣ 𝑨 ∣ 𝓦}(Y : Pred ∣ 𝑨 ∣ 𝓡)
  → Y ∈ Subuniverses 𝑨 → X ⊆ Y → Sg 𝑨 X ⊆ Y

sgIsSmallest _ _ _ X⊆Y (var v∈X) = X⊆Y v∈X
sgIsSmallest 𝑨 Y Y≤A X⊆Y (app f 𝒂 p) = fa∈Y
  where
    IH : Im 𝒂 ⊆ Y
```

```
    IH i = sgIsSmallest 𝑨 Y Y≤A X⊆Y (p i)

    fa∈Y : (f ̂ 𝑨) 𝒂 ∈ Y
    fa∈Y = Y≤A f 𝒂 IH
```

Evidently, when the element of Sg X is constructed as app $f\, \boldsymbol{a}\, p$, we may assume (the induction hypothesis) that the arguments $\boldsymbol{a}$ belong to Y. Then the result of applying f to $\boldsymbol{a}$ must also belong to Y, since Y is a subuniverse.

4.3.2 Subalgebra type

Given algebras $\mathbf{A}$: Algebra $\mathcal{W}$ S and $\mathbf{B}$: Algebra $\mathcal{U}$ S, we say that $\mathbf{B}$ is a *subalgebra* of $\mathbf{A}$, and (in the UALib) we write $\mathbf{B}$ IsSubalgebraOf $\mathbf{A}$, just in case $\mathbf{B}$ can be embedded in $\mathbf{A}$; in other terms, there exists a map $h : |\ \mathbf{A}\ | \to |\ \mathbf{B}\ |$ from the universe of $\mathbf{A}$ to the universe of $\mathbf{B}$ such that h is an embedding (i.e., is-embedding h holds) and h is a homomorphism from $\mathbf{A}$ to $\mathbf{B}$.

```
_IsSubalgebraOf_ : {𝓤 𝓦 : Universe}(𝑩 : Algebra 𝓤 𝑆)(𝑨 : Algebra 𝓦 𝑆) → 𝓞 ⊔ 𝓥 ⊔ 𝓤 ⊔ 𝓦 ̇
𝑩 IsSubalgebraOf 𝑨 = Σ h ꞉ (∣ 𝑩 ∣ → ∣ 𝑨 ∣) , is-embedding h × is-homomorphism 𝑩 𝑨 h
```

Here is some convenient syntactic sugar for the subalgebra relation.

```
_≤_ : {𝓤 𝓠 : Universe}(𝑩 : Algebra 𝓤 𝑆)(𝑨 : Algebra 𝓠 𝑆) → 𝓞 ⊔ 𝓥 ⊔ 𝓤 ⊔ 𝓠 ̇
𝑩 ≤ 𝑨 = 𝑩 IsSubalgebraOf 𝑨
```

We can now write $\mathbf{B} \leq \mathbf{A}$ to assert that $\mathbf{B}$ is a subalgebra of $\mathbf{A}$.

5 Equations and Varieties

Let S be a signature. By an *identity* or *equation* in S we mean an ordered pair of terms, written $p \approx q$, from the term algebra $\mathbf{T}$ X. If $\mathbf{A}$ is an S-algebra we say that $\mathbf{A}$ *satisfies* $p \approx q$ provided $p\ ̇\ \mathbf{A} \equiv q\ ̇\ \mathbf{A}$ holds. In this situation, we write $\mathbf{A} \models p \approx q$ and say that $\mathbf{A}$ *models* the identity $p \approx q$. If $\mathcal{K}$ is a class of algebras, all of the same signature, we write $\mathcal{K} \models p \approx q$ if, for every $\mathbf{A} \in \mathcal{K}$, $\mathbf{A} \models p \approx q$.[4]

5.1 Types for Theories and Models

The binary "models" relation $\models$ relating algebras (or classes of algebras) to the identities that they satisfy is defined in the UALib.Varieties.ModelTheory module. Agda supports the definition of infix operations and relations, and we use this to define $\models$ so that we may write, e.g., $\mathbf{A} \models p \approx q$ or $\mathcal{K} \models p \approx q$.

```
_⊧_≈_ : {𝓤 𝓧 : Universe}{X : 𝓧 ̇} → Algebra 𝓤 𝑆 → Term{𝓧}{X} → Term → 𝓤 ⊔ 𝓧 ̇

𝑨 ⊧ p ≈ q = (p ̇ 𝑨) ≡ (q ̇ 𝑨)

_⊧_≋_ : {𝓤 𝓧 : Universe}{X : 𝓧 ̇} → Pred (Algebra 𝓤 𝑆) (OV 𝓤)
  → Term{𝓧}{X} → Term → 𝓞 ⊔ 𝓥 ⊔ 𝓧 ⊔ 𝓤 ⁺ ̇
```

[4] 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 $\mathcal{K} \models p \approx q$ to denote this relation. To reiterate, if $\mathcal{K}$ is a class of S-algebras, we write $\mathcal{K} \models p \approx q$ if every $\mathbf{A} \in \mathcal{K}$ satisfies $\mathbf{A} \models p \approx q$.

```
_⊫_≋_ 𝒦 p q = {𝑨 : Algebra _ 𝑆} → 𝒦 𝑨 → 𝑨 ⊧ p ≈ q
```

The Agda UALib makes available the standard notation Th $\mathscr{K}$ for the set of identities that hold for all algebras in a class $\mathscr{K}$, as well as Mod $\mathscr{E}$ for the class of algebras that satisfy all identities in a given set $\mathscr{E}$.

```
Th : {𝓤 𝓧 : Universe}{X : 𝓧 ̇} → Pred (Algebra 𝓤 𝑆) (OV 𝓤)
  → Pred (Term{𝓧}{X} × Term) (𝓞 ⊔ 𝓥 ⊔ 𝓧 ⊔ 𝓤 ⁺)

Th 𝒦 = λ (p , q) → 𝒦 ⊫ p ≋ q

Mod : {𝓤 𝓧 : Universe}(X : 𝓧 ̇) → Pred (Term{𝓧}{X} × Term{𝓧}{X}) (𝓞 ⊔ 𝓥 ⊔ 𝓧 ⊔ 𝓤 ⁺)
  → Pred (Algebra 𝓤 𝑆) (𝓞 ⊔ 𝓥 ⊔ 𝓧 ⁺ ⊔ 𝓤 ⁺)

Mod X ℰ = λ A → ∀ p q → (p , q) ∈ ℰ → A ⊧ p ≈ q
```

5.2 Inductive types for closure operators

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

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

A *variety* is a class $\mathscr{K}$ of algebras in a fixed signature that is closed under the taking of homomorphic images (H), subalgebras (S), and arbitrary products (P). That is, $\mathscr{K}$ is a variety if and only if H S P $\mathscr{K} \subseteq \mathscr{K}$.

The UALib.Varieties.Varieties module of the Agda UALib introduces three new inductive types that represent the closure operators H, S, P. Separately, an inductive type V is defined which represents closure under all three operators. These definitions have been fine-tuned to strike a balance between faithfully representing the desired semantics and facilitating proof by induction.

