

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.

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 Cliff Bergman, Hyeyoung Shin, and Siva Somayyajula for supporting and contributing 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 algebra. The UALib is written in Agda [8], a programming language and proof assistant based on Martin-Löf Type Theory that not only supports dependent and inductive types, as well as proof tactics for proving things about the objects that inhabit these types.

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.



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

- Capretta [3] (1999) formalized the basics of universal algebra in the Calculus of Inductive Constructions using the Coq proof assistant;
- Spitters and van der Weegen [10] (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 [7] (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, the goals of these efforts seem to be the formalization of special classical algebraic structures, as opposed to the general theory of (universal) algebras. Moreover, the part of universal algebra and equational logic formalized in the UALib extends beyond the scope of prior efforts and. In particular, the library now includes a proof of Birkhoff’s variety theorem. Most other proofs of this theorem that we know of are informal and nonconstructive.²

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

Apart from the library itself, we describe the formal implementation and proof of a deep result, Garrett Birkhoff’s celebrated HSP theorem [2], 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 [4] of Carlström makes clear, it is a highly nontrivial exercise to take a well-known informal proof of a theorem

² After the completion of this work, the author learned about a constructive version of Birkhoff’s theorem that was proved by Carlström in [4]. The latter is presented in the standard, informal style of mathematical writing in the current literature, and as far as we know it was never implemented formally and type-checked with a proof assistant. Nonetheless, a comparison of the version of the theorem presented in [4] to the Agda proof we give here would be interesting. We remark briefly on this in §8.

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 only what seem to us the most important or noteworthy components of the UALib. Of course, space does not permit us to cover all aspects of the 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 describing special notation conventions and some of the more important types defined in the UALib. Finally, we highly recommend the book by Bergman [1] for background in universal algebra, and the notes by Escardó [6] for background in type theory and Agda.

The paper is structured as follows. Section 2 introduces Sigma types in Agda (for us, the most important dependent type in the language) as well as Agda's universe hierarchy. Section 3 introduces the Agda UALib by explaining how algebraic structures are represented in the library. Section 4 describes the approach we take to quotient types and quotient algebra types. Section 5 defines new types that represent *homomorphisms*, *terms*, and *subalgebras*, and Section 6 does the same for *equational theories* and *models*. This is also where we present free algebras, both in the informal theory (§6.5) and represented as types in Agda (§6.6). Finally, Section 7 presents Birkhoff's theorem, describing the structure of the proof, with pointers to places in the documentation and source code where one can find the complete formal proof. The final section of the paper includes a few remarks about current work and future directions.

To conclude this introduction, we provide the following links to official sources of information about the Agda UALib.

- 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.

These sources include complete proofs of every theorem we mention here.

2 Agda Preliminaries

For the benefit of readers who are not proficient in Agda and/or type theory, we briefly describe some of the most important types and features of Agda used in the UALib. Of necessity, some descriptions will be terse, but are usually accompanied by pointers to relevant sections of Appendix §A or the html documentation.

We begin by highlighting some of the key parts of the `UALib.Prelude.Preliminaries` module of the UALib. This module imports everything we need from Martin Escardó's Type Topology library [6], 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 [5, §2].

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. Every Agda program in the UALib begins with the following pragma, which has the effects described below.

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

(1)

- `-without-K` disables Streicher's K axiom; see [11];

127 ■ *exact-split* makes Agda accept only definitions that are *judgmental* or *definitional* equalities;
 128 see [13];
 129 ■ *safe* ensures that nothing is postulated outright—every non-MLTT axiom has to be an
 130 explicit assumption (e.g., an argument to a function or module); see [12] and [13].
 131 Throughout this paper we take assumptions 1–3 for granted without mentioning them explicitly.

132 2.1 Agda’s universe hierarchy

133 The Agda UALib adopts the notation of Martin Escardo’s Type Topology library [6]. In
 134 particular, *universe levels*³ are denoted by capitalized script letters from the second half of the
 135 alphabet, e.g., \mathcal{U} , \mathcal{V} , \mathcal{W} , etc. Also defined in Type Topology are the operators \cdot and $+$. These
 136 map a universe level \mathcal{U} to the universe $\mathcal{U} \cdot := \text{Set } \mathcal{U}$ and the level $\mathcal{U}^+ := \text{Isuc } \mathcal{U}$, respectively.
 137 Thus, $\mathcal{U} \cdot$ is simply an alias for the universe $\text{Set } \mathcal{U}$, and we have $\mathcal{U} \cdot : (\mathcal{U}^+) \cdot$.

138 The hierarchy of universes in Agda is structured as $\mathcal{U} \cdot : \mathcal{U}^+$, $\mathcal{U}^+ : \mathcal{U}^{++}$, etc. This
 139 means that the universe $\mathcal{U} \cdot$ has type \mathcal{U}^+ , and \mathcal{U}^+ has type \mathcal{U}^{++} , and so on. It is important
 140 to note, however, this does *not* imply that $\mathcal{U} \cdot : \mathcal{U}^{++}$. In other words, Agda’s universe
 141 hierarchy is *noncumulative*. This makes it possible to treat universe levels more generally and
 142 precisely, which is nice. On the other hand, a noncumulative hierarchy can sometimes make
 143 for a nonfun proof assistant. Luckily, there are ways to circumvent noncumulativity without
 144 introducing logical inconsistencies into the type theory. §A.2 describes some domain specific
 145 tools that we developed for this purpose. (See also [5, §3.3] for more details).

146 2.2 Dependent pairs

147 Given universes \mathcal{U} and \mathcal{V} , a type $X : \mathcal{U} \cdot$, and a type family $Y : X \rightarrow \mathcal{V} \cdot$, the *Sigma type* (or
 148 *dependent pair type*) is denoted by $\Sigma(x : X), Y(x)$ and generalizes the Cartesian product $X \times Y$
 149 by allowing the type $Y(x)$ of the second argument of the ordered pair (x, y) to depend on the
 150 value x of the first. That is, $\Sigma(x : X), Y(x)$ is inhabited by pairs (x, y) such that $x : X$ and $y :$
 151 $Y(x)$.

152 Agda’s default syntax for a Sigma type is $\Sigma \lambda(x : X) \rightarrow Y$, but we prefer the notation
 153 $\Sigma x : X, Y$, which is closer to the standard syntax described in the preceding paragraph.
 154 Fortunately, this preferred notation is available in the Type Topology library (see [6, Σ types]).⁴

155 Convenient notations for the first and second projections out of a product are $\lfloor _ \rfloor$ and $\lfloor _ \rfloor$,
 156 respectively. However, to improve readability or to avoid notation clashes with other modules,
 157 we sometimes use more standard alternatives, such as pr_1 and pr_2 , or fst and snd , or some
 158 combination of these. The definitions are standard so we omit them (see [5] for details).

159 3 Algebras

160 A standard way to define algebraic structures in type theory is using record types. However, we
 161 feel the dependent pair (or Sigma) type (§2.2) is more natural, as it corresponds semantically
 162 to the existential quantifier of logic. Therefore, many of the important types of the UALib are
 163 defined as Sigma types. In this section, we use function types and Sigma types to define the
 164 types of *operations* (§3.1), *signatures* (§3.2), *algebras* (§3.3), and *product algebras* (§3.4).

³ See agda.readthedocs.io/en/v2.6.1.2/language/universe-levels.html.

