The Agda UALib and Birkhoff's Theorem in Martin-Löf 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 dependent type theory using the Agda programming language and proof assistant. This paper describes the UALib and demonstrates how it makes Agda more accessible to working mathematicians (like ourselves) as a tool for formally verifying "known" results in general algebra and related fields, as well as for discovering new theorems in these areas. The library already includes a substantial collection of definitions, theorems, and proofs from universal algebra and as such provides many examples that exhibit the power of inductive and dependent types for representing and reasoning about 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 UALib and the formal proof of Birkhoff's theorem, discussing some of the challenges we faced and how these hurdles were overcome. In so doing, we illustrate the effectiveness of dependent type theory, Agda, and the UALib for proving and verifying theorems in universal algebra.

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

Supplementary Material
Documentation: `https://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 Martin Hötzel Escardo 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.

Contents

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 Agda UALib is written in Agda [6], 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 handful of other successful efforts to formalize parts of universal algebra in type theory, 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 [7] (2011) formalized the basics of universal algebra and some classical algebraic structures, also in the Calculus of Inductive Constructions using the Coq proof assistant, and promoted the use of type classes as a preferable alternative to setoids.
- Gunther, et al [5] developed what seems to have been the most extensive library formalizing universal algebra to date, including the formalization of some equational logic; this project (like the UALib) uses Martin-Löf dependent 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, fields, and modules. However, the goals of these other projects do not seem to be a formal library of definitions, theorems, and proofs about *general* (universal) algebra.

Although the UALib project was initiated relatively recently (by the author of this paper, with valuable contributions from Siva Somayyajula), the part of universal algebra and equational logic that we formalize already extends beyond the scope of prior efforts such as those mentioned above. In particular, the UALib includes the only formal, constructive, machine-checked proof of Birkhoff's variety theorem that we know of. We remark that, with the exception of [4], all other proofs of Birkhoff's theorem are informal and not known to be constructive.

The seminal idea for this project was the observation that, on the one hand, a number of basic and important constructs in universal algebra can be defined recursively, and theorems about them have easy proofs by structural induction, while, on the other hand, inductive types (of dependent type theory) make possible very precise formal representations of recursively defined objects, and often yield very short, elegant and 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. These observations suggest that much can be gained from implementing universal algebra in a language, like Martin-Löf type theory, that supports dependent and inductive types.

1.1 Objectives

The first goal of the project is to express the foundations of universal algebra constructively, in type theory, and to formally verify the foundations using the Agda proof assistant. Thus we aim to codify the edifice upon which our mathematical research stands, and demonstrate that advancements in our field can be expressed in type theory and formally verified in a way that we, and other working mathematicians, can easily understand and check the results. We hope the library inspires and encourages others to formalize and verify their own mathematics research so that we may more easily understand and verify their results.

Our field is deep and broad, so codifying all of its foundations may seem like a daunting task and a risky investment of time and resources. However, we believe the subject is well served by a new, modern, *constructive* presentation of its foundations. Finally, the mere act of reinterpreting the foundations in an alternative system offers a fresh perspective, and this often leads to deeper insights and new discoveries.

Indeed, we wish to emphasize that our ultimate objective is not merely to translate existing results into a new more modern and formal language. Rather, an important goal of the UALib project is a system that is useful for conducting research in mathematics, and that is how we intend to use our library now that we have achieved our initial objective of implementing a substantial part of the foundations of universal algebra in Agda.

In our own mathematics research, experience has taught us that a proof assistant equipped with specialized libraries for universal algebra, as well as domain-specific tactics to automate proof idioms of our field, would be extremely valuable and powerful tool. Thus, we aim to build a library that serves as an indispensable part of our research tool kit.

1.2 Contributions

Apart from the library itself, we describes the formal implementation and proof of a deep result in universal algebra, which was among the first major results of our subject—namely, Garrett Birkhoff's celebrated HSP Theorem [2]. This theorem says that a *variety* (a class of algebras closed under quotients, subalgebras, and products) is an equational class. More precisely, a class $\mathscr{K}$ of algebras is closed under the taking of quotients, subalgebras, and arbitrary products if and only if $\mathscr{K}$ is the class of all algebras that satisfy some set of equations.

The fact that we now have a proof of Birkhoff's Theorem in Agda 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 our proof is *constructive*. Judging from the paper [4] by Carlström, for example, it is evidently a nontrivial matter to take a well-known informal proof of a theorem like Birkhoff's (as presented in, e.g., [1]) and show that it can be formalized using only constructive logical assumptions—that is, without appealing to the Law of the Excluded Middle and (therefore) without the Axiom of Choice.

2 Prelude

2.1 Preliminaries

This section describes the UALib.Prelude.Preliminaries module of the Agda UALib.
Notation. Here are some acronyms that we use frequently.

- MHE = Martin Hötzel Escardo
- MLTT = Martin-Löf Type Theory

2.1.1 Options

All Agda programs begin by setting some options and by importing from existing libraries (in our case, the Agda Standard Library and the Type Topology library by MHE). In particular, logical axioms and deduction rules can be specified according to what one wishes to assume.

For example, each Agda source code file in the UALib begins with the following line:

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

These options control certain foundational assumptions that Agda assumes when type-checking the program to verify its correctness.[1]

- `without-K` disables Streicher's K axiom; see also the in the section on axiom K in the Agda Language Reference.
- `exact-split` makes Agda accept only those definitions that behave like so-called *judgmental* or *definitional* equalities. MHE explains this by saying it "makes sure that pattern matching corresponds to Martin-Löf eliminators;" see also the Pattern matching and equality section of the Agda Tools documentation.
- `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 also this section of the Agda Tools documentation and the Safe Agda section of the Agda Language Reference.

2.1.2 Modules

The OPTIONS directive is followed by some imports or the start of a module. For example, the Preliminaries module begins with `module UALib.Prelude.Preliminaries where`.

Sometimes we want to pass in parameters that will be assumed throughout the module. For instance, when working with algebras we often assume they come from a particular fixed signature S, which we could fix as a parameter at the start of a module. We'll see many examples later, but here's an example: `module _ {𝑆 : Signature 𝓞 𝓥} where`.

2.1.3 Imports from Type Topology

Throughout we use many of the nice tools that MHE has developed and made available in the Type Topology repository of Agda code for the "Univalent Foundations" of mathematics. We import these now.

```
open import universes public
```

[1] *Useful Tip*. To type-check a file that imports another file when the latter still has some unmet proof obligations, one must replace the `--safe` flag with `--allow-unsolved-metas`; i.e., use `{-# OPTIONS --without-K --exact-split --safe #-}`, but this is never done in publicly released versions of the UALib.

```
open import Identity-Type renaming (_≡_ to infix 0 _≡_ ; refl to 𝓇ℯ𝒻ℓ) public

pattern refl x = 𝓇ℯ𝒻ℓ {x = x}

open import Sigma-Type renaming (_,_ to infixr 50 _,_) public

open import MGS-MLTT using (_∘_; domain; codomain; transport; _≡⟨_⟩_; _■;
  pr₁; pr₂; -Σ; 𝕁; Π; ¬; _×_; id; _∼_; _+_; 𝟘; 𝟙; 𝟚; _⇔_;
  lr-implication; rl-implication; id; _⁻¹; ap) public

open import MGS-Equivalences using (is-equiv; inverse; invertible) public

open import MGS-Subsingleton-Theorems using (funext; global-hfunext; dfunext;
  is-singleton; is-subsingleton; is-prop; Univalence; global-dfunext;
  univalence-gives-global-dfunext; _●_; _≃_; Π-is-subsingleton; Σ-is-subsingleton;
  logically-equivalent-subsingletons-are-equivalent) public

open import MGS-Powerset
  renaming (_∈_ to _∈₀_; _⊆_ to _⊆₀_; ∈-is-subsingleton to ∈₀-is-subsingleton)
  using (𝓟; equiv-to-subsingleton; powersets-are-sets'; subset-extensionality'; propext; _holds; Ω) public

open import MGS-Embeddings
  using (Nat; NatΠ; NatΠ-is-embedding; is-embedding; pr₁-embedding; ∘-embedding; is-set;
  _↪_; embedding-gives-ap-is-equiv; embeddings-are-lc; ×-is-subsingleton; id-is-embedding) public

open import MGS-Solved-Exercises using (to-subtype-≡) public

open import MGS-Unique-Existence using (∃!; -∃!) public

open import MGS-Subsingleton-Truncation using (_·_; to-Σ-≡; equivs-are-embeddings;
  invertibles-are-equivs; fiber; ⊆-refl-consequence; hfunext) public
```

Notice that we carefully specify which definitions and theorems we want to import from each module. This is not absolutely necessary, but it helps us avoid name clashes and, more importantly, makes explicit on which components of the type theory our development depends.

2.1.4 Special notation for Agda universes

The first import in the list of open import directives above imports the universes module from Martin Escardo's Type Topology library. This provides, among other things, an elegant notation for type universes that we have fully adopted and we use it throughout the Agda UALib.[2]

Following MHE, we refer to universes using capitalized script letters from near the end of the alphabet, e.g., $\mathcal{U}$, $\mathcal{V}$, $\mathcal{W}$, $\mathcal{X}$, $\mathcal{Y}$, $\mathcal{Z}$, etc. Also, in the universes module MHE defines the $^{\cdot}$ operator which maps a universe $\mathcal{U}$ (i.e., a level) to Set $\mathcal{U}$, and the latter has type Set (lsuc $\mathcal{U}$). The level lzero is renamed $\mathcal{U}_0$, so $\mathcal{U}_0{}^{\cdot}$ is an alias for Set lzero.[3]

Thus, $\mathcal{U}\ ^{\cdot}$ is simply an alias for Set $\mathcal{U}$, and we have Set $\mathcal{U}$: Set (lsuc $\mathcal{U}$). Finally, Set (lsuc lzero) is denoted by Set $\mathcal{U}_0\ ^{+}$ which we (and MHE) denote by $\mathcal{U}_0\ ^{+}\ ^{\cdot}$.

Table 1 translates between standard Agda syntax and MHE/UALib notation.

---

[2] We won't discuss every line of the universes module of the Type Topology library. Instead we merely touch upon the few lines of code that define the notational devices we adopt throughout the UALib. For those who wish for more details, MHE has made available an excellent set of notes from his course, MGS 2019. We highly recommend Martin's notes to anyone who wants more information than we provide here about Martin-Löf Type Theory and the Univalent Foundations/HoTT extensions thereof.

[3] Incidentally, Set lzero corresponds to `Sort 0` in the Lean proof assistant language.

Standard Agda	MHE/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	$\mathcal{U}_0$
Set lzero	$\mathcal{U}_0$ ̇
lsuc lzero	$\mathcal{U}_0{}^{+}$
Set (lsuc lzero)	$\mathcal{U}_0{}^{+}$ ̇
Setω	$\mathcal{U}\omega$

Table 1 Special notation for universe levels

To justify the introduction of this somewhat nonstandard notation for universe levels, MHE 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.

There will be many occasions calling for a type living in the universe that is the least upper bound of two universes, say, $\mathcal{U}$ ̇ and $\mathcal{V}$ ̇ . The universe ($\mathcal{U} \sqcup \mathcal{V}$) ̇ denotes this least upper bound.[4] Here $\mathcal{U} \sqcup \mathcal{V}$ is used to denote the universe level corresponding to the least upper bound of the levels $\mathcal{U}$ and $\mathcal{V}$, where the _⊔_ is an Agda primitive designed for precisely this purpose.

2.1.5 Dependent pair type

Dependent pair types (or *Sigma types*) are defined in the Type Topology library as a record, as follows:

```
record Σ {𝓤 𝓥} {X : 𝓤 ̇ } (Y : X → 𝓥 ̇ ) : 𝓤 ⊔ 𝓥 ̇ where
  constructor
   _,_
  field
   pr₁ : X
   pr₂ : Y pr₁
infixr 50 _,_
```

We prefer the notation Σ x ꞉ X , y, which is closer to the standard syntax appearing in the literature than Agda's default syntax (Σ λ(x ꞉ X) → y). MHE makes the preferred notation available by making the index type explicit, as follows.

```
-Σ : {𝓤 𝓥 : Universe} (X : 𝓤 ̇ ) (Y : X → 𝓥 ̇ ) → 𝓤 ⊔ 𝓥 ̇
-Σ X Y = Σ Y
syntax -Σ X (λ x → y) = Σ x ꞉ X , y
infixr -1 -Σ
```

N.B. The symbol ꞉ used here is not the same as the ordinary colon symbol (:), despite how similar they may appear. The symbol in the expression Σ x ꞉ X , y above is obtained by typing `\:4` in agda2-mode.

[4] Actually, because _⊔_ has higher precedence than _ ̇, we could omit parentheses here and simply write $\mathcal{U} \sqcup \mathcal{V}$ ̇.

Our preferred notations for the first and second projections of a product are |_| and ||_||, respectively; however, we sometimes use more standard alternatives, such as pr_1 and pr_2, or fst and snd, or some combination of these, to improve readability, or to avoid notation clashes with other modules.

```
module _ {𝓤 : Universe} where
  |_| fst : {X : 𝓤 ̇ }{Y : X → 𝓥 ̇} → Σ Y → X
  | x , y | = x
  fst (x , y) = x
  ||_|| snd : {X : 𝓤 ̇ }{Y : X → 𝓥 ̇ } → (z : Σ Y) → Y (pr₁ z)
  || x , y || = y
  snd (x , y) = y
```

2.1.6 Dependent function type

The so-called **dependent function type** (or "Pi type") is defined in the Type Topology library as follows.

```
Π : {X : 𝓤 ̇ } (A : X → 𝓥 ̇ ) → 𝓤 ⊔ 𝓥 ̇
Π {𝓤} {𝓥} {X} A = (x : X) → A x
```

To make the syntax for Π conform to the standard notation for Pi types, MHE uses the same trick as the one used above for Sigma types.

```
-Π : {𝓤 𝓥 : Universe} (X : 𝓤 ̇ ) (Y : X → 𝓥 ̇ ) → 𝓤 ⊔ 𝓥 ̇
-Π X Y = Π Y
syntax -Π A (λ x → b) = Π x ꞉ A , b
infixr -1 -Π
```

2.1.7 Truncation and sets

In general, we may have many inhabitants of a given type and, via the 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. Then, given two inhabitants a and b of type X, we may ask whether $a \equiv_x b$.

Suppose p and q inhabit the identity 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 $r : p \equiv_{x1} q$ that the proof of $a \equiv_x b$ is unique. (This property is sometimes called *uniqueness of identity proofs*.)

Perhaps we have two proofs, say, $r\ \ s : p \equiv_{x1} q$. Then of course our next question will be whether $r \equiv_{x2} s$ has a proof! But at some level we may decide that the potential to distinguish two proofs of an identity in a meaningful way (so-called *proof relevance*) is not useful or desirable. 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.

We will see some examples of trunction later when we require it to complete some of the UALib modules leading up to the proof of Birkhoff's HSP Theorem. Readers who want more details should refer to Section 34 and 35 of MHE's notes, or Guillaume Brunerie, Truncations and truncated higher inductive types, or Section 7.1 of the HoTT book.

We take this opportunity to say what it means in type theory to say that a type is a *set*. A type X : $\mathcal{U}$ ̇ with an identity relation $\equiv_x$ is called a **set** (or 0-**groupoid**) if for every pair $a\ b$: X of elements of type X there is at most one proof of $a \equiv_x b$. This is formalized in the Type Topology library as follows:[5]

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

2.2 Equality

This section describes the UALib.Prelude.Equality module of the Agda UALib.

2.2.1 refl

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
```

We being the Prelude.Equality module by formalizing the proof that ≡ is an equivalence relation.

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

module UALib.Prelude.Equality where

open import UALib.Prelude.Preliminaries using (𝓞; 𝓥; Universe; _̇; _⊔_; _⁺; _≡_; refl; Σ; -Σ; _×_; _,_;
  is-subsingleton; is-prop; ∣_∣; ∥_∥; 𝟙; pr₁; pr₂; ap) public

module _ {𝓤 : Universe}{X : 𝓤 ̇ } where
  ≡-rfl : (x : X) → x ≡ x
  ≡-rfl x = refl _
  ≡-sym : (x y : X) → x ≡ y → y ≡ x
  ≡-sym x y (refl _) = refl _
  ≡-trans : (x y z : X) → x ≡ y → y ≡ z → x ≡ z
  ≡-trans x y z (refl _) (refl _) = refl _
  ≡-Trans : (x : X){y : X}(z : X) → x ≡ y → y ≡ z → x ≡ z
  ≡-Trans x {y} z (refl _) (refl _) = refl _
```

The only difference between `≡-trans` and `≡-Trans` is that the second argument to `≡-Trans` is implicit so we can omit it when applying `≡-Trans`. This is sometimes convenient; after all, `≡-Trans` is used to prove that the first and last arguments are the same, and often we don't care about the middle argument.

2.2.2 Functions preserve refl

A function is well defined only if it maps equivalent elements to a single element and we often use this nature of functions in Agda proofs. If we have a function $f : X \to Y$, two elements $x\ x' : X$ of the domain, and an identity proof $p : x \equiv x'$, then we obtain a proof of $f\ x \equiv f\ x'$ by simply applying the ap function like so, $\mathsf{ap}\ f\ p : f\ x \equiv f\ x'$.

[5] As MHE explains, “at this point, with the definition of these notions, we are entering the realm of univalent mathematics, but not yet needing the univalence axiom.”

MHE defines ap in the Type Topology library so we needn't redefine it here. Instead, we define two variations of ap that are sometimes useful.

```
ap-cong : {𝓤 𝓦 : Universe} {A : 𝓤 ̇} {B : 𝓦 ̇}
          {f g : A → B} {a b : A}
 →        f ≡ g → a ≡ b
          -----------------------
 →        f a ≡ g b
ap-cong (refl _) (refl _) = refl _
```

We sometimes need a version of ap-cong that works for dependent types, such as the following (which we borrow from the `Relation/Binary/Core.agda` module of the Agda Standard Library, transcribed into MHE/UALib notation of course).

```
cong-app : {𝓤 𝓦 : Universe} {A : 𝓤 ̇} {B : A → 𝓦 ̇}
           {f g : (a : A) → B a}
 →         f ≡ g → (a : A)
           --------------------
 →         f a ≡ g a
cong-app (refl _) a = refl _
```

2.2.3 ≡-intro and ≡-elim for pairs

We conclude the Equality module with some occasionally useful introduction and elimination rules for the equality relation on (nondependent) pair types.

```
≡-elim-left : {𝓤 𝓦 : Universe} {A₁ A₂ : 𝓤 ̇} {B₁ B₂ : 𝓦 ̇}
 →            (A₁ , B₁) ≡ (A₂ , B₂)
              ----------------------
 →            A₁ ≡ A₂
≡-elim-left e = ap pr₁ e

≡-elim-right : {𝓤 𝓦 : Universe} {A₁ A₂ : 𝓤 ̇}{B₁ B₂ : 𝓦 ̇}
 →             (A₁ , B₁) ≡ (A₂ , B₂)
               -----------------------
 →             B₁ ≡ B₂
≡-elim-right e = ap pr₂ e

≡-×-intro : {𝓤 𝓦 : Universe} {A₁ A₂ : 𝓤 ̇} {B₁ B₂ : 𝓦 ̇}
 →          A₁ ≡ A₂ → B₁ ≡ B₂
            ------------------------
 →          (A₁ , B₁) ≡ (A₂ , B₂)
≡-×-intro (refl _ ) (refl _ ) = (refl _ )

≡-×-int : {𝓤 𝓦 : Universe} {A : 𝓤 ̇} {B : 𝓦 ̇}
          (a a' : A)(b b' : B)
 →        a ≡ a' →  b ≡ b'
          ------------------------
 →        (a , b) ≡ (a' , b')
≡-×-int a a' b b' (refl _ ) (refl _ ) = (refl _ )
```

2.3 Inverses

This section presents the UALib.Prelude.Inverses module of the Agda UALib. Here we define (the syntax of) a type for the (semantic concept of) **inverse image** of a function.

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

module UALib.Prelude.Inverses where

open import UALib.Prelude.Equality public

open import UALib.Prelude.Preliminaries using (_⁻¹; funext; _∘_; _·_; id; fst; snd; is-set; is-embedding;
  transport; to-Σ-≡; invertible; equivs-are-embeddings; invertibles-are-equivs; fiber; 𝓇ℯ𝒻𝓁) public

module _ {𝓤 𝓦 : Universe} where

  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

  ImageIsImage : {A : 𝓤 ̇ }{B : 𝓦 ̇ }
                 (f : A → B) (b : B) (a : A)
   →             b ≡ f a
                 -------------------------
   →             Image f ∋ b
  ImageIsImage {A}{B} f b a b≡fa = eq b a b≡fa
```

Note that an inhabitant of Image f ∋ 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 ∋ b), always has a witness a : A, and a proof that $b \equiv f\ a$, so a (pseudo)inverse can actually be *computed*.

For convenience, we define a pseudo-inverse function, which we call Inv, that takes b : B and (a , p) : Image f ∋ b and returns a.

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

Of course, we can prove that Inv f is really the (right-)inverse of f.

```
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
```

2.3.1 Surjective functions

An epic (or surjective) function from type A : 𝓤 ̇ to type B : 𝓦 ̇ is as an inhabitant of the Epic type, which we define as follows.

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

We obtain the right-inverse (or pseudoinverse) of an epic function f by applying the function EpicInv (which we now define) to the function f along with a proof, *fE* : Epic *f*, that f is surjective.

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

The function defined by EpicInv *f fE* is indeed the right-inverse of *f*.

```
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))
```

2.3.2 Injective functions

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

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

Again, we obtain a pseudoinverse. Here it is obtained by applying the function MonicInv to g and a proof that g is monic.

```
--The (pseudo-)inverse of a monic function
MonicInv : {A : 𝓤 ̇ } {B : 𝓦 ̇ }
           (f : A → B) → Monic f
           -----------------------------
  →        (b : B) → Image f ∋ b → A

MonicInv f _ = λ b Imf∋b → Inv f b Imf∋b
```

The function defined by MonicInv f fM is the left-inverse of *f*.

```
--The (psudo-)inverse of a monic is the left inverse.
MonicInvIsLeftInv : {A : 𝓤 ̇ }{B : 𝓦 ̇ }
                    (f : A → B) (fmonic : Monic f)(x : A)
                    ----------------------------------------
  →                 (MonicInv f fmonic) (f x) (im x) ≡ x

MonicInvIsLeftInv f fmonic x = refl _
```

2.3.3 Bijective functions

Finally, bijective functions are defined.

```
Bijective : {A : 𝓤 ̇ }{B : 𝓦 ̇ }(f : A → B) → 𝓤 ⊔ 𝓦 ̇
Bijective f = Epic f × Monic f

Inverse : {A : 𝓤 ̇ } {B : 𝓦 ̇ }
```

```
           (f : A → B) → Bijective f
           -------------------------
  →        B → A
Inverse f fbi b = Inv f b (fst( fbi ) b)
```

2.3.4 Injective functions are set embeddings

This is the first point at which truncation comes into play. An embedding is defined in the Type Topology library as follows:

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

where

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

and

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

This is a natural way to represent what we usually mean in mathematics by embedding. It does not correspond simply to an injective map. However, if we assume that the codomain type, B, is a *set* (i.e., has *unique identity proofs*), then we can prove that a injective (i.e., *monic*) function into B is an embedding as follows:

```
module _ {𝓤 𝓦 : Universe} where

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

 monic-into-set-is-embedding {A} Bset f fmon b (a , fa≡b) (a' , fa'≡b) = γ
  where
   faa' : f a ≡ f a'
   faa' = ≡-Trans (f a) (f a') fa≡b (fa'≡b ⁻¹)

   aa' : a ≡ a'
   aa' = fmon a a' faa'

   𝒜 : A → 𝓦 ̇
   𝒜 a = f a ≡ b

   arg1 : Σ p ꞉ (a ≡ a') , (transport 𝒜 p fa≡b) ≡ fa'≡b
   arg1 = aa' , Bset (f a') b (transport 𝒜 aa' fa≡b) fa'≡b

   γ : a , fa≡b ≡ a' , fa'≡b
   γ = to-Σ-≡ arg1
```

Of course, invertible maps are embeddings.

```
invertibles-are-embeddings : {X : 𝓤 ̇ } {Y : 𝓦 ̇ }(f : X → Y)
  →                          invertible f → is-embedding f
invertibles-are-embeddings f fi = equivs-are-embeddings f (invertibles-are-equivs f fi)
```

Finally, if we have a proof `p : is-embedding f` that the map `f` is an embedding, here's a tool that makes it easier to apply `p`.

```
-- Embedding elimination (makes it easier to apply is-embedding)
embedding-elim : {X : 𝓤 ̇ } {Y : 𝓦 ̇ }{f : X → Y}
  →              is-embedding → (x x' : X)
                 ----------------------
  →              f x ≡ f x' → x ≡ x'
embedding-elim {f = f} femb x x' fxfx' = γ
  where
    fibx : fiber f (f x)
    fibx = x , 𝓻ℯ𝒻ℓ
    fibx' : fiber f (f x)
    fibx' = x' , ((fxfx') ⁻¹)
    iss-fibffx : is-subsingleton (fiber f (f x))
    iss-fibffx = femb (f x)
    fibxfibx' : fibx ≡ fibx'
    fibxfibx' = iss-fibffx fibx fibx'
    γ : x ≡ x'
    γ = ap pr₁ fibxfibx'
```

2.4 Extensionality

This section describes the UALib.Prelude.Extensionality module of the Agda UALib.

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

module UALib.Prelude.Extensionality where

open import UALib.Prelude.Inverses public

open import UALib.Prelude.Preliminaries
  using (_∼_; 𝓤ω; Π; Ω; 𝓟; ⊆-refl-consequence; _∈₀_; _⊆₀_; _holds) public
```

2.4.1 Function extensionality

Extensional equality of functions, or function extensionality, means that any two point-wise equal functions are equal. As MHE points out, this is known to be not provable or disprovable in Martin-Löf type theory. It is an independent statement, which MHE abbreviates as `funext`. Here is how this notion is given a type in the Type Topology library

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

For readability we occasionally use the following alias for the `funext` type.

```
extensionality : ∀ 𝓤 𝓦 → 𝓤 ⁺ ⊔ 𝓦 ⁺ ̇
extensionality 𝓤 𝓦 = {A : 𝓤 ̇ } {B : 𝓦 ̇ } {f g : A → B} → f ∼ g → f ≡ g
```

Pointwise equality of functions is typically what one means in informal settings when one says that two functions are equal. Here is how MHE defines pointwise equality of (dependent) function in Type Topology.

```
_∼_ : {X : 𝓤 ̇ } {A : X → 𝓥 ̇ } → Π A → Π A → 𝓤 ⊔ 𝓥 ̇
f ∼ g = ∀ x → f x ≡ g x
```

In fact, if one assumes the univalence axiom, then the _∼_ relation is equivalent to equality of functions. See Function extensionality from univalence.

2.4.2 Dependent function extensionality

Extensionality for dependent function types is defined as follows.

```
dep-extensionality : ∀ 𝓤 𝓦 → 𝓤 ⁺ ⊔ 𝓦 ⁺ ̇
dep-extensionality 𝓤 𝓦 = {A : 𝓤 ̇ } {B : A → 𝓦 ̇ }
  {f g : ∀(x : A) → B x} → f ∼ g → f ≡ g
```

Sometimes we need extensionality principles that work at all universe levels, and Agda is capable of expressing such principles, which belong to the special $\mathcal{U}\omega$ type, as follows:

```
∀-extensionality : 𝓤ω
∀-extensionality = ∀ {𝓤 𝓥} → extensionality 𝓤 𝓥

∀-dep-extensionality : 𝓤ω
∀-dep-extensionality = ∀ {𝓤 𝓥} → dep-extensionality 𝓤 𝓥
```

More details about the $\mathcal{U}\omega$ type are available at agda.readthedocs.io.

```
extensionality-lemma : ∀ {𝓘 𝓤 𝓥 𝓣} →
                       {I : 𝓘 ̇ }{X : 𝓤 ̇ }{A : I → 𝓥 ̇ }
                       (p q : (i : I) → (X → A i) → 𝓣 ̇ )
                       (args : X → (Π A))
  →                    p ≡ q
                       ---------------------------------------------------
  →                    (λ i → (p i)(λ x → args x i)) ≡ (λ i → (q i)(λ x → args x i))

extensionality-lemma p q args p≡q =
  ap (λ - → λ i → (- i) (λ x → args x i)) p≡q
```

2.4.3 Function intensionality

This is the opposite of function extensionality and is defined as follows.

```
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 an analogue for dependent function types.

```
dep-intensionality -- alias (we sometimes give multiple names to a function, like this)
  dintensionality : {𝓤 𝓦 : Universe} {A : 𝓤 ̇ } {B : A → 𝓦 ̇ } {f g : (x : A) → B x}
  →                  f ≡ g → (x : A)
                     ------------------
  →                  f x ≡ g x

dintensionality (refl _) _ = refl _
dep-intensionality = dintensionality
```

3 Algebras

3.1 Operation and Signature Types

This section presents the UALib.Algebras.Signatures module of the Agda UALib.

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

open import universes using (𝓤₀)

module UALib.Algebras.Signatures where

open import UALib.Prelude.Extensionality public
open import UALib.Prelude.Preliminaries using (𝟘; 𝟚) public
```

3.1.1 Operation type

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

```
module _ {𝓤 𝓥 : Universe} where

  --The type of operations
  Op : 𝓥 ̇ → 𝓤 ̇ → 𝓤 ⊔ 𝓥 ̇
  Op I A = (I → A) → A

  --Example. the projections
  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 $I : \mathcal{V}$ ̇, 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.

3.1.2 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 𝓞 is the universe level of operation symbol types, while 𝓥 is the universe level of arity types.

In the UALib.Prelude module we define special syntax for the first and second projections—namely, ∣_∣ and ∥_∥, resp (see Subsection 2.1.5). Consequently, if {S : Signature 𝓞 𝓥 } 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.

3.1.3 Example

Here is how we might define the signature for *monoids* as a member of the type Signature 𝓞 𝓥.

```
module _ {𝓞 : Universe} where

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

```
    · : monoid-op

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

As expected, the signature for a *monoid* consists of two operation symbols, e and ·, and a function λ { e → 𝟘 ; · → 𝟚 } which maps e to the empty type 𝟘 (since e is the nullary identity) and maps · to the two element type 𝟚 (since · is binary).

3.2 Algebra Types

This section presents the UALib.Algebras.Algebras module of the Agda UALib.

```
{-# OPTIONS --without-K --exact-split --safe #-}
module UALib.Algebras.Algebras where
open import UALib.Algebras.Signatures public
open import UALib.Prelude.Preliminaries using (𝓤₀; 𝟘; 𝟚) public
```

3.2.1 Algebra types

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

```
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; in particular, A need to be a *set*. (See the discussion in Section 2.1.7.)

We take this opportunity to define the type of "0-algebras" which are algebras whose domains are *sets* (as defined in Section 2.1.7). This type is probably closer to what most of us think of when doing informal universal algebra. However, in the UALib will have so far only needed to know that a 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 wherever a proof relies on this assumption.

The type Algebra 𝓤 S itself has a type; it is 𝓞 ⊔ 𝓥 ⊔ 𝓤 ⁺ ̇. This type appears so often in the UALib that we will define the following shorthand for its universe level: OV 𝓤 = 𝓞 ⊔ 𝓥 ⊔ 𝓤 ⁺ ̇.

3.2.2 Algebras as record types

Sometimes records are more convenient than sigma types. For such cases, we might prefer the following representation of the type of algebras.

```
module _ {𝓞 𝓥 : Universe} where
  record algebra (𝓤 : Universe) (S : Signature 𝓞 𝓥) : (𝓞 ⊔ 𝓥 ⊔ 𝓤) ⁺ ̇ where
    constructor mkalg
    field
      univ : 𝓤 ̇
      op : (f : ∣ S ∣) → ((∥ S ∥ f) → univ) → univ
```

Of course, we can go back and forth between the two representations of algebras, like so.

```
module _ {𝓤 𝓞 𝓥 : Universe} {𝑆 : Signature 𝓞 𝓥} where

 open algebra

 algebra→Algebra : algebra 𝓤 𝑆 → Algebra 𝓤 𝑆
 algebra→Algebra 𝑨 = (univ 𝑨 , op 𝑨)

 Algebra→algebra : Algebra 𝓤 𝑆 → algebra 𝓤 𝑆
 Algebra→algebra 𝑨 = mkalg ∣ 𝑨 ∣ ∥ 𝑨 ∥
```

3.2.3 Operation interpretation syntax

We conclude this module by defining a convenient shorthand for the interpretation of an operation symbol that we will use often.

```
_̂_ : (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.

3.2.4 Arbitrarily many variable symbols

Finally, we will want to 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 $h_0 : X \to |\,\mathbf{A}\,|$ 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
```

3.3 Product Algebra Types

This section presents the UALib.Algebras.Products module of the Agda UALib. We define products of algebras for both the Sigma type representation (the one we use most often) and the record type representation.

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

module UALib.Algebras.Products where

open import UALib.Algebras.Algebras public

module _ {𝓤 : Universe}{𝑆 : Signature 𝓞 𝓥} where
 -- product for algebras of sigma type
 ⨅ : {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}

 ⊓ : {I : 𝓤 ̇ }(𝒜 : I → Algebra 𝓤 𝑆 ) → Algebra 𝓤 𝑆
 ⊓ = ⨅ {𝓤}
```

```
open algebra

-- product for algebras of record type
⊓' : {𝓘 : Universe}{I : 𝓘 ̇ }(𝒜 : I → algebra 𝓤 S) → algebra (𝓘 ⊔ 𝓤) S
⊓' {𝓘}{I} 𝒜 = record
                { univ = (i : I) → univ (𝒜 i)
                ; op = λ(f : ∣ S ∣)
                       (a : ∥ S ∥ f → (j : I) → univ(𝒜 j))
                       (i : I) → ((op (𝒜 i)) f)
                       λ{x → a x i}
                }
```

3.3.1 The Universe Hierarchy and Lifts

This section presents the Algebras.Lifts module of the Agda UALib.

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

module UALib.Algebras.Lifts where

open import UALib.Algebras.Products public
```

3.3.2 The noncumulative hierarchy

The hierarchy of universe levels in Agda is structured as $\mathcal{U}_0 : \mathcal{U}_1$, $\mathcal{U}_1 : \mathcal{U}_2$, $\mathcal{U}_2 : \mathcal{U}_3$, …. This means that $\mathcal{U}_0$ has type $\mathcal{U}_1$ ̇ and $\mathcal{U}_n$ has type $\mathcal{U}_{n+1}$ ̇ 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.

Luckily, there are ways to subvert noncummulativity which, when used with care, do not introduce logically consistencies into the type theory. We describe some techniques we developed for this purpose that are specifically tailored for our domain of applications.