```
data H {𝓤 𝓦 : Universe}(𝒦 : Pred (Algebra 𝓤 𝑆)(OV 𝓤)) :
  Pred (Algebra (𝓤 ⊔ 𝓦) 𝑆)(OV (𝓤 ⊔ 𝓦)) where
    hbase : {𝑨 : Algebra 𝓤 𝑆} → 𝑨 ∈ 𝒦 → lift-alg 𝑨 𝓦 ∈ H 𝒦
    hlift : {𝑨 : Algebra 𝓤 𝑆} → 𝑨 ∈ H{𝓤}{𝓤} 𝒦 → lift-alg 𝑨 𝓦 ∈ H 𝒦
    hhimg : {𝑨 𝑩 : Algebra _ 𝑆} → 𝑨 ∈ H{𝓤}{𝓦} 𝒦 → 𝑩 is-hom-image-of 𝑨 → 𝑩 ∈ H 𝒦
    hiso : {𝑨 : Algebra _ 𝑆}{𝑩 : Algebra _ 𝑆} → 𝑨 ∈ H{𝓤}{𝓤} 𝒦 → 𝑨 ≅ 𝑩 → 𝑩 ∈ H 𝒦


data S {𝓤 𝓦 : Universe}(𝒦 : Pred (Algebra 𝓤 𝑆) (OV 𝓤)) :
  Pred (Algebra (𝓤 ⊔ 𝓦) 𝑆) (OV (𝓤 ⊔ 𝓦)) where
    sbase : {𝑨 : Algebra 𝓤 𝑆} → 𝑨 ∈ 𝒦 → lift-alg 𝑨 𝓦 ∈ S 𝒦
    slift : {𝑨 : Algebra 𝓤 𝑆} → 𝑨 ∈ S{𝓤}{𝓤} 𝒦 → lift-alg 𝑨 𝓦 ∈ S 𝒦
    ssub : {𝑨 : Algebra 𝓤 𝑆}{𝑩 : Algebra _ 𝑆} → 𝑨 ∈ S{𝓤}{𝓤} 𝒦 → 𝑩 ≤ 𝑨 → 𝑩 ∈ S 𝒦
    ssubw : {𝑨 𝑩 : Algebra _ 𝑆} → 𝑨 ∈ S{𝓤}{𝓦} 𝒦 → 𝑩 ≤ 𝑨 → 𝑩 ∈ S 𝒦
    siso  : {𝑨 : Algebra 𝓤 𝑆}{𝑩 : Algebra _ 𝑆} → 𝑨 ∈ S{𝓤}{𝓤} 𝒦 → 𝑨 ≅ 𝑩 → 𝑩 ∈ S 𝒦


data P {𝓤 𝓦 : Universe} (𝒦 : Pred (Algebra 𝓤 𝑆) (OV 𝓤)) :
  Pred (Algebra (𝓤 ⊔ 𝓦) 𝑆) (OV (𝓤 ⊔ 𝓦)) where
    pbase  : {𝑨 : Algebra 𝓤 𝑆} → 𝑨 ∈ 𝒦 → lift-alg 𝑨 𝓦 ∈ P 𝒦
```

```
    pliftu : {A : Algebra 𝓤 S} → A ∈ P{𝓤}{𝓤} 𝒦 → lift-alg A 𝓦 ∈ P 𝒦
    pliftw : {A : Algebra (𝓤 ⊔ 𝓦) S} → A ∈ P{𝓤}{𝓦} 𝒦 → lift-alg A (𝓤 ⊔ 𝓦) ∈ P 𝒦
    produ  : {I : 𝓦 ̇ }{𝒜 : I → Algebra 𝓤 S} → (∀ i → (𝒜 i) ∈ P{𝓤}{𝓤} 𝒦) → ⨅ 𝒜 ∈ P 𝒦
    prodw  : {I : 𝓦 ̇ }{𝒜 : I → Algebra _ S} → (∀ i → (𝒜 i) ∈ P{𝓤}{𝓦} 𝒦) → ⨅ 𝒜 ∈ P 𝒦
    pisou  : {A : Algebra 𝓤 S}{B : Algebra _ S} → A ∈ P{𝓤}{𝓤} 𝒦 → A ≅ B → B ∈ P 𝒦
    pisow  : {A B : Algebra _ S} → A ∈ P{𝓤}{𝓦} 𝒦 → A ≅ B → B ∈ P 𝒦
```

The operator that corresponds to closure with respect to HSP is often denoted by $\mathbb{V}$; we represent it by the inductive type V, defined as follows.

```
data V {𝓤 𝓦 : Universe}(𝒦 : Pred (Algebra 𝓤 S) (OV 𝓤)) :
  Pred (Algebra (𝓤 ⊔ 𝓦) S)(OV (𝓤 ⊔ 𝓦)) where
    vbase  : {A : Algebra 𝓤 S} → A ∈ 𝒦 → lift-alg A 𝓦 ∈ V 𝒦
    vlift  : {A : Algebra 𝓤 S} → A ∈ V{𝓤}{𝓤} 𝒦 → lift-alg A 𝓦 ∈ V 𝒦
    vliftw : {A : Algebra _ S} → A ∈ V{𝓤}{𝓦} 𝒦 → lift-alg A (𝓤 ⊔ 𝓦) ∈ V 𝒦
    vhimg  : {A B : Algebra _ S} → A ∈ V{𝓤}{𝓦} 𝒦 → B is-hom-image-of A → B ∈ V 𝒦
    vssub  : {A : Algebra 𝓤 S}{B : Algebra _ S} → A ∈ V{𝓤}{𝓤} 𝒦 → B ≤ A → B ∈ V 𝒦
    vssubw : {A B : Algebra _ S} → A ∈ V{𝓤}{𝓦} 𝒦 → B ≤ A → B ∈ V 𝒦
    vprodu : {I : 𝓦 ̇}{𝒜 : I → Algebra 𝓤 S} → (∀ i → (𝒜 i) ∈ V{𝓤}{𝓤} 𝒦) → ⨅ 𝒜 ∈ V 𝒦
    vprodw : {I : 𝓦 ̇}{𝒜 : I → Algebra _ S} → (∀ i → (𝒜 i) ∈ V{𝓤}{𝓦} 𝒦) → ⨅ 𝒜 ∈ V 𝒦
    visou  : {A : Algebra 𝓤 S}{B : Algebra _ S} → A ∈ V{𝓤}{𝓤} 𝒦 → A ≅ B → B ∈ V 𝒦
    visow  : {A B : Algebra _ S} → A ∈ V{𝓤}{𝓦} 𝒦 → A ≅ B → B ∈ V 𝒦
```

Thus, if $\mathcal{K}$ is a class of S-algebras, then the *variety generated by* $\mathcal{K}$—that is, the smallest class that contains $\mathcal{K}$ and is closed under H, S, and P—is V $\mathcal{K}$.

5.3 Class products, PS ⊆ SP and ⨅ S ∈ SP

There are two main results we need to establish about the closure operators defined in the last section. The first is the inclusion PS($\mathcal{K}$) ⊆ SP($\mathcal{K}$). The second requires that we construct the product of all subalgebras of algebras in an arbitrary class, and then show that this product belongs to SP($\mathcal{K}$). These are both nontrivial formalization tasks with rather lengthy proofs, which we omit. However, the interested reader can view the complete proofs on the web page ualib.gitlab.io/UALib.Varieties.Varieties.html or by looking at the source code of the UALib.Varieties module at gitlab.com/ualib.

Here is the formal statement of the first goal, along with the first few lines of proof.[5]

```
PS⊆SP : (P{ovu}{ovu} (S{𝓤}{ovu} 𝒦)) ⊆ (S{ovu}{ovu} (P{𝓤}{ovu} 𝒦))