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

3.1 Operation type

Here is how the type of *operations* is defined in the UALib.

```
Op :  $\mathcal{V} \cdot \rightarrow \mathcal{U} \cdot \rightarrow \mathcal{U} \sqcup \mathcal{V} \cdot$ 
Op I A = (I  $\rightarrow$  A)  $\rightarrow$  A
```

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 $I \rightarrow A$ (the type of “tuples”) and codomain A . For example, the type of *projections* is defined using the `Op` type, as follows.

```
 $\pi : \{I : \mathcal{V} \cdot\} \{A : \mathcal{U} \cdot\} \rightarrow I \rightarrow \text{Op } I A$ 
 $\pi \ i \ x = x \ i$ 
```

The last two lines of the code block above codify the i -th I -ary projection operation on A .

3.2 Signature type

We define the type of (algebraic) *signatures* as follows.

```
Signature : ( $\mathcal{O} \mathcal{V} : \text{Universe}$ )  $\rightarrow$  ( $\mathcal{O} \sqcup \mathcal{V}$ )+
Signature  $\mathcal{O} \mathcal{V} = \Sigma F : \mathcal{O} \cdot, (F \rightarrow \mathcal{V} \cdot)$ 
```

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 `||_|` (§2.2) 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 :  $\mathcal{O} \cdot$  where
  e : monoid-op
  · : monoid-op

monoid-sig : Signature  $\mathcal{O} \mathcal{U}_0$ 
monoid-sig = monoid-op ,  $\lambda \{ e \rightarrow 0; \cdot \rightarrow 2 \}$ 
```

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

3.3 Algebra type

For a fixed signature $S : \text{Signature } \mathcal{O} \mathcal{V}$ and universe \mathcal{U} , we define the type of *algebras in the signature* S (or *S-algebras*) and with *domain* (or *carrier*) $A : \mathcal{U} \cdot$ as follows.⁵

```
Algebra : ( $\mathcal{U} : \text{Universe}$ )( $S : \text{Signature } \mathcal{O} \mathcal{V}$ )  $\rightarrow$   $\mathcal{O} \sqcup \mathcal{V} \sqcup \mathcal{U}$ +
Algebra  $\mathcal{U} S = \Sigma A : \mathcal{U} \cdot, ((f : | S |) \rightarrow \text{Op } (|| S || f) A)$ 
```

One could call an inhabitant of `Algebra $S \mathcal{U}$` as an “ ∞ -algebra” because its domain can be an

⁵ 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 [5] and the html documentation available at <https://ualib.gitlab.io/UALib.Algebras.Algebras.html>.

arbitrary type, say, $A : \mathcal{U} \cdot$ and need not be *truncated* at some level (§A.5). In particular, A need not be an *h-set* (as defined in §A.5).⁶

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

```

216   $\_ \hat{\_} : (f : | S |)(\mathbf{A} : \text{Algebra } \mathcal{U} S) \rightarrow (| S | f \rightarrow | \mathbf{A} |) \rightarrow | \mathbf{A} |$ 
217
218   $f \hat{\_} \mathbf{A} = \lambda x \rightarrow (| \mathbf{A} | f) x$ 

```

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} we have a surjective map $h_0 : X \rightarrow | \mathbf{A} |$ from variables onto the domain of \mathbf{A} .

```

226   $\_ \rightarrow \_ : \{ \mathcal{U} \mathcal{X} : \text{Universe} \} \rightarrow \mathcal{X} \cdot \rightarrow \text{Algebra } \mathcal{U} S \rightarrow \mathcal{X} \sqcup \mathcal{U} \cdot$ 
227   $X \rightarrow \mathbf{A} = \Sigma h : (X \rightarrow | \mathbf{A} |) , \text{Epic } h$ 

```

3.4 Product algebra type

Suppose we are given a type $I : \mathcal{F}$ (of “indices”) and an indexed family $\mathcal{A} : I \rightarrow \text{Algebra } \mathcal{U} S$ of S -algebras. Then we define the *product algebra* $\prod \mathcal{A}$ in the following natural way.⁷

```

232   $\prod : \{ \mathcal{F} : \text{Universe} \} \{ I : \mathcal{F} \cdot \} \{ \mathcal{A} : I \rightarrow \text{Algebra } \mathcal{U} S \} \rightarrow \text{Algebra } (\mathcal{F} \sqcup \mathcal{U}) S$ 
233   $\prod \{ \mathcal{F} \} \{ I \} \mathcal{A} =$ 
234   $((i : I) \rightarrow | \mathcal{A} i |) , \lambda(f : | S |)(a : | S | f \rightarrow (j : I) \rightarrow | \mathcal{A} j |)(i : I) \rightarrow (f \hat{\_} \mathcal{A} i) \lambda\{x \rightarrow a x i\}$ 

```

Here, the domain is the dependent function type $\prod | \mathcal{A} | := (i : I) \rightarrow | \mathcal{A} i |$ of “tuples”, the i -th components of which live in $| \mathcal{A} i |$, and the operations are simply the operations of the $\mathcal{A} i$, interpreted component-wise.

The Birkhoff theorem involves products of entire arbitrary (nonindexed) classes of algebras, and it is not obvious how to handle this constructively, or whether it is even possible to do so without making extra assumptions about the class. (See [4] for a discussion of this issue.) We describe our solution to this problem in § 6.3.

4 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 $\mathcal{C} \{ A \} \{ R \}$. These are defined as in the UALib as follows.

```

248   $[ \_ ] : \{ A : \mathcal{U} \cdot \} \rightarrow A \rightarrow \text{Rel } A \mathcal{R} \rightarrow \text{Pred } A \mathcal{R}$ 
249   $[ a ] R = \lambda x \rightarrow R a x$ 

```

⁶ 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 *uniqueness of identity proofs* explicitly where, and only where, a proof relies on this assumption.

⁷ To distinguish the product algebra from the standard product type available in the Agda Standard Library, instead of \prod (`\prod`) or Π (`\Pi`), we use the symbol \prod , which is typed in agda2-mode as `\Glb`.

251 $\mathcal{C} : \{A : \mathcal{U}^\bullet\} \{R : \text{Rel } A \mathcal{R}\} \rightarrow \text{Pred } A \mathcal{R} \rightarrow (\mathcal{U} \sqcup \mathcal{R}^+)$
 252 $\mathcal{C} \{A\} \{R\} = \lambda (C : \text{Pred } A \mathcal{R}) \rightarrow \Sigma a : A, C \equiv ([a] R)$

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

255 $_/_ : (A : \mathcal{U}^\bullet) \rightarrow \text{Rel } A \mathcal{R} \rightarrow \mathcal{U} \sqcup (\mathcal{R}^+)$
 256 $A / R = \Sigma C : \text{Pred } A \mathcal{R}, \mathcal{C} \{A\} \{R\} C$

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

261 $\llbracket _ \rrbracket : \{A : \mathcal{U}^\bullet\} \rightarrow A \rightarrow \{R : \text{Rel } A \mathcal{R}\} \rightarrow A / R$
 262 $\llbracket a \rrbracket \{R\} = ([a] R), a, \text{refl}$
 263 $\lceil _ \rceil : \{A : \mathcal{U}^\bullet\} \{R : \text{Rel } A \mathcal{R}\} \rightarrow A / R \rightarrow A$
 264 $\lceil a \rceil = \llbracket a \rrbracket$

269 4.1 Quotient extensionality

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

275 $\text{class-extensionality}' : \text{propext } \mathcal{R} \rightarrow \text{global-dfunext} \rightarrow \{A : \mathcal{U}^\bullet\} \{a a' : A\} \{R : \text{Rel } A \mathcal{R}\}$
 276 $\rightarrow (\forall a x \rightarrow \text{is-subsingleton } (R a x)) \rightarrow (\forall C \rightarrow \text{is-subsingleton } (\mathcal{C} C))$
 277 $\rightarrow \text{IsEquivalence } R$
 278 $\rightarrow R a a' \rightarrow (\llbracket a \rrbracket \{R\}) \equiv (\llbracket a' \rrbracket \{R\})$

281 We omit the proof. (See [5] or ualib.org for details.)

283 4.2 Compatibility

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

290 $\text{lift-rel} : \text{Rel } Z \mathcal{W} \rightarrow (\gamma \rightarrow Z) \rightarrow (\gamma \rightarrow Z) \rightarrow \mathcal{V} \sqcup \mathcal{W}^\bullet$
 291 $\text{lift-rel } R f g = \forall x \rightarrow R (f x) (g x)$

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

294 $\text{compatible-op} : \{\mathbf{A} : \text{Algebra } \mathcal{U} S\} \rightarrow |S| \rightarrow \text{Rel } |A| \mathcal{W} \rightarrow \mathcal{U} \sqcup \mathcal{V} \sqcup \mathcal{W}^\bullet$
 295 $\text{compatible-op } \{\mathbf{A}\} f R = \forall \{a\} \{b\} \rightarrow (\text{lift-rel } R) a b \rightarrow R ((f \hat{\ } \mathbf{A}) a) ((f \hat{\ } \mathbf{A}) b)$

299 Finally, to represent that all basic operations of an algebra are compatible with a given relation,
 300

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.

4.3 Congruence relations

This `UALib.Relations.Congruences` module of the Agda UALib defines a number of alternative representations of congruence relations of an algebra. We will only have occasion to use the Sigma and record type representations, which are defined as follows.