3.3.3 Lifting and lowering

Let us be more concrete about what is at issue here by giving an example. Unless we are pedantic enough to worry about which universe level each of our types inhabits, then eventually we will encounter an error like the following:

```
Birkhoff.lagda:498,20-23
(𝓤 ⁺) != (𝓞 ⁺) ⊔ (𝓥 ⁺) ⊔ ((𝓤 ⁺) ⁺)
when checking that the expression SP𝒦 has type
Pred (Σ (λ A → (f₁ : ∣ S ∣) → Op (∥ S ∥ f₁) A)) _𝓦 _2346
```

Just to confirm we all know the meaning of such errors, let's translate the one above. This error means that Agda encountered a type at universe level $\mathcal{U}^{+}$, on line 498 (in columns 20–23) of the file Birkhoff.lagda, but was expecting a type at level $\mathcal{O}^{+} \sqcup \mathcal{V}^{+} \sqcup \mathcal{U}^{++}$ instead.

To make these situations easier to deal with, the UALib offers some domain specific tools for the *lifting* and *lowering* of universe levels of the main types in the library. In particular, we have functions that will lift or lower the universes of algebra types, homomorphisms, subalgebras, and products.

Of course, messing with the universe level of a type must be done carefully to avoid making the type theory inconsistent. In particular, a necessary condition is that a type of a given universe level may not be converted to a type of lower universe level *unless the given type was obtained from lifting another type to a higher-than-necessary universe level*. If this is not clear, don't worry; just press on and soon enough there will be examples that make it clear.

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

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

Next, we give various ways to lift function types.

```
lift-dom : {𝓧 𝓨 𝓦 : Universe}{X : 𝓧 ̇}{Y : 𝓨 ̇} → (X → Y) → (Lift{𝓧}{𝓦} X → Y)
lift-dom f = λ x → (f (lower x))

lift-cod : {𝓧 𝓨 𝓦 : Universe}{X : 𝓧 ̇}{Y : 𝓨 ̇} → (X → Y) → (X → Lift{𝓨}{𝓦} Y)
lift-cod f = λ x → lift (f x)

lift-fun : {𝓧 𝓨 𝓦 𝓩 : Universe}{X : 𝓧 ̇}{Y : 𝓨 ̇} → (X → Y) → (Lift{𝓧}{𝓦} X → Lift{𝓨}{𝓩} Y)
lift-fun f = λ x → lift (f (lower x))
```

We will also need 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 _

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

Now, to be more domain-specific, we show how to lift algebraic operation types and then, finally, how to lift algebra types.

```
module _ {S : Σ F ꞉ 𝓞 ̇ , ( F → 𝓥 ̇)} where

  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)))

  open algebra

  lift-alg-record-type : {𝓤 : Universe} → algebra 𝓤 S → (𝓦 : Universe) → algebra (𝓤 ⊔ 𝓦) S
  lift-alg-record-type 𝑨 𝓦 = mkalg (Lift (univ 𝑨)) (λ (f : ∣ S ∣) → lift-op ((op 𝑨) f) 𝓦)

  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
```

(This page is almost entirely blank on purpose.)

4 Relations

4.1 Predicates

This section presents the UALib.Relations.Unary module of the Agda UALib. We need a mechanism for implementing the notion of subsets in Agda. A typical one is called Pred (for predicate). More generally, Pred A $\mathcal{U}$ can be viewed as the type of a property that elements of type A might satisfy. We write P : Pred A $\mathcal{U}$ to represent the semantic concept of a collection of elements of type A that satisfy the property P.

```
{-# OPTIONS --without-K --exact-split --safe #-}
module UALib.Relations.Unary where
open import UALib.Algebras.Lifts public
open import UALib.Prelude.Preliminaries using (¬; propext; global-dfunext ) public
```

Here is the definition, which is similar to the one found in the*Relation/Unary.agda* file of the Agda Standard Library.

```
module _ {𝓤 : Universe} where

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

4.1.1 Unary relation truncation

Section 2.1.7 describes the concepts of *truncation* and *set* for "proof-relevant" mathematics. Sometimes we will want to assume "proof-irrelevance" and assume that certain types are sets. Recall, this mean there is at most one proof that two elements are the same. For predicates, analogously, we may wish to assume that there is at most one proof that a given element satisfies the predicate.

```
Pred₀ : 𝓤 ̇ → (𝓥 : Universe) → 𝓤 ⊔ 𝓥 ⁺ ̇
Pred₀ A 𝓥 = Σ P ꞉ (A → 𝓥 ̇) , ∀ x → is-subsingleton (P x)
```

Below we will often consider predicates over the class of all algebras of a particular type. We will define the type of algebras Algebra $\mathcal{U}$ S (for some universe level $\mathcal{U}$). Like all types, Algebra $\mathcal{U}$ S itself has a type which happens to be $\mathcal{O} \sqcup \mathcal{V} \sqcup \mathcal{U}^{+}\,\dot{}$ (see Section 3.2). Therefore, the type of Pred (Algebra $\mathcal{U}$ S)$\mathcal{U}$ is $\mathcal{O} \sqcup \mathcal{V} \sqcup \mathcal{U}^{+}\,\dot{}$ as well.

The inhabitants of the type Pred (Algebra $\mathcal{U}$ S) $\mathcal{U}$ are maps of the form $\mathbf{A} \to \mathcal{U}\,\dot{}$; given an algebra $\mathbf{A}$: Algebra $\mathcal{U}$ S, we have Pred $\mathbf{A}$ $\mathcal{U}$ = $\mathbf{A} \to \mathcal{U}\,\dot{}$.

4.1.2 The membership relation

We introduce notation for denoting that x "belongs to" or "inhabits" type P, or that x "has property" P, by writing either $x \in P$ or $P\ x$ (cf. `Relation/Unary.agda` in the Agda Standard Library.

```
module _ {𝓤 𝓦 : Universe} where

  _∈_ : {A : 𝓤 ̇ } → A → Pred A 𝓦 → 𝓦 ̇
  x ∈ P = P x

  _∉_ : {A : 𝓤 ̇ } → A → Pred A 𝓦 → 𝓦 ̇
  x ∉ P = ¬ (x ∈ P)
```

```
  infix 4 _∈_ _∉_
```

The "subset" relation is denoted, as usual, with the ⊆ symbol (cf. `Relation/Unary.agda` in the Agda Standard Library.

```
_⊆_ : {𝓤 𝓦 𝓣 : Universe}{A : 𝓤 ̇ } → Pred A 𝓦 → Pred A 𝓣 → 𝓤 ⊔ 𝓦 ⊔ 𝓣 ̇
P ⊆ Q = ∀ {x} → x ∈ P → x ∈ Q

_⊇_ : {𝓤 𝓦 𝓣 : Universe}{A : 𝓤 ̇ } → Pred A 𝓦 → Pred A 𝓣 → 𝓤 ⊔ 𝓦 ⊔ 𝓣 ̇
P ⊇ Q = Q ⊆ P

infix 4 _⊆_ _⊇_
```

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 the in informal (pencil-paper) mathematics. In the Agda UALib this *subset equality* is denoted by =̇ and define as follows.

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

4.1.3 Predicates toolbox

Here is a small collection of tools that will come in handy later. Hopefully the meaning of each is self-explanatory.

```
_∈∈_ : {𝓤 𝓦 𝓣 : Universe}{A : 𝓤 ̇ } {B : 𝓦 ̇ } → (A → B) → Pred B 𝓣 → 𝓤 ⊔ 𝓣 ̇
_∈∈_ f S = (x : _) → f x ∈ S

Pred-refl :  {𝓤 𝓦 : Universe}{A : 𝓤 ̇}{P Q : Pred A 𝓦}
  →          P ≡ Q → (a : A)
             ----------------------
  →          a ∈ P → a ∈ Q
Pred-refl (refl _) _ = λ z → z

Pred-≡ : {𝓤 𝓦 : Universe}{A : 𝓤 ̇}{P Q : Pred A 𝓦} → P ≡ Q → P =̇ Q
Pred-≡ (refl _) = (λ z → z) , λ z → z

Pred-≡→⊆ : {𝓤 𝓦 : Universe}{A : 𝓤 ̇}{P Q : Pred A 𝓦} → P ≡ Q → (P ⊆ Q)
Pred-≡→⊆ (refl _) = (λ z → z)

Pred-≡→⊇ : {𝓤 𝓦 : Universe}{A : 𝓤 ̇}{P Q : Pred A 𝓦} → P ≡ Q → (P ⊇ Q)
Pred-≡→⊇ (refl _) = (λ z → z)

Pred-=̇-≡ : {𝓤 𝓦 : Universe} → propext 𝓦 → global-dfunext
  →          {A : 𝓤 ̇}{P Q : Pred A 𝓦}
  →          ((x : A) → is-subsingleton (P x))
  →          ((x : A) → is-subsingleton (Q x))
             ----------------------
  →          P =̇ Q → P ≡ Q
Pred-=̇-≡ pe gfe {A}{P}{Q} ssP ssQ (pq , qp) = gfe γ
  where
```

```
    γ : (x : A) → P x ≡ Q x
    γ x = pe (ssP x) (ssQ x) pq qp

-- Disjoint Union.
data _∪̇_ {𝓤 𝓦 : Universe}(A : 𝓤 ̇) (B : 𝓦 ̇) : 𝓤 ⊔ 𝓦 ̇ where
  inj₁ : (x : A) → A ∪̇ B
  inj₂ : (y : B) → A ∪̇ B
infixr 1 _∪̇_

-- Union.
_∪_ : {𝓤 𝓦 𝓣 : Universe}{A : 𝓤 ̇} → Pred A 𝓦 → Pred A 𝓣 → Pred A _
P ∪ Q = λ x → x ∈ P ∪̇ x ∈ Q
infixr 1 _∪_

-- The empty set.
∅ : {𝓤 : Universe}{A : 𝓤 ̇} → Pred A 𝓤₀
∅ = λ _ → 𝟘

-- Singletons.
｛_｝ : {𝓤 : Universe}{A : 𝓤 ̇} → A → Pred A _
｛ x ｝ = x ≡_

Im_⊆_ : {𝓤 𝓦 𝓣 : Universe} {A : 𝓤 ̇ } {B : 𝓦 ̇ } → (A → B) → Pred B 𝓣 → 𝓤 ⊔ 𝓣 ̇
Im_⊆_ {A = A} f S = (x : A) → f x ∈ S

img : {𝓤 : Universe}{X : 𝓤 ̇ } {Y : 𝓤 ̇ }
      (f : X → Y) (P : Pred Y 𝓤)
  →   Im f ⊆ P → X → Σ P
img {Y = Y} f P Imf⊆P = λ x₁ → f x₁ , Imf⊆P x₁
```

4.1.4 Predicate product and transport

The product Π P of a predicate P : Pred X 𝓤 is inhabited iff P x holds for all $x : X$.

```
ΠP-meaning : {𝓧 𝓤 : Universe}{X : 𝓧 ̇}{P : Pred X 𝓤}
  → Π P → (x : X) → P x
ΠP-meaning f x = f x
```

The following is a pair of useful "transport" lemmas for predicates.

```
module _ {𝓤 𝓦 : Universe} where

  cong-app-pred : {A : 𝓤 ̇ }{B₁ B₂ : Pred A 𝓦}
                  (x : A) → x ∈ B₁ → B₁ ≡ B₂
                 ------------------------------
   →                         x ∈ B₂
  cong-app-pred x x∈B₁ (refl _ ) = x∈B₁

  cong-pred : {A : 𝓤 ̇ }{B : Pred A 𝓦}
              (x y : A) → x ∈ B →  x ≡ y
              ----------------------------
   →                      y ∈ B
  cong-pred x .x x∈B (refl _ ) = x∈B
```

4.2 Binary Relation and Kernel Types

This section presents the UALib.Relations.Binary module of the Agda UALib. 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 $A \to A \to \mathcal{R}$ ̇ (for some universe $\mathcal{R}$). We define these below.

A generalization of the notion of 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.

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

module UALib.Relations.Binary where

open import UALib.Relations.Unary public

module _ {𝓤 : Universe} where

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

4.2.1 Kernels

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
```

4.2.2 Examples

```
 ker : {A B : 𝓤 ̇ } → (A → B) → 𝓤 ̇
 ker = KER{𝓤}

 ker-rel : {A B : 𝓤 ̇ } → (A → B) → Rel A 𝓤
 ker-rel = KER-rel {𝓤}

 ker-pred : {A B : 𝓤 ̇ } → (A → B) → Pred (A × A) 𝓤
 ker-pred = KER-pred {𝓤}

 --The identity relation.
 𝟎 : {A : 𝓤 ̇ } → 𝓤 ̇
 𝟎 {A} = Σ a ꞉ A , Σ b ꞉ A , a ≡ b

 --...as a binary relation...
```

```
𝟎-rel : {A : 𝓤 ̇ } → Rel A 𝓤
𝟎-rel a b = a ≡ b

--...as a binary predicate...
𝟎-pred : {A : 𝓤 ̇ } → Pred (A × A) 𝓤
𝟎-pred (a , a') = a ≡ a'

𝟎-pred' : {A : 𝓤 ̇ } → 𝓤 ̇
𝟎-pred' {A} = Σ p ꞉ (A × A) , ∣ p ∣ ≡ ∥ p ∥

--...on the domain of an algebra...

𝟎-alg-rel : {S : Signature 𝓞 𝓥}{𝑨 : Algebra 𝓤 S} → 𝓤 ̇
𝟎-alg-rel {𝑨 = 𝑨} = Σ a ꞉ ∣ 𝑨 ∣ , Σ b ꞉ ∣ 𝑨 ∣ , a ≡ b

-- The total relation A × A
𝟏 : {A : 𝓤 ̇ } → Rel A 𝓤₀
𝟏 a b = 𝟙
```

4.2.3 Properties of binary relations

```
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

is-subsingleton-valued : {𝓡 : Universe}{A : 𝓤 ̇ } → Rel A 𝓡 → 𝓤 ⊔ 𝓡 ̇
is-subsingleton-valued _≈_ = ∀ x y → is-prop (x ≈ y)
```

4.2.4 Binary relation truncation

Recall, in Section 2.1.7 we described the concept of truncation as it relates to "proof-relevant" mathematics. Given a binary relation 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[6] We use Escardo's is-subsingleton type to express this strong (truncation at level 1) assumption in the following definition: We say that (x , y) belongs to P, or that x and y are P-related if and only if both P x y *and* is-subsingleton (P x y) holds.

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

As above we define a **set** to be a type X with the following property: for all $x\ y : X$ there is at most one proof that $x \equiv y$. In other words, X is a set if and only if it satisfies the following:

```
∀ x y : X → is-subsingleton ( x ≡ y )
```

[6] This is another example of "proof-irrelevance"; indeed, proofs of P x y are indistinguishable, or rather any distinctions are irrelevant in the context of interest.

4.2.5 Implication

We denote and define implication as follows.

```
-- (syntactic sugar)
_on_ : {𝓤 𝓥 𝓦 : Universe}{A : 𝓤 ̇}{B : 𝓥 ̇}{C : 𝓦 ̇}
  → (B → B → C) → (A → B) → (A → A → C)

_*_ on g = λ x y → g x * g y


_⇒_ : {𝓤 𝓥 𝓦 𝓧 : Universe}{A : 𝓤 ̇ } {B : 𝓥 ̇ }
  → REL A B 𝓦 → REL A B 𝓧 → 𝓤 ⊔ 𝓥 ⊔ 𝓦 ⊔ 𝓧 ̇

P ⇒ Q = ∀ {i j} → P i j → Q i j

infixr 4 _⇒_
```

Here is a more general version that we borrow from the standard library and translate into MHE/UALib notation.

```
_=[_]⇒_ : {𝓤 𝓥 𝓡 𝓢 : Universe}{A : 𝓤 ̇ } {B : 𝓥 ̇ }
  → Rel A 𝓡 → (A → B) → Rel B 𝓢 → 𝓤 ⊔ 𝓡 ⊔ 𝓢 ̇

P =[ g ]⇒ Q = P ⇒ (Q on g)

infixr 4 _=[_]⇒_
```

4.3 Equivalence Relation Types

This section presents the UALib.Relations.Equivalences module of the Agda UALib. This is all standard stuff. The notions of reflexivity, symmetry, and transitivity are defined as one would hope and expect, so we present them here without further explanation.

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

module UALib.Relations.Equivalences where

open import UALib.Relations.Binary public

module _ {𝓤 𝓡 : Universe} where

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

  is-equivalence-relation : {X : 𝓤 ̇ } → Rel X 𝓡 → 𝓤 ⊔ 𝓡 ̇
  is-equivalence-relation _≈_ = is-subsingleton-valued _≈_
                                × reflexive _≈_ × symmetric _≈_ × transitive _≈_
```

4.3.1 Examples

The zero relation 𝟎-rel is equivalent to the identity relation ≡ and, of course, these are both equivalence relations. Indeed, we saw in Subsection 2.2.1 that ≡ is reflexive, symmetric, and transitive, so we simply apply the corresponding proofs where appropriate.

```
module _ {𝓤 : Universe} where

  𝟎-IsEquivalence : {A : 𝓤 ̇ } → IsEquivalence{𝓤}{A = A} 𝟎-rel
  𝟎-IsEquivalence = record { rfl = ≡-rfl; sym = ≡-sym; trans = ≡-trans }

  ≡-IsEquivalence : {A : 𝓤 ̇} → IsEquivalence{𝓤}{A = A} _≡_
  ≡-IsEquivalence = record { rfl = ≡-rfl ; sym = ≡-sym ; trans = ≡-trans }
```

Finally, it's useful to have on hand a proof of the fact that the kernel of a function is an equivalence relation.

```
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 }
```

4.4 Quotient Types

This section presents the UALib.Relations.Quotients module of the Agda UALib.

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

module UALib.Relations.Quotients where

open import UALib.Relations.Equivalences public
open import UALib.Prelude.Preliminaries using (_⇔_; id) public

module _ {𝓤 𝓡 : Universe} where
```

For a binary relation R on A, we denote a single R-class as $[\,a\,]\,R$ (the class containing a). This notation is defined in UALib as follows.

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

So, $x \in [\,a\,]\,R$ iff $R\ a\ x$, and the following elimination rule is a tautology.

```
[]-elim : {A : 𝓤 ̇ }{a x : A}{R : Rel A 𝓡}
  → R a x ⇔ (x ∈ [ a ] R)
[]-elim = id , id
```

We define type of all classes of a relation R as follows.

```
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.

```
-- relation quotient (predicate version)
_/_ : (A : 𝓤 ̇ ) → Rel A 𝓡 → 𝓤 ⊔ (𝓡 ⁺) ̇
A / R = Σ C ꞉ Pred A 𝓡 , 𝒞{A}{R} C
-- old version: A / R = Σ C ꞉ Pred A 𝓡 , Σ a ꞉ A , C ≡ ( [ a ] R )
```

We then define the following introduction rule for a relation class with designated representative.

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

--So, x ∈ [ a ]ₚ R iff R a x, and the following elimination rule is a tautology.
⟦⟧-elim : {A : 𝓤 ̇ }{a x : A}{R : Rel A 𝓡}
  → R a x ⇔ (x ∈ [ a ] R)
⟦⟧-elim = id , id
```

If the relation is reflexive, then we have the following elimination rules.

```
/-refl : {A : 𝓤 ̇}{a a' : A}{R : Rel A 𝓡}
  → reflexive R → [ a ] R ≡ [ a' ] R → R a a'
/-refl{A = A}{a}{a'}{R} rfl x = γ
  where
    a'in : a' ∈ [ a' ] R
    a'in = rfl a'
    γ : a' ∈ [ a ] R
    γ = cong-app-pred a' a'in (x ⁻¹)

/-refl' : {A : 𝓤 ̇}{a a' : A}{R : Rel A 𝓡}
  → transitive R → R a' a → ([ a ] R) ⊆ ([ a' ] R)
/-refl'{A = A}{a}{a'}{R} trn Ra'a {x} aRx = trn a' a x Ra'a aRx

⌜_⌝ : {A : 𝓤 ̇}{R : Rel A 𝓡} → A / R → A
⌜ 𝒂 ⌝ = ∣ ∥ 𝒂 ∥ ∣ -- type ⌜ and ⌝ as `\cul` and `\cur`
```

and an elimination rule for relation class representative, defined as follows.

```
/-Refl : {A : 𝓤 ̇}{a a' : A}{R : Rel A 𝓡}
  → reflexive R → ⟦ a ⟧{R} ≡ ⟦ a' ⟧ → R a a'
/-Refl rfl (refl _) = rfl _
```

Later we will need the following additional quotient tools.

```
open IsEquivalence{𝓤}{𝓡}

/-subset : {A : 𝓤 ̇}{a a' : A}{R : Rel A 𝓡}
  → IsEquivalence R → R a a' → ([ a ] R) ⊆ ([ a' ] R)
/-subset {A = A}{a}{a'}{R} Req Raa' {x} Rax = (trans Req) a' a x (sym Req a a' Raa') Rax

/-supset : {A : 𝓤 ̇}{a a' : A}{R : Rel A 𝓡}
  → IsEquivalence R → R a a' → ([ a ] R) ⊇ ([ a' ] R)
```

```
/-supset {A = A}{a}{a'}{R} Req Raa' {x} Ra'x = (trans Req) a a' x Raa' Ra'x

/-=̇ : {A : 𝓤 ̇}{a a' : A}{R : Rel A 𝓡}
  → IsEquivalence R → R a a' → ([ a ] R) =̇ ([ a' ] R)
/-=̇ {A = A}{a}{a'}{R} Req Raa' = /-subset Req Raa' , /-supset Req Raa'
```

4.4.1 Quotient extensionality

We 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.

(Recall, a type is called a **set** if it has *unique identity proofs*; as a general principle, this is sometimes referred to as "proof irrelevance" or "uniqueness of identity proofs"—two proofs of a single identity are the same.)

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

class-extensionality pe gfe {A = A}{a}{a'}{R} ssR Req Raa' =
  Pred-=̇-≡ pe gfe {A}{[ a ] R}{[ a' ] R} (ssR a) (ssR a') (/-=̇ Req Raa')

to-subtype-⟦⟧ : {A : 𝓤 ̇}{R : Rel A 𝓡}{C D : Pred A 𝓡}
                {c : 𝒞 C}{d : 𝒞 D}
  →             (∀ C → is-subsingleton (𝒞{A}{R} C))
  →             C ≡ D → (C , c) ≡ (D , d)

to-subtype-⟦⟧ {D = D}{c}{d} ssA CD = to-Σ-≡ (CD , ssA D (transport 𝒞 CD c) d)

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})

class-extensionality' pe gfe {A = A}{a}{a'}{R} ssR ssA Req Raa' = γ
  where
    CD : ([ a ] R) ≡ ([ a' ] R)
    CD = class-extensionality pe gfe {A}{a}{a'}{R} ssR Req Raa'

    γ : (⟦ a ⟧ {R}) ≡ (⟦ a' ⟧ {R})
    γ = to-subtype-⟦⟧ ssA CD
```

4.4.2 Compatibility

The following definitions and lemmas are useful for asserting and proving facts about **compatibility** of relations and functions.

```
module _ {𝓤 𝓥 𝓦 : Universe} {γ : 𝓥 ̇ } {Z : 𝓤 ̇ } where

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

  compatible-fun : (f : (γ → Z) → Z)(R : Rel Z 𝓦) → 𝓥 ⊔ 𝓤 ⊔ 𝓦 ̇
  compatible-fun f R = (lift-rel R) =[ f ]⇒ R

-- relation compatible with an operation
module _ {𝓤 𝓦 : Universe} {S : Signature 𝓞 𝓥} where
  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)
  -- alternative notation: (lift-rel R) =[ f ̂ 𝑨 ]⇒ R

--The given relation is compatible with all ops of an algebra.
  compatible : (𝑨 : Algebra 𝓤 S) → Rel ∣ 𝑨 ∣ 𝓦 → 𝓞 ⊔ 𝓤 ⊔ 𝓥 ⊔ 𝓦 ̇
  compatible 𝑨 R = ∀ f → compatible-op{𝑨} f R
```

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

4.5 Congruence Relation Types

This section presents the UALib.Relations.Congruences module of the Agda UALib. Notice that module begins by assuming a signature S : Signature 𝓞 𝓥 which is then present and available throughout the module.

```
{-# OPTIONS --without-K --exact-split --safe #-}
open import UALib.Algebras.Signatures using (Signature; 𝓞; 𝓥)

module UALib.Relations.Congruences {S : Signature 𝓞 𝓥} where

open import UALib.Relations.Quotients hiding (Signature; 𝓞; 𝓥) public

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
```

4.5.1 Example

We defined the *trivial* (or "diagonal" or "identity" or "zero") relation **0**-rel in Subsection 4.2.2, and we observed in Subsection 4.3.1 that **0**-rel is equivalent to the identity relation ≡ and that these are both equivalence relations. Therefore, in order to build a congruence of some algebra **A** out of the trivial relation, it remains to show that **0**-rel is compatible with all operations of **A**. We do this now and immediately after we construct the corresponding congruence.

```
module _ {𝓤 : Universe} {S : Signature 𝓞 𝓥} where

 𝟎-compatible-op : funext 𝓥 𝓤 → {𝑨 : Algebra 𝓤 S} (f : ∣ S ∣)
  →                 compatible-op {𝓤 = 𝓤}{𝑨 = 𝑨} f 𝟎-rel
 𝟎-compatible-op fe {𝑨} f ptws0 = ap (f ̂ 𝑨) (fe (λ x → ptws0 x))

 𝟎-compatible : funext 𝓥 𝓤 → {A : Algebra 𝓤 S} → compatible A 𝟎-rel
 𝟎-compatible fe {A} = λ f args → 𝟎-compatible-op fe {A} f args
```

Now that we have the ingredients required to construct a congruence, we carry out the construction as follows.

```
Δ : {𝓤 : Universe} → funext 𝓥 𝓤 → (A : Algebra 𝓤 S) → Congruence A
Δ fe A = mkcon 𝟎-rel ( 𝟎-compatible fe ) ( 𝟎-IsEquivalence )
```

4.5.2 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 denote by **A** / θ and Agda allows us to define and express quotients using the standard notation.

```
_╱_ : {𝓤 𝓡 : Universe}(A : Algebra 𝓤 S) -- type ╱ with `\---`
 →    Congruence{𝓤}{𝓡} A -- a number of times`
      -----------------------
 →    Algebra (𝓤 ⊔ 𝓡 ⁺) S
A ╱ θ = (( ∣ A ∣ / ⟨ θ ⟩ ) , -- carrier (i.e. domain or universe))
       (λ f args -- operations
        → ([ (f ̂ A) (λ i₁ → ∣ ∥ args i₁ ∥ ∣) ] ⟨ θ ⟩) ,
          ((f ̂ A) (λ i₁ → ∣ ∥ args i₁ ∥ ∣) , refl _ )
       )
      )
```

4.5.3 Examples

The zero element of a quotient can be expressed as follows.

```
Zero╱ : {𝓤 𝓡 : Universe}{A : Algebra 𝓤 S}
        (θ : Congruence{𝓤}{𝓡} A)
 →      Rel (∣ A ∣ / ⟨ θ ⟩) (𝓤 ⊔ 𝓡 ⁺)

Zero╱ θ = λ x x₁ → x ≡ x₁
```

Finally, the following elimination rule is sometimes useful.

```
╱-refl :{𝓤 𝓡 : Universe} (A : Algebra 𝓤 S)
        {θ : Congruence{𝓤}{𝓡} A} {a a' : ∣ A ∣}
```

```
    →    ⟦ a ⟧{⟨ θ ⟩} ≡ ⟦ a' ⟧ → ⟨ θ ⟩ a a'

/-refl A {θ} (refl _) = IsEquivalence.rfl (IsEquiv θ) _
```

5 Homomorphisms

5.1 Basic Definitions

This section describes the UALib.Homomorphisms.Basic module of the Agda UALib. The definition of homomorphism in the Agda UALib is an *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 $f\ x \equiv g\ x$.

Now let's quickly dispense with the usual preliminaries so we can get down to the business of defining homomorphisms.

```
{-# OPTIONS --without-K --exact-split --safe #-}
open import UALib.Algebras.Signatures using (Signature; 𝓞; 𝓥)
module UALib.Homomorphisms.Basic {𝑆 : Signature 𝓞 𝓥} where
open import UALib.Relations.Congruences{𝑆 = 𝑆} public
open import UALib.Prelude.Preliminaries using (_≡⟨_⟩_; _∎) public
```

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 $g : A \to B$.

```
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 ∀ 𝒂 in the definition of compatible-op-map. We can get away with this in place of the fully type-annotated equivalent, (𝒂 : ∥ 𝑆 ∥ 𝑓 → ∣ 𝑨 ∣) since Agda is able to infer that the 𝒂 here must be a tuple on ∣ 𝑨 ∣ of "length" ∥ 𝑆 ∥ 𝑓 (the arity of 𝑓).

```
op_interpreted-in_and_commutes-with : {𝓠 𝓤 : Universe}
                                      (𝑓 : ∣ 𝑆 ∣) (𝑨 : Algebra 𝓠 𝑆) (𝑩 : Algebra 𝓤 𝑆)
                                      (𝑔 : ∣ 𝑨 ∣ → ∣ 𝑩 ∣) → 𝓥 ⊔ 𝓠 ⊔ 𝓤 ̇

op 𝑓 interpreted-in 𝑨 and 𝑩 commutes-with 𝑔 = compatible-op-map 𝑨 𝑩 𝑓 𝑔
```

We now define the type hom 𝑨 𝑩 of **homomorphisms** from 𝑨 to 𝑩 by first defining the property is-homomorphism, as follows.

```
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
```

5.1.1 Examples

A simple example is the identity map, which is proved to be a homomorphism as follows.

```
𝒾𝒹 : {𝓤 : Universe} (A : Algebra 𝓤 𝑆) → hom A A
𝒾𝒹 _ = (λ x → x) , λ _ _ → 𝓇ℯ𝒻𝓁

id-is-hom : {𝓤 : Universe}{𝑨 : Algebra 𝓤 𝑆} → is-homomorphism 𝑨 𝑨 (𝑖𝑑 ∣ 𝑨 ∣)
id-is-hom = λ _ _ → refl _
```

5.1.2 Equalizers in Agda

Recall, the equalizer of two functions (resp., homomorphisms) $g\ h : A \to B$ is the subset of A on which the values of the functions g and h agree. We define the **equalizer** of functions and homomorphisms in Agda as follows.

```
--Equalizers of functions
E : {𝓤 𝓦 : Universe}{A : 𝓤 ̇}{B : 𝓦 ̇}(g h : A → B) → Pred A 𝓦
E g h x = g x ≡ h x

--Equalizers of homomorphisms
EH : {𝓤 𝓦 : Universe}{A : Algebra 𝓤 S}{B : Algebra 𝓦 S}(g h : hom A B) → Pred ∣ A ∣ 𝓦
EH g h x = ∣ g ∣ x ≡ ∣ h ∣ x
```

We will define subuniverses in the UALib.Subalgebras.Subuniverses module, but we note here that the equalizer of homomorphisms *f*rom **A** to **B** will turn out to be subuniverse of **A**. Indeed, this easily follow from,

```
EH-is-closed : {𝓤 𝓦 : Universe} → funext 𝓥 𝓦
 →             {A : Algebra 𝓤 S}{B : Algebra 𝓦 S}
               (g h : hom A B) {f : ∣ S ∣ } (a : (∥ S ∥ f) → ∣ A ∣)
 →             ( (x : ∥ S ∥ f) → (a x) ∈ (EH {A = A}{B = B} g h) )
               ---------------------------------------------------
 →             ∣ g ∣ ((f ̂ A) a) ≡ ∣ h ∣ ((f ̂ A) a)

EH-is-closed fe {A}{B} g h {f} a p =
                 ∣ g ∣ ((f ̂ A) a)  ≡⟨ ∥ g ∥ f a ⟩
                 (f ̂ B)(∣ g ∣ ∘ a) ≡⟨ ap (_ ̂ B)(fe p) ⟩
                 (f ̂ B)(∣ h ∣ ∘ a) ≡⟨ (∥ h ∥ f a)⁻¹ ⟩
                 ∣ h ∣ ((f ̂ A) a)  ∎
```

5.2 Kernels of Homomorphisms

This section describes the UALib.Homomorphisms.Kernels module of the Agda UALib. The kernel of a homomorphism is a congruence and conversely for every congruence θ, there exists a homomorphism with kernel θ.

```
{-# OPTIONS --without-K --exact-split --safe #-}
open import UALib.Algebras.Signatures using (Signature; 𝓞; 𝓥)
open import UALib.Prelude.Preliminaries using (global-dfunext)

module UALib.Homomorphisms.Kernels {S : Signature 𝓞 𝓥} {gfe : global-dfunext} where

open import UALib.Homomorphisms.Basic{S = S} public

module _ {𝓤 𝓦 : Universe} where
  open Congruence

  hom-kernel-is-compatible : (A : Algebra 𝓤 S){B : Algebra 𝓦 S}
                             (h : hom A B) → compatible A (KER-rel ∣ h ∣)

  hom-kernel-is-compatible A {B} h f {a}{a'} Kerhab = γ
```

```
  where
   γ : | h | ((f ̂ A) a)      ≡ | h | ((f ̂ A) a')
   γ = | h | ((f ̂ A) a)    ≡⟨ ∥ h ∥ f a ⟩
       (f ̂ B) (| h | ∘ a)  ≡⟨ ap (λ - → (f ̂ B) -) (gfe λ x → Kerhab x) ⟩
       (f ̂ B) (| h | ∘ a') ≡⟨ (∥ h ∥ f a')⁻¹ ⟩
       | h | ((f ̂ A) a')   ∎

hom-kernel-is-equivalence : (A : Algebra 𝓤 S){B : Algebra 𝓦 S}
                            (h : hom A B) → IsEquivalence (KER-rel | h |)

hom-kernel-is-equivalence A h = map-kernel-IsEquivalence | h |
```

It is convenient to define a function that takes a homomorphism and constructs a congruence from its kernel. We call this function hom-kernel-congruence, but since we will use it often we also give it a short alias—kercon.

```
kercon -- (alias)
  hom-kernel-congruence : (A : Algebra 𝓤 S){B : Algebra 𝓦 S}
                          (h : hom A B) → Congruence A

hom-kernel-congruence A {B} h = mkcon (KER-rel | h |)
                                  (hom-kernel-is-compatible A {B} h)
                                    (hom-kernel-is-equivalence A {B} h)
kercon = hom-kernel-congruence -- (alias)

quotient-by-hom-kernel : (A : Algebra 𝓤 S){B : Algebra 𝓦 S}
                         (h : hom A B) → Algebra (𝓤 ⊔ 𝓦 ⁺) S

quotient-by-hom-kernel A{B} h = A ╱ (hom-kernel-congruence A{B} h)

-- NOTATION.
_[_]/ker_ : (A : Algebra 𝓤 S)(B : Algebra 𝓦 S)(h : hom A B) → Algebra (𝓤 ⊔ 𝓦 ⁺) S
A [ B ]/ker h = quotient-by-hom-kernel A {B} h


epi : {𝓤 𝓦 : Universe} → Algebra 𝓤 S → Algebra 𝓦 S → 𝓞 ⊔ 𝓥 ⊔ 𝓤 ⊔ 𝓦 ̇
epi A B = Σ g ꞉ (| A | → | B | ) , is-homomorphism A B g × Epic g

epi-to-hom : {𝓤 𝓦 : Universe}(A : Algebra 𝓤 S){B : Algebra 𝓦 S}
 →                      epi A B → hom A B
epi-to-hom A ϕ = | ϕ | , fst ∥ ϕ ∥

module _ {𝓤 𝓦 : Universe} where

  open Congruence

  canonical-projection : (A : Algebra 𝓤 S) (θ : Congruence{𝓤}{𝓦} A)
  →                      epi A (A ╱ θ)

  canonical-projection A θ = cπ , cπ-is-hom , cπ-is-epic
    where
     cπ : | A | → | A ╱ θ |
     cπ a = ⟦ a ⟧ -- ([ a ] (KER-rel | h |)) , ?
```

```
    cπ-is-hom : is-homomorphism A (A ╱ θ) cπ
    cπ-is-hom f a = γ
      where
       γ : cπ ((f ̂ A) a) ≡ (f ̂ (A ╱ θ)) (λ x → cπ (a x))
       γ = cπ ((f ̂ A) a) ≡⟨ refl ⟩
            ⟦ (f ̂ A) a ⟧ ≡⟨ refl ⟩
            (f ̂ (A ╱ θ)) (λ x → ⟦ a x ⟧) ≡⟨ refl ⟩
            (f ̂ (A ╱ θ)) (λ x → cπ (a x)) ∎

    cπ-is-epic : Epic cπ
    cπ-is-epic (.(⟨ θ ⟩ a) , a , refl _) = Image_∋_.im a

πᵏ -- alias
  kernel-quotient-projection : {𝓤 𝓦 : Universe} -- (pe : propext 𝓦)
                               (A : Algebra 𝓤 S){B : Algebra 𝓦 S}
                               (h : hom A B)
                               ----------------------------------
  →                            epi A (A [ B ]/ker h)

kernel-quotient-projection A {B} h = canonical-projection A (kercon A{B} h)

πᵏ = kernel-quotient-projection
```

5.3 Homomorphism Theorems

This section describes the UALib.Homomorphisms.Noether module of the Agda UALib.