PS⊆SP (pbase (sbase x)) = sbase (pbase x)
PS⊆SP (pbase (slift{A} x)) = slift (S⊆SP{𝓤}{ovu}{𝒦} (slift x))
PS⊆SP (pbase {B} (ssub{A} sA B≤A)) = ...
```

Evidently, the proof is by induction on the form of an inhabitant of PS($\mathcal{K}$), handling in turn each way such an inhabitant can be constructed. (See [4] or `ualib.org` for details.)

Next we formally state and prove that, given an arbitrary class $\mathcal{K}$ of algebras, the product of all algebras in the class S($\mathcal{K}$) belongs to SP($\mathcal{K}$).[6] The type that serves to index the class

---

[5] For legibility we use ovu as a shorthand for $\mathcal{O} \sqcup \mathcal{V} \sqcup \mathcal{U}^{+}$.

[6] This turned out to be a nontrivial exercise. In fact, it is not even immediately obvious (at least not to this author) how one should express the product of an entire arbitrary class of algebras as a dependent type.

(and the product of its members) is the following.

$\mathfrak{I}$: $\{\boldsymbol{\mathcal{U}}$: Universe$\}$ → Pred (Algebra $\boldsymbol{\mathcal{U}}$ S)(ov $\boldsymbol{\mathcal{U}}$) → (ov $\boldsymbol{\mathcal{U}}$) ̇
$\mathfrak{I}$ $\{\boldsymbol{\mathcal{U}}\}$ $\mathscr{K}$ = Σ **A** : (Algebra $\boldsymbol{\mathcal{U}}$ S) , **A** ∈ $\mathscr{K}$

Taking the product over this index type $\mathfrak{I}$ requires a function like the following, which takes an index (i : $\mathfrak{I}$) and returns the corresponding algebra.

$\mathfrak{A}$: $\{\boldsymbol{\mathcal{U}}$: Universe$\}\{\mathscr{K}$: Pred (Algebra $\boldsymbol{\mathcal{U}}$ S)(ov $\boldsymbol{\mathcal{U}}$)$\}$ → $\mathfrak{I}$ $\mathscr{K}$ → Algebra $\boldsymbol{\mathcal{U}}$ S
$\mathfrak{A}\{\boldsymbol{\mathcal{U}}\}\{\mathscr{K}\}$ = λ (i : ($\mathfrak{I}$ $\mathscr{K}$)) → | i |

Finally, the product of all members of $\mathscr{K}$ is represented by the following type.

class-product : $\{\boldsymbol{\mathcal{U}}$: Universe$\}$ → Pred (Algebra $\boldsymbol{\mathcal{U}}$ S)(ov $\boldsymbol{\mathcal{U}}$) → Algebra (ov $\boldsymbol{\mathcal{U}}$) S
class-product $\{\boldsymbol{\mathcal{U}}\}$ $\mathscr{K}$ = ⨅ ($\mathfrak{A}\{\boldsymbol{\mathcal{U}}\}\{\mathscr{K}\}$)

If p : **A** ∈ $\mathscr{K}$ is a proof that **A** belongs to $\mathscr{K}$, then we can view the pair (**A**, p) ∈ $\mathfrak{I}$ $\mathscr{K}$ as an index over the class, and $\mathfrak{A}$(**A** , p) as the result of projecting the product onto the (**A**, p)-th component.

5.4 Equation preservation

This section describes parts of the UALib.Varieties.Preservation module of the Agda UALib, in which it is proved that identities are preserved by closure operators H, S, and P. This will establish the easy direction of Birkhoff's HSP theorem. We present only the formal statements, omitting proofs. (See [4] or `ualib.org` for details.) For example, the assertion that H preserves identities is stated formally as follows:

H-id1 : $\{\boldsymbol{\mathcal{U}}\ \boldsymbol{\mathfrak{X}}$: Universe$\}\{X : \boldsymbol{\mathfrak{X}}$ ̇ $\}\{\mathscr{K}$: Pred (Algebra $\boldsymbol{\mathcal{U}}$ S)(ov $\boldsymbol{\mathcal{U}}$)$\}$
(p q : Term$\{\boldsymbol{\mathfrak{X}}\}\{X\}$) (1)
→ $\mathscr{K}$ ⊧ p ≋ q → H$\{\boldsymbol{\mathcal{U}}\}\{\boldsymbol{\mathcal{U}}\}$ $\mathscr{K}$ ⊧ p ≋ q

The proof is by induction and handles each of the four constructors of H (hbase, hlift, hhimg, and hiso) in turn. Of course, the UALib.Varieties.Preservation module of the UALib contains a complete proof of (1), which can be viewed in the source code or the html documentation.[7] The facts that S, P, and V preserve identities are presented in the UALib in a similar way (and named S-id1, P-id1, V-id1, respectively). As usual, the full proofs are available in the UALib source code and documentation.[7]

5.5 The free algebra in theory

In this section, we formalize, for a given class $\mathscr{K}$ of S-algebras, the (relatively) *free algebra in* SP($\mathscr{K}$) *over* X. Let Θ($\mathscr{K}$, **A**) := {θ ∈ Con **A** : **A** / θ ∈ S $\mathscr{K}$} and ψ($\mathscr{K}$, **A**) := ⋂ Θ($\mathscr{K}$, **A**). Since the term algebra **T** X is free for (and in) the class $\mathscr{A\ell g}(S)$ of all S-algebras, it is free for every subclass $\mathscr{K}$ of $\mathscr{A\ell g}(S)$. Although **T** X is not necessarily a member of $\mathscr{K}$, if we form the quotient $\mathfrak{F}$:= (**T** X) / ψ($\mathscr{K}$, **T** X), of **T** X modulo the congruence

However, after a number of failed attempts, the right type revealed itself. Now that we have it, it seems almost obvious.

[7] See the source code file Preservation.lagda, or the html documentation page ualib.gitlab.io/UALib.Varieties.Preservation.html.

$\psi(\mathcal{K}, \mathbf{T}\ X) := \bigcap \{\theta \in \mathsf{Con}\ (\mathbf{T}\ X) : (\mathbf{T}\ X) \mathbin{/} \theta \in \mathsf{S}(\mathcal{K})\}$,

then it is not hard to see that $\mathfrak{F}$ is a subdirect product of the algebras in $\{(\ \mathbf{T}\ X)\ /\ \theta\ \}$, where θ ranges over $\Theta(\mathcal{K}, \mathbf{T}\ X)$, so $\mathfrak{F}$ belongs to $\mathsf{SP}(\mathcal{K})$, and it follows that $\mathfrak{F}$ satisfies all the identities modeled by $\mathcal{K}$. Indeed, for each pair $p\ q : \mathbf{T}\ X$, if $\mathcal{K} \models p \approx q$, then p and q must belong to the same $\psi(\mathcal{K}, \mathbf{T}\ X)$-class, so p and q are identified in the quotient $\mathfrak{F}$.

The algebra $\mathfrak{F}$ so defined is called the *free algebra over* $\mathcal{K}$ *generated by* X and (because of what we just observed) we say that $\mathfrak{F}$ is free *in* $\mathsf{SP}(\mathcal{K})$.

5.6 The free algebra in Agda

To represent $\mathfrak{F}$ as a type in Agda, we must formally construct the congruence $\psi(\mathcal{K}, \mathbf{T}\ X)$. We begin by defining the collection **Timg** of homomorphic images of the term algebra that belong to a given class $\mathcal{K}$.