```
Con : {U : Universe} (A : Algebra U S) → O ⊔ V ⊔ U ⊔ + .
Con {U} A = Σ θ : ( Rel | A | U ) , 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
```

4.4 Quotient algebras

An important construction in universal algebra is the quotient of an algebra **A** with respect to a congruence relation θ of **A**. This quotient is typically denoted by \mathbf{A} / θ 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 _ )
          )
        )
```

5 Homomorphisms, terms, and subalgebras

5.1 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 $f, g : X \rightarrow Y$ are extensionally equal if 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 **A** and **B**, to commute with a function $g : A \rightarrow B$.


```

346 compatible-op-map : {Q U : Universe}{A : Algebra Q S}(B : Algebra U S)
347   (f : | S |)(g : | A | → | B |) → V ⊔ U ⊔ Q .
348 compatible-op-map A B f g = ∀ a → g ((f ^ A) a) ≡ (f ^ B) (g o a)

```

349 Note the appearance of the shorthand $\forall a$ in the definition of `compatible-op-map`. We can get
 350 away with this in place of a : $\parallel S \parallel f \rightarrow | A |$ since Agda is able to infer that the a here must
 351 be a tuple on $| A |$ of “length” $\parallel S \parallel f$ (the arity of f).
 352

353 Next we will define the type `hom A B` of *homomorphisms* from A to B in terms of the
 354 property `is-homomorphism`.
 355

```

356 is-homomorphism : {Q U : Universe}{A : Algebra Q S}(B : Algebra U S)
357   → (| A | → | B |) → Q ⊔ V ⊔ Q ⊔ U .
358 is-homomorphism A B g = ∀ (f : | S |) → compatible-op-map A B f g
359
360 hom : {Q U : Universe} → Algebra Q S → Algebra U S → Q ⊔ V ⊔ Q ⊔ U .
361 hom A B = Σ g : (| A | → | B |) , is-homomorphism A B g
362
363

```

364 5.2 Terms

365 We define an inductive datatype called `Term` which, not surprisingly, represents the type of
 366 *terms* in a given signature. Here the type $X : \mathfrak{X} \cdot$ represents an arbitrary collection of variable
 367 symbols.
 368

```

369 data Term {X : Universe}{X : X ·} : Q ⊔ V ⊔ X + · where
370   generator : X → Term{X}{X}
371   node : (f : | S |)(args : ∥ S ∥ f → Term{X}{X}) → Term

```

372 Terms can be viewed as acting on other terms and we can form an algebraic structure whose
 373 domain and basic operations are both the collection of term operations. We call this the *term*
 374 *algebra* and denote it by $\mathbf{T} X$.
 375

```

376 T : {X : Universe}{X : X ·} → Algebra (Q ⊔ V ⊔ X + ·) S
377 T {X} X = Term{X}{X} , node

```

378 The term algebra is *absolutely free*, or *universal*, for algebras in the signature S . That is, for
 379 every S -algebra A , every map $h : X \rightarrow | A |$ lifts to a homomorphism from $\mathbf{T} X$ to A , and the
 380 induced homomorphism is unique. This is proved by induction on the structure of terms, as
 381 follows.
 382
 383

```

384 free-lift : {X U : Universe}{X : X ·}{A : Algebra U S}(h : X → | A |) → | T X | → | A |
385 free-lift _ h (generator x) = h x
386 free-lift A h (node f args) = (f ^ A) λ i → free-lift A h (args i)
387
388 lift-hom : {X U : Universe}{X : X ·}{A : Algebra U S}(h : X → | A |) → hom (T X) A
389 lift-hom A h = free-lift A h , λ f a → ap ( _ ^ A ) refl
390
391 free-unique : {X U : Universe}{X : X ·} → funext V U
392   → (A : Algebra U S)(g h : hom (T X) A)
393   → (∀ x → | g | (generator x) ≡ | h | (generator x))
394   → (t : Term{X}{X})
395   → | g | t ≡ | h | t
396

```

```

397
398   free-unique _ _ _ p (generator x) = p x
399   free-unique fe A g h p (node f args) =
400     | g | (node f args)           ≡⟨ || g || f args ⟩
401     (f ^ A)(λ i → | g | (args i)) ≡⟨ ap ( _ ^ A ) γ ⟩
402     (f ^ A)(λ i → | h | (args i)) ≡⟨ (|| h || f args)⊥ ⟩
403     | h | (node f args) ■
404   where γ = fe λ i → free-unique fe A g h p (args i)

```

5.3 Subalgebras

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.

```

410   Subuniverses : {Q U : Universe}(A : Algebra Q S) → Pred (Pred | A | U) (Q ⊔ V ⊔ Q ⊔ U)
411   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.

```

416   data Sg {U W : Universe}(A : Algebra U S)(X : Pred | A | W) :
417     Pred | A | (Q ⊔ V ⊔ W ⊔ U) where
418     var : ∀ {v} → v ∈ X → v ∈ Sg A X
419     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 (the proof object is simply `app!`) and the proof that `Sg X` is the smallest subuniverse containing `X` is not much harder; it proceeds by induction on the shape of elements of `Sg X`.

```

425   sglsSmallest : {U W R : Universe}(A : Algebra U S){X : Pred | A | W}{Y : Pred | A | R}
426     → Y ∈ Subuniverses A → X ⊆ Y → Sg A X ⊆ Y
427
428   sglsSmallest _ _ _ XinY (var Xv) = XinY Xv
429   sglsSmallest A Y YsubA XinY (app f a SgXa) = fa∈Y
430   where
431     IH : Im a ⊆ Y
432     IH i = sglsSmallest A Y YsubA XinY (SgXa i)
433
434     fa∈Y : (f ^ A) a ∈ Y
435     fa∈Y = YsubA f a IH

```

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

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

```

445   _IsSubalgebraOf_ : {U W : Universe}(B : Algebra U S)(A : Algebra W S) → Q ⊔ V ⊔ U ⊔ W ·
446   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.