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

open import UALib.Algebras.Signatures using (Signature; 𝓞; 𝓥)
open import UALib.Prelude.Preliminaries using (global-dfunext)

module UALib.Homomorphisms.Noether {S : Signature 𝓞 𝓥}{gfe : global-dfunext} where

open import UALib.Homomorphisms.Kernels{S = S}{gfe} hiding (global-dfunext) public
```

5.3.1 The First Homomorphism Theorem

Here is a version of the so-called *First Homomorphism Theorem* (aka, the *First Isomorphism Theorem*).

```
open Congruence

FirstIsomorphismTheorem : {𝓤 𝓦 : Universe}
                          (A : Algebra 𝓤 S)(B : Algebra 𝓦 S)
                          (ϕ : hom A B) (ϕE : Epic ∣ ϕ ∣ )
                           -- extensionality assumptions:
                             {pe : propext 𝓦}
                             (Bset : is-set ∣ B ∣)
 →                           (∀ a x → is-subsingleton (⟨ kercon A{B} ϕ ⟩ a x))
 →                           (∀ C → is-subsingleton (𝒞{A = ∣ A ∣}{⟨ kercon A{B} ϕ ⟩} C))
```

```
                 ------------------------------------------------------------
 →               Σ f : (epi (A [ B ]/ker ϕ) B) , ( | ϕ | ≡ | f | ∘ | πᵏ A {B} ϕ | ) × is-embedding | f |

FirstIsomorphismTheorem {𝓤}{𝓦} A B ϕ ϕE {pe} Bset ssR ssA = (fmap , fhom , fepic) , commuting , femb
  where
   θ : Congruence A
   θ = kercon A{B} ϕ

   A/θ : Algebra (𝓤 ⊔ 𝓦 ⁺) S
   A/θ = A [ B ]/ker ϕ

   fmap : | A/θ | → | B |
   fmap a = | ϕ | ⌜ a ⌝

   fhom : is-homomorphism A/θ B fmap
   fhom f a = | ϕ | ( fst ∥ (f ̂ A/θ) a ∥ ) ≡⟨ refl ⟩
              | ϕ | ( (f ̂ A) (λ x → ⌜ (a x) ⌝) ) ≡⟨ ∥ ϕ ∥ f (λ x → ⌜ (a x) ⌝) ⟩
               (f ̂ B) (| ϕ | ∘ (λ x → ⌜ (a x) ⌝)) ≡⟨ ap (λ - → (f ̂ B) -) (gfe λ x → refl) ⟩
               (f ̂ B) (λ x → fmap (a x)) ∎

   fepic : Epic fmap
   fepic b = γ
    where
     a : | A |
     a = EpicInv | ϕ | ϕE b

     a/θ : | A/θ |
     a/θ = ⟦ a ⟧

     bfa : b ≡ fmap a/θ
     bfa = (cong-app (EpicInvIsRightInv gfe | ϕ | ϕE) b)⁻¹

     γ : Image fmap ∋ b
     γ = Image_∋_.eq b a/θ bfa


   commuting : | ϕ | ≡ fmap ∘ | πᵏ A {B} ϕ |
   commuting = refl

   fmon : Monic fmap
   fmon (.(⟨ θ ⟩ a) , a , refl _) (.(⟨ θ ⟩ a') , a' , refl _) faa' = γ
    where
     aθa' : ⟨ θ ⟩ a a'
     aθa' = faa'

     γ : (⟨ θ ⟩ a , a , refl) ≡ (⟨ θ ⟩ a' , a' , refl)
     γ = class-extensionality' pe gfe ssR ssA (IsEquiv θ) aθa'

   femb : is-embedding fmap
   femb = monic-into-set-is-embedding Bset fmap fmon
```

5.3.2 Homomorphism composition

The composition of homomorphisms is again a homomorphism. For convenience, we give a few versions of this theorem which differ only with respect to which of their arguments are implicit.

```
module _ {𝓠 𝓤 𝓦 : Universe} where

  -- composition of homomorphisms 1
  HCompClosed : (A : Algebra 𝓠 S)(B : Algebra 𝓤 S)(C : Algebra 𝓦 S)
   →            hom A B → hom B C
                --------------------
   →            hom A C

  HCompClosed (A , FA) (B , FB) (C , FC) (g , ghom) (h , hhom) = h ∘ g , γ
   where
    γ : (f : ∣ S ∣)(a : ∥ S ∥ f → A) → (h ∘ g)(FA f a) ≡ FC f (h ∘ g ∘ a)

    γ f a = (h ∘ g) (FA f a) ≡⟨ ap h ( ghom f a ) ⟩
           h (FB f (g ∘ a)) ≡⟨ hhom f ( g ∘ a ) ⟩
           FC f (h ∘ g ∘ a) ∎

  -- composition of homomorphisms 2
  HomComp : (A : Algebra 𝓠 S){B : Algebra 𝓤 S}(C : Algebra 𝓦 S)
   →            hom A B → hom B C
                --------------------
   →            hom A C
  HomComp A {B} C f g = HCompClosed A B C f g

  -- composition of homomorphisms 3
∘-hom : {𝓧 𝓨 𝓩 : Universe}
        (A : Algebra 𝓧 S)(B : Algebra 𝓨 S)(C : Algebra 𝓩 S)
        {f : ∣ A ∣ → ∣ B ∣} {g : ∣ B ∣ → ∣ C ∣}
 →      is-homomorphism{𝓧}{𝓨} A B f → is-homomorphism{𝓨}{𝓩} B C g
        ---------------------------------------------------
 →      is-homomorphism{𝓧}{𝓩} A C (g ∘ f)

∘-hom A B C {f} {g} fhom ghom = ∥ HCompClosed A B C (f , fhom) (g , ghom) ∥

-- composition of homomorphisms 4
∘-Hom : {𝓧 𝓨 𝓩 : Universe}
        (A : Algebra 𝓧 S){B : Algebra 𝓨 S}(C : Algebra 𝓩 S)
        {f : ∣ A ∣ → ∣ B ∣} {g : ∣ B ∣ → ∣ C ∣}
 →      is-homomorphism{𝓧}{𝓨} A B f → is-homomorphism{𝓨}{𝓩} B C g
        -------------------------------------------------
 →      is-homomorphism{𝓧}{𝓩} A C (g ∘ f)

∘-Hom A {B} C {f} {g} = ∘-hom A B C {f} {g}


trans-hom : {𝓧 𝓨 𝓩 : Universe}
        (A : Algebra 𝓧 S)(B : Algebra 𝓨 S)(C : Algebra 𝓩 S)
        (f : ∣ A ∣ → ∣ B ∣ )(g : ∣ B ∣ → ∣ C ∣ )
```

```
    → is-homomorphism{𝔛}{𝔜} A B f → is-homomorphism{𝔜}{ℨ} B C g
      --------------------------------------------------------
    → is-homomorphism{𝔛}{ℨ} A C (g ∘ f)
  trans-hom {𝔛}{𝔜}{ℨ} A B C f g = ∘-hom {𝔛}{𝔜}{ℨ} A B C {f}{g}
```

5.3.3 Homomorphism decomposition

If g : hom **A** **B**, h : hom **A** **C**, h is surjective, and ker $h \subseteq$ ker g, then there exists ϕ : hom **C** **B**} such that $g = \phi \circ h$, that is, such that the following diagram commutes.

```
A---- h -->>C
 \          .
  \        .
   g     ∃ϕ
    \    .
     \ .
      V
      B
```

This, or some variation of it, is sometimes referred to as the *Second Isomorphism Theorem*. We formalize its statement and proof as follows. (Notice that the proof is constructive.)

```
  homFactor : {𝓤 : Universe} → funext 𝓤 𝓤 → {A B C : Algebra 𝓤 S}
                  (g : hom A B) (h : hom A C)
    →             ker-pred ∣ h ∣ ⊆ ker-pred ∣ g ∣ → Epic ∣ h ∣
                    ------------------------------
    →               Σ ϕ ꞉ (hom C B) , ∣ g ∣ ≡ ∣ ϕ ∣ ∘ ∣ h ∣

  homFactor fe {A = A , FA}{B = B , FB}{C = C , FC}
   (g , ghom) (h , hhom) Kh⊆Kg hEpi = (ϕ , ϕIsHomCB) , g≡ϕ∘h
    where
     hInv : C → A
     hInv = λ c → (EpicInv h hEpi) c

     ϕ : C → B
     ϕ = λ c → g ( hInv c )

     ξ : (x : A) → ker-pred h (x , hInv (h x))
     ξ x = ( cong-app (EpicInvIsRightInv fe h hEpi) ( h x ) )⁻¹

     g≡ϕ∘h : g ≡ ϕ ∘ h
     g≡ϕ∘h = fe λ x → Kh⊆Kg (ξ x)

     ζ : (f : ∣ S ∣)(c : ∥ S ∥ f → C)(x : ∥ S ∥ f)
      → c x ≡ (h ∘ hInv)(c x)

     ζ f c x = (cong-app (EpicInvIsRightInv fe h hEpi) (c x))⁻¹

     ι : (f : ∣ S ∣)(c : ∥ S ∥ f → C)
      → (λ x → c x) ≡ (λ x → h (hInv (c x)))

     ι f c = ap (λ - → - ∘ c)(EpicInvIsRightInv fe h hEpi)⁻¹
```

```
useker : (f : ∣ S ∣) (c : ∥ S ∥ f → C)
  → g (hInv (h (FA f (hInv ∘ c)))) ≡ g(FA f (hInv ∘ c))

useker = λ f c
  → Kh⊆Kg (cong-app
             (EpicInvIsRightInv fe h hEpi)
             (h(FA f(hInv ∘ c)))
          )

ϕIsHomCB : (f : ∣ S ∣)(a : ∥ S ∥ f → C)
  → ϕ (FC f a) ≡ FB f (ϕ ∘ a)

ϕIsHomCB f c =
  g (hInv (FC f c))                ≡⟨ i   ⟩
  g (hInv (FC f (h ∘ (hInv ∘ c)))) ≡⟨ ii  ⟩
  g (hInv (h (FA f (hInv ∘ c))))   ≡⟨ iii ⟩
  g (FA f (hInv ∘ c))              ≡⟨ iv  ⟩
  FB f (λ x → g (hInv (c x)))      ∎
  where
    i   = ap (g ∘ hInv) (ap (FC f) (ι f c))
    ii  = ap (λ - → g (hInv -)) (hhom f (hInv ∘ c))⁻¹
    iii = useker f c
    iv  = ghom f (hInv ∘ c)
```

5.4 Products and Homomorphisms

This section describes the UALib.Homomorphisms.Products module of the Agda UALib.

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

open import UALib.Algebras.Signatures using (Signature; 𝓞; 𝓥)
open import UALib.Prelude.Preliminaries using (global-dfunext)

module UALib.Homomorphisms.Products {S : Signature 𝓞 𝓥}{gfe : global-dfunext} where

open import UALib.Homomorphisms.Noether{S = S}{gfe} public

⊓-hom : global-dfunext → {𝓠 𝓤 𝓘 : Universe}
         {I : 𝓘 ̇}{𝒜 : I → Algebra 𝓠 S}{ℬ : I → Algebra 𝓤 S}
  →      ((i : I) → hom (𝒜 i)(ℬ i))
         -------------------------
  →      hom (⊓ 𝒜) (⊓ ℬ)

⊓-hom gfe {𝓠}{𝓤}{𝓘}{I}{𝒜}{ℬ} homs = ϕ , ϕhom
  where
    ϕ : ∣ ⊓ 𝒜 ∣ → ∣ ⊓ ℬ ∣
    ϕ = λ x i → ∣ homs i ∣ (x i)

    ϕhom : is-homomorphism (⊓ 𝒜) (⊓ ℬ) ϕ
    ϕhom 𝑓 𝒂 = gfe (λ i → ∥ homs i ∥ 𝑓 (λ x → 𝒂 x i))
```

5.4.1 Projection homomorphisms

Later we will need a proof of the fact that projecting out of a product algebra onto one of its factors is a homomorphism.

```
⨅-projection-hom : {𝓤 𝓘 : Universe}
                   {I : 𝓘 ̇}{𝒜 : I → Algebra 𝓤 𝑆}
                   ----------------------------
 →                 (i : I) → hom (⨅ 𝒜) (𝒜 i)

⨅-projection-hom {𝓤}{𝓘}{I}{𝒜} i = ϕi , ϕihom
 where
  ϕi : ∣ ⨅ 𝒜 ∣ → ∣ 𝒜 i ∣
  ϕi = λ x → x i

  ϕihom : is-homomorphism (⨅ 𝒜) (𝒜 i) ϕi
  ϕihom 𝑓 𝒂 = ϕi ((𝑓 ̂ ⨅ 𝒜) 𝒂) ≡⟨ 𝓇ℯ𝒻𝓁 ⟩
           ((𝑓 ̂ ⨅ 𝒜) 𝒂) i ≡⟨ 𝓇ℯ𝒻𝓁 ⟩
           (𝑓 ̂ 𝒜 i) (λ x → 𝒂 x i) ≡⟨ 𝓇ℯ𝒻𝓁 ⟩
           (𝑓 ̂ 𝒜 i) (λ x → ϕi (𝒂 x)) ∎
```

(Of course, we could prove a more general result involving projections onto multiple factors, but so far the single-factor result has sufficed.)

5.5 Isomorphism Type

This section describes the UALib.Homomorphisms.Isomorphisms module of the Agda UALib. We implement (the extensional version of) the notion of isomorphism between algebraic structures.

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

open import UALib.Algebras.Signatures using (Signature; 𝓞; 𝓥)
open import UALib.Prelude.Preliminaries using (global-dfunext)

module UALib.Homomorphisms.Isomorphisms {𝑆 : Signature 𝓞 𝓥}{gfe : global-dfunext} where

open import UALib.Homomorphisms.Products{𝑆 = 𝑆}{gfe} public
open import UALib.Prelude.Preliminaries using (is-equiv; hfunext; Nat; NatΠ; NatΠ-is-embedding) public

_≅_ : {𝓤 𝓦 : Universe} (𝑨 : Algebra 𝓤 𝑆) (𝑩 : Algebra 𝓦 𝑆) → 𝓞 ⊔ 𝓥 ⊔ 𝓤 ⊔ 𝓦 ̇
𝑨 ≅ 𝑩 = Σ f ꞉ (hom 𝑨 𝑩) , Σ g ꞉ (hom 𝑩 𝑨) , ((∣ f ∣ ∘ ∣ g ∣) ∼ ∣ 𝑖𝑑 𝑩 ∣) × ((∣ g ∣ ∘ ∣ f ∣) ∼ ∣ 𝑖𝑑 𝑨 ∣)
```

Recall, $f \sim g$ means f and g are extensionally equal; i.e., $\forall\, x, f\, x \equiv g\, x$.

5.5.1 Isomorphism toolbox

Here are some useful definitions and theorems for working with isomorphisms of algebraic structures.

```
module _ {𝓤 𝓦 : Universe}{𝑨 : Algebra 𝓤 𝑆}{𝑩 : Algebra 𝓦 𝑆} where

 ≅-hom : (ϕ : 𝑨 ≅ 𝑩) → hom 𝑨 𝑩
 ≅-hom ϕ = ∣ ϕ ∣

 ≅-inv-hom : (ϕ : 𝑨 ≅ 𝑩) → hom 𝑩 𝑨
```

```
≅-inv-hom ϕ = fst ∥ ϕ ∥

≅-map : (ϕ : A ≅ B) → ∣ A ∣ → ∣ B ∣
≅-map ϕ = ∣ ≅-hom ϕ ∣

≅-map-is-homomorphism : (ϕ : A ≅ B) → is-homomorphism A B (≅-map ϕ)
≅-map-is-homomorphism ϕ = ∥ ≅-hom ϕ ∥

≅-inv-map : (ϕ : A ≅ B) → ∣ B ∣ → ∣ A ∣
≅-inv-map ϕ = ∣ ≅-inv-hom ϕ ∣

≅-inv-map-is-homomorphism : (ϕ : A ≅ B) → is-homomorphism B A (≅-inv-map ϕ)
≅-inv-map-is-homomorphism ϕ = ∥ ≅-inv-hom ϕ ∥

≅-map-invertible : (ϕ : A ≅ B) → invertible (≅-map ϕ)
≅-map-invertible ϕ = (≅-inv-map ϕ) , (∥ snd ∥ ϕ ∥ ∥ , ∣ snd ∥ ϕ ∥ ∣)

≅-map-is-equiv : (ϕ : A ≅ B) → is-equiv (≅-map ϕ)
≅-map-is-equiv ϕ = invertibles-are-equivs (≅-map ϕ) (≅-map-invertible ϕ)

≅-map-is-embedding : (ϕ : A ≅ B) → is-embedding (≅-map ϕ)
≅-map-is-embedding ϕ = equivs-are-embeddings (≅-map ϕ) (≅-map-is-equiv ϕ)
```

5.5.2 Isomorphism is an equivalence relation

```
REFL-≅ ID≅ : {𝓤 : Universe} (A : Algebra 𝓤 S) → A ≅ A
ID≅ A = 𝑖𝑑 A , 𝑖𝑑 A , (λ a → 𝓇ℯ𝒻ℓ) , (λ a → 𝓇ℯ𝒻ℓ)
REFL-≅ = ID≅

refl-≅ id≅ : {𝓤 : Universe} {A : Algebra 𝓤 S} → A ≅ A
id≅ {𝓤}{A} = ID≅ {𝓤} A
refl-≅ = id≅

sym-≅ : {𝓠 𝓤 : Universe}{A : Algebra 𝓠 S}{B : Algebra 𝓤 S}
  →      A ≅ B → B ≅ A
sym-≅ h = fst ∥ h ∥ , fst h , ∥ snd ∥ h ∥ ∥ , ∣ snd ∥ h ∥ ∣

trans-≅ : {𝓠 𝓤 𝓦 : Universe}
           (A : Algebra 𝓠 S)(B : Algebra 𝓤 S)(C : Algebra 𝓦 S)
  →        A ≅ B → B ≅ C
           ----------------
  →        A ≅ C

trans-≅ A B C ab bc = f , g , α , β
  where
    f1 : hom A B
    f1 = ∣ ab ∣
    f2 : hom B C
    f2 = ∣ bc ∣
    f : hom A C
    f = HCompClosed A B C f1 f2
```

```
    g1 : hom C B
    g1 = fst ∥ bc ∥
    g2 : hom B A
    g2 = fst ∥ ab ∥
    g : hom C A
    g = HCompClosed C B A g1 g2

    f1~g2 : ∣ f1 ∣ ∘ ∣ g2 ∣ ∼ ∣ 𝒾𝒹 B ∣
    f1~g2 = ∣ snd ∥ ab ∥ ∣

    g2~f1 : ∣ g2 ∣ ∘ ∣ f1 ∣ ∼ ∣ 𝒾𝒹 A ∣
    g2~f1 = ∥ snd ∥ ab ∥ ∥

    f2~g1 : ∣ f2 ∣ ∘ ∣ g1 ∣ ∼ ∣ 𝒾𝒹 C ∣
    f2~g1 = ∣ snd ∥ bc ∥ ∣

    g1~f2 : ∣ g1 ∣ ∘ ∣ f2 ∣ ∼ ∣ 𝒾𝒹 B ∣
    g1~f2 = ∥ snd ∥ bc ∥ ∥

    α : ∣ f ∣ ∘ ∣ g ∣ ∼ ∣ 𝒾𝒹 C ∣
    α x = (∣ f ∣ ∘ ∣ g ∣) x ≡⟨ 𝓇ℯ𝒻ℓ ⟩
          ∣ f2 ∣ ( (∣ f1 ∣ ∘ ∣ g2 ∣) (∣ g1 ∣ x)) ≡⟨ ap ∣ f2 ∣ (f1~g2 (∣ g1 ∣ x)) ⟩
          ∣ f2 ∣ ( ∣ 𝒾𝒹 B ∣ (∣ g1 ∣ x)) ≡⟨ 𝓇ℯ𝒻ℓ ⟩
          (∣ f2 ∣ ∘ ∣ g1 ∣) x          ≡⟨ f2~g1 x ⟩
          ∣ 𝒾𝒹 C ∣ x ∎
    β : ∣ g ∣ ∘ ∣ f ∣ ∼ ∣ 𝒾𝒹 A ∣
    β x = (ap ∣ g2 ∣ (g1~f2 (∣ f1 ∣ x))) · g2~f1 x

TRANS-≅ : {𝓠 𝓤 𝓦 : Universe}
          {A : Algebra 𝓠 𝑆}{B : Algebra 𝓤 𝑆}{C : Algebra 𝓦 𝑆}
 →         A ≅ B → B ≅ C
          ----------------
 →        A ≅ C
TRANS-≅ {A = A}{B = B}{C = C} = trans-≅ A B C

Trans-≅ : {𝓠 𝓤 𝓦 : Universe}
          (A : Algebra 𝓠 𝑆){B : Algebra 𝓤 𝑆}(C : Algebra 𝓦 𝑆)
 →         A ≅ B → B ≅ C
          ----------------
 →        A ≅ C
Trans-≅ A {B} C = trans-≅ A B C
```

5.5.3 Lift is an algebraic invariant

Fortunately, the lift operation preserves isomorphism (i.e., it's an "algebraic invariant"), which is why it's a workable solution to the "level hell" problem we mentioned earlier.

```
open Lift

--An algebra is isomorphic to its lift to a higher universe level
lift-alg-≅ : {𝓤 𝓦 : Universe}{A : Algebra 𝓤 𝑆} → A ≅ (lift-alg A 𝓦)
lift-alg-≅ = (lift , λ _ _ → 𝓇ℯ𝒻ℓ) ,
```

```
                        (lower , λ _ _ → refl) ,
                        (λ _ → refl) , (λ _ → refl)

lift-alg-hom : (𝓧 : Universe){𝓨 : Universe}
               (𝓩 : Universe){𝓦 : Universe}
               (A : Algebra 𝓧 S) (B : Algebra 𝓨 S)
 →             hom A B
               -------------------------
 →             hom (lift-alg A 𝓩) (lift-alg B 𝓦)
lift-alg-hom 𝓧 𝓩 {𝓦} A B (f , fhom) = lift ∘ f ∘ lower , γ
 where
  lh : is-homomorphism (lift-alg A 𝓩) A lower
  lh = λ _ _ → refl
  lABh : is-homomorphism (lift-alg A 𝓩) B (f ∘ lower)
  lABh = ∘-hom (lift-alg A 𝓩) A B {lower}{f} lh fhom
  Lh : is-homomorphism B (lift-alg B 𝓦) lift
  Lh = λ _ _ → refl
  γ : is-homomorphism (lift-alg A 𝓩) (lift-alg B 𝓦) (lift ∘ (f ∘ lower))
  γ = ∘-hom (lift-alg A 𝓩) B (lift-alg B 𝓦) {f ∘ lower}{lift} lABh Lh

lift-alg-iso : (𝓧 : Universe){𝓨 : Universe}
               (𝓩 : Universe){𝓦 : Universe}
               (A : Algebra 𝓧 S) (B : Algebra 𝓨 S)
 →             A ≅ B
               -------------------------
 →             (lift-alg A 𝓩) ≅ (lift-alg B 𝓦)

lift-alg-iso 𝓧 {𝓨} 𝓩 {𝓦} A B A≅B = TRANS-≅ (TRANS-≅ lA≅A A≅B) lift-alg-≅
 where
  lA≅A : (lift-alg A 𝓩) ≅ A
  lA≅A = sym-≅ lift-alg-≅
```

5.5.4 Lift associativity

The lift is also associative, up to isomorphism at least.

```
lift-alg-assoc : {𝓤 𝓦 𝓘 : Universe}{A : Algebra 𝓤 S}
 → lift-alg A (𝓦 ⊔ 𝓘) ≅ (lift-alg (lift-alg A 𝓦) 𝓘)
lift-alg-assoc {𝓤} {𝓦} {𝓘} {A} = TRANS-≅ (TRANS-≅ ζ lift-alg-≅) lift-alg-≅
 where
  ζ : lift-alg A (𝓦 ⊔ 𝓘) ≅ A
  ζ = sym-≅ lift-alg-≅

lift-alg-associative : {𝓤 𝓦 𝓘 : Universe}(A : Algebra 𝓤 S)
 → lift-alg A (𝓦 ⊔ 𝓘) ≅ (lift-alg (lift-alg A 𝓦) 𝓘)
lift-alg-associative{𝓤}{𝓦}{𝓘} A = lift-alg-assoc{𝓤}{𝓦}{𝓘}{A}
```

5.5.5 Products preserve isomorphisms

```
⊓≅ : global-dfunext → {𝓠 𝓤 𝓘 : Universe}
     {I : 𝓘 ̇}{𝒜 : I → Algebra 𝓠 S}{ℬ : I → Algebra 𝓤 S}
```

```
  → ((i : I) → (𝒜 i) ≅ (ℬ i))
    ----------------------------
  → ⨅ 𝒜 ≅ ⨅ ℬ

⨅≅ gfe {𝓠}{𝓤}{𝓘}{I}{𝒜}{ℬ} AB = γ
  where
   F : ∀ i → ∣ 𝒜 i ∣ → ∣ ℬ i ∣
   F i = ∣ fst (AB i) ∣
   Fhom : ∀ i → is-homomorphism (𝒜 i) (ℬ i) (F i)
   Fhom i = ∥ fst (AB i) ∥

   G : ∀ i → ∣ ℬ i ∣ → ∣ 𝒜 i ∣
   G i = fst ∣ snd (AB i) ∣
   Ghom : ∀ i → is-homomorphism (ℬ i) (𝒜 i) (G i)
   Ghom i = snd ∣ snd (AB i) ∣

   F∼G : ∀ i → (F i) ∘ (G i) ∼ (∣ 𝒾𝒹 (ℬ i) ∣)
   F∼G i = fst ∥ snd (AB i) ∥

   G∼F : ∀ i → (G i) ∘ (F i) ∼ (∣ 𝒾𝒹 (𝒜 i) ∣)
   G∼F i = snd ∥ snd (AB i) ∥

   ϕ : ∣ ⨅ 𝒜 ∣ → ∣ ⨅ ℬ ∣
   ϕ a i = F i (a i)

   ϕhom : is-homomorphism (⨅ 𝒜) (⨅ ℬ) ϕ
   ϕhom 𝑓 𝒂 = gfe (λ i → (Fhom i) 𝑓 (λ x → 𝒂 x i))

   ψ : ∣ ⨅ ℬ ∣ → ∣ ⨅ 𝒜 ∣
   ψ b i = ∣ fst ∥ AB i ∥ ∣ (b i)

   ψhom : is-homomorphism (⨅ ℬ) (⨅ 𝒜) ψ
   ψhom 𝑓 𝒃 = gfe (λ i → (Ghom i) 𝑓 (λ x → 𝒃 x i))

   ϕ∼ψ : ϕ ∘ ψ ∼ ∣ 𝒾𝒹 (⨅ ℬ) ∣
   ϕ∼ψ 𝒃 = gfe λ i → F∼G i (𝒃 i)

   ψ∼ϕ : ψ ∘ ϕ ∼ ∣ 𝒾𝒹 (⨅ 𝒜) ∣
   ψ∼ϕ 𝒂 = gfe λ i → G∼F i (𝒂 i)

   γ : ⨅ 𝒜 ≅ ⨅ ℬ
   γ = (ϕ , ϕhom) , ((ψ , ψhom) , ϕ∼ψ , ψ∼ϕ)
```

A nearly identical proof goes through for isomorphisms of lifted products.

```
lift-alg-⨅≅ : global-dfunext → {𝓠 𝓤 𝓘 𝓩 : Universe}
              {I : 𝓘 ̇}{𝒜 : I → Algebra 𝓠 𝑆}{ℬ : (Lift{𝓘}{𝓩} I) → Algebra 𝓤 𝑆}
 →            ((i : I) → (𝒜 i) ≅ (ℬ (lift i)))
              ---------------------------
 →            lift-alg (⨅ 𝒜) 𝓩 ≅ ⨅ ℬ

lift-alg-⨅≅ gfe {𝓠}{𝓤}{𝓘}{𝓩}{I}{𝒜}{ℬ} AB = γ
  where
```

```
F : ∀ i → ∣ 𝒜 i ∣ → ∣ ℬ (lift i) ∣
F i = ∣ fst (AB i) ∣
Fhom : ∀ i → is-homomorphism (𝒜 i) (ℬ (lift i)) (F i)
Fhom i = ∥ fst (AB i) ∥

G : ∀ i → ∣ ℬ (lift i) ∣ → ∣ 𝒜 i ∣
G i = fst ∣ snd (AB i) ∣
Ghom : ∀ i → is-homomorphism (ℬ (lift i)) (𝒜 i) (G i)
Ghom i = snd ∣ snd (AB i) ∣

F∼G : ∀ i → (F i) ∘ (G i) ∼ (∣ 𝑖𝑑 (ℬ (lift i)) ∣)
F∼G i = fst ∥ snd (AB i) ∥

G∼F : ∀ i → (G i) ∘ (F i) ∼ (∣ 𝑖𝑑 (𝒜 i) ∣)
G∼F i = snd ∥ snd (AB i) ∥

ϕ : ∣ ⨅ 𝒜 ∣ → ∣ ⨅ ℬ ∣
ϕ a i = F (lower i) (a (lower i))

ϕhom : is-homomorphism (⨅ 𝒜) (⨅ ℬ) ϕ
ϕhom 𝑓 𝒂 = gfe (λ i → (Fhom (lower i)) 𝑓 (λ x → 𝒂 x (lower i)))

ψ : ∣ ⨅ ℬ ∣ → ∣ ⨅ 𝒜 ∣
ψ b i = ∣ fst ∥ AB i ∥ ∣ (b (lift i))

ψhom : is-homomorphism (⨅ ℬ) (⨅ 𝒜) ψ
ψhom 𝑓 𝒃 = gfe (λ i → (Ghom i) 𝑓 (λ x → 𝒃 x (lift i)))

ϕ∼ψ : ϕ ∘ ψ ∼ ∣ 𝑖𝑑 (⨅ ℬ) ∣
ϕ∼ψ 𝒃 = gfe λ i → F∼G (lower i) (𝒃 i)

ψ∼ϕ : ψ ∘ ϕ ∼ ∣ 𝑖𝑑 (⨅ 𝒜) ∣
ψ∼ϕ 𝒂 = gfe λ i → G∼F i (𝒂 i)

A≅B : ⨅ 𝒜 ≅ ⨅ ℬ
A≅B = (ϕ , ϕhom) , ((ψ , ψhom) , ϕ∼ψ , ψ∼ϕ)

γ : lift-alg (⨅ 𝒜) 𝔛 ≅ ⨅ ℬ
γ = Trans-≅ (lift-alg (⨅ 𝒜) 𝔛) (⨅ ℬ) (sym-≅ lift-alg-≅) A≅B
```

5.5.6 Embedding tools

Here are some useful tools for working with embeddings.

```
embedding-lift-nat : {𝓠 𝓤 𝓘 : Universe} → hfunext 𝓘 𝓠 → hfunext 𝓘 𝓤
 →                    {I : 𝓘 ̇}{A : I → 𝓠 ̇}{B : I → 𝓤 ̇}
                      (h : Nat A B)
 →                    ((i : I) → is-embedding (h i))
                      ----------------------------
 →                    is-embedding(NatΠ h)

embedding-lift-nat hfiq hfiu h hem = NatΠ-is-embedding hfiq hfiu h hem
```

```
embedding-lift-nat' : {𝓠 𝓤 𝓘 : Universe} → hfunext 𝓘 𝓠 → hfunext 𝓘 𝓤
  →                   {I : 𝓘 ̇}{𝒜 : I → Algebra 𝓠 S}{ℬ : I → Algebra 𝓤 S}
                      (h : Nat (fst ∘ 𝒜) (fst ∘ ℬ))
  →                   ((i : I) → is-embedding (h i))
                      -------------------------------
  →                   is-embedding(NatΠ h)

embedding-lift-nat' hfiq hfiu h hem = NatΠ-is-embedding hfiq hfiu h hem

embedding-lift :      {𝓠 𝓤 𝓘 : Universe} → hfunext 𝓘 𝓠 → hfunext 𝓘 𝓤
  →                   {I : 𝓘 ̇} -- global-dfunext → {𝓠 𝓤 𝓘 : Universe}{I : 𝓘 ̇}
                      {𝒜 : I → Algebra 𝓠 S}{ℬ : I → Algebra 𝓤 S}
  →                   (h : ∀ i → ∣ 𝒜 i ∣ → ∣ ℬ i ∣)
  →                   ((i : I) → is-embedding (h i))
                      --------------------------------------------
  →                   is-embedding(λ (x : ∣ ⨅ 𝒜 ∣) (i : I) → (h i) (x i))
embedding-lift {𝓠} {𝓤} {𝓘} hfiq hfiu {I} {𝒜} {ℬ} h hem =
  embedding-lift-nat' {𝓠} {𝓤} {𝓘} hfiq hfiu {I} {𝒜} {ℬ} h hem
```

5.5.7 Isomorphism, intensionally

This is not used so much, and this section may be absent from future releases of the library.