```
Timg : Pred (Algebra 𝓤 S) ov𝓤 → ov𝓤 ⊔ 𝔛 ⁺ ·
Timg 𝒦 = Σ 𝐀 : (Algebra 𝓤 S) , Σ ϕ : hom (𝐓 X) 𝐀 , (𝐀 ∈ 𝒦) × Epic ∣ ϕ ∣
```

The inhabitants of this Sigma type represent algebras $\mathbf{A} \in \mathcal{K}$ such that there exists a surjective homomorphism ϕ : hom $(\mathbf{T}\ X)\ \mathbf{A}$. Thus, **Timg** represents the collection of all homomorphic images of $\mathbf{T}\ X$ that belong to $\mathcal{K}$. Of course, this is the entire class $\mathcal{K}$, since the term algebra is absolutely free. Nonetheless, this representation of $\mathcal{K}$ is useful since it endows each element with extra information. Indeed, each inhabitant of **Timg** $\mathcal{K}$ is a quadruple, $(\mathbf{A}\ ,\ \phi\ ,\ ka,\ p)$, where $\mathbf{A}$ is an S-algebra, ϕ is a homomorphism from $\mathbf{T}\ X$ to $\mathbf{A}$, ka is a proof that $\mathbf{A}$ belongs to $\mathcal{K}$, and p is a proof that the underlying map $\mid \phi \mid$ is epic.

Next we define the congruence relation modulo which $\mathbf{T}\ X$ yields the relatively free algebra, $\mathfrak{F}\ \mathcal{K}\ X$. We start by letting ψ be the collection of all identities $(p,\ q)$ satisfied by all subalgebras of algebras in $\mathcal{K}$,

```
ψ : (𝒦 : Pred (Algebra 𝓤 S) ov𝓤) → Pred (∣ 𝐓 X ∣ × ∣ 𝐓 X ∣) ov𝓤
ψ 𝒦 (p , q) = ∀(𝐀 : Algebra 𝓤 S) → (sA : 𝐀 ∈ S{𝓤}{𝓤} 𝒦)
                 → ∣ lift-hom 𝐀 (fst(𝕏 𝐀)) ∣ p ≡ ∣ lift-hom 𝐀 (fst(𝕏 𝐀)) ∣ q,
```

which we convert into a relation by currying, ψRel $\mathcal{K}\ p\ q = \psi\ \mathcal{K}\ (p\ ,\ q)$. The relation ψRel is an equivalence relation and is compatible with the operations of $\mathbf{T}\ X$, which are just the terms themselves.[8] Therefore, from ψRel we can construct a congruence relation of the term algebra $\mathbf{T}\ X$. The inhabitants of this congruence are pairs of terms representing identities satisfied by all subalgebras of algebras in the class. We call this ψCon and define it using the Congruence constructor mkcon as follows.

```
ψCon : (𝒦 : Pred (Algebra 𝓤 S) ov𝓤) → Congruence (𝐓 X)
ψCon 𝒦 = mkcon (ψRel 𝒦) (ψcompatible 𝒦) ψIsEquivalence
```

The free algebra $\mathfrak{F}$ is then defined as the quotient $\mathbf{T}\ X \mathbin{/} (\psi\mathsf{Con}\ \mathcal{K})$.

```
𝔉 : Algebra 𝓕 S
𝔉 = 𝐓 X ╱ (ψCon 𝒦)
```

where $\boldsymbol{\mathcal{F}} = (\mathfrak{X} \sqcup \mathsf{ov}\boldsymbol{u})^+$ happens to be the universe level of $\mathfrak{F}$. The domain of the free algebra is $\mid \mathbf{T}\ X \mid\ /\ \langle\ \psi\mathsf{Con}\ \mathcal{K}\ \rangle$, which is $\Sigma\ \mathsf{C} : \_\ ,\ \Sigma\ p : \mid \mathbf{T}\ X \mid\ ,\ \mathsf{C} \equiv ([\ p\ ]\ \langle\ \psi\mathsf{Con}\ \mathcal{K}\ \rangle)$, by definition; i.e., the collection $\{\mathsf{C} : \exists\ p \in \mid \mathbf{T}\ X \mid,\ \mathsf{C} \equiv [\ p\ ]\ \langle\ \psi\mathsf{Con}\ \mathcal{K}\ \rangle\}$ of $\langle\ \psi\mathsf{Con}\ \mathcal{K}\ \rangle$-classes of $\mathbf{T}\ X$.

---

[8] We omit the easy proofs, but see the UALib.Birkhoff.FreeAlgebra module for details.

6 Birkhoff's HSP Theorem

This section presents the UALib.Birkhoff module of the Agda UALib. In §**??**, we present a formal representation of the *free algebra* in S (P $\mathcal{K}$) over X, for a given class $\mathcal{K}$ of S-algebras. In §6.1, we state a number of lemmas needed in §6.2 where, finally, we present the statement and proof of Birkhoff's HSP theorem.

6.1 HSP Lemmas

This subsection gives formal statements of four lemmas that we will string together in §6.2 to complete the proof of Birkhoff's theorem.

The first hurdle is the lift-alg-V-closure lemma, which says that if an algebra **A** belongs to the variety $\mathbb{V}$, then so does its lift. This dispenses with annoying universe level problems that arise later—a minor techinical issue, but the proof is long and tedious, not to mention uninteresting. (See [4] or `ualib.org` for details.) The next fact that must be formalized is the inclusion SP($\mathcal{K}$) ⊆ V($\mathcal{K}$), which also suffers from the unfortunate defect of being boring, so we omit this one as well.

After these first two lemmas are formally verified (see [4] or `ualib.org` for proofs), we arrive at a step in the formalization of Birkhoff's theorem that turns out to be surprisingly nontrivial. We must show that the relatively free algebra $\mathfrak{F}$ embeds in the product $\mathfrak{C}$ of all subalgebras of algebras in the given class $\mathcal{K}$. We begin by constructing $\mathfrak{C}$, using the class-product types described in §5.3.

```
ℑs : ov𝓤 ̇ – for indexing over all subalgebras of algebras in 𝒦
ℑs = ℑ (S{𝓤}{𝓤} 𝒦)
𝔄s : ℑs → Algebra 𝓤 S
𝔄s = λ (i : ℑs) → ∣ i ∣
ℭ : Algebra ov𝓤 S – the product of all subalgebras of algebras in 𝒦
ℭ = ⨅ 𝔄s
```

Observe that the elements of $\mathfrak{C}$ are maps from ℑs to {𝔄s i : $i \in$ ℑs}.

Next, we construct an embedding 𝔣 from $\mathfrak{F}$ into $\mathfrak{C}$ using a UALib tool called 𝔉-free-lift.

```
𝔥₀ : X → ∣ ℭ ∣
𝔥₀ x = λ i → (fst (𝕏 (𝔄s i))) x
ϕ𝔠 : hom (𝑻 X) ℭ
ϕ𝔠 = lift-hom ℭ 𝔥₀
𝔣 : hom 𝔉 ℭ
𝔣 = 𝔉-free-lift ℭ 𝔥₀ , λ f 𝒂 → ∥ ϕ𝔠 ∥ f (λ i → ⌜ 𝒂 i ⌝)
```

The hard part is showing that 𝔣 is a monomorphism. For lack of space, we must omit the inner workings of the proof, and settle for the outer shell. (See [4] or `ualib.org` for details.) For ease of notation, let Ψ = ψRel $\mathcal{K}$.

```
monf : Monic ∣ 𝔣 ∣
monf (.(Ψ p) , p , refl _) (.(Ψ q) , q , refl _) fpq = γ
  where
  ... details omitted ...
  γ : ( Ψ p , p , 𝓇ℯ𝒻ℓ) ≡ ( Ψ q , q , 𝓇ℯ𝒻ℓ)
  γ = class-extensionality' pe gfe ssR ssA ψIsEquivalence pΨq
```

Assuming the foregoing, the proof that 𝔉 is (isomorphic to) a subalgebra of ℭ would be completed as follows.

```
𝔉≤ℭ : is-set ∣ ℭ ∣ → 𝔉 ≤ ℭ
𝔉≤ℭ Cset = ∣ 𝔣 ∣ , (emb𝔣 , ∥ 𝔣 ∥)
  where
    emb𝔣 : is-embedding ∣ 𝔣 ∣
    emb𝔣 = monic-into-set-is-embedding Cset ∣ 𝔣 ∣ mon𝔣
```

With the foregoing results in hand, along with what we proved earlier—namely, that PS(𝒦) ⊆ SP(𝒦) ⊆ V(𝒦)—it is not hard to show that 𝔉 belongs to SP(𝒦), and hence to V(𝒦).