```

448  _≤_ : {U Q : Universe} (B : Algebra U S) (A : Algebra Q S) → Q ⊔ V ⊔ U ⊔ Q .
449
450  B ≤ A = B IsSubalgebraOf A
451

```

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

6 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 $p \cdot \mathbf{A} \equiv q \cdot \mathbf{A}$ 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 of the same signature, we write $\mathcal{K} \models p \approx q$ if $\mathbf{A} \models p \approx q$ for all $\mathbf{A} \in \mathcal{K}$.⁸

6.1 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 $\mathcal{K} \models p \approx q$.

```

466  _|=≈_| : {U X : Universe} {X : X ·} → Algebra U S → Term {X} {X} → Term → U ⊔ X ·
467  A |= p ≈ q = (p · A) ≡ (q · A)
468
469  _|=≈_| : {U X : Universe} {X : X ·} → Pred (Algebra U S) (OV U)
470  → Term {X} {X} → Term → Q ⊔ V ⊔ X ⊔ U + ·
471  _|=≈_| K p q = {A : Algebra _ S} → K A → A |= p ≈ q

```

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

```

478  Th : {U X : Universe} {X : X ·} → Pred (Algebra U S) (OV U)
479  → Pred (Term {X} {X} × Term) (Q ⊔ V ⊔ X ⊔ U +)
480  Th K = λ (p , q) → K |= p ≈ q
481
482  Mod : {U X : Universe} (X : X ·) → Pred (Term {X} {X} × Term {X} {X}) (Q ⊔ V ⊔ X ⊔ U +)
483  → Pred (Algebra U S) (Q ⊔ V ⊔ X + ⊔ U +)
484  Mod X E = λ A → ∀ p q → (p , q) ∈ E → A |= p ≈ q

```

6.2 Inductive types for closure operators

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

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

⁸ 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.

A *variety* is a class \mathcal{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, \mathcal{K} is a variety if and only if $\mathbf{HSP} \mathcal{K} \subseteq \mathcal{K}$.

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**.⁹ 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 $\mathbf{V} \mathcal{K}$.

6.3 Class products, $\mathbf{PS} \subseteq \mathbf{SP}$ and $\prod \mathbf{S} \in \mathbf{SP}$

We need to establish two important facts about the closure operators defined in the last section. The first is the inclusion $\mathbf{PS}(\mathcal{K}) \subseteq \mathbf{SP}(\mathcal{K})$. 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 $\mathbf{SP}(\mathcal{K})$. 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

⁹ See the `UAlib.Varieties.Varieties` module or the html documentation at ualib.gitlab.io/UAlib.Varieties.Varieties.html for the definition.

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.¹⁰

```

535 PS⊆SP : (P{ovu}{ovu} (S{u}{ovu} K)) ⊆ (S{ovu}{ovu} (P{u}{ovu} K))
536
537 PS⊆SP (pbase (sbase x)) = sbase (pbase x)
538
539 PS⊆SP (pbase (slift{A} x)) = slift (S⊆SP{u}{ovu}{K} (slift x))
540
541 PS⊆SP (pbase {B} (ssub{A} sA B≤A)) = ...

```

Evidently, the proof is by induction on the form of an inhabitant of $\text{PS}(\mathcal{K})$, handling in turn each way such an inhabitant can be constructed. (See [5] 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 $\text{S}(\mathcal{K})$ belongs to $\text{SP}(\mathcal{K})$.¹¹ The type that serves to index the class (and the product of its members) is the following.

```

544 J : {u : Universe} → Pred (Algebra u S)(ov u) → (ov u) ·
545 J {u} K = Σ A : (Algebra u S) , A ∈ K

```

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

```

546 A : {u : Universe}{K : Pred (Algebra u S)(ov u)} → J K → Algebra u S
547 A{u}{K} = λ (i : (J K)) → | i |

```

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

```

548 class-product : {u : Universe} → Pred (Algebra u S)(ov u) → Algebra (ov u) S
549 class-product {u} K = Π ( A{u}{K} )

```

Observe that the elements of the product are the maps from \mathcal{J} to $\{A \ i : i \in \mathcal{J}\}$. If $p : A \in \mathcal{K}$ is a proof that A belongs to \mathcal{K} , then we can view the pair $(A, p) \in \mathcal{J} \mathcal{K}$ as an index over the class, and $A(A, p)$ as the result of projecting the product onto the (A, p) -th component.

6.4 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 [5] or ualib.org for details.) For example, the assertion that `H` preserves identities is stated formally as follows:

```

574 H-id1 : {u x : Universe}{X : x}{K : Pred (Algebra u S)(ov u)}
575 (p q : Term{x}{X})
576
577 → K ⊨ p ≈ q → H{u}{K} K ⊨ p ≈ q

```

The proof is by induction and handles each of the four constructors of `H` (`hbase`, `hlift`, `hhimg`,

¹⁰For legibility we use `ovu` as a shorthand for $\mathbf{0} \sqcup \mathcal{V} \sqcup \mathcal{U}^+$.

¹¹This 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. However, after a number of failed attempts, the right type revealed itself. Now that we have it, it seems almost obvious.

and `hiso`) in turn. Of course, the `UALib.Varieties.Preservation` module of the UALib contains a complete proof of (2), which can be viewed in the source code or the html documentation.¹² 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.¹²

6.5 The free algebra in theory

In this section, we formalize, for a given class \mathcal{K} of S -algebras, the (relatively) *free algebra in* $\mathbf{SP}(\mathcal{K})$ over X . Let $\Theta(\mathcal{K}, \mathbf{A}) := \{\theta \in \mathbf{Con} \mathbf{A} : \mathbf{A} / \theta \in \mathbf{S} \mathcal{K}\}$ and $\psi(\mathcal{K}, \mathbf{A}) := \bigcap \Theta(\mathcal{K}, \mathbf{A})$. Since the term algebra $\mathbf{T} X$ is free for (and in) the class $\mathbf{Alg}(S)$ of all S -algebras, it is free for every subclass \mathcal{K} of $\mathbf{Alg}(S)$. Although $\mathbf{T} X$ is not necessarily a member of \mathcal{K} , if we form the quotient $\mathfrak{F} := (\mathbf{T} X) / \psi(\mathcal{K}, \mathbf{T} X)$, of $\mathbf{T} X$ modulo the congruence

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

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 $\mathbf{SP}(\mathcal{K})$, 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 $\mathbf{SP}(\mathcal{K})$.

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

$$\begin{aligned} \psi : (\mathcal{K} : \mathbf{Pred} (\mathbf{Algebra} \mathcal{U} S) \mathbf{ov}) &\rightarrow \mathbf{Pred} (| \mathbf{T} X | \times | \mathbf{T} X |) \mathbf{ov} \\ \psi \mathcal{K} (p, q) &= \forall (\mathbf{A} : \mathbf{Algebra} \mathcal{U} S) \rightarrow (sA : \mathbf{A} \in \mathbf{S}\{\mathcal{U}\}\{\mathcal{U}\} \mathcal{K}) \\ &\rightarrow | \mathbf{lift-hom} \mathbf{A} (\mathbf{fst}(\mathcal{X} \mathbf{A})) | p \equiv | \mathbf{lift-hom} \mathbf{A} (\mathbf{fst}(\mathcal{X} \mathbf{A})) | q, \end{aligned}$$

We convert the predicate ψ into a relation by currying, $\psi \mathbf{Rel} \mathcal{K} p q = \psi \mathcal{K} (p, q)$. The latter is an equivalence relation that is compatible with the operations of $\mathbf{T} X$.¹³ Therefore, from $\psi \mathbf{Rel}$ we can construct a congruence of the term algebra $\mathbf{T} X$, the inhabitants of which are pairs of terms representing identities satisfied by all subalgebras of algebras in the class.

$$\begin{aligned} \psi \mathbf{Con} : (\mathcal{K} : \mathbf{Pred} (\mathbf{Algebra} \mathcal{U} S) \mathbf{ov}) &\rightarrow \mathbf{Congruence} (\mathbf{T} X) \\ \psi \mathbf{Con} \mathcal{K} &= \mathbf{mkcon} (\psi \mathbf{Rel} \mathcal{K}) (\psi \mathbf{compatible} \mathcal{K}) \psi \mathbf{IsEquivalence} \end{aligned}$$

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

$$\begin{aligned} \mathfrak{F} : \mathbf{Algebra} \mathfrak{F} S \\ \mathfrak{F} &= \mathbf{T} X / (\psi \mathbf{Con} \mathcal{K}) \end{aligned}$$

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

¹² See the source code file `Preservation.lagda`, or the html documentation page ualib.gitlab.io/UALib.Varieties.Preservation.html.

¹³ We omit the easy proofs, but see the `UALib.Birkhoff.FreeAlgebra` module for details.

7 Birkhoff's HSP Theorem

This section presents the `UALib.Birkhoff` module of the Agda UALib; §7.1 describes the key components of the proof of Birkhoff's theorem, and §7.2 reveals how these components fit together to complete the proof.

7.1 HSP Lemmas

This subsection gives formal statements of four lemmas that we will string together in §7.2 to complete the proof of Birkhoff's theorem.

The first hurdle is the `lift-alg-V-closure` lemma, which says that if an algebra \mathbf{A} belongs to the variety \mathbb{V} , then so does its lift. This dispenses with annoying universe level problems that arise later—a minor technical issue, but the proof is long and tedious, not to mention uninteresting. (See [5] or `ualib.org` for details.) The next fact that must be formalized is the inclusion $\mathbf{SP}(\mathcal{K}) \subseteq \mathbf{V}(\mathcal{K})$, 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 [5] 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 §6.3.