```
--Isomorphism
_≅'_ : {𝓤 𝓦 : Universe} (𝑨 : Algebra 𝓤 S) (𝑩 : Algebra 𝓦 S) → 𝓞 ⊔ 𝓥 ⊔ 𝓤 ⊔ 𝓦 ̇
𝑨 ≅' 𝑩 = Σ f ꞉ (hom 𝑨 𝑩) , Σ g ꞉ (hom 𝑩 𝑨) , ((∣ f ∣ ∘ ∣ g ∣) ≡ ∣ 𝒾𝒹 𝑩 ∣) × ((∣ g ∣ ∘ ∣ f ∣) ≡ ∣ 𝒾𝒹 𝑨 ∣)
-- An algebra is (intensionally) isomorphic to itself
id≅' : {𝓤 : Universe} (𝑨 : Algebra 𝓤 S) → 𝑨 ≅' 𝑨
id≅' 𝑨 = 𝒾𝒹 𝑨 , 𝒾𝒹 𝑨 , 𝓇ℯ𝒻𝓁 , 𝓇ℯ𝒻𝓁

iso→embedding : {𝓤 𝓦 : Universe}{𝑨 : Algebra 𝓤 S}{𝑩 : Algebra 𝓦 S}
  → (ϕ : 𝑨 ≅ 𝑩) → is-embedding (fst ∣ ϕ ∣)
iso→embedding {𝓤}{𝓦}{𝑨}{𝑩} ϕ = γ
  where
    f : hom 𝑨 𝑩
    f = ∣ ϕ ∣
    g : hom 𝑩 𝑨
    g = ∣ snd ϕ ∣

    finv : invertible ∣ f ∣
    finv = ∣ g ∣ , (snd ∥ snd ϕ ∥ , fst ∥ snd ϕ ∥)

    γ : is-embedding ∣ f ∣
    γ = equivs-are-embeddings ∣ f ∣ (invertibles-are-equivs ∣ f ∣ finv)
```

5.6 Homomorphic Image Type

This section describes the UALib.Homomorphisms.HomomorphicImages module of the Agda UALib.

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

```
open import UALib.Algebras.Signatures using (Signature; 𝓞; 𝓥)
open import UALib.Prelude.Preliminaries using (global-dfunext)

module UALib.Homomorphisms.HomomorphicImages {𝑆 : Signature 𝓞 𝓥}{gfe : global-dfunext} where

open import UALib.Homomorphisms.Isomorphisms{𝑆 = 𝑆}{gfe} public
```

5.6.1 Images of a single algebra

We begin with what seems to be (for our purposes at least) the most useful way to represent, in Martin-Löf type theory, the class of **homomomrphic images** of an algebra.

```
HomImage : {𝓤 𝓦 : Universe}{𝑨 : Algebra 𝓤 𝑆}(𝑩 : Algebra 𝓦 𝑆)(ϕ : hom 𝑨 𝑩) → ∣ 𝑩 ∣ → 𝓤 ⊔ 𝓦 ̇
HomImage 𝑩 ϕ = λ b → Image ∣ ϕ ∣ ∋ b

HomImagesOf : {𝓤 𝓦 : Universe} → Algebra 𝓤 𝑆 → 𝓞 ⊔ 𝓥 ⊔ 𝓤 ⊔ 𝓦 ⁺ ̇
HomImagesOf {𝓤}{𝓦} 𝑨 = Σ 𝑩 ꞉ (Algebra 𝓦 𝑆) , Σ ϕ ꞉ (∣ 𝑨 ∣ → ∣ 𝑩 ∣) , is-homomorphism 𝑨 𝑩 ϕ × Epic ϕ
```

5.6.2 Images of a class of algebras

Here are a few more definitions, derived from the one above, that will come in handy.

```
_is-hom-image-of_ : {𝓤 𝓦 : Universe} (𝑩 : Algebra 𝓦 𝑆)
  → (𝑨 : Algebra 𝓤 𝑆) → 𝓞 ⊔ 𝓥 ⊔ 𝓤 ⊔ 𝓦 ⁺ ̇

_is-hom-image-of_ {𝓤}{𝓦} 𝑩 𝑨 = Σ 𝑪ϕ ꞉ (HomImagesOf{𝓤}{𝓦} 𝑨) , ∣ 𝑪ϕ ∣ ≅ 𝑩

_is-hom-image-of-class_ : {𝓤 : Universe} → Algebra 𝓤 𝑆 → Pred (Algebra 𝓤 𝑆)(𝓤 ⁺) → 𝓞 ⊔ 𝓥 ⊔ 𝓤 ⁺ ̇
_is-hom-image-of-class_ {𝓤} 𝑩 𝒦 = Σ 𝑨 ꞉ (Algebra 𝓤 𝑆) , (𝑨 ∈ 𝒦) × (𝑩 is-hom-image-of 𝑨)

HomImagesOfClass : {𝓤 : Universe} → Pred (Algebra 𝓤 𝑆) (𝓤 ⁺) → 𝓞 ⊔ 𝓥 ⊔ 𝓤 ⁺ ̇
HomImagesOfClass 𝒦 = Σ 𝑩 ꞉ (Algebra _ 𝑆) , (𝑩 is-hom-image-of-class 𝒦)

all-ops-in_and_commute-with : {𝓤 𝓦 : Universe}
                              (𝑨 : Algebra 𝓤 𝑆)(𝑩 : Algebra 𝓦 𝑆)
  →                           (∣ 𝑨 ∣ → ∣ 𝑩 ∣) → 𝓞 ⊔ 𝓥 ⊔ 𝓤 ⊔ 𝓦 ̇

all-ops-in 𝑨 and 𝑩 commute-with g = is-homomorphism 𝑨 𝑩 g
```

5.6.3 Lifting tools

```
open Lift

lift-function : (𝓧 : Universe){𝓨 : Universe}
                (𝓩 : Universe){𝓦 : Universe}
                (A : 𝓧 ̇)(B : 𝓨 ̇) → (f : A → B)
                -----------------------------------
  →             Lift{𝓧}{𝓩} A → Lift{𝓨}{𝓦} B

lift-function 𝓧 {𝓨} 𝓩 {𝓦} A B f = λ la → lift (f (lower la))


lift-of-alg-epic-is-epic : (𝓧 : Universe){𝓨 : Universe}
```

```
                        (𝔛 : Universe){𝔚 : Universe}
                        (A : Algebra 𝔛 S)(B : Algebra 𝔜 S)
                        (f : hom A B) → Epic ∣ f ∣
                        -----------------------------------------
 →                      Epic ∣ lift-alg-hom 𝔛 𝔛{𝔚} A B f ∣

lift-of-alg-epic-is-epic 𝔛 {𝔜} 𝔛 {𝔚} A B f fepi = lE
  where
   lA : Algebra (𝔛 ⊔ 𝔛) S
   lA = lift-alg A 𝔛
   lB : Algebra (𝔜 ⊔ 𝔚) S
   lB = lift-alg B 𝔚

   lf : hom (lift-alg A 𝔛) (lift-alg B 𝔚)
   lf = lift-alg-hom 𝔛 𝔛 A B f

   lE : (y : ∣ lB ∣ ) → Image ∣ lf ∣ ∋ y
   lE y = ξ
    where
     b : ∣ B ∣
     b = lower y

     ζ : Image ∣ f ∣ ∋ b
     ζ = fepi b

     a : ∣ A ∣
     a = Inv ∣ f ∣ b ζ

     η : y ≡ ∣ lf ∣ (lift a)
     η = y      ≡⟨ (intensionality lift∼lower) y ⟩
         lift b ≡⟨ ap lift (InvIsInv ∣ f ∣ (lower y) ζ)⁻¹ ⟩
         lift (∣ f ∣ a)  ≡⟨ (ap (λ - → lift (∣ f ∣ ( - a)))) (lower∼lift{𝔚 = 𝔚}) ⟩
         lift (∣ f ∣ ((lower{𝔚 = 𝔚} ∘ lift) a)) ≡⟨ refl ⟩
         (lift ∘ ∣ f ∣ ∘ lower{𝔚 = 𝔚}) (lift a) ≡⟨ refl ⟩
         ∣ lf ∣ (lift a) ∎
     ξ : Image ∣ lf ∣ ∋ y
     ξ = eq y (lift a) η


lift-alg-hom-image : {𝔛 𝔜 𝔛 𝔚 : Universe}
                     {A : Algebra 𝔛 S}{B : Algebra 𝔜 S}
 →                   B is-hom-image-of A
                     -----------------------------------
 →                   (lift-alg B 𝔚) is-hom-image-of (lift-alg A 𝔛)

lift-alg-hom-image {𝔛}{𝔜}{𝔛}{𝔚}{A}{B} ((C , ϕ , ϕhom , ϕepic) , C≅B) = γ
  where
   lA : Algebra (𝔛 ⊔ 𝔛) S
   lA = lift-alg A 𝔛
   lB lC : Algebra (𝔜 ⊔ 𝔚) S
   lB = lift-alg B 𝔚
```

```
lC = lift-alg C 𝒲

lϕ : hom lA lC
lϕ = (lift-alg-hom 𝒳 ℨ A C) (ϕ , ϕhom)

lϕepic : Epic | lϕ |
lϕepic = lift-of-alg-epic-is-epic 𝒳 ℨ A C (ϕ , ϕhom) ϕepic

lCϕ : HomImagesOf {𝒳 ⊔ ℨ}{𝒴 ⊔ 𝒲} lA
lCϕ = lC , | lϕ | , ∥ lϕ ∥ , lϕepic

lC≅lB : lC ≅ lB
lC≅lB = lift-alg-iso 𝒴 𝒲 C B C≅B

γ : lB is-hom-image-of lA
γ = lCϕ , lC≅lB
```

6 Terms

6.1 Basic Definitions

This section describes the UALib.Terms.Basic module of the Agda UALib.

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

open import UALib.Algebras using (Signature; 𝓞; 𝓥; Algebra; _↠_)
open import UALib.Prelude.Preliminaries using (global-dfunext; Universe; _̇)

module UALib.Terms.Basic
  {𝑆 : Signature 𝓞 𝓥}{gfe : global-dfunext}
  {𝕏 : {𝓤 𝓧 : Universe}{X : 𝓧 ̇ }(𝑨 : Algebra 𝓤 𝑆) → X ↠ 𝑨}
  where

open import UALib.Homomorphisms.HomomorphicImages{𝑆 = 𝑆}{gfe} hiding (Universe; _̇) public
```

6.1.1 The inductive type of terms

We define a type called Term which represents the type of terms of a given signature. As usual, the type $X : \mathcal{U}\,^{\cdot}$ 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

open Term
```

6.2 The Term Algebra

This section describes the UALib.Terms.Free module of the Agda UALib.

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

open import UALib.Algebras using (Signature; 𝓞; 𝓥; Algebra; _↠_)
open import UALib.Prelude.Preliminaries using (global-dfunext; Universe; _̇)

module UALib.Terms.Free
  {𝑆 : Signature 𝓞 𝓥}{gfe : global-dfunext}
  {𝕏 : {𝓤 𝓧 : Universe}{X : 𝓧 ̇ }(𝑨 : Algebra 𝓤 𝑆) → X ↠ 𝑨}
  where

open import UALib.Terms.Basic{𝑆 = 𝑆}{gfe}{𝕏} hiding (Algebra) public
```

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 **T** X. In the Agda this algebra can be defined quite simply, as one would hope and expect.

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

6.2.1 The universal property

The Term algebra is **absolutely free** (or "universal") **for** algebras in the signature S. That is, for every S-algebra $\mathbf{A}$,

1. every map $h : X \to |\mathbf{A}|$ lifts to a homomorphism from $\mathbf{T}\ X$ to $\mathbf{A}$, and
2. the induced homomorphism is unique.

```
--1.a. Every map (X → A) lifts.
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)

--1.b. The lift is (extensionally) a hom
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

--2. The lift to (free → A) is (extensionally) unique.
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)
```

6.2.2 Lifting and imaging tools

Next we note the easy fact that the lift induced by h_0 agrees with h_0 on X and that the lift is surjective if h_0 is.

```
lift-agrees-on-X : {𝓧 𝓤 : Universe}{X : 𝓧 ̇}
                   (𝑨 : Algebra 𝓤 𝑆)(h₀ : X → ∣ 𝑨 ∣)(x : X)
                   ----------------------------------------
 →                 h₀ x ≡ ∣ lift-hom 𝑨 h₀ ∣ (generator x)

lift-agrees-on-X _ h₀ x = 𝓇ℯ𝒻𝓁

lift-of-epi-is-epi : {𝓧 𝓤 : Universe}{X : 𝓧 ̇}
                     (𝑨 : Algebra 𝓤 𝑆)(h₀ : X → ∣ 𝑨 ∣)
                     ------------------------
 →                   Epic h₀ → Epic ∣ lift-hom 𝑨 h₀ ∣

lift-of-epi-is-epi {𝓧}{𝓤}{X} 𝑨 h₀ hE y = γ
```

```
  where
   h₀pre : Image h₀ ∋ y
   h₀pre = hE y

   h₀⁻¹y : X
   h₀⁻¹y = Inv h₀ y (hE y)

   η : y ≡ | lift-hom A h₀ | (generator h₀⁻¹y)
   η =
    y        ≡⟨ (InvIsInv h₀ y h₀pre)⁻¹ ⟩
    h₀ h₀⁻¹y ≡⟨ lift-agrees-on-X A h₀ h₀⁻¹y ⟩
    | lift-hom A h₀ | (generator h₀⁻¹y) ∎

   γ : Image | lift-hom A h₀ | ∋ y
   γ = eq y (generator h₀⁻¹y) η
```

Since it's absolutely free, **T** X is the domain of a homomorphism to any algebra we like. The following function makes it easy to lay our hands on such homomorphisms.

```
Thom-gen : {𝓧 𝓤 : Universe}{X : 𝓧 ̇} (C : Algebra 𝓤 S)
 →        Σ h ꞉ (hom (T X) C), Epic | h |
Thom-gen {𝓧}{𝓤}{X} C = h , lift-of-epi-is-epi C h₀ hE
  where
   h₀ : X → | C |
   h₀ = fst (𝕏 C)

   hE : Epic h₀
   hE = snd (𝕏 C)

   h : hom (T X) C
   h = lift-hom C h₀
```

6.3 Term Operations

This section describes the UALib.Terms.Operations module of the Agda UALib.

```
{-# OPTIONS --without-K --exact-split --safe #-}
open import UALib.Algebras using (Signature; 𝓞; 𝓥; Algebra; _↠_)
open import UALib.Prelude.Preliminaries using (global-dfunext; Universe; _̇)

module UALib.Terms.Operations
 {S : Signature 𝓞 𝓥}{gfe : global-dfunext}
 {𝕏 : {𝓤 𝓧 : Universe}{X : 𝓧 ̇ }(A : Algebra 𝓤 S) → X ↠ A}
  where

open import UALib.Terms.Free{S = S}{gfe}{𝕏} public
```

When we interpret a term in an algebra we call the result a **term operation**. Given a term p : Term and an algebra **A**, we denote by p ̇ **A** the **interpretation** of p in **A**. This is defined inductively as follows:

1. if p is x : X (a variable) and if a : X → | **A** | is a tuple of elements of | **A** |, then define (p ̇ **A**) a = a x;
2. if p = f s, where f : | S | is an operation symbol and s : ‖ S ‖ f → **T** X is an (‖ S ‖ f)-tuple of terms and a : X → | **A** | is a tuple from **A**, then define (p ̇ **A**) a = ((f s) ̇ **A**) a = (f ̂ **A**) λ i → ((s i) ̇ **A**) a

In the Agda UALib term interpretation is defined as follows.

```
_̇_ : {𝔛 𝓤 : Universe}{X : 𝔛 ̇ } → Term{𝔛}{X} → (𝑨 : Algebra 𝓤 𝑆) → (X → ∣ 𝑨 ∣) → ∣ 𝑨 ∣
((generator x) ̇ 𝑨) 𝒂 = 𝒂 x
((node f args) ̇ 𝑨) 𝒂 = (f ̂ 𝑨) λ i → (args i ̇ 𝑨) 𝒂
```

Observe that intepretation of a term is the same as free-lift (modulo argument order).

```
free-lift-interpretation : {𝔛 𝓤 : Universe}{X : 𝔛 ̇ }
                           (𝑨 : Algebra 𝓤 𝑆)(h : X → ∣ 𝑨 ∣)(p : Term)
 →                         (p ̇ 𝑨) h ≡ free-lift 𝑨 h p

free-lift-interpretation 𝑨 h (generator x) = 𝓇ℯ𝒻ℓ
free-lift-interpretation 𝑨 h (node f args) = ap (f ̂ 𝑨) (gfe λ i → free-lift-interpretation 𝑨 h (args i))

lift-hom-interpretation : {𝔛 𝓤 : Universe}{X : 𝔛 ̇ }
                          (𝑨 : Algebra 𝓤 𝑆)(h : X → ∣ 𝑨 ∣)(p : Term)
 →                        (p ̇ 𝑨) h ≡ ∣ lift-hom 𝑨 h ∣ p

lift-hom-interpretation = free-lift-interpretation
```

Here we want $(\boldsymbol{t} : X \to \mid \mathbf{T}(X) \mid) \to ((p\ \dot{}\ \mathbf{T}(X))\ \boldsymbol{t}) \equiv p\ \boldsymbol{t}$ …, but what is $(p\ \dot{}\ \mathbf{T}(X))\ \boldsymbol{t}$? By definition, it depends on the form of p as follows:

- if $p \equiv$ (generator x), then $(p\ \dot{}\ \mathbf{T}(X))\ \boldsymbol{t} \equiv$ ((generator x) $\dot{}\ \mathbf{T}(X))\ \boldsymbol{t} \equiv \boldsymbol{t}\ x$
- if $p \equiv$ (node f $args$), then $(p\ \dot{}\ \mathbf{T}(X))\ \boldsymbol{t} \equiv$ ((node f $args$) $\dot{}\ \mathbf{T}(X))\ \boldsymbol{t} \equiv (f\ \hat{}\ \mathbf{T}(X))\ \lambda\ i \to (args\ i\ \dot{}\ \mathbf{T}(X))\ \boldsymbol{t}$.

We claim that if $p : \mid \mathbf{T}(X) \mid$ then there exists $\mathfrak{p} : \mid \mathbf{T}(X) \mid$ and $\boldsymbol{t} : X \to \mid \mathbf{T}(X) \mid$ such that $p \equiv (\mathfrak{p}\ \dot{}\ \mathbf{T}(X))\ \boldsymbol{t}$. We prove this fact as follows.

```
term-op-interp1 : {𝔛 : Universe}{X : 𝔛 ̇}(f : ∣ 𝑆 ∣)(args : ∥ 𝑆 ∥ f → Term)
 →                node f args ≡ (f ̂ 𝑻 X) args
term-op-interp1 = λ f args → 𝓇ℯ𝒻ℓ

term-op-interp2 : {𝔛 : Universe}{X : 𝔛 ̇}(f : ∣ 𝑆 ∣){a1 a2 : ∥ 𝑆 ∥ f → Term{𝔛}{X}}
 →                a1 ≡ a2 → node f a1 ≡ node f a2
term-op-interp2 f a1≡a2 = ap (node f) a1≡a2

term-op-interp3 : {𝔛 : Universe}{X : 𝔛 ̇}(f : ∣ 𝑆 ∣){a1 a2 : ∥ 𝑆 ∥ f → Term}
 →                a1 ≡ a2 → node f a1 ≡ (f ̂ 𝑻 X) a2
term-op-interp3 f {a1}{a2} a1a2 = (term-op-interp2 f a1a2) ∙ (term-op-interp1 f a2)

term-gen : {𝔛 : Universe}{X : 𝔛 ̇}(p : ∣ 𝑻 X ∣) → Σ 𝓅 ꞉ ∣ 𝑻 X ∣ , p ≡ (𝓅 ̇ 𝑻 X) generator
term-gen (generator x) = (generator x) , 𝓇ℯ𝒻ℓ
term-gen (node f args) = node f (λ i → ∣ term-gen (args i) ∣) ,
                                   term-op-interp3 f (gfe λ i → ∥ term-gen (args i) ∥)

tg : {𝔛 : Universe}{X : 𝔛 ̇}(p : ∣ 𝑻 X ∣) → Σ 𝓅 ꞉ ∣ 𝑻 X ∣ , p ≡ (𝓅 ̇ 𝑻 X) generator
tg p = term-gen p

term-equality : {𝔛 : Universe}{X : 𝔛 ̇}(p q : ∣ 𝑻 X ∣)
 →              p ≡ q → (∀ t → (p ̇ 𝑻 X) t ≡ (q ̇ 𝑻 X) t)
term-equality p q (refl _) _ = refl _

term-equality' : {𝓤 𝔛 : Universe}{X : 𝔛 ̇}{𝑨 : Algebra 𝓤 𝑆}(p q : ∣ 𝑻 X ∣)
```

```
 →                    p ≡ q → (∀ a → (p ̇ A) a ≡ (q ̇ A) a)
term-equality' p q (refl _) _ = refl _

term-gen-agreement : {𝓧 : Universe}{X : 𝓧 ̇}(p : ∣ 𝑻 X ∣)
 →                    (p ̇ 𝑻 X) generator ≡ (∣ term-gen p ∣ ̇ 𝑻 X) generator
term-gen-agreement (generator x) = 𝓇ℯ𝒻𝓁
term-gen-agreement {𝓧}{X}(node f args) = ap (f ̂ 𝑻 X) (gfe λ x → term-gen-agreement (args x))

term-agreement : {𝓧 : Universe}{X : 𝓧 ̇}(p : ∣ 𝑻 X ∣)
 →                    p ≡ (p ̇ 𝑻 X) generator
term-agreement p = snd (term-gen p) ∙ (term-gen-agreement p)⁻¹
```

6.3.1 Interpretation of terms in product algebras

```
interp-prod : {𝓧 𝓤 : Universe} → funext 𝓥 𝓤
 → {X : 𝓧 ̇}(p : Term){I : 𝓤 ̇}
   (𝒜 : I → Algebra 𝓤 S)(x : X → ∀ i → ∣ (𝒜 i) ∣)
   --------------------------------------------------------
 → (p ̇ (⨅ 𝒜)) x ≡ (λ i → (p ̇ 𝒜 i) (λ j → x j i))

interp-prod _ (generator x₁) 𝒜 x = 𝓇ℯ𝒻𝓁

interp-prod fe (node f t) 𝒜 x =
  let IH = λ x₁ → interp-prod fe (t x₁) 𝒜 x in
    (f ̂ ⨅ 𝒜)(λ x₁ → (t x₁ ̇ ⨅ 𝒜) x)                    ≡⟨ ap (f ̂ ⨅ 𝒜)(fe IH) ⟩
    (f ̂ ⨅ 𝒜)(λ x₁ → (λ i₁ → (t x₁ ̇ 𝒜 i₁)(λ j₁ → x j₁ i₁))) ≡⟨ 𝓇ℯ𝒻𝓁 ⟩
    (λ i₁ → (f ̂ 𝒜 i₁) (λ x₁ → (t x₁ ̇ 𝒜 i₁) (λ j₁ → x j₁ i₁))) ∎

interp-prod2 : {𝓤 𝓧 : Universe} → global-dfunext
 → {X : 𝓧 ̇}(p : Term){I : 𝓤 ̇ }(𝒜 : I → Algebra 𝓤 S)
   ----------------------------------------------------------------------
 → (p ̇ ⨅ 𝒜) ≡ λ(args : X → ∣ ⨅ 𝒜 ∣) → (λ i → (p ̇ 𝒜 i)(λ x → args x i))

interp-prod2 _ (generator x₁) 𝒜 = 𝓇ℯ𝒻𝓁

interp-prod2 gfe {X} (node f t) 𝒜 = gfe λ (tup : X → ∣ ⨅ 𝒜 ∣) →
    let IH = λ x → interp-prod gfe (t x) 𝒜 in
    let tA = λ z → t z ̇ ⨅ 𝒜 in
      (f ̂ ⨅ 𝒜)(λ s → tA s tup)                   ≡⟨ ap (f ̂ ⨅ 𝒜)(gfe λ x → IH x tup) ⟩
      (f ̂ ⨅ 𝒜)(λ s → λ j → (t s ̇ 𝒜 j)(λ ℓ → tup ℓ j)) ≡⟨ 𝓇ℯ𝒻𝓁 ⟩
      (λ i → (f ̂ 𝒜 i)(λ s → (t s ̇ 𝒜 i)(λ ℓ → tup ℓ i))) ∎
```

6.4 Compatibility of Terms

This section describes the UALib.Terms.Compatibility module of the Agda UALib. Here we prove that every term commutes with every homomorphism and is compatible with every congruence.

```
{-# OPTIONS --without-K --exact-split --safe #-}
open import UALib.Algebras using (Signature; 𝓞; 𝓥; Algebra; _↠_)
open import UALib.Prelude.Preliminaries using (global-dfunext; Universe; _̇)
```

```
module UALib.Terms.Compatibility
  {𝑆 : Signature 𝓞 𝓥}{gfe : global-dfunext}
  {𝕏 : {𝓤 𝓧 : Universe}{X : 𝓧 ̇ }(𝑨 : Algebra 𝓤 𝑆) → X ↠ 𝑨}
  where

open import UALib.Terms.Operations{𝑆 = 𝑆}{gfe}{𝕏} public
```

6.4.1 Homomorphism compatibility

We first prove an extensional version of this fact.

```
comm-hom-term : {𝓤 𝓦 𝓧 : Universe} → funext 𝓥 𝓦
 →              {X : 𝓧 ̇}(𝑨 : Algebra 𝓤 𝑆) (𝑩 : Algebra 𝓦 𝑆)
                (h : hom 𝑨 𝑩) (t : Term) (a : X → ∣ 𝑨 ∣)
                ------------------------------------------
 →              ∣ h ∣ ((t ̇ 𝑨) a) ≡ (t ̇ 𝑩) (∣ h ∣ ∘ a)

comm-hom-term _ 𝑨 𝑩 h (generator x) a = 𝓻𝓮𝒻𝓵

comm-hom-term fe 𝑨 𝑩 h (node f args) a =
  ∣ h ∣((f ̂ 𝑨) λ i₁ → (args i₁ ̇ 𝑨) a)   ≡⟨ ∥ h ∥ f ( λ r → (args r ̇ 𝑨) a ) ⟩
  (f ̂ 𝑩)(λ i₁ → ∣ h ∣((args i₁ ̇ 𝑨) a)) ≡⟨ ap (_ ̂ 𝑩)(fe (λ i₁ → comm-hom-term fe 𝑨 𝑩 h (args i₁) a))⟩
  (f ̂ 𝑩)(λ r → (args r ̇ 𝑩)(∣ h ∣ ∘ a)) ∎
```

Here is an intensional version.

```
comm-hom-term-intensional : global-dfunext → {𝓤 𝓦 𝓧 : Universe}{X : 𝓧 ̇}
 →                          (𝑨 : Algebra 𝓤 𝑆) (𝑩 : Algebra 𝓦 𝑆)(h : hom 𝑨 𝑩) (t : Term)
                            ----------------------------------------
 →                          ∣ h ∣ ∘ (t ̇ 𝑨) ≡ (t ̇ 𝑩) ∘ (λ a → ∣ h ∣ ∘ a)

comm-hom-term-intensional gfe 𝑨 𝑩 h (generator x) = 𝓻𝓮𝒻𝓵

comm-hom-term-intensional gfe {X = X} 𝑨 𝑩 h (node f args) = γ
  where
   γ : ∣ h ∣ ∘ (λ a → (f ̂ 𝑨) (λ i → (args i ̇ 𝑨) a))
       ≡ (λ a → (f ̂ 𝑩)(λ i → (args i ̇ 𝑩) a)) ∘ _∘_ ∣ h ∣
   γ = (λ a → ∣ h ∣ ((f ̂ 𝑨)(λ i → (args i ̇ 𝑨) a))) ≡⟨ gfe (λ a → ∥ h ∥ f ( λ r → (args r ̇ 𝑨) a )) ⟩
       (λ a → (f ̂ 𝑩)(λ i → ∣ h ∣ ((args i ̇ 𝑨) a))) ≡⟨ ap (λ - → (λ a → (f ̂ 𝑩)(- a))) ih ⟩
       (λ a → (f ̂ 𝑩)(λ i → (args i ̇ 𝑩) a)) ∘ _∘_ ∣ h ∣ ∎
    where
     IH : (a : X → ∣ 𝑨 ∣)(i : ∥ 𝑆 ∥ f) → (∣ h ∣ ∘ (args i ̇ 𝑨)) a ≡ ((args i ̇ 𝑩) ∘ _∘_ ∣ h ∣) a
     IH a i = intensionality (comm-hom-term-intensional gfe 𝑨 𝑩 h (args i)) a

     ih : (λ a → (λ i → ∣ h ∣ ((args i ̇ 𝑨) a))) ≡ (λ a → (λ i → ((args i ̇ 𝑩) ∘ _∘_ ∣ h ∣) a))
     ih = gfe λ a → gfe λ i → IH a i
```

6.4.2 Congruence compatibility

If t : Term, θ : Con $\mathbf{A}$, then $a\ \theta\ b \to t(a)\ \theta\ t(b))$. The statement and proof of this obvious but important fact may be formalized in Agda as follows.

```
compatible-term : {𝓤 : Universe}{X : 𝓤 ̇}
                  (A : Algebra 𝓤 S)(t : Term{𝓤}{X})(θ : Con A)
                   -------------------------------------------
 →                compatible-fun (t ̇ A) ∣ θ ∣

compatible-term A (generator x) θ p = p x

compatible-term A (node f args) θ p = snd ∥ θ ∥ f λ x → (compatible-term A (args x) θ) p

compatible-term' : {𝓤 : Universe} {X : 𝓤 ̇}
                   (A : Algebra 𝓤 S)(t : Term{𝓤}{X}) (θ : Con A)
                   ----------------------------------------------
 →                 compatible-fun (t ̇ A) ∣ θ ∣

compatible-term' A (generator x) θ p = p x
compatible-term' A (node f args) θ p = snd ∥ θ ∥ f λ x → (compatible-term' A (args x) θ) p
```

7 Subalgebras

7.1 Types for Subuniverses

This section describes the UALib.Subalgebras.Subuniverses module of the Agda UALib. We show how to represent in Agda *subuniverses* of a given algebra or collection of algebras.

As usual, we start with the required options and imports.

```
{-# OPTIONS --without-K --exact-split --safe #-}
open import UALib.Algebras using (Signature; 𝓞; 𝓥; Algebra; _↠_)
open import UALib.Prelude.Preliminaries using (global-dfunext; Universe; _̇)

module UALib.Subalgebras.Subuniverses
  {𝑆 : Signature 𝓞 𝓥}{gfe : global-dfunext}
  {𝕏 : {𝓤 𝓧 : Universe}{X : 𝓧 ̇ }(𝑨 : Algebra 𝓤 𝑆) → X ↠ 𝑨}
  where

open import UALib.Terms.Compatibility{𝑆 = 𝑆}{gfe}{𝕏} public

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

SubunivAlg : {𝓠 𝓤 : Universe} (𝑨 : Algebra 𝓠 𝑆)(B : Pred ∣ 𝑨 ∣ 𝓤)
  → B ∈ Subuniverses 𝑨
  → Algebra (𝓠 ⊔ 𝓤) 𝑆
SubunivAlg 𝑨 B B∈SubA = Σ B , λ 𝑓 x → (𝑓 ̂ 𝑨)(∣_∣ ∘ x) ,
                                      B∈SubA 𝑓 (∣_∣ ∘ x)(∥_∥ ∘ x)

record Subuniverse {𝓠 𝓤 : Universe}{𝑨 : Algebra 𝓠 𝑆} : 𝓞 ⊔ 𝓥 ⊔ (𝓠 ⊔ 𝓤) ⁺ ̇ where
  constructor mksub
  field
    sset : Pred ∣ 𝑨 ∣ 𝓤
    isSub : sset ∈ Subuniverses 𝑨
```

7.2 Subuniverse Generation

This section describes the UALib.Subalgebras.Generation module of the Agda UALib. Here we define the inductive type of the subuniverse generated by a given collection of elements from the domain of an algebra, and prove that what we have defined is indeed the smallest subuniverse containing the given elements.

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

open import UALib.Algebras using (Signature; 𝓞; 𝓥; Algebra; _↠_)
open import UALib.Prelude.Preliminaries using (global-dfunext; Universe; _̇)


module UALib.Subalgebras.Generation
  {𝑆 : Signature 𝓞 𝓥}{gfe : global-dfunext}
  {𝕏 : {𝓤 𝓧 : Universe}{X : 𝓧 ̇ }(𝑨 : Algebra 𝓤 𝑆) → X ↠ 𝑨}
  where
```

```
open import UALib.Subalgebras.Subuniverses{𝑆 = 𝑆}{gfe}{𝕏} public

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

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

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

-- By induction on x ∈ Sg X, show x ∈ Y
sgIsSmallest _ _ _ X⊆Y (var v∈X) = X⊆Y v∈X

sgIsSmallest 𝑨 Y YIsSub X⊆Y (app f {a} ima⊆SgX) = app∈Y
  where
    -- First, show the args are in Y
    ima⊆Y : Im a ⊆ Y
    ima⊆Y i = sgIsSmallest 𝑨 Y YIsSub X⊆Y (ima⊆SgX i)

    --Since Y is a subuniverse of 𝑨, it contains the application
    app∈Y : (f ̂ 𝑨) a ∈ Y -- of f to said args.
    app∈Y = YIsSub f a ima⊆Y
```

7.3 Subuniverse Lemmas

This section describes the UALib.Subalgebras.Properties module of the Agda UALib. Here we formalize and prove a few basic properties of subuniverses.

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

open import UALib.Algebras using (Signature; 𝓞; 𝓥; Algebra; _↠_)
open import UALib.Prelude.Preliminaries using (global-dfunext; Universe; _̇)


module UALib.Subalgebras.Properties
  {𝑆 : Signature 𝓞 𝓥}{gfe : global-dfunext}
  {𝕏 : {𝓤 𝓧 : Universe}{X : 𝓧 ̇ }(𝑨 : Algebra 𝓤 𝑆) → X ↠ 𝑨}
  where


open import UALib.Subalgebras.Generation{𝑆 = 𝑆}{gfe}{𝕏} renaming (generator to ℊ) public

open import Relation.Unary using (⋂) public
```

```
sub-inter-is-sub : {𝓠 𝓤 : Universe}(𝑨 : Algebra 𝓠 𝑆)
                   (I : 𝓤 ̇)(𝒜 : I → Pred ∣ 𝑨 ∣ 𝓤)
  →                ((i : I) → 𝒜 i ∈ Subuniverses 𝑨)
    -------------------------------------
  → ⋂ I 𝒜 ∈ Subuniverses 𝑨

sub-inter-is-sub 𝑨 I 𝒜 Ai-is-Sub f a ima⊆⋂A = α
  where
    α : (f ̂ 𝑨) a ∈ ⋂ I 𝒜
    α i = Ai-is-Sub i f a λ j → ima⊆⋂A j i
```

7.3.1 Conservativity of term operations

```
sub-term-closed : {𝓧 𝓠 𝓤 : Universe}{X : 𝓧 ̇}(𝑨 : Algebra 𝓠 𝑆)(B : Pred ∣ 𝑨 ∣ 𝓤)
  → B ∈ Subuniverses 𝑨
  → (t : Term)(b : X → ∣ 𝑨 ∣)
  → (∀ x → b x ∈ B)
    ---------------------------
  → ((t ̇ 𝑨) b) ∈ B

sub-term-closed _ _ B≤A (generator x) b b∈B = b∈B x

sub-term-closed 𝑨 B B≤A (node f t) b b∈B =
  B≤A f (λ z → (t z ̇ 𝑨) b)
        (λ x → sub-term-closed 𝑨 B B≤A (t x) b b∈B)
```

7.3.2 Term images

```
data TermImage {𝓠 𝓤 : Universe}(𝑨 : Algebra 𝓠 𝑆)(Y : Pred ∣ 𝑨 ∣ 𝓤) : Pred ∣ 𝑨 ∣ (𝓞 ⊔ 𝓥 ⊔ 𝓠 ⊔ 𝓤) where
  var : ∀ {y : ∣ 𝑨 ∣} → y ∈ Y → y ∈ TermImage 𝑨 Y
  app : (f : ∣ 𝑆 ∣) (t : ∥ 𝑆 ∥ f → ∣ 𝑨 ∣) → (∀ i → t i ∈ TermImage 𝑨 Y)
      ---------------------------------------------------------------
   →              (f ̂ 𝑨) t ∈ TermImage 𝑨 Y

--1. TermImage is a subuniverse
TermImageIsSub : {𝓠 𝓤 : Universe}
                 {𝑨 : Algebra 𝓠 𝑆}{Y : Pred ∣ 𝑨 ∣ 𝓤}
                 -----------------------------------
  → TermImage 𝑨 Y ∈ Subuniverses 𝑨

TermImageIsSub = app -- λ f a x → app f a x

--2. Y ⊆ TermImageY
Y⊆TermImageY : {𝓠 𝓤 : Universe}
               {𝑨 : Algebra 𝓠 𝑆}{Y : Pred ∣ 𝑨 ∣ 𝓤}
               -----------------------------------
  →              Y ⊆ TermImage 𝑨 Y

Y⊆TermImageY {a} a∈Y = var a∈Y

-- 3. Sg^A(Y) is the smallest subuniverse containing Y
```

```
-- Proof: see `sgIsSmallest`

SgY⊆TermImageY : {𝓠 𝓤 : Universe}
                 (𝑨 : Algebra 𝓠 𝑆) (Y : Pred ∣ 𝑨 ∣ 𝓤)
                 --------------------------------------
 → Sg 𝑨 Y ⊆ TermImage 𝑨 Y

SgY⊆TermImageY 𝑨 Y = sgIsSmallest 𝑨 (TermImage 𝑨 Y) TermImageIsSub Y⊆TermImageY
```

7.4 Homomorphisms and Subuniverses

This section describes the UALib.Subalgebras.Homomorphisms module of the Agda UALib. Here we mechanize the interaction between homomorphisms and subuniverses—two central cogs of the universal algebra machine.