```
𝔉∈SP : is-set ∣ ℭ ∣ → 𝔉 ∈ (S{ov𝓾}{ov𝓾⁺} (P{𝓤}{ov𝓾} 𝒦))
𝔉∈SP Cset = ssub spC (𝔉≤ℭ Cset)
  where
    spC : ℭ ∈ (S{ov𝓾}{ov𝓾} (P{𝓤}{ov𝓾} 𝒦))
    spC = (class-prod-s-∈-sp hfe)

𝔉∈𝕍 : is-set ∣ ℭ ∣ → 𝔉 ∈ 𝕍
𝔉∈𝕍 Cset = SP⊆V' (𝔉∈SP Cset)
```

6.2 The HSP Theorem

It is now all but trivial to use what we have already proved and piece together a complete formal, type-theoretic proof of Birkhoff's celebrated HSP theorem asserting that every variety is defined by a set of identities (is an "equational class").

```
module Birkhoffs-Theorem
  {𝒦 : Pred (Algebra 𝓤 S) ov𝓾}
    -- extensionality assumptions
    {hfe : hfunext ov𝓾 ov𝓾}
    {pe : propext ov𝓾}
    -- truncation assumptions:
    {ssR : ∀ p q → is-subsingleton ((ψRel 𝒦) p q)}
    {ssA : ∀ C → is-subsingleton (𝒞{ov𝓾}{ov𝓾}{∣ 𝐓 X ∣}{ψRel 𝒦} C)}
  where

  -- Birkhoff's theorem: every variety is an equational class.
  birkhoff : is-set ∣ ℭ ∣ → Mod X (Th 𝕍) ⊆ 𝕍

  birkhoff Cset {𝐀} MThVA = γ
    where
      ϕ : Σ h ꞉ (hom 𝔉 𝐀) , Epic ∣ h ∣
      ϕ = (𝔉-lift-hom 𝐀 ∣ 𝕏 𝐀 ∣) , 𝔉-lift-of-epic-is-epic 𝐀 ∣ 𝕏 𝐀 ∣ ∥ 𝕏 𝐀 ∥

      AiF : 𝐀 is-hom-image-of 𝔉
      AiF = (𝐀 , ∣ fst ϕ ∣ , (∥ fst ϕ ∥ , snd ϕ) ) , refl-≅

      γ : 𝐀 ∈ 𝕍
      γ = vhimg (𝔉∈𝕍 Cset) AiF
```

Some readers might worry that we haven't quite acheived our goal because what we just proved (birkhoff) is not an "if and only if" assertion. Those fears are quickly put to rest by noting that the converse—that every equational class is closed under HSP—was already proved in the Equation Preservation module. Indeed, there we proved the following identity preservation lemmas:

- (H-id1) $\mathcal{K} \models p \approx q \to \mathsf{H}\ \mathcal{K} \models p \approx q$
- (S-id1) $\mathcal{K} \models p \approx q \to \mathsf{S}\ \mathcal{K} \models p \approx q$
- (P-id1) $\mathcal{K} \models p \approx q \to \mathsf{P}\ \mathcal{K} \models p \approx q$

From these it follows that every equational class is a variety.

7 Conclusions and future work

As mentioned in § 1, Carlström in [3] proved a constructive version of Birkhoff's theorem. We would like to know how the two new hypothesis that Carlström adds to the classical theorem in order to make it constructive compares with the assumptions we make in our Agda proof.

References

1. G Birkhoff. On the structure of abstract algebras. *Proceedings of the Cambridge Philosophical Society*, 31(4):433–454, Oct 1935.
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. URL: `http://dx.doi.org/10.1007/3-540-48256-3_10`, `doi:10.1007/3-540-48256-3_10`.
3. Jesper Carlström. A constructive version of birkhoff's theorem. *Mathematical Logic Quarterly*, 54(1):27–34, 2008. URL: `https://onlinelibrary.wiley.com/doi/abs/10.1002/malq.200710023`, `arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/malq.200710023`, `doi:https://doi.org/10.1002/malq.200710023`.
4. William DeMeo. The agda universal algebra library and birkhoff's theorem in martin-löf dependent type theory, 2021. `arXiv:2101.10166`.
5. Martín Hötzel Escardó. Introduction to univalent foundations of mathematics with agda. *CoRR*, abs/1911.00580, 2019. URL: `http://arxiv.org/abs/1911.00580`, `arXiv:1911.00580`.
6. 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). URL: `http://www.sciencedirect.com/science/article/pii/S1571066118300768`, `doi:https://doi.org/10.1016/j.entcs.2018.10.010`.
7. Ulf Norell. Dependently typed programming in agda. In *Proceedings of the 6th International Conference on Advanced Functional Programming*, AFP'08, pages 230–266, Berlin, Heidelberg, 2009. Springer-Verlag. URL: `http://dl.acm.org/citation.cfm?id=1813347.1813352`.
8. 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`.
9. Bas Spitters and Eelis van der Weegen. Type classes for mathematics in type theory. *CoRR*, abs/1102.1323, 2011. URL: `http://arxiv.org/abs/1102.1323`, `arXiv:1102.1323`.
10. The Agda Team. Agda Language Reference: Sec. Axiom K, 2021. URL: `https://agda.readthedocs.io/en/v2.6.1/language/without-k.html`.
11. The Agda Team. Agda Language Reference: Sec. Safe Agda, 2021. URL: `https://agda.readthedocs.io/en/v2.6.1/language/safe-agda.html#safe-agda`.
12. The Agda Team. Agda Tools Documentation: Sec. Pattern matching and equality, 2021. URL: `https://agda.readthedocs.io/en/v2.6.1/tools/command-line-options.html#pattern-matching-and-equality`.

A Agda Prerequisites

For the benefit of readers who are not proficient in Agda and/or type theory, we describe some of the most important types and features of Agda used in the UALib.

We begin by highlighting some of the key parts of UALib.Prelude.Preliminaries of the Agda UALib. This module imports everything we need from Martin Escardo's Type Topology library [5], defines some other basic types and proves some of their properties. We do not cover the entire Preliminaries module here, but call attention to aspects that differ from standard Agda syntax. For more details, see [4, §2].

A.1 Options and imports

Agda programs typically begin by setting some options and by importing from existing libraries. Options are specified with the OPTIONS *pragma* and control the way Agda behaves by, for example, specifying which logical foundations should be assumed when the program is type-checked to verify its correctness. All Agda programs in the UALib begin with the pragma

{-# OPTIONS *–without-K –exact-split –safe* #-} (2)

This has the following effects:

1. `without-K` disables Streicher's K axiom; see [10];
2. `exact-split` makes Agda accept only definitions that are *judgmental* or *definitional* equalities. As Escardó explains, this "makes sure that pattern matching corresponds to Martin-Löf eliminators;" for more details see [12];
3. `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 [11] and [12].

Throughout this paper we take assumptions 1–3 for granted without mentioning them explicitly.

The Agda UALib adopts the notation of the Type Topology library. In particular, universes are denoted by capitalized script letters from the second half of the alphabet, e.g., $\mathcal{U}$, $\mathcal{V}$, $\mathcal{W}$, etc. Also defined in Type Topology are the operators $^\cdot$ and $^+$. These map a universe $\mathcal{U}$ to $\mathcal{U}\ ^\cdot$:= Set $\mathcal{U}$ and $\mathcal{U}\ ^+$:= lsuc $\mathcal{U}$, respectively. Thus, $\mathcal{U}\ ^\cdot$ is simply an alias for Set $\mathcal{U}$, and we have $\mathcal{U}\ ^\cdot : (\mathcal{U}\ ^+)\ ^\cdot$. Table 1 translates between standard Agda syntax and Type Topology/UALib notation.

A.2 Agda's universe hierarchy

The hierarchy of universe levels in Agda is structured as $\mathcal{U}_0 : \mathcal{U}_1, \quad \mathcal{U}_1 : \mathcal{U}_2, \quad \mathcal{U}_2 : \mathcal{U}_3, \ldots$. This means that $\mathcal{U}_0$ has type $\mathcal{U}_1\ ^\cdot$ and $\mathcal{U}_n$ has type $\mathcal{U}_{n+1}\ ^\cdot$ for each n. It is important to note, however, this does *not* imply that $\mathcal{U}_0 : \mathcal{U}_2$ and $\mathcal{U}_0 : \mathcal{U}_3$, and so on. In other words, Agda's universe hierarchy is *noncummulative*. This makes it possible to treat universe levels more generally and precisely, which is nice. On the other hand, it is this author's experience that a noncummulative hierarchy can sometimes make for a nonfun proof assistant. (See [4, §3.3] for a more detailed discussion.)

Because of the noncummulativity of Agda's universe level hierarchy, certain proof verification (i.e., type-checking) tasks may seem unnecessarily difficult. Luckily there are ways to circumvent noncummulativity without introducing logical inconsistencies into the type theory. We present here some domain specific tools that we developed for this purpose. (See [4, §3.3] for more details).

A general Lift record type, similar to the one found in the Level module of the Agda Standard Library, is defined as follows.