```

 $\mathfrak{I}s$  : ovu · — for indexing over all subalgebras of algebras in  $\mathcal{K}$ 
 $\mathfrak{I}s$  =  $\mathfrak{I}$  ( $\mathbf{S}\{\mathcal{U}\}\{\mathcal{U}\}$   $\mathcal{K}$ )
 $\mathfrak{A}s$  :  $\mathfrak{I}s \rightarrow \mathbf{Algebra} \mathcal{U} S$ 
 $\mathfrak{A}s$  =  $\lambda (i : \mathfrak{I}s) \rightarrow | i |$ 
 $\mathfrak{C}$  :  $\mathbf{Algebra} \text{ovu} S$  — the product of all subalgebras of algebras in  $\mathcal{K}$ 
 $\mathfrak{C}$  =  $\prod \mathfrak{A}s$ 

```

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

```

 $\mathfrak{h}_0$  :  $X \rightarrow | \mathfrak{C} |$ 
 $\mathfrak{h}_0$   $x$  =  $\lambda i \rightarrow (\mathbf{fst} (\mathfrak{X} (\mathfrak{A}s i))) x$ 
 $\phi_{\mathfrak{C}}$  :  $\mathbf{hom} (\mathbf{T} X) \mathfrak{C}$ 
 $\phi_{\mathfrak{C}}$  =  $\mathbf{lift-hom} \mathfrak{C} \mathfrak{h}_0$ 
 $\mathfrak{f}$  :  $\mathbf{hom} \mathfrak{F} \mathfrak{C}$ 
 $\mathfrak{f}$  =  $\mathfrak{F}$ -free-lift  $\mathfrak{C} \mathfrak{h}_0$  ,  $\lambda f \mathbf{a} \rightarrow || \phi_{\mathfrak{C}} || f (\lambda i \rightarrow \ulcorner \mathbf{a} i \urcorner)$ 

```

The hard part is showing that \mathfrak{f} is a monomorphism. For lack of space, we must omit the inner workings of the proof, and settle for the outer shell. (See [5] or `ualib.org` for details.) For ease of notation, let $\Psi = \psi\mathbf{Rel} \mathcal{K}$.

```

 $\mathbf{monf}$  :  $\mathbf{Monic} | \mathfrak{f} |$ 
 $\mathbf{monf}$  ( $\mathfrak{f} (\Psi p)$  ,  $p$  ,  $\mathbf{refl} \_$ ) ( $\mathfrak{f} (\Psi q)$  ,  $q$  ,  $\mathbf{refl} \_$ )  $f p q$  =  $\gamma$ 
  where
  ... details omitted ...
   $\gamma$  : ( $\Psi p$  ,  $p$  ,  $\mathbf{refl} \_$ )  $\equiv$  ( $\Psi q$  ,  $q$  ,  $\mathbf{refl} \_$ )
   $\gamma$  = class-extensionality'  $p e g f e s s R s s A \psi\mathbf{IsEquivalence} p \Psi q$ 

```

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


```

671    $\mathfrak{F} \leq \mathcal{C} : \text{is-set} \mid \mathcal{C} \mid \rightarrow \mathfrak{F} \leq \mathcal{C}$ 
672    $\mathfrak{F} \leq \mathcal{C} \text{ Cset} = \mid \mathfrak{f} \mid , (\text{embf} , \parallel \mathfrak{f} \parallel)$ 
673   where
674     embf : is-embedding  $\mid \mathfrak{f} \mid$ 
675     embf = monic-into-set-is-embedding Cset  $\mid \mathfrak{f} \mid$  monf

```

676 Once we have all of the foregoing at hand, it is not hard to show that \mathfrak{F} belongs to $\text{SP}(\mathcal{K})$, and
 677 hence to $\text{V}(\mathcal{K})$. (See [5] or ualib.org for details.)

678 7.2 The HSP Theorem

679 It is now all but trivial to use what we have already proved and piece together a complete proof
 680 of Birkhoff’s celebrated HSP theorem asserting that every variety is defined by a set of identities.

```

681   – Birkhoff’s theorem: every variety is an equational class.
682   birkhoff : is-set  $\mid \mathcal{C} \mid \rightarrow \text{Mod } X (\text{Th } \mathbb{V}) \subseteq \mathbb{V}$ 
683   birkhoff Cset  $\{\mathbf{A}\} \alpha = \gamma$ 
684   where
685      $\phi : \Sigma h : (\text{hom } \mathfrak{F} \mathbf{A}) , \text{Epic} \mid h \mid$ 
686      $\phi = (\mathfrak{F}\text{-lift-hom } \mathbf{A} \mid \mathbb{X} \mathbf{A} \mid) , \mathfrak{F}\text{-lift-of-epic-is-epic } \mathbf{A} \mid \mathbb{X} \mathbf{A} \mid \parallel \mathbb{X} \mathbf{A} \parallel$ 
687
688     AiF :  $\mathbf{A}$  is-hom-image-of  $\mathfrak{F}$ 
689     AiF =  $(\mathbf{A} , \mid \text{fst } \phi \mid , (\parallel \text{fst } \phi \parallel , \text{snd } \phi) ) , \text{refl-}\cong$ 
690
691      $\gamma : \mathbf{A} \in \mathbb{V}$ 
692      $\gamma = \text{vhimg } (\mathfrak{F} \in \mathbb{V} \text{ Cset}) \text{ AiF}$ 

```

694 Some readers might worry that we haven’t quite achieved our goal because what we just proved
 695 (birkhoff) is not an “if and only if” assertion. Those fears are quickly put to rest by noting that
 696 the converse—that every equational class is closed under HSP—is straightforward, as discussed
 697 in §6.4. Indeed, in the `UALib.Varieties.Preservation` module of the UALib we provide proofs of