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

open import UALib.Algebras using (Signature; 𝓞; 𝓥; Algebra; _↠_)
open import UALib.Prelude.Preliminaries using (global-dfunext; Universe; _̇)

module UALib.Subalgebras.Homomorphisms
 {𝑆 : Signature 𝓞 𝓥}{gfe : global-dfunext}
 {𝕏 : {𝓤 𝓧 : Universe}{X : 𝓧 ̇ }(𝑨 : Algebra 𝓤 𝑆) → X ↠ 𝑨}
 where

open import UALib.Subalgebras.Properties{𝑆 = 𝑆}{gfe}{𝕏} public
```

7.4.1 Homomorphic images are subuniverses

The image of a homomorphism is a subuniverse of its codomain.

```
hom-image-is-sub : {𝓤 𝓦 : Universe} → global-dfunext
 → {𝑨 : Algebra 𝓤 𝑆} {𝑩 : Algebra 𝓦 𝑆} (ϕ : hom 𝑨 𝑩)
   -------------------------------------------------
 → (HomImage 𝑩 ϕ) ∈ Subuniverses 𝑩

hom-image-is-sub gfe {𝑨}{𝑩} ϕ f b b∈Imf = eq ((f ̂ 𝑩) b) ((f ̂ 𝑨) ar) γ
 where
  ar : ∥ 𝑆 ∥ f → ∣ 𝑨 ∣
  ar = λ x → Inv ∣ ϕ ∣(b x)(b∈Imf x)

  ζ : ∣ ϕ ∣ ∘ ar ≡ b
  ζ = gfe (λ x → InvIsInv ∣ ϕ ∣(b x)(b∈Imf x))

  γ : (f ̂ 𝑩) b ≡ ∣ ϕ ∣((f ̂ 𝑨)(λ x → Inv ∣ ϕ ∣(b x)(b∈Imf x)))

  γ = (f ̂ 𝑩) b ≡⟨ ap (f ̂ 𝑩)(ζ ⁻¹) ⟩
      (f ̂ 𝑩)(∣ ϕ ∣ ∘ ar) ≡⟨(∥ ϕ ∥ f ar)⁻¹ ⟩
      ∣ ϕ ∣((f ̂ 𝑨) ar) ∎
```

7.4.2 Uniqueness property for homomorphisms

Here we formalize the proof that homomorphisms are uniquely determined by their values on a generating set.

```
HomUnique : {𝓤 𝓦 : Universe} → funext 𝓥 𝓤 → {𝑨 𝑩 : Algebra 𝓤 𝑆}
            (X : Pred ∣ 𝑨 ∣ 𝓤) (g h : hom 𝑨 𝑩)
 →          (∀ (x : ∣ 𝑨 ∣) → x ∈ X → ∣ g ∣ x ≡ ∣ h ∣ x)
            ---------------------------------------------
 →          (∀ (a : ∣ 𝑨 ∣) → a ∈ Sg 𝑨 X → ∣ g ∣ a ≡ ∣ h ∣ a)

HomUnique _ _ _ _ gx≡hx a (var x) = (gx≡hx) a x

HomUnique {𝓤}{𝓦} fe {𝑨}{𝑩} X g h gx≡hx a (app 𝑓 {𝒂} ima⊆SgX) =
  ∣ g ∣ ((𝑓 ̂ 𝑨) 𝒂)   ≡⟨ ∥ g ∥ 𝑓 𝒂 ⟩
  (𝑓 ̂ 𝑩)(∣ g ∣ ∘ 𝒂 ) ≡⟨ ap (𝑓 ̂ 𝑩)(fe induction-hypothesis) ⟩
  (𝑓 ̂ 𝑩)(∣ h ∣ ∘ 𝒂)  ≡⟨ ( ∥ h ∥ 𝑓 𝒂 )⁻¹ ⟩
  ∣ h ∣ ((𝑓 ̂ 𝑨) 𝒂 ) ∎
  where induction-hypothesis = λ x → HomUnique{𝓤}{𝓦} fe {𝑨}{𝑩} X g h gx≡hx (𝒂 x) ( ima⊆SgX x )
```

7.5 Types for Subalgebra

This section describes the UALib.Subalgebras.Subalgebras module of the Agda UALib. Here we define a *subalgebra* of an algebra as well as the collection of all subalgebras of a given class of algebras.

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

open import UALib.Algebras using (Signature; 𝓞; 𝓥; Algebra; _↠_)
open import UALib.Prelude.Preliminaries using (global-dfunext; Universe; _̇)

module UALib.Subalgebras.Subalgebras
  {𝑆 : Signature 𝓞 𝓥}{gfe : global-dfunext}
  {𝕏 : {𝓤 𝓧 : Universe}{X : 𝓧 ̇ }(𝑨 : Algebra 𝓤 𝑆) → X ↠ 𝑨}
  where

open import UALib.Subalgebras.Homomorphisms {𝑆 = 𝑆}{gfe}{𝕏} public
open import UALib.Prelude.Preliminaries using (∘-embedding; id-is-embedding)


_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

SUBALGEBRA : {𝓤 𝓦 : Universe} → Algebra 𝓦 𝑆 → 𝓞 ⊔ 𝓥 ⊔ 𝓦 ⊔ 𝓤 ⁺ ̇
SUBALGEBRA {𝓤} 𝑨 = Σ 𝑩 ꞉ (Algebra 𝓤 𝑆) , 𝑩 IsSubalgebraOf 𝑨

subalgebra : {𝓤 𝓦 : Universe} → Algebra 𝓤 𝑆 → 𝓞 ⊔ 𝓥 ⊔ 𝓤 ⊔ 𝓦 ⁺ ̇
subalgebra {𝓤}{𝓦} 𝑨 = Σ 𝑩 ꞉ (Algebra 𝓦 𝑆) , 𝑩 IsSubalgebraOf 𝑨

Subalgebra : {𝓤 : Universe} → Algebra 𝓤 𝑆 → 𝓞 ⊔ 𝓥 ⊔ 𝓤 ⁺ ̇
Subalgebra {𝓤} = SUBALGEBRA {𝓤}{𝓤}

getSub : {𝓤 𝓦 : Universe}{𝑨 : Algebra 𝓦 𝑆} → SUBALGEBRA{𝓤}{𝓦} 𝑨 → Algebra 𝓤 𝑆
getSub SA = ∣ SA ∣
```

7.5.1 Examples

The equalizer of two homomorphisms is a subuniverse.

```
EH-is-subuniverse : {𝓤 𝓦 : Universe} → funext 𝓥 𝓦 →
                    {A : Algebra 𝓤 S}{B : Algebra 𝓦 S}
                    (g h : hom A B) → Subuniverse {A = A}

EH-is-subuniverse fe {A} {B} g h = mksub (EH {B = B} g h) λ f a x → EH-is-closed fe {A}{B} g h {f} a x
```

Observe that the type universe level 𝓞 ⊔ 𝓥 ⊔ 𝓤 ⁺ arises quite often throughout the ualib since it is the level of the type Algebra 𝓤 S of an algebra in the signature S and domain of type 𝓤 ̇. Let us define, once and for all, a simple notation for this universe level.

```
OV : Universe → Universe
OV 𝓤 = 𝓞 ⊔ 𝓥 ⊔ 𝓤 ⁺
```

To clean up and simplify the presentation, we will sometimes write OV 𝓤 in place of 𝓞 ⊔ 𝓥 ⊔ 𝓤 ⁺.

7.5.2 Subalgebras of a class

```
_IsSubalgebraOfClass_ : {𝓤 𝓠 𝓦 : Universe}(B : Algebra 𝓤 S)
  → Pred (Algebra 𝓠 S) 𝓦 → 𝓞 ⊔ 𝓥 ⊔ 𝓦 ⊔ (𝓤 ⊔ 𝓠) ⁺ ̇
_IsSubalgebraOfClass_ {𝓤} B 𝒦 = Σ A ꞉ (Algebra _ S) , Σ SA ꞉ (SUBALGEBRA{𝓤} A) , (A ∈ 𝒦) × (B ≅ ∣ SA ∣)

_is-subalgebra-of-class_ : {𝓤 𝓦 : Universe}(B : Algebra (𝓤 ⊔ 𝓦) S)
  → Pred (Algebra 𝓤 S) (OV 𝓤) → (OV (𝓤 ⊔ 𝓦)) ̇
_is-subalgebra-of-class_ {𝓤}{𝓦} B 𝒦 = Σ A ꞉ (Algebra 𝓤 S) , Σ SA ꞉ (subalgebra{𝓤}{𝓦} A) , (A ∈ 𝒦) × (B ≅ ∣ SA ∣)

SUBALGEBRAOFCLASS : {𝓤 𝓠 𝓦 : Universe} → Pred (Algebra 𝓠 S) 𝓦 → 𝓞 ⊔ 𝓥 ⊔ 𝓦 ⊔ (𝓠 ⊔ 𝓤) ⁺ ̇
SUBALGEBRAOFCLASS {𝓤} 𝒦 = Σ B ꞉ (Algebra 𝓤 S) , B IsSubalgebraOfClass 𝒦

SubalgebraOfClass : {𝓤 𝓠 : Universe} → Pred (Algebra 𝓠 S)(𝓞 ⊔ 𝓥 ⊔ 𝓠 ⁺) → 𝓞 ⊔ 𝓥 ⊔ (𝓠 ⊔ 𝓤) ⁺ ̇
SubalgebraOfClass {𝓤}{𝓠} = SUBALGEBRAOFCLASS {𝓤}{𝓠}{𝓞 ⊔ 𝓥 ⊔ 𝓠 ⁺}

getSubOfClass : {𝓤 𝓠 𝓦 : Universe}{𝒦 : Pred (Algebra 𝓠 S) 𝓦} → SUBALGEBRAOFCLASS 𝒦 → Algebra 𝓤 S
getSubOfClass SAC = ∣ SAC ∣

SUBALGEBRAOFCLASS' : {𝓤 𝓠 𝓦 : Universe} → Pred (Algebra 𝓠 S) 𝓦 → 𝓞 ⊔ 𝓥 ⊔ 𝓦 ⊔ (𝓠 ⊔ 𝓤) ⁺ ̇
SUBALGEBRAOFCLASS' {𝓤}{𝓠} 𝒦 = Σ A ꞉ (Algebra 𝓠 S) , (A ∈ 𝒦) × SUBALGEBRA{𝓤}{𝓠} A
```

7.5.2.1 Syntactic sugar

We use the symbol ≤ to denote the subalgebra relation.

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

7.5.3 Subalgebra lemmata

```
--Transitivity of IsSubalgebra (explicit args)
TRANS-≤ : {𝓧 𝓨 𝓩 : Universe}(A : Algebra 𝓧 S)(B : Algebra 𝓨 S)(C : Algebra 𝓩 S)
  → B ≤ A → C ≤ B
        ----------------------
```

```
  → C ≤ A

TRANS-≤ A B C BA CB =
  ∣ BA ∣ ∘ ∣ CB ∣ , ∘-embedding (fst ∥ BA ∥) (fst ∥ CB ∥) , ∘-hom C B A {∣ CB ∣}{∣ BA ∣}(snd ∥ CB ∥) (snd ∥ BA ∥)

--Transitivity of IsSubalgebra (implicit args)
Trans-≤ : {𝓧 𝓨 𝓩 : Universe}(A : Algebra 𝓧 𝑆){B : Algebra 𝓨 𝑆}(C : Algebra 𝓩 𝑆)
  → B ≤ A → C ≤ B → C ≤ A
Trans-≤ A {B} C = TRANS-≤ A B C

--Transitivity of IsSubalgebra (implicit args)
trans-≤ : {𝓧 𝓨 𝓩 : Universe}{A : Algebra 𝓧 𝑆}{B : Algebra 𝓨 𝑆}{C : Algebra 𝓩 𝑆}
  → B ≤ A → C ≤ B → C ≤ A
trans-≤ {A = A}{B = B}{C = C} = TRANS-≤ A B C
transitivity-≤ : {𝓧 𝓨 𝓩 : Universe}(A : Algebra 𝓧 𝑆){B : Algebra 𝓨 𝑆}{C : Algebra 𝓩 𝑆}
  → A ≤ B → B ≤ C → A ≤ C
transitivity-≤ A {B}{C} A≤B B≤C = ∣ B≤C ∣ ∘ ∣ A≤B ∣ , ∘-embedding (fst ∥ B≤C ∥) (fst ∥ A≤B ∥) , ∘-hom A B C {∣ A≤B ∣}{∣ B≤C ∣}(snd

--Reflexivity of IsSubalgebra (explicit arg)
REFL-≤ : {𝓤 : Universe}(A : Algebra 𝓤 𝑆) → A ≤ A
REFL-≤ A = id ∣ A ∣ , id-is-embedding , id-is-hom

--Reflexivity of IsSubalgebra (implicit arg)
refl-≤ : {𝓤 : Universe}{A : Algebra 𝓤 𝑆} → A ≤ A
refl-≤ {A = A} = REFL-≤ A

--Reflexivity of IsSubalgebra (explicit arg)
ISO-≤ : {𝓧 𝓨 𝓩 : Universe}(A : Algebra 𝓧 𝑆)(B : Algebra 𝓨 𝑆)(C : Algebra 𝓩 𝑆)
  → B ≤ A → C ≅ B
        ---------------------
  → C ≤ A
ISO-≤ A B C B≤A C≅B = h , hemb , hhom
  where
   f : ∣ C ∣ → ∣ B ∣
   f = fst ∣ C≅B ∣
   g : ∣ B ∣ → ∣ A ∣
   g = ∣ B≤A ∣
   h : ∣ C ∣ → ∣ A ∣
   h = g ∘ f

   hemb : is-embedding h
   hemb = ∘-embedding (fst ∥ B≤A ∥) (iso→embedding C≅B)

   hhom : is-homomorphism C A h
   hhom = ∘-hom C B A {f}{g} (snd ∣ C≅B ∣) (snd ∥ B≤A ∥)

Iso-≤ : {𝓧 𝓨 𝓩 : Universe}(A : Algebra 𝓧 𝑆){B : Algebra 𝓨 𝑆}(C : Algebra 𝓩 𝑆)
  → B ≤ A → C ≅ B → C ≤ A
Iso-≤ A {B} C = ISO-≤ A B C

iso-≤ : {𝓧 𝓨 𝓩 : Universe}{A : Algebra 𝓧 𝑆}{B : Algebra 𝓨 𝑆}(C : Algebra 𝓩 𝑆)
```

```
  → B ≤ A → C ≅ B → C ≤ A
iso-≤ {A = A} {B = B} C = ISO-≤ A B C

trans-≤-≅ : {𝔛 𝔜 𝔛 : Universe}(A : Algebra 𝔛 S){B : Algebra 𝔜 S}(C : Algebra ℨ S)
  → A ≤ B → A ≅ C → C ≤ B
trans-≤-≅ {𝔛}{𝔜}{ℨ} A {B} C A≤B B≅C = ISO-≤ B A C A≤B (sym-≅ B≅C)

TRANS-≤-≅ : {𝔛 𝔜 ℨ : Universe}(A : Algebra 𝔛 S){B : Algebra 𝔜 S}(C : Algebra ℨ S)
  → A ≤ B → B ≅ C → A ≤ C
TRANS-≤-≅ {𝔛}{𝔜}{ℨ} A {B} C A≤B B≅C = h , hemb , hhom
  where
   f : ∣ A ∣ → ∣ B ∣
   f = ∣ A≤B ∣
   g : ∣ B ∣ → ∣ C ∣
   g = fst ∣ B≅C ∣

   h : ∣ A ∣ → ∣ C ∣
   h = g ∘ f

   hemb : is-embedding h
   hemb = ∘-embedding (iso→embedding B≅C)(fst ∥ A≤B ∥)

   hhom : is-homomorphism A C h
   hhom = ∘-hom A B C {f}{g} (snd ∥ A≤B ∥) (snd ∣ B≅C ∣) -- ISO-≤ A B C A≤B B≅C

mono-≤ : {𝓤 𝓠 𝓦 : Universe}(B : Algebra 𝓤 S){𝒦 𝒦' : Pred (Algebra 𝓠 S) 𝓦}
  → 𝒦 ⊆ 𝒦' → B IsSubalgebraOfClass 𝒦 → B IsSubalgebraOfClass 𝒦'
mono-≤ B KK' KB = ∣ KB ∣ , fst ∥ KB ∥ , KK' (∣ snd ∥ KB ∥ ∣) , ∥ (snd ∥ KB ∥) ∥

lift-alg-is-sub : {𝓤 : Universe}{𝒦 : Pred (Algebra 𝓤 S) (OV 𝓤)} {B : Algebra 𝓤 S}
  → B IsSubalgebraOfClass 𝒦
  → (lift-alg B 𝓤) IsSubalgebraOfClass 𝒦
lift-alg-is-sub {𝓤}{𝒦}{B} (A , (sa , (KA , B≅sa))) = A , sa , KA , trans-≅ _ _ _ (sym-≅ lift-alg-≅) B≅sa

lift-alg-lift-≤-lower : {𝔛 𝔜 ℨ : Universe}(A : Algebra 𝔛 S){B : Algebra 𝔜 S}
  → B ≤ A → (lift-alg B ℨ) ≤ A
lift-alg-lift-≤-lower {𝔛}{𝔜}{ℨ} A {B} B≤A = iso-≤{𝔛}{𝔜}{ℨ = (𝔜 ⊔ ℨ)}{A}{B} (lift-alg B ℨ) B≤A (sym-≅ lift-alg-≅)

lift-alg-lower-≤-lift : {𝔛 𝔜 ℨ : Universe}(A : Algebra 𝔛 S){B : Algebra 𝔜 S}
  → B ≤ A → B ≤ (lift-alg A ℨ)
lift-alg-lower-≤-lift {𝔛}{𝔜}{ℨ} A {B} B≤A = γ
  where
   lA : Algebra (𝔛 ⊔ ℨ) S
   lA = lift-alg A ℨ

   A≅lA : A ≅ lA
   A≅lA = lift-alg-≅

   f : ∣ B ∣ → ∣ A ∣
   f = ∣ B≤A ∣

   g : ∣ A ∣ → ∣ lA ∣
```

```
    g = ≅-map A≅lA

    h : ∣ B ∣ → ∣ lA ∣
    h = g ∘ f

    hemb : is-embedding h
    hemb = ∘-embedding (≅-map-is-embedding A≅lA) (fst ∥ B≤A ∥)

    hhom : is-homomorphism B lA h
    hhom = ∘-hom B A lA {f}{g} (snd ∥ B≤A ∥) (snd ∣ A≅lA ∣)

    γ : B IsSubalgebraOf lift-alg A 𝔛
    γ = h , hemb , hhom

lift-alg-sub-lift : {𝓤 𝓦 : Universe}(A : Algebra 𝓤 S){C : Algebra (𝓤 ⊔ 𝓦) S}
 → C ≤ A → C ≤ (lift-alg A 𝓦)
lift-alg-sub-lift {𝓤}{𝓦} A {C} C≤A = γ
  where
    lA : Algebra (𝓤 ⊔ 𝓦) S
    lA = lift-alg A 𝓦

    A≅lA : A ≅ lA
    A≅lA = lift-alg-≅

    f : ∣ C ∣ → ∣ A ∣
    f = ∣ C≤A ∣

    g : ∣ A ∣ → ∣ lA ∣
    g = ≅-map A≅lA

    h : ∣ C ∣ → ∣ lA ∣
    h = g ∘ f

    hemb : is-embedding h
    hemb = ∘-embedding (≅-map-is-embedding A≅lA) (fst ∥ C≤A ∥)

    hhom : is-homomorphism C lA h
    hhom = ∘-hom C A lA {f}{g} (snd ∥ C≤A ∥) (snd ∣ A≅lA ∣)

    γ : C IsSubalgebraOf lift-alg A 𝓦
    γ = h , hemb , hhom

lift-alg-≤ lift-alg-lift-≤-lift : {𝓧 𝓨 𝔛 𝓦 : Universe}(A : Algebra 𝓧 S){B : Algebra 𝓨 S}
 → A ≤ B → (lift-alg A 𝔛) ≤ (lift-alg B 𝓦)
lift-alg-≤ {𝓧}{𝓨}{𝔛}{𝓦} A {B} A≤B =
  transitivity-≤ lA {B}{lB} (transitivity-≤ lA {A}{B} lA≤A A≤B) B≤lB
  where
    lA : Algebra (𝓧 ⊔ 𝔛) S
    lA = (lift-alg A 𝔛)
    lB : Algebra (𝓨 ⊔ 𝓦) S
    lB = (lift-alg B 𝓦)
    lA≤A : lA ≤ A
```

```
  lA≤A = lift-alg-lift-≤-lower A {A} refl-≤
  B≤lB : B ≤ lB
  B≤lB = lift-alg-lower-≤-lift B {B} refl-≤

lift-alg-lift-≤-lift = lift-alg-≤ -- (alias)
```

7.6 WWMD

This section describes the UALib.Subalgebras.WWMD module of the Agda UALib. In his Type Topology library, Martin Escardo gives a nice formalization of the notion of subgroup and its properties. In this module we merely do for general algebras what Martin does for groups.

This is our first foray into univalent foundations, and our first chance to put Voevodsky's univalence axiom to work.

The present module is called WWMD.[7] Evidently, as one sees from the lengthy import statement that starts with open import Prelude.Preliminaries, we import many definitions and theorems from the Type Topology library here. Most of these will not be discussed. (See www.cs.bham.ac.uk/ mhe/HoTT-UF-in-Agda-Lecture-Notes to learn about the imported modules.)

The WWMD module can be safely skipped. As of 22 Jan 2021, it plays no role in any of the other modules of the Agda UALib.

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

open import UALib.Algebras using (Signature; 𝓞; 𝓥; Algebra; _↠_)
open import UALib.Prelude.Preliminaries using (global-dfunext; Universe; _̇)

module UALib.Subalgebras.WWMD
 {S : Signature 𝓞 𝓥}{gfe : global-dfunext}
 {𝕏 : {𝓤 𝓧 : Universe}{X : 𝓧 ̇ }(A : Algebra 𝓤 S) → X ↠ A}
 where

open import UALib.Subalgebras.Homomorphisms {S = S}{gfe}{𝕏} public
open import UALib.Prelude.Preliminaries using (∘-embedding; id-is-embedding; Univalence; Π-is-subsingleton;
 ∈₀-is-subsingleton; pr₁-embedding; embedding-gives-ap-is-equiv; equiv-to-subsingleton; powersets-are-sets';
 lr-implication; rl-implication; subset-extensionality'; inverse; ×-is-subsingleton; _≃_;
 logically-equivalent-subsingletons-are-equivalent; _•_)

module mhe_subgroup_generalization {𝓦 : Universe} {A : Algebra 𝓦 S} (ua : Univalence) where

 op-closed : (∣ A ∣ → 𝓦 ̇) → 𝓞 ⊔ 𝓥 ⊔ 𝓦 ̇
 op-closed B = (f : ∣ S ∣)(a : ∥ S ∥ f → ∣ A ∣)
  → ((i : ∥ S ∥ f) → B (a i)) → B ((f ̂ A) a)

 subuniverse : 𝓞 ⊔ 𝓥 ⊔ 𝓦 ⁺ ̇
 subuniverse = Σ B ꞉ (𝓟 ∣ A ∣) , op-closed ( _∈₀ B)

 being-op-closed-is-subsingleton : (B : 𝓟 ∣ A ∣)
  → is-subsingleton (op-closed ( _∈₀ B ))
 being-op-closed-is-subsingleton B = Π-is-subsingleton gfe
```

[7] For lack of a better alternative, we named this module with the acronym for "What would Martin do?"

```
    (λ f → Π-is-subsingleton gfe
      (λ a → Π-is-subsingleton gfe
        (λ _ → ∈₀-is-subsingleton B ((f ̂ 𝐀) a))))

pr₁-is-embedding : is-embedding ∣_∣
pr₁-is-embedding = pr₁-embedding being-op-closed-is-subsingleton

--so equality of subalgebras is equality of their underlying
--subsets in the powerset:
ap-pr₁ : (B C : subuniverse) → B ≡ C → ∣ B ∣ ≡ ∣ C ∣
ap-pr₁ B C = ap ∣_∣

ap-pr₁-is-equiv : (B C : subuniverse) → is-equiv (ap-pr₁ B C)
ap-pr₁-is-equiv =
  embedding-gives-ap-is-equiv ∣_∣ pr₁-is-embedding

subuniverse-is-a-set : is-set subuniverse
subuniverse-is-a-set B C = equiv-to-subsingleton
                              (ap-pr₁ B C , ap-pr₁-is-equiv B C)
                              (powersets-are-sets' ua ∣ B ∣ ∣ C ∣)

subuniverse-equality-gives-membership-equiv : (B C : subuniverse)
  → B ≡ C
    -----------------------------------
  → ( x : ∣ 𝐀 ∣ ) → (x ∈₀ ∣ B ∣) ⇔ (x ∈₀ ∣ C ∣)
subuniverse-equality-gives-membership-equiv B C B≡C x =
  transport (λ - → x ∈₀ ∣ - ∣) B≡C ,
    transport (λ - → x ∈₀ ∣ - ∣ ) ( B≡C ⁻¹ )

membership-equiv-gives-carrier-equality : (B C : subuniverse)
  → ((x : ∣ 𝐀 ∣) → x ∈₀ ∣ B ∣ ⇔ x ∈₀ ∣ C ∣)
   -----------------------------------------
  → ∣ B ∣ ≡ ∣ C ∣
membership-equiv-gives-carrier-equality B C φ =
  subset-extensionality' ua α β
    where
     α : ∣ B ∣ ⊆₀ ∣ C ∣
     α x = lr-implication (φ x)

     β : ∣ C ∣ ⊆₀ ∣ B ∣
     β x = rl-implication (φ x)

membership-equiv-gives-subuniverse-equality : (B C : subuniverse)
  → (( x : ∣ 𝐀 ∣ ) → x ∈₀ ∣ B ∣ ⇔ x ∈₀ ∣ C ∣)
    ---------------------------------------
  →                B ≡ C
membership-equiv-gives-subuniverse-equality B C =
  inverse (ap-pr₁ B C)
  (ap-pr₁-is-equiv B C)
        ∘ (membership-equiv-gives-carrier-equality B C)
```

```
membership-equiv-is-subsingleton : (B C : subuniverse)
  →        is-subsingleton (( x : ∣ 𝐀 ∣) → x ∈₀ ∣ B ∣ ⇔ x ∈₀ ∣ C ∣)
membership-equiv-is-subsingleton B C =
  Π-is-subsingleton gfe
   (λ x → ×-is-subsingleton
    (Π-is-subsingleton gfe (λ _ → ∈₀-is-subsingleton ∣ C ∣ x ))
      (Π-is-subsingleton gfe (λ _ → ∈₀-is-subsingleton ∣ B ∣ x )))

subuniverse-equality : (B C : subuniverse)
  →        (B ≡ C) ≃ ((x : ∣ 𝐀 ∣) → (x ∈₀ ∣ B ∣) ⇔ (x ∈₀ ∣ C ∣))

subuniverse-equality B C =
  logically-equivalent-subsingletons-are-equivalent _ _
    (subuniverse-is-a-set B C)
     (membership-equiv-is-subsingleton B C)
      (subuniverse-equality-gives-membership-equiv B C ,
       membership-equiv-gives-subuniverse-equality B C)

carrier-equality-gives-membership-equiv : (B C : subuniverse)
  → ∣ B ∣ ≡ ∣ C ∣
                    ----------------------------------------
  → ( ( x : ∣ 𝐀 ∣ ) → x ∈₀ ∣ B ∣ ⇔ x ∈₀ ∣ C ∣ )
carrier-equality-gives-membership-equiv B C (refl _) x = id , id

--so we have...
carrier-equiv : (B C : subuniverse)
  →        ((x : ∣ 𝐀 ∣) → x ∈₀ ∣ B ∣ ⇔ x ∈₀ ∣ C ∣) ≃ (∣ B ∣ ≡ ∣ C ∣)
carrier-equiv B C =
  logically-equivalent-subsingletons-are-equivalent _ _
   (membership-equiv-is-subsingleton B C)
    (powersets-are-sets' ua ∣ B ∣ ∣ C ∣)
     (membership-equiv-gives-carrier-equality B C ,
      carrier-equality-gives-membership-equiv B C)

-- ...which yields an alternative subuniverse equality lemma.
subuniverse-equality' : (B C : subuniverse)
  → (B ≡ C) ≃ (∣ B ∣ ≡ ∣ C ∣)
subuniverse-equality' B C =
  (subuniverse-equality B C) ● (carrier-equiv B C)
```

(This page is almost entirely blank on purpose.)

8 Equations and Varieties

8.1 Types for Theories and Models

This section presents the UALib.Varieties.ModelTheory module of the Agda UALib.

Having set the stage for the entrance of Equational Logic, Cliff Bergman proclaims (in [1, Section 4.4]), "Now, finally, we can formalize the idea we have been using since the first page of this text," and proceeds to define **identities of terms** as follows (paraphrasing for notational consistency):

> Let S be a signature. An **identity** or **equation** in S is 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** $p \approx q$ if $p \cdot \mathbf{A} \equiv q \cdot \mathbf{A}$. 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 $\mathbf{A} \in \mathcal{K}$, $\mathbf{A} \models p \approx q$.

Notation. In the Agda UALib, 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 ⊧ and ≈. As a reasonable alternative to what we would normally express informally as $\mathcal{K} \models p \approx q$, we have settled on $\mathcal{K} \models p \approxeq q$ to denote this relation. To reiterate, if $\mathcal{K}$ is a class of S-algebras, we write $\mathcal{K} \models p \approxeq q$ if every $\mathbf{A} \in \mathcal{K}$ satisfies $\mathbf{A} \models p \approx q$.

Unicode Hints. To produce the symbols ≈ and ⊧ in Emacs agda2-mode, type `\~~` and `\models` (resp.). The symbol ≋ is produced in Emacs agda2-mode with `\~~~`.

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

open import UALib.Algebras using (Signature; 𝓞; 𝓥; Algebra; _↠_)
open import UALib.Prelude.Preliminaries using (global-dfunext; Universe; _̇)

module UALib.Varieties.ModelTheory
  {𝑆 : Signature 𝓞 𝓥}{gfe : global-dfunext}
  {𝕏 : {𝓤 𝓧 : Universe}{X : 𝓧 ̇ }(𝑨 : Algebra 𝓤 𝑆) → X ↠ 𝑨}
  where

open import UALib.Subalgebras.Subalgebras{𝑆 = 𝑆}{gfe}{𝕏} public
```

8.1.1 The models relation

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

After ⊧ is defined, we prove just a couple of useful facts about ⊧. However, more is proved about the models relation in the next section (Section 8.2).

```
_⊧_≈_ : {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 ⁺ ̇

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

8.1.2 Equational theories and models

The set of identities that hold for all algebras in a class $\mathscr{K}$ is denoted by Th $\mathscr{K}$, which we define as follows.

```
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
```

The class of algebras that satisfy all identities in a given set $\mathscr{E}$ is denoted by Mod $\mathscr{E}$. We given three nearly equivalent definitions for this below. The only distinction between these is whether the arguments are explicit (so must appear in the argument list) or implicit (so we may let Agda do its best to guess the argument).

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


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

8.1.3 Computing with ⊧

We have formally defined $\mathbf{A} \models p \approx q$, which represents the assertion that $p \approx q$ holds when this identity is interpreted in the algebra $\mathbf{A}$; syntactically, $p \,\dot{}\, \mathbf{A} \equiv q \,\dot{}\, \mathbf{A}$. Hopefully we already grasp the semantic meaning of these strings of symbols, but our understanding is at best tenuous unless we have a handle on their computational meaning, since this tells us how we can *use* the definitions. So we pause to emphasize that we interpret the expression $p \,\dot{}\, \mathbf{A} \equiv q \,\dot{}\, \mathbf{A}$ as an *extensional equality*, by which we mean that for each *assignment function* $\boldsymbol{a} : X \to |\,\mathbf{A}\,|$—assigning values in the domain of $\mathbf{A}$ to the variable symbols in X—we have $(p \,\dot{}\, \mathbf{A})\, \boldsymbol{a} \equiv (q \,\dot{}\, \mathbf{A})\, \boldsymbol{a}$.

The binary relation ⊧ would be practically useless if it were not an *algebraic invariant* (i.e., invariant under isomorphism), and this fact is proved by showing that a certain term operation identity—namely, $(p \,\dot{}\, \mathbf{B}) \equiv (q \,\dot{}\, \mathbf{B})$—holds *extensionally*, in the sense of the previous paragraph.