```
record Lift {𝓤 𝓦 : Universe} (X : 𝓤 ̇) : 𝓤 ⊔ 𝓦 ̇ where
  constructor lift
  field lower : X
open Lift
```

It is useful to know that lift and lower compose to the identity.

```
lower∼lift : {𝓧 𝓦 : Universe}{X : 𝓧 ̇} → lower{𝓧}{𝓦} ∘ lift ≡ id X
lower∼lift = refl _
```

Similarly, lift ∘ lower ≡ *id* (Lift{𝓧}{𝓦}.

A.2.1 The lift of an algebra

More domain-specifically, here is how we lift types of operations and algebras.

```
lift-op : {𝓤 : Universe}{I : 𝓥 ̇}{A : 𝓤 ̇}
  → ((I → A) → A) → (𝓦 : Universe)→((I → Lift{𝓤}{𝓦}A) → Lift{𝓤}{𝓦}A)
lift-op f 𝓦 = λ x → lift (f (λ i → lower (x i)))

lift-∞-algebra lift-alg : {𝓤 : Universe} → Algebra 𝓤 S → (𝓦 : Universe) → Algebra (𝓤 ⊔ 𝓦) S
lift-∞-algebra 𝐀 𝓦 = Lift ∣ 𝐀 ∣ , (λ (f : ∣ S ∣) → lift-op (∥ 𝐀 ∥ f) 𝓦)
lift-alg = lift-∞-algebra
```

A.3 Dependent pairs and projections

Given universes $\mathcal{U}$ and $\mathcal{V}$, a type $X : \mathcal{U}\,^{\cdot}$, and a type family $Y : \mathrm{X} \to \mathcal{V}\,^{\cdot}$, the *Sigma type* (or *dependent pair type*) is denoted by $\Sigma(x : X), Y(x)$ and generalizes the Cartesian product $X \times Y$ by allowing the type $Y(x)$ of the second argument of the ordered pair $(x,\ y)$ to depend on the value x of the first. That is, $\Sigma(x : X), Y(x)$ is inhabited by pairs $(x,\ y)$ such that $x : X$ and $y : Y(x)$.

Agda's default syntax for a Sigma type is $\Sigma\ \lambda(x : X) \to Y$, but we prefer the notation $\Sigma\ x : X\ ,\ Y$, which is closer to the standard syntax described in the preceding paragraph. Fortunately, this preferred notation is available in the Type Topology library (see [5, Σ types]).[9]

Convenient notations for the first and second projections out of a product are ∣_∣ and ∥_∥, respectively. However, to improve readability or to avoid notation clashes with other modules, we sometimes use more standard alternatives, such as pr_1 and pr_2, or fst and snd, or some combination of these. The definitions are standard so we omit them (see [4] for details).

A.4 Equality

Perhaps the most important types in type theory are the equality types. The *definitional equality* we use is a standard one and is often referred to as "reflexivity" or "refl". In our case, it is defined in the Identity-Type module of the Type Topology library, but apart from syntax it is equivalent to the identity type used in most other Agda libraries. Here is the definition.

```
data _≡_ {𝓤} {X : 𝓤 ̇ } : X → X → 𝓤 ̇ where
  refl : {x : X} → x ≡ x
```

[9] The symbol : in the expression $\Sigma\ x : X\ ,\ Y$ is not the ordinary colon (:); rather, it is the symbol obtained by typing `\:4` in agda2-mode.

A.5 Function extensionality and intensionality

Extensional equality of functions, or *function extensionality*, is a principle that is often assumed in the Agda UALib. It asserts that two point-wise equal functions are equal and is defined in the Type Topology library in the following natural way:

```
funext : ∀ 𝓤 𝓥 → (𝓤 ⊔ 𝓥)⁺ ̇
funext 𝓤 𝓥 = {X : 𝓤 ̇ } {Y : 𝓥 ̇ } {f g : X → Y} → f ∼ g → f ≡ g
```

where $f \sim g$ denotes pointwise equality, that is, $\forall\ x \to f\ x \equiv g\ x$.

Pointwise equality of functions is typically what one means in informal settings when one says that two functions are equal. However, as Escardó notes in [5], function extensionality is known to be not provable or disprovable in Martin-Löf Type Theory. It is an independent axiom which may be assumed (or not) without making the logic inconsistent.

Dependent and polymorphic notions of function extensionality are also defined in the UALib and Type Topology libraries (see [4, §2.4] and [5, §17-18]).

Function intensionality is the opposite of function extensionality and it comes in handy whenever we have a definitional equality and need a point-wise equality.

```
intensionality : {𝓤 𝓦 : Universe} {A : 𝓤 ̇ } {B : 𝓦 ̇ } {f g : A → B}
  →              f ≡ g → (x : A)  →  f x ≡ g x
intensionality (refl _ ) _ = refl _
```

Of course, the intensionality principle has dependent and polymorphic analogues defined in the Agda UALib, but we omit the definitions. See [4, §2.4] for details.

A.6 Truncation and sets

In general, we may have many inhabitants of a given type, hence (via Curry-Howard correspondence) many proofs of a given proposition. For instance, suppose we have a type X and an identity relation $\equiv_x$ on X so that, given two inhabitants of X, say, $a\ b : X$, we can form the type $a \equiv_x b$. Suppose p and q inhabit the type $a \equiv_x b$; that is, p and q are proofs of $a \equiv_x b$, in which case we write $p\ q : a \equiv_x b$. Then we might wonder whether and in what sense are the two proofs p and q the "same." We are asking about an identity type on the identity type $\equiv_x$, and whether there is some inhabitant r of this type; i.e., whether there is a proof $r : p \equiv_{x1} q$. If such a proof exists for all $p\ q : a \equiv_x b$, then we say that the proof of $a \equiv_x b$ is *unique*. As a property of the types X and $\equiv_x$, this is sometimes called *uniqueness of identity proofs*.

Perhaps we have two proofs, say, $r\ s : p \equiv_{x1} q$. Then it is natural to wonder whether $r \equiv_{x2} s$ has a proof! However, we may decide that at some level the potential to distinguish two proofs of an identity in a meaningful way (so-called *proof relevance*) is not useful or interesting. At that point, say, at level k, we might assume that there is at most one proof of any identity of the form $p \equiv_{xk} q$. This is called *truncation*.

In *homotopy type theory* [8], a type X with an identity relation $\equiv_x$ is called a *set* (or *0-groupoid* or *h-set*) if for every pair $a\ b : X$ of elements of type X there is at most one proof of $a \equiv_x b$. This notion is formalized in the Type Topology library as follows:

```
is-set : 𝓤 ̇ → 𝓤 ̇
is-set X = (x y : X) → is-subsingleton (x ≡ y)
```
(3)

where