```

699   ■ (H-id1)  $\mathcal{K} \models p \approx q \rightarrow \mathbf{H} \mathcal{K} \models p \approx q$ 
700   ■ (S-id1)  $\mathcal{K} \models p \approx q \rightarrow \mathbf{S} \mathcal{K} \models p \approx q$ 
701   ■ (P-id1)  $\mathcal{K} \models p \approx q \rightarrow \mathbf{P} \mathcal{K} \models p \approx q$ 

```

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

703 8 Current and future directions

704 Now that we have accomplished our initial goal of formalizing Birkhoff’s HSP theorem in Agda,
 705 our next goal is to support current mathematical research by formalizing some recently proved
 706 theorems. For example, we intend to formalize theorems about the computational complexity
 707 of decidable properties of algebraic structures. Part of this effort will naturally involve further
 708 work on the library itself, and may lead to new observations about dependent type theory.

709 One natural question is whether there are any objects of our research that cannot be repres-
 710 ented constructively in type theory. As mentioned in § 1.1, a constructive version of Birkhoff’s
 711 theorem was presented by Carlström in [4]. We should determine how the two new hypotheses
 712 required by Carlström compare with the assumptions we make in our Agda proof.

713 Finally, as one of the goals of the Agda UALib project is to make computer formalization of
 714 mathematics more accessible to mathematicians working in universal algebra and model theory,
 715 we welcome feedback from the community and are happy to field questions about the UALib,
 716 how it is installed, and how it can be used to prove theorems that are not yet part of the library.

References

- 1 Clifford Bergman. *Universal Algebra: fundamentals and selected topics*, volume 301 of *Pure and Applied Mathematics (Boca Raton)*. CRC Press, Boca Raton, FL, 2012.
- 2 G Birkhoff. On the structure of abstract algebras. *Proceedings of the Cambridge Philosophical Society*, 31(4):433–454, Oct 1935.
- 3 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.
- 4 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>.
- 5 William DeMeo. The Agda Universal Algebra Library and Birkhoff’s Theorem in Dependent Type Theory. *CoRR*, abs/2101.10166, 2021. source code: <https://gitlab.com/ualib/ualib.gitlab.io>. URL: <https://arxiv.org/abs/2101.10166>, arXiv:2101.10166.
- 6 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.
- 7 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>.
- 8 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>.
- 9 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>.
- 10 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.
- 11 The Agda Team. Agda Language Reference section on Axiom K, 2021. URL: <https://agda.readthedocs.io/en/v2.6.1/language/without-k.html>.
- 12 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>.
- 13 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>.

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.

A.1 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.

Universe levels 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 level \mathcal{U} to the universe $\mathcal{U} \cdot := \text{Set } \mathcal{U}$ and the level $\mathcal{U}^+ := \text{Isuc } \mathcal{U}$, respectively. Thus,

Standard Agda	Type Topology/UALib
<code>Level</code>	<code>Universe</code>
<code>$\mathcal{U} : \text{Level}$</code>	<code>$\mathcal{U} : \text{Universe}$</code>
<code><code>Set</code> \mathcal{U}</code>	<code>$\mathcal{U} \cdot$</code>
<code><code>Isuc</code> \mathcal{U}</code>	<code>\mathcal{U}^+</code>
<code><code>Set</code> (<code><code>Isuc</code> \mathcal{U}</code>)</code>	<code>\mathcal{U}^{++}</code>
<code><code>Izero</code></code>	<code>\mathcal{U}_0</code>
<code><code>Setω</code></code>	<code>\mathcal{U}_ω</code>

■ **Table 1** Special notation for universe levels

$\mathcal{U} \cdot$ is simply an alias for the universe `Set \mathcal{U}` , and we have $\mathcal{U} \cdot : (\mathcal{U}^+)$. Table 1 translates between standard Agda syntax and Type Topology/UALib notation.

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 $\mathcal{U} \cdot$ is written `Set \mathcal{U}` in standard Agda), but in univalent mathematics the types in $\mathcal{U} \cdot$ need not be sets, so the standard Agda notation can be misleading.

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 corresponding type is $(\mathcal{U} \sqcup \mathcal{V}) \cdot$, or, more simply, $\mathcal{U} \sqcup \mathcal{V} \cdot$ (since `_` has higher precedence than `_`).

A.2 Tools for noncumulative universes

We present here some general and some domain-specific tools that we developed for dealing with the noncumulativity of universes in Agda. (See [5, §3.3] for more details).

First, 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 { $\mathcal{U} \mathcal{W} : \text{Universe}$ } ( $X : \mathcal{U} \cdot$ ) :  $\mathcal{U} \sqcup \mathcal{W} \cdot$  where
  constructor lift
  field lower :  $X$ 
open Lift
```

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

```
lower~lift : { $\mathcal{X} \mathcal{W} : \text{Universe}$ } { $X : \mathcal{X} \cdot$ } → lower{ $\mathcal{X}$ } { $\mathcal{W}$ } ∘ lift ≡ id X
lower~lift = refl _
```

Similarly, `lift ∘ lower ≡ id (Lift{ \mathcal{X} } { \mathcal{W} })`.

A.2.1 The lift of an algebra

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

```
lift-op : { $\mathcal{U} : \text{Universe}$ } { $I : \mathcal{V} \cdot$ } { $A : \mathcal{U} \cdot$ }
  → (( $I \rightarrow A$ ) →  $A$ ) → ( $\mathcal{W} : \text{Universe}$ ) → (( $I \rightarrow \text{Lift}\{\mathcal{U}\}\{\mathcal{W}\} A$ ) →  $\text{Lift}\{\mathcal{U}\}\{\mathcal{W}\} A$ )
lift-op f  $\mathcal{W}$  =  $\lambda x \rightarrow \text{lift} (f (\lambda i \rightarrow \text{lower} (x i)))$ 

lift-∞-algebra lift-alg : { $\mathcal{U} : \text{Universe}$ } → Algebra  $\mathcal{U} S$  → ( $\mathcal{W} : \text{Universe}$ ) → Algebra ( $\mathcal{U} \sqcup \mathcal{W}$ ) S
```

```

799 lift-∞-algebra A W = Lift | A | , (λ (f : | S |) → lift-op (|| A || f) W)
800 lift-alg = lift-∞-algebra

```

801 A.3 Equality

802 Perhaps the most important types in type theory are the equality types. The *definitional*
803 *equality* we use is a standard one and is often referred to as “reflexivity” or “refl”. In our case,
804 it is defined in the Identity-Type module of the Type Topology library, but apart from syn-
805 tax it is equivalent to the identity type used in most other Agda libraries. Here is the definition.
806

```

807 data _≡_ {U} {X : U ·} : X → X → U · where
808   refl : {x : X} → x ≡ x

```

809 A.4 Function extensionality and intensionality

810 Extensional equality of functions, or *function extensionality*, is a principle that is often assumed
811 in the Agda UALib. It asserts that two point-wise equal functions are equal and is defined in
812 the Type Topology library in the following natural way:
813

```

814 funext : ∀ U V → (U ⊔ V)+ ·
815 funext U V = {X : U ·} {Y : V ·} {f g : X → Y} → f ~ g → f ≡ g

```

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

817 Pointwise equality of functions is typically what one means in informal settings when one
818 says that two functions are equal. However, as Escardó notes in [6], function extensionality
819 is known to be not provable or disprovable in Martin-Löf Type Theory. It is an independent
820 axiom which may be assumed (or not) without making the logic inconsistent.