```
⊧-≅
  ⊧-transport : {𝓠 𝓤 𝓧 : Universe}{X : 𝓧 ̇}{𝐀 : Algebra 𝓠 S}{𝐁 : Algebra 𝓤 S}
                (p q : Term{𝓧}{X}) → (𝐀 ⊧ p ≈ q) → (𝐀 ≅ 𝐁) → 𝐁 ⊧ p ≈ q
⊧-transport {𝓠}{𝓤}{𝓧}{X}{𝐀}{𝐁} p q Apq (f , g , f∼g , g∼f) = γ
  where
    γ : (p ̇ 𝐁) ≡ (q ̇ 𝐁)
    γ = gfe λ x →
        (p ̇ 𝐁) x ≡⟨ refl ⟩
        (p ̇ 𝐁) (∣ 𝒾𝒹 𝐁 ∣ ∘ x) ≡⟨ ap (λ - → (p ̇ 𝐁) -) (gfe λ i → ((f∼g)(x i))⁻¹) ⟩
```

```
        (p ̇ B) ((| f | ∘ | g |) ∘ x) ≡⟨ (comm-hom-term gfe A B f p (| g | ∘ x))⁻¹ ⟩
        | f | ((p ̇ A) (| g | ∘ x)) ≡⟨ ap (λ - → | f | (- (| g | ∘ x))) Apq ⟩
        | f | ((q ̇ A) (| g | ∘ x)) ≡⟨ comm-hom-term gfe A B f q (| g | ∘ x) ⟩
        (q ̇ B) ((| f | ∘ | g |) ∘ x) ≡⟨ ap (λ - → (q ̇ B) -) (gfe λ i → (f∼g) (x i)) ⟩
        (q ̇ B) x ■

⊧-≅ = ⊧-transport -- (alias)
```

The ⊧ relation is also compatible with the lift operation.

```
lift-alg-⊧ : {𝓤 𝓦 𝓧 : Universe}{X : 𝓧 ̇}
    (A : Algebra 𝓤 S)(p q : Term{𝓧}{X})
  → A ⊧ p ≈ q → (lift-alg A 𝓦) ⊧ p ≈ q
lift-alg-⊧ A p q Apq = ⊧-≅ p q Apq lift-alg-≅

lower-alg-⊧ : {𝓤 𝓦 𝓧 : Universe}{X : 𝓧 ̇}(A : Algebra 𝓤 S)
                             (p q : Term{𝓧}{X})
  →                          lift-alg A 𝓦 ⊧ p ≈ q → A ⊧ p ≈ q
lower-alg-⊧ {𝓤}{𝓦}{𝓧}{X} A p q lApq = ⊧-≅ p q lApq (sym-≅ lift-alg-≅)
```

8.2 Equational Logic Types

This section presents the UALib.Varieties.EquationalLogic module of the Agda UALib. We establish some important features of the "models" relation, which demonstrate the nice relationships ⊧ has with the other protagonists of our story, H, S, and P.

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

open import UALib.Algebras using (Signature; 𝓞; 𝓥; Algebra; _↠_)
open import UALib.Prelude.Preliminaries using (global-dfunext; Universe; _̇)

module UALib.Varieties.EquationalLogic
  {S : Signature 𝓞 𝓥}{gfe : global-dfunext}
  {𝕏 : {𝓤 𝓧 : Universe}{X : 𝓧 ̇ }(A : Algebra 𝓤 S) → X ↠ A}
  where

open import UALib.Varieties.ModelTheory {S = S}{gfe}{𝕏} public
open import UALib.Prelude.Preliminaries using (∘-embedding; domain; embeddings-are-lc) public
```

8.2.1 Product transport

We prove that identities satisfied by all factors of a product are also satisfied by the product.

```
product-id-compatibility -- (alias)
  products-preserve-identities : {𝓤 𝓧 : Universe}{X : 𝓧 ̇}(p q : Term{𝓧}{X})
                                 (I : 𝓤 ̇ ) (𝒜 : I → Algebra 𝓤 S)
  →                              ((i : I) → (𝒜 i) ⊧ p ≈ q)
                                 -----------------------------------
  →                              ⨅ 𝒜 ⊧ p ≈ q

products-preserve-identities p q I 𝒜 𝒜pq = γ
  where
    γ : (p ̇ ⨅ 𝒜) ≡ (q ̇ ⨅ 𝒜)
```

```
    γ = gfe λ a →
      (p ̇ ⨅ 𝒜) a ≡⟨ interp-prod gfe p 𝒜 a ⟩
      (λ i → ((p ̇ (𝒜 i)) (λ x → (a x) i))) ≡⟨ gfe (λ i → cong-app (𝒜pq i) (λ x → (a x) i)) ⟩
      (λ i → ((q ̇ (𝒜 i)) (λ x → (a x) i))) ≡⟨ (interp-prod gfe q 𝒜 a)⁻¹ ⟩
      (q ̇ ⨅ 𝒜) a ∎
product-id-compatibility = products-preserve-identities

lift-product-id-compatibility -- (alias)
  lift-products-preserve-ids : {𝓤 𝓦 𝓧 : Universe}{X : 𝓧 ̇}
                               (p q : Term{𝓧}{X})
                               (I : 𝓤 ̇ ) (𝒜 : I → Algebra 𝓤 S)
  →                            ((i : I) → (lift-alg (𝒜 i) 𝓦) ⊧ p ≈ q)
                               -------------------------------------
  →                            ⨅ 𝒜 ⊧ p ≈ q

lift-products-preserve-ids {𝓤}{𝓦} p q I 𝒜 lApq = products-preserve-identities p q I 𝒜 Aipq
  where
    Aipq : (i : I) → (𝒜 i) ⊧ p ≈ q
    Aipq i = ⊧-≅ p q (lApq i) (sym-≅ lift-alg-≅)
lift-product-id-compatibility = lift-products-preserve-ids

class-product-id-compatibility -- (alias)
  products-in-class-preserve-identities : {𝓤 𝓧 : Universe}{X : 𝓧 ̇}
                                          {𝒦 : Pred (Algebra 𝓤 S) (OV 𝓤)}
                                          (p q : Term{𝓧}{X})
                                          (I : 𝓤 ̇)(𝒜 : I → Algebra 𝓤 S)
  →                                       𝒦 ⊧ p ≋ q → ((i : I) → 𝒜 i ∈ 𝒦)
                                          ------------------------------------------
  →                                       ⨅ 𝒜 ⊧ p ≈ q

products-in-class-preserve-identities p q I 𝒜 α K𝒜 = γ
  where
    β : ∀ i → (𝒜 i) ⊧ p ≈ q
    β i = α (K𝒜 i)

    γ : (p ̇ ⨅ 𝒜) ≡ (q ̇ ⨅ 𝒜)
    γ = products-preserve-identities p q I 𝒜 β
class-product-id-compatibility = products-in-class-preserve-identities
```

8.2.2 Subalgebra transport

We show that identities modeled by a class of algebras is also modeled by all subalgebras of the class. In other terms, every term equation $p \approx q$ that is satisfied by all $\mathbf{A} \in \mathcal{K}$ is also satisfied by every subalgebra of a member of $\mathcal{K}$.

```
S-⊧ -- (alias)
  subalgebras-preserve-identities : {𝓤 𝓠 𝓧 : Universe}{X : 𝓧 ̇}
                                    {𝒦 : Pred (Algebra 𝓠 S) (OV 𝓠)}
                                    (p q : Term{𝓧}{X})
                                    (𝐁 : SubalgebraOfClass{𝓤}{𝓠} 𝒦)
  →                                 𝒦 ⊧ p ≋ q
```

```
   →                                   -------------
                                       | B | ⊧ p ≈ q

  subalgebras-preserve-identities {𝓤}{X = X} p q (B , A , SA , (KA , BisSA)) Kpq = γ
   where
    B' : Algebra 𝓤 S
    B' = | SA |

    h' : hom B' A
    h' = (| snd SA | , snd ∥ snd SA ∥ )

    f : hom B B'
    f = | BisSA |

    h : hom B A
    h = HCompClosed B B' A f h'

    hem : is-embedding | h |
    hem = ∘-embedding h'em fem
     where
      h'em : is-embedding | h' |
      h'em = fst ∥ snd SA ∥

      fem : is-embedding | f |
      fem = iso→embedding BisSA

    β : A ⊧ p ≈ q
    β = Kpq KA

    ξ : (b : X → | B | ) → | h | ((p ̇ B) b) ≡ | h | ((q ̇ B) b)
    ξ b =
     | h |((p ̇ B) b) ≡⟨ comm-hom-term gfe B A h p b ⟩
     (p ̇ A)(| h | ∘ b) ≡⟨ intensionality β (| h | ∘ b) ⟩
     (q ̇ A)(| h | ∘ b) ≡⟨ (comm-hom-term gfe B A h q b)⁻¹ ⟩
     | h |((q ̇ B) b) ∎

    hlc : {b b' : domain | h |} → | h | b ≡ | h | b' → b ≡ b'
    hlc hb≡hb' = (embeddings-are-lc | h | hem) hb≡hb'

    γ : B ⊧ p ≈ q
    γ = gfe λ b → hlc (ξ b)

  S-⊧ = subalgebras-preserve-identities
  -- subalgebra-id-compatibility = subalgebras-preserve-identities
```

8.2.3 Homomorphism transport

Recall that an identity is satisfied by all algebras in a class if and only if that identity is compatible with all homomorphisms from the term algebra $\mathbf{T}\ X$ into algebras of the class. More precisely, if $\mathcal{K}$ is a class of S-algebras and p, q terms in the language of S, then,

$$\mathcal{K} \models p \approx q \quad \Leftrightarrow \quad \forall\, \mathbf{A} \in \mathcal{K}, \forall\, h : \mathsf{hom}\ (\mathbf{T}\ X)\ \mathbf{A},\ h \circ (p\ ^{\cdot}\ (\mathbf{T}\ X)) = h \circ (q\ ^{\cdot}\ (\mathbf{T}\ X)).$$

We now formalize this result in Agda and we prove that identities satisfied by all algberas in a class are also satsified by all homomorphic images of algebras in the class.

```
-- ⇒ (the "only if" direction)
id-class-hom-compatibility -- (alias)
  identities-compatible-with-homs : {𝓤 𝓧 : Universe}{X : 𝓧 ̇}
                                    {𝒦 : Pred (Algebra 𝓤 S) (OV 𝓤)}
                                    (p q : Term) → 𝒦 ⊧ p ≋ q
                                  ------------------------------------------
  →                                 ∀ (A : Algebra 𝓤 S)(KA : 𝒦 A)(h : hom (T X) A)
                                    → ∣ h ∣ ∘ (p ̇ T X) ≡ ∣ h ∣ ∘ (q ̇ T X)

identities-compatible-with-homs {X = X} p q α A KA h = γ
  where
   β : ∀(a : X → ∣ T X ∣ ) → (p ̇ A)(∣ h ∣ ∘ a) ≡ (q ̇ A)(∣ h ∣ ∘ a)
   β a = intensionality (α KA) (∣ h ∣ ∘ a)

   ξ : ∀(a : X → ∣ T X ∣ ) → ∣ h ∣ ((p ̇ T X) a) ≡ ∣ h ∣ ((q ̇ T X) a)
   ξ a =
    ∣ h ∣ ((p ̇ T X) a) ≡⟨ comm-hom-term gfe (T X) A h p a ⟩
    (p ̇ A)(∣ h ∣ ∘ a) ≡⟨ β a ⟩
    (q ̇ A)(∣ h ∣ ∘ a) ≡⟨ (comm-hom-term gfe (T X) A h q a)⁻¹ ⟩
    ∣ h ∣ ((q ̇ T X) a) ∎

   γ : ∣ h ∣ ∘ (p ̇ T X) ≡ ∣ h ∣ ∘ (q ̇ T X)
   γ = gfe ξ
id-class-hom-compatibility = identities-compatible-with-homs

-- ⇐ (the "if" direction)
class-hom-id-compatibility -- (alias)
  homs-compatible-with-identities : {𝓤 𝓧 : Universe}{X : 𝓧 ̇}
                                    {𝒦 : Pred (Algebra 𝓤 S) (OV 𝓤)}
                                    (p q : Term)
  →                                 ( ∀ (A : Algebra 𝓤 S)(KA : A ∈ 𝒦) (h : hom (T X) A)
                                           → ∣ h ∣ ∘ (p ̇ T X) ≡ ∣ h ∣ ∘ (q ̇ T X) )
                                    ----------------------------------------
  →                                 𝒦 ⊧ p ≋ q

homs-compatible-with-identities {X = X} p q β {A} KA = γ
    where
     h : (a : X → ∣ A ∣) → hom (T X) A
     h a = lift-hom A a

     γ : A ⊧ p ≈ q
     γ = gfe λ a →
      (p ̇ A) a ≡⟨ refl ⟩
      (p ̇ A)(∣ h a ∣ ∘ g) ≡⟨(comm-hom-term gfe (T X) A (h a) p g)⁻¹ ⟩
      (∣ h a ∣ ∘ (p ̇ T X)) g ≡⟨ ap (λ - → - g) (β A KA (h a)) ⟩
      (∣ h a ∣ ∘ (q ̇ T X)) g ≡⟨ (comm-hom-term gfe (T X) A (h a) q g) ⟩
      (q ̇ A)(∣ h a ∣ ∘ g) ≡⟨ refl ⟩
```

```
        (q ̇ A) a ∎
  class-hom-id-compatibility = homs-compatible-with-identities

  compatibility-of-identities-and-homs : {𝓤 𝓧 : Universe}{X : 𝓧 ̇}
                                         {𝒦 : Pred (Algebra 𝓤 S) (OV 𝓤)}
                                         (p q : Term{𝓧}{X})
    -------------------------------------------------
    → (𝒦 ⊧ p ≋ q) ⇔ (∀(A : Algebra 𝓤 S)(KA : A ∈ 𝒦)(hh : hom (𝑻 X) A)
                                     → ∣ hh ∣ ∘ (p ̇ 𝑻 X) ≡ ∣ hh ∣ ∘ (q ̇ 𝑻 X))

  compatibility-of-identities-and-homs p q = identities-compatible-with-homs p q ,
                                             homs-compatible-with-identities p q
```

Here's a simpler special case of the previous result that suffices when we're interested in just a single algebra, rather than a class of algebras.

```
  hom-id-compatibility : {𝓤 𝓧 : Universe}{X : 𝓧 ̇}
                         (p q : Term{𝓧}{X})
                         (A : Algebra 𝓤 S)(ϕ : hom (𝑻 X) A)
    →                    A ⊧ p ≈ q
                         ------------------
    →                    ∣ ϕ ∣ p ≡ ∣ ϕ ∣ q

  hom-id-compatibility {X = X} p q A ϕ β = ∣ ϕ ∣ p ≡⟨ ap ∣ ϕ ∣ (term-agreement p) ⟩
                            ∣ ϕ ∣ ((p ̇ 𝑻 X) ℊ) ≡⟨ (comm-hom-term gfe (𝑻 X) A ϕ p ℊ) ⟩
                            (p ̇ A) (∣ ϕ ∣ ∘ ℊ)  ≡⟨ intensionality β (∣ ϕ ∣ ∘ ℊ) ⟩
                            (q ̇ A) (∣ ϕ ∣ ∘ ℊ)  ≡⟨ (comm-hom-term gfe (𝑻 X) A ϕ q ℊ)⁻¹ ⟩
                            ∣ ϕ ∣ ((q ̇ 𝑻 X) ℊ) ≡⟨ (ap ∣ ϕ ∣ (term-agreement q))⁻¹ ⟩
                            ∣ ϕ ∣ q             ∎
```

8.3 Inductive Types H, S, P, V

This section presents the UALib.Varieties.Varieties module of the Agda UALib. Fix a signature S, let $\mathcal{K}$ be a class of S-algebras, and define

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

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

An algebra is a homomorphic image (resp., subalgebra; resp., product) of every algebra to which it is isomorphic. Thus, the class H $\mathcal{K}$ (resp., S $\mathcal{K}$; resp., P $\mathcal{K}$) is closed under isomorphism.

The operators H, S, and P can be composed with one another repeatedly, forming yet more closure operators.

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 H S P $\mathcal{K} \subseteq \mathcal{K}$.

This module defines what we have found to be the most useful inductive types representing the closure operators H, S, and P. Separately, we define the inductive type V for simultaneously representing closure under H, S, and P.

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

open import UALib.Algebras using (Signature; 𝓞; 𝓥; Algebra; _↠_)
open import UALib.Prelude.Preliminaries using (global-dfunext; Universe; _̇)

module UALib.Varieties.Varieties
  {𝑆 : Signature 𝓞 𝓥}{gfe : global-dfunext}
  {𝕏 : {𝓤 𝓧 : Universe}{X : 𝓧 ̇ }(𝑨 : Algebra 𝓤 𝑆) → X ↠ 𝑨}
  where

open import UALib.Varieties.EquationalLogic{𝑆 = 𝑆}{gfe}{𝕏} public
```

8.3.1 Homomorphism closure

We define the inductive type H to represent classes of algebras that include all homomorphic images of algebras in the class; i.e., classes that are closed under the taking of homomorphic images.

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

8.3.2 Subalgebra closure

The most useful inductive type that we have found for representing classes of algebras that are closed under the taking of subalgebras as an inductive type.

```
--Closure wrt S
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{𝓤}{𝓦} 𝒦
  slift : {𝑨 : Algebra 𝓤 𝑆} → 𝑨 ∈ S{𝓤}{𝓤} 𝒦 → lift-alg 𝑨 𝓦 ∈ S{𝓤}{𝓦} 𝒦
  ssub : {𝑨 : Algebra 𝓤 𝑆}{𝑩 : Algebra (𝓤 ⊔ 𝓦) 𝑆} → 𝑨 ∈ S{𝓤}{𝓤} 𝒦 → 𝑩 ≤ 𝑨 → 𝑩 ∈ S{𝓤}{𝓦} 𝒦
  ssubw : {𝑨 𝑩 : Algebra (𝓤 ⊔ 𝓦) 𝑆} → 𝑨 ∈ S{𝓤}{𝓦} 𝒦 → 𝑩 ≤ 𝑨 → 𝑩 ∈ S{𝓤}{𝓦} 𝒦
  siso  : {𝑨 : Algebra 𝓤 𝑆}{𝑩 : Algebra (𝓤 ⊔ 𝓦) 𝑆} → 𝑨 ∈ S{𝓤}{𝓤} 𝒦 → 𝑨 ≅ 𝑩 → 𝑩 ∈ S{𝓤}{𝓦} 𝒦
```

8.3.3 Product closure

The most useful inductive type that we have found for representing classes of algebras closed under arbitrary products is the following.

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

8.3.4 Varietal closure

A class $\mathcal{K}$ of $\mathcal{S}$-algebras is called a **variety** if it is closed under each of the closure operators H, S, and P introduced above; the corresponding closure operator is often denoted $\mathbb{V}$, but we will typically denote it by V.

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

We now define V as an inductive type.

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

8.3.5 Closure properties

The types defined above represent operators with useful closure properties. We now prove a handful of such properties since we will need them later.

```
-- P is a closure operator, in particular, it's expansive...
P-expa : {𝓤 : Universe}{𝒦 : Pred (Algebra 𝓤 𝑆)(OV 𝓤)}
  → 𝒦 ⊆ P{𝓤}{𝓤} 𝒦
P-expa{𝓤}{𝒦} {𝑨} KA = pisou{𝑨 = (lift-alg 𝑨 𝓤)}{𝑩 = 𝑨} (pbase KA) (sym-≅ lift-alg-≅)

-- ...and idempotent...
P-idemp : {𝓤 𝓦 : Universe}{𝒦 : Pred (Algebra 𝓤 𝑆)(OV 𝓤)}
  →        P{𝓤}{𝓦} (P{𝓤}{𝓤} 𝒦) ⊆ P{𝓤}{𝓦} 𝒦

P-idemp (pbase x) = pliftu x
P-idemp {𝓤} (pliftu x) = pliftu (P-idemp{𝓤}{𝓤} x)
P-idemp (pliftw x) = pliftw (P-idemp x)
P-idemp {𝓤} (produ x) = produ (λ i → P-idemp{𝓤}{𝓤} (x i))
P-idemp (prodw x) = prodw (λ i → P-idemp (x i))
P-idemp {𝓤} (pisou x x₁) = pisou (P-idemp{𝓤}{𝓤} x) x₁
P-idemp (pisow x x₁) = pisow (P-idemp x) x₁

module _ {𝓤 𝓦 : Universe}{𝒦 : Pred (Algebra 𝓤 𝑆)(OV 𝓤)} where

  -- An idempotence variant that handles universes more generally (we need this later)
  P-idemp' : --{𝓤 : Universe}{𝓦 : Universe}{𝒦 : Pred (Algebra 𝓤 𝑆)(OV 𝓤)}
             P{𝓤 ⊔ 𝓦}{𝓤 ⊔ 𝓦} (P{𝓤}{𝓤 ⊔ 𝓦} 𝒦) ⊆ P{𝓤}{𝓤 ⊔ 𝓦} 𝒦

  P-idemp' (pbase x) = pliftw x
  P-idemp' (pliftu x) = pliftw (P-idemp' x)
  P-idemp' (pliftw x) = pliftw (P-idemp' x)
```

```
  P-idemp' (produ x) = prodw (λ i → P-idemp' (x i))
  P-idemp' (prodw x) = prodw (λ i → P-idemp' (x i))
  P-idemp' (pisou x x₁) = pisow (P-idemp' x) x₁
  P-idemp' (pisow x x₁) = pisow (P-idemp' x) x₁

-- S is a closure operator

-- In particular, it's monotone.
S-mono : {𝓤 𝓦 : Universe}{𝒦 𝒦' : Pred (Algebra 𝓤 𝑆)(OV 𝓤)}
 → 𝒦 ⊆ 𝒦' → S{𝓤}{𝓦} 𝒦 ⊆ S{𝓤}{𝓦} 𝒦'
S-mono ante (sbase x) = sbase (ante x)
S-mono {𝓤}{𝓦}{𝒦}{𝒦'} ante (slift{𝑨} x) = slift{𝓤}{𝓦}{𝒦'} (S-mono{𝓤}{𝓤} ante x)
S-mono ante (ssub{𝑨}{𝑩} sA B≤A) = ssub (S-mono ante sA) B≤A
S-mono ante (ssubw{𝑨}{𝑩} sA B≤A) = ssubw (S-mono ante sA) B≤A
S-mono ante (siso x x₁) = siso (S-mono ante x) x₁
```

We sometimes want to go back and forth between our two representations of subalgebras of algebras in a class. The tools subalgebra→S and S→subalgebra were designed with that purpose in mind.

```
  subalgebra→S : {𝓤 𝓦 : Universe}{𝒦 : Pred (Algebra 𝓤 𝑆)(OV 𝓤)}
                 {𝑪 : Algebra (𝓤 ⊔ 𝓦) 𝑆} → 𝑪 IsSubalgebraOfClass 𝒦
                ----------------------------------------------------------
   →             𝑪 ∈ S{𝓤}{𝓦} 𝒦

  subalgebra→S {𝓤}{𝓦}{𝒦}{𝑪} (𝑨 , ((𝑩 , B≤A) , KA , C≅B)) = ssub sA C≤A
    where
     C≤A : 𝑪 ≤ 𝑨
     C≤A = Iso-≤ 𝑨 𝑪 B≤A C≅B

     slAu : lift-alg 𝑨 𝓤 ∈ S{𝓤}{𝓤} 𝒦
     slAu = sbase KA

     sA : 𝑨 ∈ S{𝓤}{𝓤} 𝒦
     sA = siso slAu (sym-≅ lift-alg-≅)


  module _ {𝓤 : Universe}{𝒦 : Pred (Algebra 𝓤 𝑆)(OV 𝓤)} where

   S→subalgebra : {𝑩 : Algebra 𝓤 𝑆} → 𝑩 ∈ S{𝓤}{𝓤} 𝒦 → 𝑩 IsSubalgebraOfClass 𝒦
   S→subalgebra (sbase{𝑩} x) = 𝑩 , (𝑩 , refl-≤) , x , (sym-≅ lift-alg-≅)
   S→subalgebra (slift{𝑩} x) = ∣ BS ∣ , SA , KA , TRANS-≅ (sym-≅ lift-alg-≅) B≅SA
     where
      BS : 𝑩 IsSubalgebraOfClass 𝒦
      BS = S→subalgebra x

      SA : SUBALGEBRA ∣ BS ∣
      SA = fst ∥ BS ∥

      KA : ∣ BS ∣ ∈ 𝒦
      KA = ∣ snd ∥ BS ∥ ∣

      B≅SA : 𝑩 ≅ ∣ SA ∣
      B≅SA = ∥ snd ∥ BS ∥ ∥
```

```
S→subalgebra {B} (ssub{A} sA B≤A) = γ
  where
   AS : A IsSubalgebraOfClass 𝒦
   AS = S→subalgebra sA
   SA : SUBALGEBRA ∣ AS ∣
   SA = fst ∥ AS ∥
   B≤SA : B ≤ ∣ SA ∣
   B≤SA = TRANS-≤-≅ B ∣ SA ∣ B≤A (∥ snd ∥ AS ∥ ∥)
   B≤AS : B ≤ ∣ AS ∣
   B≤AS = transitivity-≤ B{∣ SA ∣}{∣ AS ∣} B≤SA ∥ SA ∥
   γ : B IsSubalgebraOfClass 𝒦
   γ = ∣ AS ∣ , (B , B≤AS) , ∣ snd ∥ AS ∥ ∣ , refl-≅

S→subalgebra {B} (ssubw{A} sA B≤A) = γ
  where
   AS : A IsSubalgebraOfClass 𝒦
   AS = S→subalgebra sA
   SA : SUBALGEBRA ∣ AS ∣
   SA = fst ∥ AS ∥
   B≤SA : B ≤ ∣ SA ∣
   B≤SA = TRANS-≤-≅ B ∣ SA ∣ B≤A (∥ snd ∥ AS ∥ ∥)
   B≤AS : B ≤ ∣ AS ∣
   B≤AS = transitivity-≤ B{∣ SA ∣}{∣ AS ∣} B≤SA ∥ SA ∥
   γ : B IsSubalgebraOfClass 𝒦
   γ = ∣ AS ∣ , (B , B≤AS) , ∣ snd ∥ AS ∥ ∣ , refl-≅

S→subalgebra {B} (siso{A} sA A≅B) = γ
  where
   AS : A IsSubalgebraOfClass 𝒦
   AS = S→subalgebra sA
   SA : SUBALGEBRA ∣ AS ∣
   SA = fst ∥ AS ∥
   A≅SA : A ≅ ∣ SA ∣
   A≅SA = snd ∥ snd AS ∥
   γ : B IsSubalgebraOfClass 𝒦
   γ = ∣ AS ∣ , SA , ∣ snd ∥ AS ∥ ∣ , (TRANS-≅ (sym-≅ A≅B) A≅SA)
```

Next we observe that lifting to a higher universe does not break the property of being a subalgebra of an algebra of a class. In other words, if we lift a subalgebra of an algebra in a class, the result is still a subalgebra of an algebra in the class.

```
lift-alg-subP : {𝓤 𝓦 : Universe}{𝒦 : Pred (Algebra 𝓤 S)(OV 𝓤)}{B : Algebra 𝓤 S}
              → B IsSubalgebraOfClass (P{𝓤}{𝓤} 𝒦)
                ---------------------------------------------------
              → (lift-alg B 𝓦) IsSubalgebraOfClass (P{𝓤}{𝓦} 𝒦)

lift-alg-subP {𝓤} {𝓦} {𝒦} {B} (A , (C , C≤A) , pA , B≅C ) = γ
              where
               lA lB lC : Algebra (𝓤 ⊔ 𝓦) S
               lA = lift-alg A 𝓦
```

```
    lB = lift-alg 𝑩 𝓦
    lC = lift-alg 𝑪 𝓦

    lC≤lA : lC ≤ lA
    lC≤lA = lift-alg-lift-≤-lift 𝑪 {𝑨} C≤A
    plA : lA ∈ P{𝓤}{𝓦} 𝒦
    plA = pliftu pA

    γ : lB IsSubalgebraOfClass (P{𝓤}{𝓦} 𝒦)
    γ = lA , (lC , lC≤lA) , plA , (lift-alg-iso 𝓤 𝓦 𝑩 𝑪 B≅C)
```

The next lemma would be too obvoius to care about were it not for the fact that we'll need it later, so it too must be formalized.

```
S⊆SP : {𝓤 𝓦 : Universe}{𝒦 : Pred (Algebra 𝓤 𝑆)(OV 𝓤)}
 → S{𝓤}{𝓦} 𝒦 ⊆ S{𝓤 ⊔ 𝓦}{𝓤 ⊔ 𝓦} (P{𝓤}{𝓦} 𝒦)

S⊆SP {𝓤} {𝓦} {𝒦} {.(lift-alg 𝑨 𝓦)} (sbase{𝑨} x) =
  siso spllA (sym-≅ lift-alg-≅)
    where
    llA : Algebra (𝓤 ⊔ 𝓦) 𝑆
    llA = lift-alg (lift-alg 𝑨 𝓦) (𝓤 ⊔ 𝓦)

    spllA : llA ∈ S{𝓤 ⊔ 𝓦}{𝓤 ⊔ 𝓦} (P{𝓤}{𝓦} 𝒦)
    spllA = sbase{𝓤 = (𝓤 ⊔ 𝓦)}{𝓦 = (𝓤 ⊔ 𝓦)} (pbase x)

S⊆SP {𝓤} {𝓦} {𝒦} {.(lift-alg 𝑨 𝓦)} (slift{𝑨} x) =
  subalgebra→S{𝓤 ⊔ 𝓦}{𝓤 ⊔ 𝓦}{P{𝓤}{𝓦} 𝒦}{lift-alg 𝑨 𝓦} lAsc
    where
    splAu : 𝑨 ∈ S{𝓤}{𝓤} (P{𝓤}{𝓤} 𝒦)
    splAu = S⊆SP{𝓤}{𝓤} x

    Asc : 𝑨 IsSubalgebraOfClass (P{𝓤}{𝓤} 𝒦)
    Asc = S→subalgebra{𝓤}{P{𝓤}{𝓤} 𝒦}{𝑨} splAu

    lAsc : (lift-alg 𝑨 𝓦) IsSubalgebraOfClass (P{𝓤}{𝓦} 𝒦)
    lAsc = lift-alg-subP Asc

S⊆SP {𝓤} {𝓦} {𝒦} {𝑩} (ssub{𝑨} sA B≤A) =
  ssub{𝓤 ⊔ 𝓦}{𝓤 ⊔ 𝓦} lAsp (lift-alg-sub-lift 𝑨 B≤A)
    where
    lA : Algebra (𝓤 ⊔ 𝓦) 𝑆
    lA = lift-alg 𝑨 𝓦

    splAu : 𝑨 ∈ S{𝓤}{𝓤} (P{𝓤}{𝓤} 𝒦)
    splAu = S⊆SP{𝓤}{𝓤} sA

    Asc : 𝑨 IsSubalgebraOfClass (P{𝓤}{𝓤} 𝒦)
    Asc = S→subalgebra{𝓤}{P{𝓤}{𝓤} 𝒦}{𝑨} splAu

    lAsc : lA IsSubalgebraOfClass (P{𝓤}{𝓦} 𝒦)
    lAsc = lift-alg-subP Asc
```

```
      lAsp : lA ∈ S{𝓤 ⊔ 𝓦}{𝓤 ⊔ 𝓦} (P{𝓤}{𝓦} 𝒦)
      lAsp = subalgebra→S{𝓤 ⊔ 𝓦}{𝓤 ⊔ 𝓦}{P{𝓤}{𝓦} 𝒦}{lA} lAsc

  S⊆SP {𝓤} {𝓦} {𝒦} {𝐁} (ssubw{𝐀} sA B≤A) = γ
   where
    spA : 𝐀 ∈ S{𝓤 ⊔ 𝓦}{𝓤 ⊔ 𝓦} (P{𝓤}{𝓦} 𝒦)
    spA = S⊆SP sA
    γ : 𝐁 ∈ S{𝓤 ⊔ 𝓦}{𝓤 ⊔ 𝓦} (P{𝓤}{𝓦} 𝒦)
    γ = ssubw{𝓤 ⊔ 𝓦}{𝓤 ⊔ 𝓦} spA B≤A

  S⊆SP {𝓤} {𝓦} {𝒦} {𝐁} (siso{𝐀} sA A≅B) = siso{𝓤 ⊔ 𝓦}{𝓤 ⊔ 𝓦} lAsp lA≅B
   where
    lA : Algebra (𝓤 ⊔ 𝓦) S
    lA = lift-alg 𝐀 𝓦

    splAu : 𝐀 ∈ S{𝓤}{𝓤} (P{𝓤}{𝓤} 𝒦)
    splAu = S⊆SP{𝓤}{𝓤} sA

    lAsc : lA IsSubalgebraOfClass (P{𝓤}{𝓦} 𝒦)
    lAsc = lift-alg-subP (S→subalgebra{𝓤}{P{𝓤}{𝓤} 𝒦}{𝐀} splAu)

    lAsp : lA ∈ S{𝓤 ⊔ 𝓦}{𝓤 ⊔ 𝓦} (P{𝓤}{𝓦} 𝒦)
    lAsp = subalgebra→S{𝓤 ⊔ 𝓦}{𝓤 ⊔ 𝓦}{P{𝓤}{𝓦} 𝒦}{lA} lAsc

    lA≅B : lA ≅ 𝐁
    lA≅B = Trans-≅ lA 𝐁 (sym-≅ lift-alg-≅) A≅B
```

We need to formalize one more lemma before arriving the short term objective of this section, which is the proof of the inclusion PS⊆SP.