```
is-subsingleton : 𝓤 ̇ → 𝓤 ̇
is-subsingleton X = (x y : X) → x ≡ y
```
(4)

Truncation is used in various places in the UALib, and it is required in the proof of Birkhoff's theorem. Consult [5, §34-35] or [8, §7.1] for more details.

A.7 Inverses, Epics and Monics

This section describes some of the more important parts of the UALib.Prelude.Inverses module. In § A.7.1, we define an inductive datatype that represents our semantic notion of the *inverse image* of a function. In § A.7.2 we define types for *epic* and *monic* functions. Finally, in Subsections A.8, we consider the type of *embeddings* (defined in [5, §26]), and determine how this type relates to our type of monic functions.

A.7.1 Inverse image type

```
data Image_∋_ {A : 𝓤 ̇ }{B : 𝓦 ̇ }(f : A → B) : B → 𝓤 ⊔ 𝓦 ̇
  where
  im : (x : A) → Image f ∋ f x
  eq : (b : B) → (a : A) → b ≡ f a → Image f ∋ b
```

Note that an inhabitant of Image $f \ni b$ is a dependent pair $(a\ ,\ p)$, where $a : A$ and $p : b \equiv f\ a$ is a proof that f maps a to b. Thus, a proof that b belongs to the image of f (i.e., an inhabitant of Image $f \ni b$), is always accompanied by a witness $a : A$, and a proof that $b \equiv f\ a$, so the inverse of a function f can actually be *computed* at every inhabitant of the image of f.

We define an inverse function, which we call Inv, which, when given $b : B$ and a proof $(a\ ,\ p)$: Image $f \ni b$ that b belongs to the image of f, produces a (a preimage of b under f).

```
Inv : {A : 𝓤 ̇ }{B : 𝓦 ̇ }(f : A → B)(b : B) → Image f ∋ b → A
Inv f .(f a) (im a) = a
Inv f _ (eq _ a _) = a
```

Thus, the inverse is computed by pattern matching on the structure of the third explicit argument, which has (inductive) type Image $f \ni b$. Since there are two constructors, im and eq, that argument must take one of two forms. Either it has the form im a (in which case the second explicit argument is .($f\ a$)),[10] or it has the form eq $b\ a\ p$, where p is a proof of $b \equiv f\ a$. (The underscore characters replace b and p in the definition since Inv doesn't care about them; it only needs to extract and return the preimage a.)

We can formally prove that Inv f is the right-inverse of f, as follows. Again, we use pattern matching and structural induction.

```
InvIsInv : {A : 𝓤 ̇ } {B : 𝓦 ̇ } (f : A → B)
           (b : B) (b∈Imgf : Image f ∋ b)
           ─────────────────
  →        f (Inv f b b∈Imgf) ≡ b
InvIsInv f .(f a) (im a) = refl _
InvIsInv f b (eq b a b≡fa) = b≡fa ⁻¹
```

Here we give names to all the arguments for readability, but most of them could be replaced with underscores.

[10] The dotted pattern is used when the form of the argument is forced... todo: fix this sentence

A.7.2 Epic and monic function types

Given universes $\mathcal{U}$, $\mathcal{W}$, types $A : \mathcal{U}\ ^\cdot$ and $B : \mathcal{W}\ ^\cdot$, and $f : A \to B$, we say that f is an *epic* (or *surjective*) *function from A to B* provided we can produce an element (or proof or witness) of type Epic f, where

```
Epic : {A : 𝓤 ̇ } {B : 𝓦 ̇ } (f : A → B) → 𝓤 ⊔ 𝓦 ̇
Epic f = ∀ y → Image f ∋ y
```

We obtain the (right-) inverse of an epic function f by applying the following function to f and a proof that f is epic.

```
EpicInv : {A : 𝓤 ̇ } {B : 𝓦 ̇ }
          (f : A → B) → Epic f
          ----------------
     →    B → A
EpicInv f p b = Inv f b (p b)
```

The function defined by EpicInv f p is indeed the right-inverse of f, as we now prove.

```
EpicInvIsRightInv : funext 𝓦 𝓦 → {A : 𝓤 ̇ } {B : 𝓦 ̇ }
                    (f : A → B) (fE : Epic f)
                    ----------------
     →              f ∘ (EpicInv f fE) ≡ id B
EpicInvIsRightInv fe f fE = fe (λ x → InvIsInv f x (fE x))
```

Similarly, we say that $f : A \to B$ is a *monic* (or *injective*) function from A to B if we have a proof of Monic f, where

```
Monic : {A : 𝓤 ̇ } {B : 𝓦 ̇ }(f : A → B) → 𝓤 ⊔ 𝓦 ̇
Monic f = ∀ a₁ a₂ → f a₁ ≡ f a₂ → a₁ ≡ a₂
```

As one would hope and expect, the *left*-inverse of a monic function is derived from a proof p : Monic f in a similar way. (See MonicInv and MonicInvIsLeftInv in [4, §2.3] for details.)

A.8 Monic functions are set embeddings

An *embedding*, as defined in [5, §26], is a function with *subsingleton fibers*. The meaning of this will be clear from the definition, which involves the three functions is-embedding, is-subsingleton, and fiber. The second of these is defined in (4); the other two are defined as follows.

```
is-embedding : {X : 𝓤 ̇ } {Y : 𝓥 ̇ } → (X → Y) → 𝓤 ⊔ 𝓥 ̇
is-embedding f = (y : codomain f) → is-subsingleton (fiber f y)

fiber : {X : 𝓤 ̇ } {Y : 𝓥 ̇ } (f : X → Y) → Y → 𝓤 ⊔ 𝓥 ̇
fiber f y = Σ x ꞉ domain f , f x ≡ y
```

This is not simply a *monic* function (§A.7.2), and it is important to understand why not. Suppose $f : X \to Y$ is a monic function from X to Y, so we have a proof p : Monic f. To prove f is an embedding we must show that for every $y : Y$ we have is-subsingleton (fiber f y). That is, for all $y : Y$, we must prove the following implication:

$$\frac{(x\ x' : X) \quad (p : f\ x \equiv y) \quad (q : f\ x' \equiv y) \quad (m : \text{Monic } f)}{(x\ ,\ p) \equiv (x'\ ,\ q)}$$

By m, p, and q, we have $r : x \equiv x'$. Thus, in order to prove f is an embedding, we must somehow show that the proofs p and q (each of which entails $f\ x \equiv y$) are the same. However, there is no axiom or deduction rule in MLTT to indicate that $p \equiv q$ must hold; indeed, the two proofs may differ.

One way we could resolve this is to assume that the codomain type, B, is a *set*, i.e., has *unique identity proofs*. Recall the definition (3) of is-set from the Type Topology library. If the codomain of $f : A \to B$ is a set, and if p and q are two proofs of an equality in B, then $p \equiv q$, and we can use this to prove that a injective function into B is an embedding.

```
monic-into-set-is-embedding : {A : 𝓤 ̇}{B : 𝓦 ̇} → is-set B
  →                             (f : A → B) → Monic f
                                ————————————
  →                             is-embedding
```

We omit the formal proof for lack of space, but see [4, §2.3].

A.9 Unary Relations (predicates)

We need a mechanism for implementing the notion of subsets in Agda. A typical one is called Pred (for predicate). More generally, Pred A 𝓤 can be viewed as the type of a property that elements of type A might satisfy. We write P : Pred A 𝓤 to represent the semantic concept of a collection of elements of type A that satisfy the property P. Here is the definition, which is similar to the one found in the `Relation/Unary.agda` file of the Agda Standard Library.