821 Dependent and polymorphic notions of function extensionality are also defined in the UALib
822 and Type Topology libraries (see [5, §2.4] and [6, §17-18]).

823 *Function intensionality* is the opposite of function extensionality and it comes in handy
824 whenever we have a definitional equality and need a point-wise equality.
825
826

```

827 intensionality : {U W : Universe} {A : U ·} {B : W ·} {f g : A → B}
828   → f ≡ g → (x : A) → f x ≡ g x
829 intensionality (refl _) _ = refl _

```

830 Of course, the intensionality principle has dependent and polymorphic analogues defined in the
831 Agda UALib, but we omit the definitions. See [5, §2.4] for details.
832

833 A.5 Truncation and sets

834 In general, we may have many inhabitants of a given type, hence (via Curry-Howard corres-
835 pondence) many proofs of a given proposition. For instance, suppose we have a type X and an
836 identity relation \equiv_x on X so that, given two inhabitants of X , say, $a b : X$, we can form the
837 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
838 which case we write $p q : a \equiv_x b$. Then we might wonder whether and in what sense are the
839 two proofs p and q the “same.” We are asking about an identity type on the identity type \equiv_x ,
840 and whether there is some inhabitant r of this type; i.e., whether there is a proof $r : p \equiv_{x1} q$.
841 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
842 property of the types X and \equiv_x , this is sometimes called *uniqueness of identity proofs*.

843 Perhaps we have two proofs, say, $r s : p \equiv_{x1} q$. Then it is natural to wonder whether r
844 $\equiv_{x2} s$ 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* [9], 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 :  $\mathcal{U} \rightarrow \mathcal{U}$ 
is-set  $X = (x\ y : X) \rightarrow \text{is-subsingleton } (x \equiv y)$  (3)
```

where

```
is-subsingleton :  $\mathcal{U} \rightarrow \mathcal{U}$ 
is-subsingleton  $X = (x\ y : X) \rightarrow x \equiv y$  (4)
```

Truncation is used in various places in the UALib, and it is required in the proof of Birkhoff's theorem. Consult [6, §34-35] or [9, §7.1] for more details.

A.6 Inverses, Epics and Monics

This section describes some of the more important parts of the `UALib.Prelude.Inverses` module. In § A.6.1, we define an inductive datatype that represents our semantic notion of the *inverse image* of a function. In § A.6.2 we define types for *epic* and *monic* functions. Finally, in Subsections A.7, we consider the type of *embeddings* (defined in [6, §26]), and determine how this type relates to our type of monic functions.

A.6.1 Inverse image type

```
data Image  $\Rightarrow$  _ {A :  $\mathcal{U}$ } {B :  $\mathcal{W}$ } (f : A  $\rightarrow$  B) : B  $\rightarrow$   $\mathcal{U} \sqcup \mathcal{W}$ 
  where
    im : (x : A)  $\rightarrow$  Image f  $\Rightarrow$  f x
    eq : (b : B)  $\rightarrow$  (a : A)  $\rightarrow$  b  $\equiv$  f a  $\rightarrow$  Image f  $\Rightarrow$  b
```

Note that an inhabitant of `Image f \Rightarrow 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 \Rightarrow 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 $(a, p) : \text{Image } f \Rightarrow b$ that b belongs to the image of f , produces a (a preimage of b under f).

```
Inv : {A :  $\mathcal{U}$ } {B :  $\mathcal{W}$ } (f : A  $\rightarrow$  B) (b : B)  $\rightarrow$  Image f  $\Rightarrow$  b  $\rightarrow$  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 \Rightarrow 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)`),¹⁴ 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 .)

¹⁴The dotted pattern is used when the form of the argument is forced... todo: fix this sentence

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

```

889   InvInv : {A :  $\mathcal{U}$  ·} {B :  $\mathcal{W}$  ·} (f : A → B)
890           (b : B) (b ∈ Image f) → Image f ⊃ b
891
892   →      f (Inv f b (b ∈ Image f)) ≡ b
893
894   InvInv f (f a) (im a) = refl _
895   InvInv f b (eq b a b ≡ f a) = b ≡ f a-1

```

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

A.6.2 Epic and monic function types

Given universes \mathcal{U}, \mathcal{W} , types $A : \mathcal{U} \cdot$ and $B : \mathcal{W} \cdot$, and $f : A \rightarrow B$, 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

```

906   Epic : {A :  $\mathcal{U}$  ·} {B :  $\mathcal{W}$  ·} (f : A → B) →  $\mathcal{U} \sqcup \mathcal{W} \cdot$ 
907   Epic f =  $\forall y \rightarrow \text{Image } f \ni y$ 

```

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

```

912   EpicInv : {A :  $\mathcal{U}$  ·} {B :  $\mathcal{W}$  ·}
913           (f : A → B) → Epic f
914
915   →      B → A
916   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.

```

920   EpicInvRightInv : funext  $\mathcal{W} \mathcal{W} \rightarrow \{A : \mathcal{U} \cdot\} \{B : \mathcal{W} \cdot\}$ 
921                   (f : A → B) (fE : Epic f)
922
923   →      f ∘ (EpicInv f fE) ≡ id B
924   EpicInvRightInv f fE = fE (λ x → InvInv f x (fE x))

```

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

```

930   Monic : {A :  $\mathcal{U}$  ·} {B :  $\mathcal{W}$  ·} (f : A → B) →  $\mathcal{U} \sqcup \mathcal{W} \cdot$ 
931   Monic f =  $\forall a_1 a_2 \rightarrow f a_1 \equiv f a_2 \rightarrow a_1 \equiv a_2$ 

```

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 `MonicInvLeftInv` in [5, §2.3] for details.)

A.7 Monic functions are set embeddings

An *embedding*, as defined in [6, §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.

```

939 is-embedding : {X : U ·} {Y : V ·} → (X → Y) → U ⊔ V ·
940 is-embedding f = (y : codomain f) → is-subsingleton (fiber f y)
941
942 fiber : {X : U ·} {Y : V ·} (f : X → Y) → Y → U ⊔ V ·
943 fiber f y = Σ x : domain f , f x ≡ y

```

944 This is not simply a *monic* function (§A.6.2), and it is important to understand why not.
 945 Suppose $f : X \rightarrow Y$ is a monic function from X to Y , so we have a proof $p : \mathbf{Monic} f$. To prove f
 946 is an embedding we must show that for every $y : Y$ we have **is-subsingleton** (**fiber** f y). That is,
 947 for all $y : Y$, we must prove the following implication:

$$948 \frac{(x \, x' : X) \quad (p : f \, x \equiv y) \quad (q : f \, x' \equiv y) \quad (m : \mathbf{Monic} \, f)}{(x, p) \equiv (x', q)}$$

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

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

```

959 monic-into-set-is-embedding : {A : U ·} {B : W ·} → is-set B
960   → (f : A → B) → Monic f
961   -----
962   → is-embedding

```

963 We omit the formal proof for lack of space, but see [5, §2.3].
 964

965 A.8 Unary Relations (predicates)

966 We need a mechanism for implementing the notion of subsets in Agda. A typical one is called
 967 **Pred** (for predicate). More generally, **Pred** $A \, \mathbf{U}$ can be viewed as the type of a property that
 968 elements of type A might satisfy. We write $P : \mathbf{Pred} \, A \, \mathbf{U}$ to represent the semantic concept of
 969 a collection of elements of type A that satisfy the property P . Here is the definition, which is
 970 similar to the one found in the `Relation/Unary.agda` file of the Agda Standard Library.
 971