```
  lemPS⊆SP : {𝓤 𝓦 : Universe}{𝒦 : Pred (Algebra 𝓤 S)(OV 𝓤)}{hfe : hfunext 𝓦 𝓤}
             {I : 𝓦 ̇}{ℬ : I → Algebra 𝓤 S}
   →         ((i : I) → (ℬ i) IsSubalgebraOfClass 𝒦)
             ----------------------------------------------------
   →         ⨅ ℬ IsSubalgebraOfClass (P{𝓤}{𝓦} 𝒦)

  lemPS⊆SP {𝓤}{𝓦}{𝒦}{hfe}{I}{ℬ} B≤K =
   ⨅ 𝒜 , (⨅ SA , ⨅SA≤⨅𝒜 ) , (produ{𝓤}{𝓦}{I = I}{𝒜 = 𝒜} (λ i → P-expa (KA i)) ) , γ
   where
    𝒜 : I → Algebra 𝓤 S
    𝒜 = λ i → ∣ B≤K i ∣

    SA : I → Algebra 𝓤 S
    SA = λ i → ∣ fst ∥ B≤K i ∥ ∣

    KA : ∀ i → 𝒜 i ∈ 𝒦
    KA = λ i → ∣ snd ∥ B≤K i ∥ ∣

    B≅SA : ∀ i → ℬ i ≅ SA i
    B≅SA = λ i → ∥ snd ∥ B≤K i ∥ ∥
    pA : ∀ i → lift-alg (𝒜 i) 𝓦 ∈ P{𝓤}{𝓦} 𝒦
    pA = λ i → pbase (KA i)
```

```
SA≤𝒜 : ∀ i → (SA i) IsSubalgebraOf (𝒜 i)
SA≤𝒜 = λ i → snd ∣ ∥ B≤K i ∥ ∣

h : ∀ i → ∣ SA i ∣ → ∣ 𝒜 i ∣
h = λ i → ∣ SA≤𝒜 i ∣

⊓SA≤⊓𝒜 : ⊓ SA ≤ ⊓ 𝒜
⊓SA≤⊓𝒜 = i , ii , iii
  where
    i : ∣ ⊓ SA ∣ → ∣ ⊓ 𝒜 ∣
    i = λ x i → (h i) (x i)
    ii : is-embedding i
    ii = embedding-lift{𝓠 = 𝓤}{𝓤 = 𝓤}{𝓘 = 𝓦} hfe hfe {I}{SA}{𝒜}h(λ i → fst ∥ SA≤𝒜 i ∥)
    iii : is-homomorphism (⊓ SA) (⊓ 𝒜) i
    iii = λ f a → gfe λ i → (snd ∥ SA≤𝒜 i ∥) f (λ x → a x i)
γ : ⊓ ℬ ≅ ⊓ SA
γ = ⊓≅ gfe B≅SA
```

8.3.6 PS(𝒦) ⊆ SP(𝒦)

Finally, we are in a position to prove that a product of subalgebras of algebras in a class 𝒦 is a subalgebra of a product of algebras in 𝒦.

```
module _ {𝓤 : Universe}{𝒦u : Pred (Algebra 𝓤 S)(OV 𝓤)} {hfe : hfunext (OV 𝓤)(OV 𝓤)} where

  𝓾 : Universe
  𝓾 = OV 𝓤

  PS⊆SP : (P{𝓾}{𝓾} (S{𝓤}{𝓾} 𝒦u)) ⊆ (S{𝓾}{𝓾} (P{𝓤}{𝓾} 𝒦u))
  PS⊆SP (pbase (sbase x)) = sbase (pbase x)
  PS⊆SP (pbase (slift{𝐀} x)) = slift splA
    where
      splA : (lift-alg 𝐀 𝓾) ∈ S{𝓾}{𝓾} (P{𝓤}{𝓾} 𝒦u)
      splA = S⊆SP{𝓤}{𝓾}{𝒦u} (slift x)

  PS⊆SP (pbase {𝐁} (ssub{𝐀} sA B≤A)) = siso γ refl-≅
    where
      lA lB : Algebra 𝓾 S
      lA = lift-alg 𝐀 𝓾
      lB = lift-alg 𝐁 𝓾

      ζ : lB ≤ lA
      ζ = lift-alg-lift-≤-lift 𝐁{𝐀} B≤A

      spA : lA ∈ S{𝓾}{𝓾} (P{𝓤}{𝓾} 𝒦u)
      spA = S⊆SP{𝓤}{𝓾}{𝒦u} (slift sA)

      γ : (lift-alg 𝐁 𝓾) ∈ (S{𝓾}{𝓾} (P{𝓤}{𝓾} 𝒦u))
      γ = ssub{𝓤 = 𝓾} spA ζ

  PS⊆SP (pbase {𝐁} (ssubw{𝐀} sA B≤A)) = ssub{𝓤 = 𝓾} splA (lift-alg-≤ 𝐁{𝐀} B≤A)
```

```
    where
     lA lB : Algebra 𝓾 𝑆
     lA = lift-alg 𝐀 𝓾
     lB = lift-alg 𝐁 𝓾

     splA : lA ∈ S{𝓾}{𝓾} (P{𝓤}{𝓾} 𝒦𝓾)
     splA = slift{𝓾}{𝓾} (S⊆SP sA)


  PS⊆SP (pbase (siso{𝐀}{𝐁} x A≅B)) = siso splA ζ
    where
     lA lB : Algebra 𝓾 𝑆
     lA = lift-alg 𝐀 𝓾
     lB = lift-alg 𝐁 𝓾

     ζ : lA ≅ lB
     ζ = lift-alg-iso 𝓤 𝓾 𝐀 𝐁 A≅B

     splA : lA ∈ S{𝓾}{𝓾} (P{𝓤}{𝓾} 𝒦𝓾)
     splA = S⊆SP (slift x)

  PS⊆SP (pliftu x) = slift (PS⊆SP x)
  PS⊆SP (pliftw x) = slift (PS⊆SP x)

  PS⊆SP (produ{I}{𝒜} x) = γ
    where
     ξ : (i : I) → (𝒜 i) IsSubalgebraOfClass (P{𝓤}{𝓾} 𝒦𝓾)
     ξ i = S→subalgebra{𝒦 = (P{𝓤}{𝓾} 𝒦𝓾)} (PS⊆SP (x i))

     η' : ⨅ 𝒜 IsSubalgebraOfClass (P{𝓾}{𝓾} (P{𝓤}{𝓾} 𝒦𝓾))
     η' = lemPS⊆SP{𝓤 = (𝓾)}{𝓾}{𝒦 = (P{𝓤}{𝓾} 𝒦𝓾)}{hfe}{I = I}{ℬ = 𝒜} ξ

     η : ⨅ 𝒜 ∈ S{𝓾}{𝓾} (P{𝓾}{𝓾} (P{𝓤}{𝓾} 𝒦𝓾))
     η = subalgebra→S{𝓤 = (𝓾)}{𝓦 = 𝓾}{𝒦 = (P{𝓾}{𝓾} (P{𝓤}{𝓾} 𝒦𝓾))}{𝐂 = ⨅ 𝒜} η'

     γ : ⨅ 𝒜 ∈ S{𝓾}{𝓾} (P{𝓤}{𝓾} 𝒦𝓾)
     γ = (S-mono{𝓤 = (𝓾)}{𝒦 = (P{𝓾}{𝓾} (P{𝓤}{𝓾} 𝒦𝓾))}{𝒦' = (P{𝓤}{𝓾} 𝒦𝓾)} (P-idemp')) η

  PS⊆SP (prodw{I}{𝒜} x) = γ
    where
     ξ : (i : I) → (𝒜 i) IsSubalgebraOfClass (P{𝓤}{𝓾} 𝒦𝓾)
     ξ i = S→subalgebra{𝒦 = (P{𝓤}{𝓾} 𝒦𝓾)} (PS⊆SP (x i))

     η' : ⨅ 𝒜 IsSubalgebraOfClass (P{𝓾}{𝓾} (P{𝓤}{𝓾} 𝒦𝓾))
     η' = lemPS⊆SP{𝓤 = (𝓾)}{𝓾}{𝒦 = (P{𝓤}{𝓾} 𝒦𝓾)}{hfe}{I = I}{ℬ = 𝒜} ξ

     η : ⨅ 𝒜 ∈ S{𝓾}{𝓾} (P{𝓾}{𝓾} (P{𝓤}{𝓾} 𝒦𝓾))
     η = subalgebra→S{𝓤 = (𝓾)}{𝓦 = 𝓾}{𝒦 = (P{𝓾}{𝓾} (P{𝓤}{𝓾} 𝒦𝓾))}{𝐂 = ⨅ 𝒜} η'

     γ : ⨅ 𝒜 ∈ S{𝓾}{𝓾} (P{𝓤}{𝓾} 𝒦𝓾)
     γ = (S-mono{𝓤 = (𝓾)}{𝒦 = (P{𝓾}{𝓾} (P{𝓤}{𝓾} 𝒦𝓾))}{𝒦' = (P{𝓤}{𝓾} 𝒦𝓾)} (P-idemp')) η

  PS⊆SP (pisou{𝐀}{𝐁} pA A≅B) = siso{𝓾}{𝓾}{P{𝓤}{𝓾} 𝒦𝓾}{𝐀}{𝐁} spA A≅B
```

```
  where
   spA : 𝐀 ∈ S{𝓾}{𝓾} (P{𝓤}{𝓾} 𝒦u)
   spA = PS⊆SP pA

PS⊆SP (pisow{𝐀}{𝐁} pA A≅B) = siso{𝓾}{𝓾}{P{𝓤}{𝓾} 𝒦u}{𝐀}{𝐁} spA A≅B
  where
   spA : 𝐀 ∈ S{𝓾}{𝓾} (P{𝓤}{𝓾} 𝒦u)
   spA = PS⊆SP pA
```

8.3.7 More class inclusions

We conclude this module with three more inclusion relations that will have bit parts to play in our formal proof of Birkhoff's Theorem.

```
P⊆V : {𝓤 𝓦 : Universe}{𝒦 : Pred (Algebra 𝓤 S)(OV 𝓤)}
 → P{𝓤}{𝓦} 𝒦 ⊆ V{𝓤}{𝓦} 𝒦
P⊆V (pbase x) = vbase x
P⊆V{𝓤} (pliftu x) = vlift (P⊆V{𝓤}{𝓤} x)
P⊆V{𝓤}{𝓦} (pliftw x) = vliftw (P⊆V{𝓤}{𝓦} x)
P⊆V (produ x) = vprodu (λ i → P⊆V (x i))
P⊆V (prodw x) = vprodw (λ i → P⊆V (x i))
P⊆V (pisou x x₁) = visou (P⊆V x) x₁
P⊆V (pisow x x₁) = visow (P⊆V x) x₁

S⊆V : {𝓤 𝓦 : Universe}{𝒦 : Pred (Algebra 𝓤 S)(OV 𝓤)}
 → S{𝓤}{𝓦} 𝒦 ⊆ V{𝓤}{𝓦} 𝒦
S⊆V (sbase x) = vbase x
S⊆V (slift x) = vlift (S⊆V x)
S⊆V (ssub x x₁) = vssub (S⊆V x) x₁
S⊆V (ssubw x x₁) = vssubw (S⊆V x) x₁
S⊆V (siso x x₁) = visou (S⊆V x) x₁

SP⊆V : {𝓤 𝓦 : Universe}{𝒦 : Pred (Algebra 𝓤 S)(OV 𝓤)}
 → S{𝓤 ⊔ 𝓦}{𝓤 ⊔ 𝓦} (P{𝓤}{𝓦} 𝒦) ⊆ V{𝓤}{𝓦} 𝒦
SP⊆V (sbase{𝐀} PCloA) = P⊆V (pisow PCloA lift-alg-≅)
SP⊆V (slift{𝐀} x) = vliftw (SP⊆V x)
SP⊆V{𝓤}{𝓦} {𝒦} (ssub{𝐀}{𝐁} spA B≤A) = vssubw (SP⊆V spA) B≤A
SP⊆V{𝓤}{𝓦} {𝒦} (ssubw{𝐀}{𝐁} spA B≤A) = vssubw (SP⊆V spA) B≤A
SP⊆V (siso x x₁) = visow (SP⊆V x) x₁
```

8.3.8 Products of classes

Above we proved PS($\mathcal{K}$) ⊆ SP($\mathcal{K}$). It is slightly more painful to prove that the product of *all* algebras in the class S($\mathcal{K}$) is a member of SP($\mathcal{K}$). That is,

```
⨅ S (𝒦) ∈ SP (𝒦 )
```

This is mainly due to the fact that it's not obvious (at least not to this author-coder) what should be the type of the product of all members of a class of algebras. After a few false starts, eventually the right type revealed itself. Of course, now that we have it in our hands, it seems rather obvious.

We now describe the this type of product of all algebras in an arbitrary class $\mathscr{K}$ of algebras of the same signature.

```
module class-product {𝓤 : Universe}{𝒦 : Pred (Algebra 𝓤 𝑆) (OV 𝓤)} where

  -- ℑ serves as the index of the product
  ℑ : {𝓤 : Universe} → Pred (Algebra 𝓤 𝑆)(OV 𝓤) → (OV 𝓤) ̇
  ℑ {𝓤} 𝒦 = Σ 𝐀 ꞉ (Algebra 𝓤 𝑆) , 𝐀 ∈ 𝒦

   -- 𝔄 produces an algebra for each index (i : ℑ).
  𝔄 : {𝓤 : Universe}{𝒦 : Pred (Algebra 𝓤 𝑆)(OV 𝓤)} → ℑ 𝒦 → Algebra 𝓤 𝑆
  𝔄{𝓤}{𝒦} = λ (i : (ℑ 𝒦)) → ∣ i ∣

  -- The product of all members of 𝒦 can be written simply as follows:
  class-product : {𝓤 : Universe} → Pred (Algebra 𝓤 𝑆)(OV 𝓤) → Algebra (OV 𝓤) 𝑆
  class-product {𝓤} 𝒦 = ⨅ ( 𝔄{𝓤}{𝒦} )

  -- ...or, more explicitly, here is the expansion of this indexed product.
  class-product' : {𝓤 : Universe} → Pred (Algebra 𝓤 𝑆)(OV 𝓤) → Algebra (OV 𝓤) 𝑆
  class-product'{𝓤} 𝒦 = ⨅ λ (i : (Σ 𝐀 ꞉ (Algebra 𝓤 𝑆) , 𝐀 ∈ 𝒦)) → ∣ i ∣
```

Notice that, if $p : \mathbf{A} \in \mathscr{K}$, then we can think of the pair $(\mathbf{A}\ ,\ p) \in \mathfrak{I}\ \mathscr{K}$ as an index over the class, and so we can think of $\mathfrak{A}\ (\mathbf{A}\ ,\ p)$ (which is obviously $\mathbf{A}$) as the projection of the product ⨅ (𝔄{𝓤}{𝒦}) onto the $(\mathbf{A}\ ,\ p)$-th component.

8.3.9 ⨅ S(𝒦) ∈ SP(𝒦)

Finally, we prove the result that plays a leading role in the formal proof of Birkhoff's Theorem—namely, that our newly defined class product ⨅ (𝔄{𝓤}{𝒦}) belongs to SP(𝒦).

```
-- The product of all subalgebras of a class 𝒦 belongs to SP(𝒦).
module class-product-inclusions {𝓤 : Universe} {𝒦 : Pred (Algebra 𝓤 𝑆)(OV 𝓤)} where

  open class-product {𝓤 = 𝓤}{𝒦 = 𝒦}

  class-prod-s-∈-ps : class-product (S{𝓤}{𝓤} 𝒦) ∈ (P{OV 𝓤}{OV 𝓤} (S{𝓤}{OV 𝓤} 𝒦))

  class-prod-s-∈-ps = pisou{𝓤 = (OV 𝓤)}{𝓦 = (OV 𝓤)} ps⨅lA ⨅lA≅cpK
    where
      I : (OV 𝓤) ̇
      I = ℑ (S{𝓤}{𝓤} 𝒦)

      sA : (i : I) → (𝔄 i) ∈ (S{𝓤}{𝓤} 𝒦)
      sA i = ∥ i ∥

      lA llA : I → Algebra (OV 𝓤) 𝑆
      lA i = lift-alg (𝔄 i) (OV 𝓤)
      llA i = lift-alg (lA i) (OV 𝓤)

      slA : (i : I) → (lA i) ∈ (S{𝓤}{(OV 𝓤)} 𝒦)
      slA i = siso (sA i) lift-alg-≅

      psllA : (i : I) → (llA i) ∈ (P{OV 𝓤}{OV 𝓤} (S{𝓤}{(OV 𝓤)} 𝒦))
```

```
    psIIA i = pbase{𝓤 = (OV 𝓤)}{𝓦 = (OV 𝓤)} (sIA i)

    ps⊓IIA : ⊓ IIA ∈ P{OV 𝓤}{OV 𝓤} (S{𝓤}{OV 𝓤} 𝒦)
    ps⊓IIA = produ{𝓤 = (OV 𝓤)}{𝓦 = (OV 𝓤)} psIIA

    IIA≅A : (i : I) → (IIA i) ≅ (𝔄 i)
    IIA≅A i = Trans-≅ (IIA i) (𝔄 i) (sym-≅ lift-alg-≅) (sym-≅ lift-alg-≅)

    ⊓IIA≅cpK : ⊓ IIA ≅ class-product (S{𝓤}{𝓤} 𝒦)
    ⊓IIA≅cpK = ⊓≅ gfe IIA≅A

-- PS⊆SP, so the product of all subalgebras of algebras in 𝒦 belongs to SP(𝒦).
class-prod-s-∈-sp : hfunext (OV 𝓤) (OV 𝓤)
  → class-product (S{𝓤}{𝓤} 𝒦) ∈ (S{OV 𝓤}{OV 𝓤} (P{𝓤}{OV 𝓤} 𝒦))

class-prod-s-∈-sp hfe = PS⊆SP{hfe = hfe} (class-prod-s-∈-ps)
```

8.4 Equation Preservation Theorems

This section presents the UALib.Varieties.Preservation module of the Agda UALib. In this module we show that identities are preserved by closure operators H, S, and P. This will establish the easy direction of Birkhoff's HSP Theorem.

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

open import UALib.Algebras using (Signature; 𝓞; 𝓥; Algebra; _↠_)
open import UALib.Prelude.Preliminaries using (global-dfunext; Universe; _̇)

module UALib.Varieties.Preservation
  {S : Signature 𝓞 𝓥}{gfe : global-dfunext}
  {𝕏 : {𝓤 𝔛 : Universe}{X : 𝔛 ̇ }(𝑨 : Algebra 𝓤 S) → X ↠ 𝑨}
  where

open import UALib.Varieties.Varieties {S = S}{gfe}{𝕏} public
```

8.4.1 H preserves identities

```
--H preserves identities
H-id1 : {𝓤 𝔛 : Universe}{X : 𝔛 ̇}
        {𝒦 : Pred (Algebra 𝓤 S)(OV 𝓤)}
        (p q : Term{𝔛}{X})
         ---------------------------------------
  →     (𝒦 ⊧ p ≋ q) → (H{𝓤}{𝓤} 𝒦 ⊧ p ≋ q)

H-id1 p q α (hbase x) = lift-alg-⊧ _ p q (α x)

H-id1 {𝓤} p q α (hlift{𝑨} x) = γ
  where
   β : 𝑨 ⊧ p ≈ q
   β = H-id1 p q α x
   γ : lift-alg 𝑨 𝓤 ⊧ p ≈ q
```

```
    γ = lift-alg-⊧ _ p q β

  H-id1 p q α (hhimg{A}{C} HA ((B , ϕ , (ϕhom , ϕsur)) , B≅C) ) = ⊧-≅ p q γ B≅C
    where
    β : A ⊧ p ≈ q
    β = (H-id1 p q α) HA

    preim : ∀ b x → ∣ A ∣
    preim b x = (Inv ϕ (b x) (ϕsur (b x)))

    ζ : ∀ b → ϕ ∘ (preim b) ≡ b
    ζ b = gfe λ x → InvIsInv ϕ (b x) (ϕsur (b x))

    γ : (p ̇ B) ≡ (q ̇ B)
    γ = gfe λ b →
      (p ̇ B) b              ≡⟨ (ap (p ̇ B) (ζ b))⁻¹ ⟩
      (p ̇ B) (ϕ ∘ (preim b)) ≡⟨ (comm-hom-term gfe A B (ϕ , ϕhom) p (preim b))⁻¹ ⟩
      ϕ((p ̇ A)(preim b))    ≡⟨ ap ϕ (intensionality β (preim b)) ⟩
      ϕ((q ̇ A)(preim b))    ≡⟨ comm-hom-term gfe A B (ϕ , ϕhom) q (preim b) ⟩
      (q ̇ B)(ϕ ∘ (preim b)) ≡⟨ ap (q ̇ B) (ζ b) ⟩
      (q ̇ B) b              ∎

  H-id1 p q α (hiso{A}{B} x x₁) = ⊧-transport p q (H-id1 p q α x) x₁
```

The converse is almost too obvious to bother with. Nonetheless, we formalize it for completeness.

```
  H-id2 : {𝓤 𝓦 𝓧 : Universe}{X : 𝓧 ̇}{𝒦 : Pred (Algebra 𝓤 S)(OV 𝓤)}
             {p q : Term{𝓧}{X}} → (H{𝓤}{𝓦} 𝒦 ⊧ p ≋ q) → (𝒦 ⊧ p ≋ q)
  H-id2 {𝓤}{𝓦}{𝓧}{X} {𝒦} {p}{q} Hpq {A} KA = γ
    where
    lA : Algebra (𝓤 ⊔ 𝓦) S
    lA = lift-alg A 𝓦

    plA : lA ∈ H{𝓤}{𝓦} 𝒦
    plA = hbase KA

    ξ : lA ⊧ p ≈ q
    ξ = Hpq plA
    γ : A ⊧ p ≈ q
    γ = lower-alg-⊧ A p q ξ
```

8.4.2 S preserves identities

```
  S-id1 : {𝓤 𝓧 : Universe}{X : 𝓧 ̇}
          (𝒦 : Pred (Algebra 𝓤 S)(OV 𝓤))
          (p q : Term{𝓧}{X})
          ------------------------------------
    →     (𝒦 ⊧ p ≋ q) → (S{𝓤}{𝓤} 𝒦 ⊧ p ≋ q)

  S-id1 _ p q α (sbase x) = lift-alg-⊧ _ p q (α x)

  S-id1 𝒦 p q α (slift x) = lift-alg-⊧ _ p q ((S-id1 𝒦 p q α) x)
```

```
S-id1 𝒦 p q α (ssub{A}{B} sA B≤A) =
  S-⊧ p q ((B , A , (B , B≤A) , inj₂ refl , id≅)) γ
   where --Apply S-⊧ to the class 𝒦 ∪ ｛ A ｝
    β : A ⊧ p ≈ q
    β = S-id1 𝒦 p q α sA

    Apq : ｛ A ｝ ⊧ p ≋ q
    Apq (refl _) = β

    γ : (𝒦 ∪ ｛ A ｝) ⊧ p ≋ q
    γ {B} (inj₁ x) = α x
    γ {B} (inj₂ y) = Apq y

S-id1 𝒦 p q α (ssubw{A}{B} sA B≤A) =
  S-⊧ p q ((B , A , (B , B≤A) , inj₂ refl , id≅) ) γ
   where --Apply S-⊧ to the class 𝒦 ∪ ｛ A ｝
    β : A ⊧ p ≈ q
    β = S-id1 𝒦 p q α sA

    Apq : ｛ A ｝ ⊧ p ≋ q
    Apq (refl _) = β

    γ : (𝒦 ∪ ｛ A ｝) ⊧ p ≋ q
    γ {B} (inj₁ x) = α x
    γ {B} (inj₂ y) = Apq y

S-id1 𝒦 p q α (siso{A}{B} x x₁) = γ
  where
   ζ : A ⊧ p ≈ q
   ζ = S-id1 𝒦 p q α x
   γ : B ⊧ p ≈ q
   γ = ⊧-transport p q ζ x₁
```

Again, the obvious converse is barely worth the bits needed to formalize it.

```
S-id2 : {𝓤 𝓦 𝓧 : Universe}{X : 𝓧 ̇}{𝒦 : Pred (Algebra 𝓤 S)(OV 𝓤)}
          {p q : Term{𝓧}{X}} → (S{𝓤}{𝓦} 𝒦 ⊧ p ≋ q) → (𝒦 ⊧ p ≋ q)
S-id2 {𝓤}{𝓦}{𝓧}{X} {𝒦} {p}{q} Spq {A} KA = γ
  where
   lA : Algebra (𝓤 ⊔ 𝓦) S
   lA = lift-alg A 𝓦

   plA : lA ∈ S{𝓤}{𝓦} 𝒦
   plA = sbase KA

   ξ : lA ⊧ p ≈ q
   ξ = Spq plA
   γ : A ⊧ p ≈ q
   γ = lower-alg-⊧ A p q ξ
```

8.4.3 P preserves identities

```
P-id1 : {𝓤 𝓧 : Universe}{X : 𝓧 ̇}
        {𝒦 : Pred (Algebra 𝓤 S)(OV 𝓤)}
        (p q : Term{𝓧}{X})
        -----------------------------------
 →      (𝒦 ⊧ p ≋ q) → (P{𝓤}{𝓤} 𝒦 ⊧ p ≋ q)

P-id1 p q α (pbase x) = lift-alg-⊧ _ p q (α x)
P-id1 p q α (pliftu x) = lift-alg-⊧ _ p q ((P-id1 p q α) x)
P-id1 p q α (pliftw x) = lift-alg-⊧ _ p q ((P-id1 p q α) x)
P-id1 {𝓤} {𝓧} p q α (produ{I}{𝒜} x) = γ
  where
   lA : I → Algebra 𝓤 S
   lA i = (lift-alg (𝒜 i) 𝓤)

   IH : (i : I) → (p ̇ (lA i)) ≡ (q ̇ (lA i))
   IH i = lift-alg-⊧ (𝒜 i) p q ((P-id1 p q α) (x i))

   γ : p ̇ (⨅ 𝒜) ≡ q ̇ (⨅ 𝒜)
   γ = lift-products-preserve-ids p q I 𝒜 IH

P-id1{𝓤} p q α (prodw{I}{𝒜} x) = γ
  where
   lA : I → Algebra 𝓤 S
   lA i = (lift-alg (𝒜 i) 𝓤)

   IH : (i : I) → (p ̇ (lA i)) ≡ (q ̇ (lA i))
   IH i = lift-alg-⊧ (𝒜 i) p q ((P-id1 p q α) (x i))

   γ : p ̇ (⨅ 𝒜) ≡ q ̇ (⨅ 𝒜)
   γ = lift-products-preserve-ids p q I 𝒜 IH

P-id1 p q α (pisou{𝐀}{𝐁} x x₁) = γ
  where
   γ : 𝐁 ⊧ p ≈ q
   γ = ⊧-transport p q (P-id1 p q α x) x₁

P-id1 p q α (pisow{𝐀}{𝐁} x x₁) = ⊧-transport p q ζ x₁
  where
   ζ : 𝐀 ⊧ p ≈ q
   ζ = P-id1 p q α x
```

…and conversely…

```
P-id2 : {𝓤 𝓦 𝓧 : Universe}{X : 𝓧 ̇}
        (𝒦 : Pred (Algebra 𝓤 S)(OV 𝓤))
        {p q : Term{𝓧}{X}}
        ------------------------------------
 →      ((P{𝓤}{𝓦} 𝒦) ⊧ p ≋ q ) → (𝒦 ⊧ p ≋ q)

P-id2 {𝓤}{𝓦} 𝒦 {p}{q} PKpq {𝐀} KA = γ
  where
```

```
    lA : Algebra (𝓤 ⊔ 𝓦) S
    lA = lift-alg 𝐀 𝓦

    plA : lA ∈ P{𝓤}{𝓦} 𝒦
    plA = pbase KA

    ξ : lA ⊧ p ≈ q
    ξ = PKpq plA
    γ : 𝐀 ⊧ p ≈ q
    γ = lower-alg-⊧ 𝐀 p q ξ
```

8.4.4 V preserves identities

```
-- V preserves identities

V-id1 : {𝓤 𝓧 : Universe}{X : 𝓧 ̇}{𝒦 : Pred (Algebra 𝓤 S)(OV 𝓤)}
        (p q : Term{𝓧}{X}) → (𝒦 ⊧ p ≋ q) → (V{𝓤}{𝓤} 𝒦 ⊧ p ≋ q)
V-id1 p q α (vbase x) = lift-alg-⊧ _ p q (α x)
V-id1 {𝓤}{𝓧}{X}{𝒦} p q α (vlift{𝐀} x) = γ
  where
    β : 𝐀 ⊧ p ≈ q
    β = (V-id1 p q α) x
    γ : lift-alg 𝐀 𝓤 ⊧ p ≈ q
    γ = lift-alg-⊧ 𝐀 p q β
V-id1 {𝓤}{𝓧}{X}{𝒦} p q α (vliftw{𝐀} x) = γ
  where
    β : 𝐀 ⊧ p ≈ q
    β = (V-id1 p q α) x
    γ : lift-alg 𝐀 𝓤 ⊧ p ≈ q
    γ = lift-alg-⊧ 𝐀 p q β
V-id1 p q α (vhimg{𝐀}{𝐂} VA ((𝐁 , ϕ , (ϕh , ϕE)) , B≅C)) = ⊧-≅ p q γ B≅C
  where
    IH : 𝐀 ⊧ p ≈ q
    IH = V-id1 p q α VA

    preim : ∀ 𝒃 x → ∣ 𝐀 ∣
    preim 𝒃 x = (Inv ϕ (𝒃 x) (ϕE (𝒃 x)))

    ζ : ∀ 𝒃 → ϕ ∘ (preim 𝒃) ≡ 𝒃 -- (𝒃 : → ∣ 𝐁 ∣)(x : X)
    ζ 𝒃 = gfe λ x → InvIsInv ϕ (𝒃 x) (ϕE (𝒃 x))

    γ : (p ̇ 𝐁) ≡ (q ̇ 𝐁)
    γ = gfe λ 𝒃 →
      (p ̇ 𝐁) 𝒃              ≡⟨ (ap (p ̇ 𝐁) (ζ 𝒃))⁻¹ ⟩
      (p ̇ 𝐁) (ϕ ∘ (preim 𝒃)) ≡⟨ (comm-hom-term gfe 𝐀 𝐁 (ϕ , ϕh) p (preim 𝒃))⁻¹ ⟩
      ϕ((p ̇ 𝐀)(preim 𝒃)) ≡⟨ ap ϕ (intensionality IH (preim 𝒃)) ⟩
      ϕ((q ̇ 𝐀)(preim 𝒃)) ≡⟨ comm-hom-term gfe 𝐀 𝐁 (ϕ , ϕh) q (preim 𝒃) ⟩
      (q ̇ 𝐁)(ϕ ∘ (preim 𝒃)) ≡⟨ ap (q ̇ 𝐁) (ζ 𝒃) ⟩
      (q ̇ 𝐁) 𝒃              ∎

V-id1{𝓤}{𝓧}{X}{𝒦} p q α ( vssub {𝐀}{𝐁} VA B≤A ) =
```

```
  S-⊧ p q ((𝐁 , 𝐀 , (𝐁 , B≤A) , inj₂ 𝓻𝓮𝒻𝓵 , id≅) ) γ
   where
    IH : 𝐀 ⊧ p ≈ q
    IH = V-id1 {𝓤}{𝓧}{X}p q α VA

    Asinglepq : ｛ 𝐀 ｝ ⊧ p ≋ q
    Asinglepq (refl _) = IH

    γ : (𝒦 ∪ ｛ 𝐀 ｝) ⊧ p ≋ q
    γ {𝐁} (inj₁ x) = α x
    γ {𝐁} (inj₂ y) = Asinglepq y

 V-id1{𝓤}{𝓧}{X}{𝒦} p q α ( vssubw {𝐀}{𝐁} VA B≤A ) =
  S-⊧ p q ((𝐁 , 𝐀 , (𝐁 , B≤A) , inj₂ 𝓻𝓮𝒻𝓵 , id≅) ) γ
   where
    IH : 𝐀 ⊧ p ≈ q
    IH = V-id1 {𝓤}{𝓧}{X}p q α VA

    Asinglepq : ｛ 𝐀 ｝ ⊧ p ≋ q
    Asinglepq (refl _) = IH

    γ : (𝒦 ∪ ｛ 𝐀 ｝) ⊧ p ≋ q
    γ {𝐁} (inj₁ x) = α x
    γ {𝐁} (inj₂ y) = Asinglepq y

 V-id1 {𝓤}{𝓧}{X} p q α (vprodu{I}{𝒜} V𝒜) = γ
  where
   IH : (i : I) → 𝒜 i ⊧ p ≈ q
   IH i = V-id1{𝓤}{𝓧}{X} p q α (V𝒜 i)

   γ : p ̇ (⨅ 𝒜) ≡ q ̇ (⨅ 𝒜)
   γ = product-id-compatibility p q I 𝒜 IH

 V-id1 {𝓤}{𝓧}{X} p q α (vprodw{I}{𝒜} V𝒜) = γ
  where
   IH : (i : I) → 𝒜 i ⊧ p ≈ q
   IH i = V-id1{𝓤}{𝓧}{X} p q α (V𝒜 i)

   γ : p ̇ (⨅ 𝒜) ≡ q ̇ (⨅ 𝒜)
   γ = product-id-compatibility p q I 𝒜 IH
 V-id1 p q α (visou{𝐀}{𝐁} VA A≅B) = ⊧-≅ p q (V-id1 p q α VA) A≅B
 V-id1 p q α (visow{𝐀}{𝐁} VA A≅B) = ⊧-≅ p q (V-id1 p q α VA) A≅B
```

Once again, and for the last time, completeness dictates that we formalize the coverse, however obvious it may be.

```
 V-id2 : {𝓤 𝓦 𝓧 : Universe}{X : 𝓧 ̇}{𝒦 : Pred (Algebra 𝓤 S)(OV 𝓤)}
          {p q : Term{𝓧}{X}} → (V{𝓤}{𝓦} 𝒦 ⊧ p ≋ q) → (𝒦 ⊧ p ≋ q)
 V-id2 {𝓤}{𝓦}{𝓧}{X} {𝒦} {p}{q} Vpq {𝐀} KA = γ
  where
   lA : Algebra (𝓤 ⊔ 𝓦) S
   lA = lift-alg 𝐀 𝓦
```

```
    vlA : lA ∈ V{𝓤}{𝓦} 𝒦
    vlA = vbase KA

    ξ : lA ⊧ p ≈ q
    ξ = Vpq vlA
    γ : 𝐀 ⊧ p ≈ q
    γ = lower-alg-⊧ 𝐀 p q ξ
```

8.4.5 Class identities

It follows from V-id1 that, if 𝒦 is a class of structures, the set of identities modeled by all structures in 𝒦 is the same as the set of identities modeled by all structures in V 𝒦.

```
-- Th (V 𝒦) is precisely the set of identities modeled by 𝒦
class-identities : {𝓤 𝓧 : Universe}{X : 𝓧 ̇}{𝒦 : Pred (Algebra 𝓤 S) (OV 𝓤)}
                   (p q : ∣ 𝑻 X ∣)
                   ----------------------------------------------------------
 →                 𝒦 ⊧ p ≋ q ⇔ ((p , q) ∈ Th (V 𝒦))

class-identities{𝓤}{𝓧}{X}{𝒦} p q = ⇒ , ⇐
 where
  ⇒ : 𝒦 ⊧ p ≋ q → p , q ∈ Th (V 𝒦)
  ⇒ = λ α VCloA → V-id1 p q α VCloA

  ⇐ : p , q ∈ Th (V 𝒦) → 𝒦 ⊧ p ≋ q
  ⇐ = λ Thpq {𝐀} KA → lower-alg-⊧ 𝐀 p q (Thpq (vbase KA))
```

9 Birkhoff's HSP Theorem

9.1 The Relatively Free Algebra

This section presents the UALib.Birkhoff.FreeAlgebra module of the Agda UALib. Recall, we proved in Section 6.2 that the term algebra $\mathbf{T}\ X$ is the absolutely free algebra in the class of all S-structures. In this section, we formalize, for a given class $\mathcal{K}$ of S-algebras, the (relatively) free algebra in S (P $\mathcal{K}$) over X. Indeed, a free algebra *for* a class $\mathcal{K}$ induces the free algebra *in* the variety generated by $\mathcal{K}$, via the following definitions:[8]

$$\Theta(\mathcal{K}, \mathbf{A}) := \{\theta \in \mathsf{Con}\,\mathbf{A} : \mathbf{A} \,/\, \theta \in \mathsf{S}\mathcal{K}\} \qquad \text{and} \qquad \psi(\mathcal{K}, \mathbf{A}) := \cap\Theta(\mathcal{K}, \mathbf{A}).$$

The free algebra *in* the variety generated by $\mathcal{K}$ is constructed by applying these definitions to the special case in which $\mathbf{A}$ is the term algebra $\mathbf{T}\ X$ of S-terms over X.