```
Pred : 𝓤 ̇ → (𝓥 : Universe) → 𝓤 ⊔ 𝓥 ⁺ ̇
Pred A 𝓥 = A → 𝓥 ̇
```

Below we will often consider predicates over the class of all algebras of a particular type. By definition, the inhabitants of the type Pred (Algebra 𝓤 S) 𝓤 are maps of the form $\mathbf{A} \to$ 𝓤 ̇.

In type theory everything is a type. As we have just seen, this includes subsets. Since the notion of equality for types is usually a nontrivial matter, it may be nontrivial to represent equality of subsets. Fortunately, it is straightforward to write down a type that represents what it means for two subsets to be equal in informal (pencil-paper) mathematics. In the UALib we denote this *subset equality* by =̇ and define it as follows.[11]

```
_=̇_ : {𝓤 𝓦 𝓣 : Universe}{A : 𝓤 ̇ } → Pred A 𝓦 → Pred A 𝓣 → 𝓤 ⊔ 𝓦 ⊔ 𝓣 ̇
P =̇ Q = (P ⊆ Q) × (Q ⊆ P)
```

A.10 Binary Relations

In set theory, a binary relation on a set A is simply a subset of the product $A \times A$. As such, we could model these as predicates over the type $A \times A$, or as relations of type $A \to A \to$ 𝓡 ̇ (for some universe 𝓡). A generalization of this notion is a binary relation is a *relation from A to B*, which we define first and treat binary relations on a single A as a special case.

[11] Our notation and definition representing the semantic concept "x belongs to P," or "x has property P," is standard. We write either $x \in P$ or $P\ x$. Similarly, the "subset" relation is denoted, as usual, with the $\subseteq$ symbol (cf. in the Agda Standard Library). The relations $\in$ and $\subseteq$ are defined in the Agda UALib in a was similar to that found in the `Relation/Unary.agda` module of the Agda Standard Library. (See [4, §4.1.2].)

```
REL : {𝓡 : Universe} → 𝓤 ̇ → 𝓡 ̇ → (𝓝 : Universe) → (𝓤 ⊔ 𝓡 ⊔ 𝓝 ⁺) ̇
REL A B 𝓝 = A → B → 𝓝 ̇
```

The notions of reflexivity, symmetry, and transitivity are defined as one would hope and expect, so we present them here without further explanation.

```
reflexive : {𝓡 : Universe}{X : 𝓤 ̇ } → Rel X 𝓡 → 𝓤 ⊔ 𝓡 ̇
reflexive _≈_ = ∀ x → x ≈ x

symmetric : {𝓡 : Universe}{X : 𝓤 ̇ } → Rel X 𝓡 → 𝓤 ⊔ 𝓡 ̇
symmetric _≈_ = ∀ x y → x ≈ y → y ≈ x

transitive : {𝓡 : Universe}{X : 𝓤 ̇ } → Rel X 𝓡 → 𝓤 ⊔ 𝓡 ̇
transitive _≈_ = ∀ x y z → x ≈ y → y ≈ z → x ≈ z
```

A.11 Kernels of functions

The kernel of a function can be defined in many ways. For example,

```
KER : {𝓡 : Universe} {A : 𝓤 ̇ } {B : 𝓡 ̇ } → (A → B) → 𝓤 ⊔ 𝓡 ̇
KER {𝓡} {A} g = Σ x ꞉ A , Σ y ꞉ A , g x ≡ g y
```

or as a unary relation (predicate) over the Cartesian product,

```
KER-pred : {𝓡 : Universe} {A : 𝓤 ̇}{B : 𝓡 ̇} → (A → B) → Pred (A × A) 𝓡
KER-pred g (x , y) = g x ≡ g y
```

or as a relation from A to B,

```
Rel : 𝓤 ̇ → (𝓝 : Universe) → 𝓤 ⊔ 𝓝 ⁺ ̇
Rel A 𝓝 = REL A A 𝓝

KER-rel : {𝓡 : Universe}{A : 𝓤 ̇ } {B : 𝓡 ̇ } → (A → B) → Rel A 𝓡
KER-rel g x y = g x ≡ g y
```

A.12 Equivalence Relations

Types for equivalence relations are defined in the UALib.Relations.Equivalences module of the Agda UALib using a record type, as follows:

```
record IsEquivalence {A : 𝓤 ̇ } (_≈_ : Rel A 𝓡) : 𝓤 ⊔ 𝓡 ̇ where
  field
    rfl   : reflexive _≈_
    sym   : symmetric _≈_
    trans : transitive _≈_
```

For example, here is how we construct an equivalence relation out of the kernel of a function.

```
map-kernel-IsEquivalence : {𝓦 : Universe}{A : 𝓤 ̇}{B : 𝓦 ̇}
                           (f : A → B) → IsEquivalence (KER-rel f)

map-kernel-IsEquivalence {𝓦} f =
  record { rfl = λ x → 𝓇ℯ𝒻ℓ
         ; sym = λ x y x₁ → ≡-sym{𝓦} (f x) (f y) x₁
         ; trans = λ x y z x₁ x₂ → ≡-trans (f x) (f y) (f z) x₁ x₂ }
```

Standard Agda	Type Topology/UALib
Level	Universe
$\mathcal{U}$: Level	$\mathcal{U}$: Universe
Set $\mathcal{U}$	$\mathcal{U}$ ̇
lsuc $\mathcal{U}$	$\mathcal{U}^{+}$
Set (lsuc $\mathcal{U}$)	$\mathcal{U}^{+}$ ̇
lzero	$\mathcal{U}_0$
Setω	$\mathcal{U}\omega$

Table 1 Special notation for universe levels

A.13 Relation truncation

Here we discuss a special technical issue that will arise when working with quotients, specifically when we must determine whether two equivalence classes are equal. Given a binary relation[12] P, it may be necessary or desirable to assume that there is at most one way to prove that a given pair of elements is P-related. This is an example of *proof-irrelevance*; indeed, under this assumption, proofs of P x y are indistinguishable, or rather distinctions are irrelevant in given context.

In the UALib, the is-subsingleton type of Type Topology is used to express the assertion that a given type is a set, or *0-truncated*. Above we defined truncation for a type with an identity relation, but the general principle can be applied to arbitrary binary relations. Indeed, we say that P is a *0-truncated binary relation* on X if for all $x\ y :\ X$ we have is-subsingleton (P x y).

```
Rel₀ : 𝓤 ̇ → (𝓝 : Universe) → 𝓤 ⊔ 𝓝 ⁺ ̇
Rel₀ A 𝓝 = Σ P : (A → A → 𝓝 ̇) , ∀ x y → is-subsingleton (P x y)
```

Thus, a *set*, as defined in §A.6, is a type X along with an equality relation ≡ of type Rel₀ X $\mathcal{N}$, for some $\mathcal{N}$.

A.14 Nonstandard notation and syntax

The notation we adopt is that of the Type Topology library of Martín Escardó. Here we give a few more details and a table (Table 1) which translates between standard Agda syntax and Type Topology/UALib notation.

Many occasions call for the universe that is the least upper bound of two universes, say, $\mathcal{U}$ and $\mathcal{V}$. This is denoted by $\mathcal{U} \sqcup \mathcal{V}$ in standard Agda syntax, and in our notation the correponding type is ($\mathcal{U} \sqcup \mathcal{V}$) ̇, or, more simply, $\mathcal{U} \sqcup \mathcal{V}$ ̇ (since _⊔_ has higher precedence than _ ̇).

To justify the introduction of this somewhat nonstandard notation for universe levels, Escardó points out that the Agda library uses Level for universes (so what we write as $\mathcal{U}$ ̇ is written Set $\mathcal{U}$ in standard Agda), but in univalent mathematics the types in $\mathcal{U}$ ̇ need not be sets, so the standard Agda notation can be misleading.

In addition to the notation described in §A.1 above, the level lzero is renamed $\mathcal{U}_0$, so $\mathcal{U}_0$ ̇ is an alias for Set lzero. (For those familiar the Lean proof assistant, $\mathcal{U}_0$ ̇ (i.e., Set lzero) is analogous to Lean's `Sort 0`.)

[12] Binary relations, as represented in Agda in general and the UALib in particular, are described in §A.10.