```

972 Pred : U · → (V : Universe) → U ⊔ V + ·
973 Pred A V = A → V ·

```

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

976 In type theory everything is a type. As we have just seen, this includes subsets. Since
 977 the notion of equality for types is usually a nontrivial matter, it may be nontrivial to represent
 978 equality of subsets. Fortunately, it is straightforward to write down a type that represents what
 979 it means for two subsets to be equal in informal (pencil-paper) mathematics. In the UALib we
 980 denote this *subset equality* by $='$ and define it as follows.¹⁵
 981
 982

¹⁵ 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


```

983    $\_ = \_ : \{ \mathcal{U} \ \mathcal{W} \ \mathcal{T} : \mathbf{Universe} \} \{ A : \mathcal{U} \cdot \} \rightarrow \mathbf{Pred} \ A \ \mathcal{W} \rightarrow \mathbf{Pred} \ A \ \mathcal{T} \rightarrow \mathcal{U} \sqcup \mathcal{W} \sqcup \mathcal{T} \cdot$ 
984    $P = \cdot \ Q = (P \subseteq Q) \times (Q \subseteq P)$ 

```

985 A.9 Binary Relations

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

```

991    $\mathbf{REL} : \{ \mathcal{R} : \mathbf{Universe} \} \rightarrow \mathcal{U} \cdot \rightarrow \mathcal{R} \cdot \rightarrow (\mathcal{N} : \mathbf{Universe}) \rightarrow (\mathcal{U} \sqcup \mathcal{R} \sqcup \mathcal{N}^+) \cdot$ 
992    $\mathbf{REL} \ A \ B \ \mathcal{N} = A \rightarrow B \rightarrow \mathcal{N} \cdot$ 

```

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

```

995    $\mathbf{reflexive} : \{ \mathcal{R} : \mathbf{Universe} \} \{ X : \mathcal{U} \cdot \} \rightarrow \mathbf{Rel} \ X \ \mathcal{R} \rightarrow \mathcal{U} \sqcup \mathcal{R} \cdot$ 
996    $\mathbf{reflexive} \ \_ \approx \_ = \forall x \rightarrow x \approx x$ 
997
998    $\mathbf{symmetric} : \{ \mathcal{R} : \mathbf{Universe} \} \{ X : \mathcal{U} \cdot \} \rightarrow \mathbf{Rel} \ X \ \mathcal{R} \rightarrow \mathcal{U} \sqcup \mathcal{R} \cdot$ 
999    $\mathbf{symmetric} \ \_ \approx \_ = \forall x \ y \rightarrow x \approx y \rightarrow y \approx x$ 
1000
1001    $\mathbf{transitive} : \{ \mathcal{R} : \mathbf{Universe} \} \{ X : \mathcal{U} \cdot \} \rightarrow \mathbf{Rel} \ X \ \mathcal{R} \rightarrow \mathcal{U} \sqcup \mathcal{R} \cdot$ 
1002    $\mathbf{transitive} \ \_ \approx \_ = \forall x \ y \ z \rightarrow x \approx y \rightarrow y \approx z \rightarrow x \approx z$ 

```

1003 A.10 Kernels of functions

1004 The kernel of a function can be defined in many ways. For example,

```

1005
1006    $\mathbf{KER} : \{ \mathcal{R} : \mathbf{Universe} \} \{ A : \mathcal{U} \cdot \} \{ B : \mathcal{R} \cdot \} \rightarrow (A \rightarrow B) \rightarrow \mathcal{U} \sqcup \mathcal{R} \cdot$ 
1007    $\mathbf{KER} \ \{ \mathcal{R} \} \{ A \} \ g = \Sigma \ x : A , \Sigma \ y : A , g \ x \equiv g \ y$ 

```

1008 or as a unary relation (predicate) over the Cartesian product,

```

1009    $\mathbf{KER-pred} : \{ \mathcal{R} : \mathbf{Universe} \} \{ A : \mathcal{U} \cdot \} \{ B : \mathcal{R} \cdot \} \rightarrow (A \rightarrow B) \rightarrow \mathbf{Pred} \ (A \times A) \ \mathcal{R}$ 
1010    $\mathbf{KER-pred} \ g \ (x , y) = g \ x \equiv g \ y$ 

```

1011 or as a relation from A to B ,

```

1012    $\mathbf{Rel} : \mathcal{U} \cdot \rightarrow (\mathcal{N} : \mathbf{Universe}) \rightarrow \mathcal{U} \sqcup \mathcal{N}^+ \cdot$ 
1013    $\mathbf{Rel} \ A \ \mathcal{N} = \mathbf{REL} \ A \ A \ \mathcal{N}$ 

```

```

1014    $\mathbf{KER-rel} : \{ \mathcal{R} : \mathbf{Universe} \} \{ A : \mathcal{U} \cdot \} \{ B : \mathcal{R} \cdot \} \rightarrow (A \rightarrow B) \rightarrow \mathbf{Rel} \ A \ \mathcal{R}$ 
1015    $\mathbf{KER-rel} \ g \ x \ y = g \ x \equiv g \ y$ 

```

1021 A.11 Equivalence Relations

1022 Types for equivalence relations are defined in the `UALib.Relations.Equivalences` module of the
 1023 Agda UALib using a `record` type, as follows:

```

1024
1025   record  $\mathbf{IsEquivalence} \ \{ A : \mathcal{U} \cdot \} \ (\_ \approx \_ : \mathbf{Rel} \ A \ \mathcal{R}) : \mathcal{U} \sqcup \mathcal{R} \cdot$  where  

1026     field

```

\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 [5, §4.1.2].)

```

1027   rfl  : reflexive _≈_
1028   sym  : symmetric _≈_
1029   trans : transitive _≈_

```

1030 For example, here is how we construct an equivalence relation out of the kernel of a function.
 1031
 1032

```

1033   map-kernel-IsEquivalence : {W : Universe}{A : U .}{B : W .}
1034                               (f : A → B) → IsEquivalence (KER-rel f)
1035
1036   map-kernel-IsEquivalence {W} f =
1037     record { rfl = λ x → refl
1038               ; sym = λ x y x1 → ≡-sym{W} (f x) (f y) x1
1039               ; trans = λ x y z x1 x2 → ≡-trans (f x) (f y) (f z) x1 x2 }

```

1040 A.12 Relation truncation

1041 Here we discuss a special technical issue that will arise when working with quotients, specifically
 1042 when we must determine whether two equivalence classes are equal. Given a binary relation¹⁶
 1043 P , it may be necessary or desirable to assume that there is at most one way to prove that a
 1044 given pair of elements is P -related. This is an example of *proof-irrelevance*; indeed, under this
 1045 assumption, proofs of $P\ x\ y$ are indistinguishable, or rather distinctions are irrelevant in given
 1046 context.

1047 In the UALib, the `is-subsingleton` type of Type Topology is used to express the assertion
 1048 that a given type is a set, or *0-truncated*. Above we defined truncation for a type with an
 1049 identity relation, but the general principle can be applied to arbitrary binary relations. Indeed,
 1050 we say that P is a *0-truncated binary relation* on X if for all $x\ y : X$ we have `is-subsingleton` $(P\ x\ y)$.
 1051

```

1052   Rel0 : U . → (N : Universe) → U ⊔ N+ .
1053   Rel0 A N = Σ P : (A → A → N .) , ∀ x y → is-subsingleton (P x y)

```

1054 Thus, a *set*, as defined in §A.5, is a type X along with an equality relation \equiv of type `Rel0 X N`,
 1055 for some N .
 1056

¹⁶ Binary relations, as represented in Agda in general and the UALib in particular, are described in §A.9.