Since $\mathbf{T}\ X$ is free for the class of all S-algebras, it is free for every subclass $\mathcal{K}$ of S-algebras. Of course, $\mathbf{T}\ X$ is not necessarily a member of $\mathcal{K}$, but if we form the quotient of $\mathbf{T}\ X$ modulo the congruence $\psi(\mathcal{K}, \mathbf{T}\ X)$, which we denote and define by $\mathfrak{F} := \mathbf{T}\ X \,/\, (\psi\mathsf{Con}\ \mathcal{K})$, then it should be clear 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 $\mathfrak{F} \in$ S (P $\mathcal{K}$).

The $\mathfrak{F}$ that we just defined is called the **free algebra over** $\mathcal{K}$ **generated by** X and, by what we just observed we may say that $\mathfrak{F}$ is free *in* S (P $\mathcal{K}$).

To represent $\mathfrak{F}$ as a type in Agda, we start with the congruence $\psi(\mathcal{K}, \mathbf{T}\ X) = \cap\,\Theta(\mathcal{K}, \mathbf{T}\ X)$, where $\Theta(\mathcal{K}, \mathbf{T}\ X) := \{\ \theta \in \mathsf{Con}(\mathbf{T}\ X) : \mathbf{A} \,/\, \theta \in (\mathsf{S}\ \mathcal{K})\}$.[9]

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

open import UALib.Algebras using (Signature; 𝓞; 𝓥; Algebra; _↠_)
open import UALib.Prelude.Preliminaries using (global-dfunext; Universe; _̇)

module UALib.Birkhoff.FreeAlgebra
  {S : Signature 𝓞 𝓥}{gfe : global-dfunext}
  {𝕏 : {𝓤 𝓧 : Universe}{X : 𝓧 ̇ }(A : Algebra 𝓤 S) → X ↠ A}
  where

open import UALib.Varieties.Preservation {S = S}{gfe}{𝕏} public
```

9.1.1 The free algebra

In this subsection we define the relatively free algebra in Agda. Throughout this section, we assume we have two ambient universes $\mathcal{U}$ and $\mathfrak{X}$, as well as a type $X : \mathfrak{X}\ ^{\cdot}$ at our disposal. As usual, this is accomplished with the module directive.

```
module the-free-algebra {𝓤 𝓧 : Universe}{X : 𝓧 ̇} where
```

We begin by defining the collection Timg of homomorphic images of the term algebra.

```
-- H (𝑻 X) (hom images of 𝑻 X)
Timg : Pred (Algebra 𝓤 S) (OV 𝓤) → 𝓞 ⊔ 𝓥 ⊔ (𝓤 ⊔ 𝓧)⁺ ̇
Timg 𝒦 = Σ 𝑨 ꞉ (Algebra 𝓤 S) , Σ ϕ ꞉ hom (𝑻 X) 𝑨 , (𝑨 ∈ 𝒦) × Epic ∣ ϕ ∣
```

8 If $\Theta(\mathcal{K}, \mathbf{A})$ is empty, then $\psi(\mathcal{K}, \mathbf{A}) = 1$ and $\mathbf{A} \,/\, \psi(\mathcal{K}, \mathbf{A})$ is trivial.

9 Since X is not a subset of $\mathfrak{F}$, technically it doesn't make sense to say "X generates $\mathfrak{F}$." But as long as $\mathcal{K}$ contains a nontrivial algebra, $\psi(\mathcal{K}, \mathbf{T}\ X) \cap X^2$ will be nonempty, and we can identify X with $X \,/\, \psi(\mathcal{K}, \mathbf{T}X)$ which does belong to $\mathfrak{F}$.

The Sigma type we use to define **T**img is the type of algebras $\mathbf{A} \in \mathcal{K}$ such that there exists a surjective homomorphism ϕ : hom ($\mathbf{T}$ X) $\mathbf{A}$. This is precisely the collection of all homomorphic images of $\mathbf{T}$ X.

An inhabitant of type **T**img is a quadruple, ($\mathbf{A}$, ϕ , ka , ϕE), where $\mathbf{A}$ is an algebra, ϕ : hom ($\mathbf{T}$ X) $\mathbf{A}$ is a homomorphism, ka : $\mathbf{A} \in \mathcal{K}$ is a proof that $\mathbf{A}$ belongs to $\mathcal{K}$, and ϕE is a proof that the underlying map $|\ \phi\ |$ is surjective.

Next we define a function mkti that takes an arbitrary algebra $\mathbf{A}$ and returns an inhabitant of **T**img, which is a proof that $\mathbf{A}$ is a homomorphic image of $\mathbf{T}$ X.

```
-- Every algebra is a hom image of T X.
mkti : {𝒦 : Pred (Algebra 𝓤 S) (OV 𝓤)}(A : Algebra 𝓤 S)
  → A ∈ 𝒦 → Timg 𝒦
mkti A KA = (A , fst thg , KA , snd thg)
  where
    thg : Σ h ꞉ (hom (T X) A), Epic | h |
    thg = Thom-gen A
```

Occasionally we want to extract the homomorphism ϕ from an inhabitant of **T**img, so we define.

```
-- The hom part of a hom image of T X.
Tϕ : (𝒦 : Pred (Algebra 𝓤 S) (OV 𝓤))(ti : Timg 𝒦)
  → hom (T X) | ti |
Tϕ _ ti = fst ∥ ti ∥
```

Finally, it is time to define the congruence relation modulo which $\mathbf{T}$ X will yield 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) (OV 𝓤)) → Pred (∣ T X ∣ × ∣ T X ∣) (OV 𝓤)
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
```

We convert the predicate ψ into a relation by Currying.

```
ψRel : (𝒦 : Pred (Algebra 𝓤 S) (OV 𝓤)) → Rel ∣ (T X) ∣ (OV 𝓤)
ψRel 𝒦 p q = ψ 𝒦 (p , q)
```

We will want to express ψRel as a congruence of the term algebra $\mathbf{T}$ X, so we must prove that ψRel is compatible with the operations of $\mathbf{T}$ X (which are jsut the terms themselves) and that ψRel an equivalence relation.

```
ψcompatible : (𝒦 : Pred (Algebra 𝓤 S) (OV 𝓤))
  → compatible (T X) (ψRel 𝒦)
ψcompatible 𝒦 f {i} {j} iψj A sA = γ
  where
    ti : Timg (S{𝓤}{𝓤} 𝒦)
    ti = mkti A sA

    ϕ : hom (T X) A
    ϕ = fst ∥ ti ∥

    γ : ∣ ϕ ∣ ((f ̂ T X) i) ≡ ∣ ϕ ∣ ((f ̂ T X) j)
    γ = ∣ ϕ ∣ ((f ̂ T X) i) ≡⟨ ∥ ϕ ∥ f i ⟩
```

```
    (f ̂ A) (| ϕ | ∘ i) ≡⟨ ap (f ̂ A) (gfe λ x → ((iψj x) A sA)) ⟩
    (f ̂ A) (| ϕ | ∘ j) ≡⟨ (∥ ϕ ∥ f j)⁻¹ ⟩
    | ϕ | ((f ̂ T X) j) ∎

ψRefl : {𝒦 : Pred (Algebra 𝓤 S) (OV 𝓤)} → reflexive (ψRel 𝒦)
ψRefl = λ x C ϕ → refl

ψSymm : {𝒦 : Pred (Algebra 𝓤 S) (OV 𝓤)} → symmetric (ψRel 𝒦)
ψSymm p q pψRelq C ϕ = (pψRelq C ϕ)⁻¹

ψTrans : {𝒦 : Pred (Algebra 𝓤 S) (OV 𝓤)} → transitive (ψRel 𝒦)
ψTrans p q r pψq qψr C ϕ = (pψq C ϕ) · (qψr C ϕ)

ψIsEquivalence : {𝒦 : Pred (Algebra 𝓤 S) (OV 𝓤)} → IsEquivalence (ψRel 𝒦)
ψIsEquivalence = record { rfl = ψRefl ; sym = ψSymm ; trans = ψTrans }
```

We have collected all the pieces necessary to express the collection of identities satisfied by all algebras in the class as a congruence relation of the term algebra. We call this congruence ψCon and define it using the Congruence constructor mkcon.

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

9.1.2 The relatively free algebra

We will denote the relatively free algebra by 𝔉 or 𝔽 and construct it as the quotient T X ╱ (ψCon 𝒦).

```
open the-free-algebra

module the-relatively-free-algebra
  {𝓤 𝔛 : Universe}{X : 𝔛 ̇}
  {𝒦 : Pred (Algebra 𝓤 S) (OV 𝓤)} where

  𝓕 : Universe -- (universe level of the relatively free algebra)
  𝓕 = (𝔛 ⊔ (OV 𝓤))⁺

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

The domain, | 𝔉 |, is defined by

(| T X | ╱ ⟨ θ ⟩) = Σ C ꞉ _ , Σ p ꞉ | T X | , C ≡ ([p] ≈)

which is the collection { C : ∃ p ∈ | T X | , C ≡ [p] } of θ-classes of T X.

```
𝔉-free-lift : {𝓦 : Universe}(A : Algebra 𝓦 S)
              (h₀ : X → | A |) → | 𝔉 | → | A |

𝔉-free-lift {𝓦}A h₀ (_ , x , _) = (free-lift{𝔛}{𝓦} A h₀) x

𝔉-free-lift-interpretation : (A : Algebra 𝓤 S)
                             (h₀ : X → | A |)(x : | 𝔉 |)
                             -------------------------------------
 →                           (⌜ x ⌝ ̇ A) h₀ ≡ 𝔉-free-lift A h₀ x
```

```
𝔉-free-lift-interpretation 𝐀 f x = free-lift-interpretation 𝐀 f ⌜ x ⌝

𝔉-lift-hom : {𝓦 : Universe}(𝐀 : Algebra 𝓦 S)
             (h₀ : X → ∣ 𝐀 ∣) → hom 𝔉 𝐀

𝔉-lift-hom 𝐀 h₀ = f , fhom
  where
   f : ∣ 𝔉 ∣ → ∣ 𝐀 ∣
   f = 𝔉-free-lift 𝐀 h₀

   ϕ : hom (𝑻 X) 𝐀
   ϕ = lift-hom 𝐀 h₀

   fhom : is-homomorphism 𝔉 𝐀 f
   fhom 𝑓 𝒂 = ∥ ϕ ∥ 𝑓 (λ i → ⌜ 𝒂 i ⌝ )

𝔉-lift-agrees-on-X : {𝓦 : Universe}(𝐀 : Algebra 𝓦 S)
                     (h₀ : X → ∣ 𝐀 ∣)(x : X)
                     ----------------------------------------
 →                   h₀ x ≡ ( ∣ 𝔉-lift-hom 𝐀 h₀ ∣ ⟦ 𝑔 x ⟧ )

𝔉-lift-agrees-on-X _ h₀ x = 𝓇ℯ𝒻𝓁

𝔉-lift-of-epic-is-epic : {𝓦 : Universe}(𝐀 : Algebra 𝓦 S)
                         (h₀ : X → ∣ 𝐀 ∣) → Epic h₀
                         --------------------------------
 →                       Epic ∣ 𝔉-lift-hom 𝐀 h₀ ∣

𝔉-lift-of-epic-is-epic 𝐀 h₀ hE y = γ
  where
   h₀pre : Image h₀ ∋ y
   h₀pre = hE y

   h₀⁻¹y : X
   h₀⁻¹y = Inv h₀ y (hE y)

   η : y ≡ ( ∣ 𝔉-lift-hom 𝐀 h₀ ∣ ⟦ 𝑔 (h₀⁻¹y) ⟧ )
   η = y ≡⟨ (InvIsInv h₀ y h₀pre)⁻¹ ⟩
       h₀ h₀⁻¹y ≡⟨ (𝔉-lift-agrees-on-X) 𝐀 h₀ h₀⁻¹y ⟩
       ∣ 𝔉-lift-hom 𝐀 h₀ ∣ ⟦ (𝑔 h₀⁻¹y) ⟧ ∎

   γ : Image ∣ 𝔉-lift-hom 𝐀 h₀ ∣ ∋ y
   γ = eq y (⟦ 𝑔 h₀⁻¹y ⟧) η

𝑻-canonical-projection : (θ : Congruence{OV 𝔛}{𝔘} (𝑻 X)) → epi (𝑻 X) ((𝑻 X) ╱ θ)
𝑻-canonical-projection θ = canonical-projection (𝑻 X) θ

𝔉-canonical-projection : epi (𝑻 X) 𝔉
𝔉-canonical-projection = canonical-projection (𝑻 X) (ψCon 𝒦)
```

```
πℱ : hom (𝐓 X) ℱ
πℱ = epi-to-hom (𝐓 X) {ℱ} ℱ-canonical-projection

πℱ-X-defined : (g : hom (𝐓 X) ℱ)
  → ((x : X) → ∣ g ∣ (ℊ x) ≡ ⟦ ℊ x ⟧)
  → (t : ∣ 𝐓 X ∣ )
               ---------------------------------
  → ∣ g ∣ t ≡ ⟦ t ⟧

πℱ-X-defined g gx t = free-unique gfe ℱ g πℱ gπℱ-agree-on-X t
  where
    gπℱ-agree-on-X : ((x : X) → ∣ g ∣ (ℊ x) ≡ ∣ πℱ ∣ ( ℊ x ))
    gπℱ-agree-on-X x = gx x


X↪ℱ : X → ∣ ℱ ∣
X↪ℱ x = ⟦ ℊ x ⟧


ψlem : (p q : ∣ 𝐓 X ∣ )
  → ∣ lift-hom ℱ X↪ℱ ∣ p ≡ ∣ lift-hom ℱ X↪ℱ ∣ q
        ----------------------------------------------
  → (p , q) ∈ ψ 𝒦

ψlem p q gpgq 𝐀 sA = γ
    where
      g : hom (𝐓 X) ℱ
      g = lift-hom ℱ (X↪ℱ)

      h₀ : X → ∣ 𝐀 ∣
      h₀ = fst (𝕏 𝐀)

      f : hom ℱ 𝐀
      f = ℱ-lift-hom 𝐀 h₀

      h ϕ : hom (𝐓 X) 𝐀
      h = HomComp (𝐓 X) 𝐀 g f
      ϕ = 𝐓ϕ (S 𝒦) (mkti 𝐀 sA)

        --(homs from 𝐓 X to 𝐀 that agree on X are equal)
      lift-agreement : (x : X) → h₀ x ≡ ∣ f ∣ ⟦ ℊ x ⟧
      lift-agreement x = ℱ-lift-agrees-on-X 𝐀 h₀ x
      fgx≡ϕ : (x : X) → (∣ f ∣ ∘ ∣ g ∣) (ℊ x) ≡ ∣ ϕ ∣ (ℊ x)
      fgx≡ϕ x = (lift-agreement x)⁻¹
      h≡ϕ : ∀ t → (∣ f ∣ ∘ ∣ g ∣) t ≡ ∣ ϕ ∣ t
      h≡ϕ t = free-unique gfe 𝐀 h ϕ fgx≡ϕ t

      γ : ∣ ϕ ∣ p ≡ ∣ ϕ ∣ q
      γ = ∣ ϕ ∣ p ≡⟨ (h≡ϕ p)⁻¹ ⟩ (∣ f ∣ ∘ ∣ g ∣) p
                  ≡⟨ 𝓇ℯ𝒻𝓁 ⟩ ∣ f ∣ ( ∣ g ∣ p )
                  ≡⟨ ap ∣ f ∣ gpgq ⟩ ∣ f ∣ ( ∣ g ∣ q )
                  ≡⟨ h≡ϕ q ⟩ ∣ ϕ ∣ q ∎
```

9.2 HSP Lemmas

This section presents the UALib.Birkhoff.Lemmata of the Agda UALib. Here we establish some facts that will be needed in the proof of Birkhoff's HSP Theorem.
Warning: not all of these are very interesting!

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

open import UALib.Algebras using (Signature; 𝓞; 𝓥; Algebra; _↠_)
open import UALib.Prelude.Preliminaries using (global-dfunext; Universe; _̇)

module UALib.Birkhoff.Lemmata
  {S : Signature 𝓞 𝓥}{gfe : global-dfunext}
  {𝕏 : {𝓤 𝓧 : Universe}{X : 𝓧 ̇ }(A : Algebra 𝓤 S) → X ↠ A}
  {𝓤 : Universe} {X : 𝓤 ̇}
  where

open import UALib.Birkhoff.FreeAlgebra {S = S}{gfe}{𝕏} public
```

9.2.1 Lemma 0: V is closed under lift

We begin with the first of a number of facts that later, in Section 9.3, we use to prove Birkhoff's HSP Theorem.

```
open the-free-algebra {𝓤}{𝓤}{X}

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


  -- NOTATION.
  ovu ovu+ ovu++ : Universe
  ovu = 𝓞 ⊔ 𝓥 ⊔ 𝓤 ⁺
  ovu+ = ovu ⁺
  ovu++ = ovu ⁺ ⁺
```

Next we prove the lift-alg-V-closure lemma, which says that if an algebra 𝑨 belongs to the variety 𝕍, then so does its lift. This enables us to quickly dispense with any universe level problems that will inevitably arise later—a minor techinical issue, but the proof is long and tedious, not to mention uninteresting.

```
open Lift
lift-alg-V-closure -- (alias)
  VlA : {𝑨 : Algebra ovu S}
    → 𝑨 ∈ V{𝓤}{ovu} 𝒦
      ---------------------------------
    → lift-alg 𝑨 ovu+ ∈ V{𝓤}{ovu+} 𝒦
```

```
VlA (vbase{𝐀} x) = visow (vbase{𝓤}{𝓦 = ovu+} x) A≅B
  where
  A≅B : lift-alg 𝐀 ovu+ ≅ lift-alg (lift-alg 𝐀 ovu) ovu+
  A≅B = lift-alg-associative 𝐀

VlA (vlift{𝐀} x) = visow (vlift{𝓤}{𝓦 = ovu+} x) A≅B
  where
  A≅B : lift-alg 𝐀 ovu+ ≅ lift-alg (lift-alg 𝐀 ovu) ovu+
  A≅B = lift-alg-associative 𝐀

VlA (vliftw{𝐀} x) = visow (VlA x) A≅B
  where
  A≅B : (lift-alg 𝐀 ovu+) ≅ lift-alg (lift-alg 𝐀 ovu) ovu+
  A≅B = lift-alg-associative 𝐀

VlA (vhimg{𝐀}{𝐁} x hB) = vhimg (VlA x) (lift-alg-hom-image hB)

VlA (vssub{𝐀}{𝐁} x B≤A) = vssubw (vlift x) lB≤lA
  where
  lB≤lA : lift-alg 𝐁 ovu+ ≤ lift-alg 𝐀 ovu+
  lB≤lA = lift-alg-≤ 𝐁{𝐀} B≤A

VlA (vssubw{𝐀}{𝐁} x B≤A) = vssubw vlA lB≤lA
  where
  vlA : (lift-alg 𝐀 ovu+) ∈ V{𝓤}{ovu+} 𝒦
  vlA = VlA x

  lB≤lA : (lift-alg 𝐁 ovu+) ≤ (lift-alg 𝐀 ovu+)
  lB≤lA = lift-alg-≤ 𝐁{𝐀} B≤A

VlA (vprodu{I}{𝒜} x) = visow (vprodw vlA) (sym-≅ B≅A)
  where
  𝐈 : ovu+ ̇
  𝐈 = Lift{ovu}{ovu+} I

  lA+ : Algebra ovu+ S
  lA+ = lift-alg (⨅ 𝒜) ovu+

  lA : 𝐈 → Algebra ovu+ S
  lA i = lift-alg (𝒜 (lower i)) ovu+

  vlA : (i : 𝐈) → (lA i) ∈ V{𝓤}{ovu+} 𝒦
  vlA i = vlift (x (lower i))

  iso-components : (i : I) → 𝒜 i ≅ lA (lift i)
  iso-components i = lift-alg-≅

  B≅A : lA+ ≅ ⨅ lA
  B≅A = lift-alg-⨅≅ gfe iso-components

VlA (vprodw{I}{𝒜} x) = visow (vprodw vlA) (sym-≅ B≅A)
  where
```

```
    I : ovu+ ̇
    I = Lift{ovu}{ovu+} I

    lA+ : Algebra ovu+ S
    lA+ = lift-alg (⊓ 𝒜) ovu+

    lA : I → Algebra ovu+ S
    lA i = lift-alg (𝒜 (lower i)) ovu+

    vlA : (i : I) → (lA i) ∈ V{𝓤}{ovu+} 𝒦
    vlA i = VlA (x (lower i))

    iso-components : (i : I) → 𝒜 i ≅ lA (lift i)
    iso-components i = lift-alg-≅

    B≅A : lA+ ≅ ⊓ lA
    B≅A = lift-alg-⊓≅ gfe iso-components

  VlA (visou{A}{B} x A≅B) = visow (vlift x) lA≅lB
    where
    lA≅lB : (lift-alg A ovu+) ≅ (lift-alg B ovu+)
    lA≅lB = lift-alg-iso 𝓤 ovu+ A B A≅B

  VlA (visow{A}{B} x A≅B) = visow vlA lA≅lB
    where
    lA lB : Algebra ovu+ S
    lA = lift-alg A ovu+
    lB = lift-alg B ovu+

    vlA : lA ∈ V{𝓤}{ovu+} 𝒦
    vlA = VlA x

    lA≅lB : lA ≅ lB
    lA≅lB = lift-alg-iso ovu ovu+ A B A≅B

  lift-alg-V-closure = VlA -- (alias)
```

9.2.2 Lamma 1: SP($\mathscr{K}$) ⊆ V($\mathscr{K}$)

Next we formalize the obvious fact that SP($\mathscr{K}$) ⊆ V($\mathscr{K}$). Unfortunately, the formal proof is neither trivial nor interesting.

```
  SP⊆V' : S{ovu}{ovu+} (P{𝓤}{ovu} 𝒦) ⊆ V{𝓤}{ovu+} 𝒦

  SP⊆V' (sbase{A} x) = γ
    where
    llA lA+ : Algebra ovu+ S
    lA+ = lift-alg A ovu+
    llA = lift-alg (lift-alg A ovu) ovu+

    vllA : llA ∈ V{𝓤}{ovu+} 𝒦
```

```
    vllA = lift-alg-V-closure (SP⊆V (sbase x))

    llA≅lA+ : llA ≅ lA+
    llA≅lA+ = sym-≅ (lift-alg-associative A)

    γ : lA+ ∈ (V{𝓤}{ovu+} 𝒦)
    γ = visow vllA llA≅lA+

SP⊆V' (slift{A} x) = lift-alg-V-closure (SP⊆V x)

SP⊆V' (ssub{A}{B} spA B≤A) = vssubw vlA B≤lA
  where
    lA : Algebra ovu+ S
    lA = lift-alg A ovu+

    vlA : lA ∈ V{𝓤}{ovu+} 𝒦
    vlA = lift-alg-V-closure (SP⊆V spA)

    B≤lA : B ≤ lA
    B≤lA = (lift-alg-lower-≤-lift {ovu}{ovu+}{ovu+} A {B}) B≤A

SP⊆V' (ssubw{A}{B} spA B≤A) = vssubw (SP⊆V' spA) B≤A

SP⊆V' (siso{A}{B} x A≅B) = visow (lift-alg-V-closure vA) lA≅B
  where
    lA : Algebra ovu+ S
    lA = lift-alg A ovu+

    plA : A ∈ S{ovu}{ovu}(P{𝓤}{ovu} 𝒦)
    plA = x

    vA : A ∈ V{𝓤}{ovu} 𝒦
    vA = SP⊆V x

    lA≅B : lA ≅ B
    lA≅B = Trans-≅ lA B (sym-≅ lift-alg-≅) A≅B
```

9.2.3 Lemma 2: $\mathbb{F} \leq \sqcap\, \mathbf{S}(\mathscr{K})$

Now we come to a step in the Agda formalization of Birkhoff's theorem that turns out to be surprisingly nontrivial—namely, we need to prove that the relatively free algebra $\mathbb{F}$ embeds in the product $\mathfrak{C}$ of all subalgebras of algebras in the given class $\mathscr{K}$. To prepare for this, we arm ourselves with a small arsenal of notation.

```
open the-relatively-free-algebra {𝓤 = 𝓤}{𝔛 = 𝓤}{X = X} {𝒦 = 𝒦}
open class-product {𝓤 = 𝓤}{𝒦 = 𝒦}

-- NOTATION.

-- 𝔽 is the relatively free algebra
𝔽 : Algebra ovu+ S
𝔽 = 𝔉 -- 𝒦
```

```
-- 𝕍 is HSP(𝒦)
𝕍 : Pred (Algebra ovu+ 𝑆) ovu++
𝕍 = V{𝓤}{ovu+} 𝒦

ℑs : ovu ̇
ℑs = ℑ (S{𝓤}{𝓤} 𝒦)

𝔄s : ℑs → Algebra 𝓤 𝑆
𝔄s = λ (i : ℑs) → ∣ i ∣

SK𝔄 : (i : ℑs) → (𝔄s i) ∈ S{𝓤}{𝓤} 𝒦
SK𝔄 = λ (i : ℑs) → ∥ i ∥

-- ℭ is the product of all subalgebras of 𝒦.
ℭ : Algebra ovu 𝑆
ℭ = ⨅ 𝔄s
-- elements of ℭ are mappings from ℑs to {𝔄s i : i ∈ ℑs}
𝔥₀ : X → ∣ ℭ ∣
𝔥₀ x = λ i → (fst (𝕏 (𝔄s i))) x -- fst (𝕏 ℭ)
                                 -- 𝔄1
ϕ𝔠 : hom (𝑻 X) ℭ                 -- 77
ϕ𝔠 = lift-hom ℭ 𝔥₀               -- /
                                 -- 𝑻 -----ϕ≡h --->> ℭ -->> 𝔄2
𝔤 : hom (𝑻 X) 𝔽                  -- \ 77 ⋮
𝔤 = lift-hom 𝔽 (X↪𝔉) -- \ /
                                 -- g ∃f
𝔣 : hom 𝔽 ℭ                      -- \ /
𝔣 = 𝔉-free-lift ℭ 𝔥₀ ,           -- \ /
    λ f a → ∥ ϕ𝔠 ∥ f (λ i → ⌜ a i ⌝) -- V 1/
                                  -- 𝔽= 𝑻/ψ

𝔤-⟦⟧ : ∀ p → ∣ 𝔤 ∣ p ≡ ⟦ p ⟧
𝔤-⟦⟧ p = π𝔉-X-defined 𝔤 (𝔉-lift-agrees-on-X 𝔉 X↪𝔉) p

--Projection out of the product ℭ onto the specified (i-th) factor.
𝔭 : (i : ℑs) → ∣ ℭ ∣ → ∣ 𝔄s i ∣
𝔭 i 𝒂 = 𝒂 i

𝔭hom : (i : ℑs) → hom ℭ (𝔄s i)
𝔭hom = ⨅-projection-hom {I = ℑs}{𝒜 = 𝔄s}

-- the composition: 𝔽 --| 𝔣 |--> ℭ --(𝔭 i)--> 𝔄s i
𝔭𝔣 : ∀ i → ∣ 𝔽 ∣ → ∣ 𝔄s i ∣
𝔭𝔣 i = (𝔭 i) ∘ ∣ 𝔣 ∣

𝔭𝔣hom : (i : ℑs) → hom 𝔽 (𝔄s i)
𝔭𝔣hom i = HomComp 𝔽 (𝔄s i) 𝔣 (𝔭hom i)

𝔭ϕ𝔠 : ∀ i → ∣ 𝑻 X ∣ → ∣ 𝔄s i ∣
𝔭ϕ𝔠 i = ∣ 𝔭hom i ∣ ∘ ∣ ϕ𝔠 ∣

𝔓 : ∀ i → hom (𝑻 X) (𝔄s i)
```

```
𝔓 i = HomComp (𝐓 X) (𝔄s i) ϕc (𝔭hom i)

𝔭ϕc≡𝔓 : ∀ i p → (𝔭ϕc i) p ≡ ∣ 𝔓 i ∣ p
𝔭ϕc≡𝔓 i p = 𝓇ℯ𝒻𝓁

-- The class of subalgebras of products of 𝒦.
SP𝒦 : Pred (Algebra (ovu) S) (OV (ovu))
SP𝒦 = S{ovu}{ovu}(P{𝓤}{ovu} 𝒦)
```

9.2.4 Lemma 3: 𝔽 ≤ ℭ

Armed with these tools, we proceed to the proof that the free algebra 𝔽 is a subalgebra of the product ℭ of all subalgebras of algebras in 𝒦. The hard part of the proof is showing that 𝔣 : hom 𝔽ℭ is a monomorphism. Let's dispense with that first.

```
Ψ : Rel ∣ 𝐓 X ∣ (OV 𝓤)
Ψ = ψRel 𝒦

mon𝔣 : Monic ∣ 𝔣 ∣
mon𝔣 (.(Ψ p) , p , refl _) (.(Ψ q) , q , refl _) 𝔣pq = γ
  where

    pΨq : Ψ p q
    pΨq 𝐀 sA = γ'
      where
        𝔭A : hom 𝔽 𝐀
        𝔭A = 𝔭𝔣hom (𝐀 , sA)

        𝔣pq : ∣ 𝔭A ∣ ⟦ p ⟧ ≡ ∣ 𝔭A ∣ ⟦ q ⟧
        𝔣pq = ∣ 𝔭A ∣ ⟦ p ⟧    ≡⟨ 𝓇ℯ𝒻𝓁 ⟩
              ∣ 𝔭hom (𝐀 , sA) ∣ ( ∣ 𝔣 ∣ ⟦ p ⟧ ) ≡⟨ ap (λ - → (∣ 𝔭hom (𝐀 , sA) ∣ -)) 𝔣pq ⟩
              ∣ 𝔭hom (𝐀 , sA) ∣ ( ∣ 𝔣 ∣ ⟦ q ⟧ ) ≡⟨ 𝓇ℯ𝒻𝓁 ⟩
              ∣ 𝔭A ∣ ⟦ q ⟧    ∎

        h₀ : X → ∣ 𝐀 ∣
        h₀ = (𝔭 (𝐀 , sA)) ∘ 𝔥₀

        h ϕ : hom (𝐓 X) 𝐀
        h = HomComp (𝐓 X) 𝐀 𝔤 𝔭A

        ϕ = lift-hom 𝐀 h₀

        --(homs from 𝐓 X to 𝐀 that agree on X are equal)
        lift-agreement : (x : X) → h₀ x ≡ ∣ 𝔭A ∣ ⟦ 𝑔 x ⟧
        lift-agreement x = 𝔉-lift-agrees-on-X 𝐀 h₀ x

        𝔣gx≡ϕ : (x : X) → (∣ 𝔭A ∣ ∘ ∣ 𝔤 ∣) (𝑔 x) ≡ ∣ ϕ ∣ (𝑔 x)
        𝔣gx≡ϕ x = (∣ 𝔭A ∣ ∘ ∣ 𝔤 ∣) (𝑔 x) ≡⟨ 𝓇ℯ𝒻𝓁 ⟩
                  ∣ 𝔭A ∣ ( ∣ 𝔤 ∣ (𝑔 x) ) ≡⟨ ap ∣ 𝔭A ∣ (𝔤-⟦⟧ (𝑔 x)) ⟩
                  ∣ 𝔭A ∣ (⟦ 𝑔 x ⟧) ≡⟨ (lift-agreement x)⁻¹ ⟩
                    h₀ x ≡⟨ 𝓇ℯ𝒻𝓁 ⟩
                  ∣ ϕ ∣ (𝑔 x) ∎
```

```
        h≡ϕ' : ∀ t → (∣ 𝔭A ∣ ∘ ∣ 𝔤 ∣) t ≡ ∣ ϕ ∣ t
        h≡ϕ' t = free-unique gfe 𝐀 h ϕ 𝔣gx≡ϕ t

        SPu : Pred (Algebra 𝓤 S) (OV 𝓤)
        SPu = S{𝓤}{𝓤} (P{𝓤}{𝓤} 𝒦)
        i : ℑs
        i = (𝐀 , sA)
        𝔄i : Algebra 𝓤 S
        𝔄i = 𝔄s i
        sp𝔄i : 𝔄i ∈ SPu
        sp𝔄i = S⊆SP{𝓤}{𝓤} sA

        γ' : ∣ ϕ ∣ p ≡ ∣ ϕ ∣ q
        γ' = ∣ ϕ ∣ p              ≡⟨ (h≡ϕ' p)⁻¹ ⟩
             (∣ 𝔭A ∣ ∘ ∣ 𝔤 ∣) p ≡⟨ 𝓇ℯ𝒻𝓁 ⟩
             ∣ 𝔭A ∣ ( ∣ 𝔤 ∣ p ) ≡⟨ ap ∣ 𝔭A ∣ (𝔤-⟦⟧ p) ⟩
             ∣ 𝔭A ∣ ⟦ p ⟧       ≡⟨ 𝔣pq ⟩
             ∣ 𝔭A ∣ ⟦ q ⟧       ≡⟨ (ap ∣ 𝔭A ∣ (𝔤-⟦⟧ q))⁻¹ ⟩
             ∣ 𝔭A ∣ ( ∣ 𝔤 ∣ q ) ≡⟨ h≡ϕ' q ⟩
             ∣ ϕ ∣ q            ∎

    γ : ( Ψ p , p , 𝓇ℯ𝒻𝓁) ≡ ( Ψ q , q , 𝓇ℯ𝒻𝓁)
    γ = class-extensionality' pe gfe ssR ssA ψIsEquivalence pΨq
```

With that out of the way, the proof that 𝔽 is (isomorphic to) a subalgebra of ℭ is all but complete.

```
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
```

9.2.5 Lemma 4: 𝔽 ∈ V(𝒦)

Now, with this result in hand, along with what we proved earlier—namely, PS(𝒦) ⊆ SP(𝒦) ⊆ HSP(𝒦) ≡ 𝕍—it is not hard to show that 𝔽 belongs to SP(𝒦), and hence to 𝕍.

```
open class-product-inclusions {𝓤 = 𝓤}{𝒦 = 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)
```

9.3 The HSP Theorem

This section presents the UALib.Birkhoff.Theorem module of the Agda UALib. We now have all the pieces in place so that it is all but trivial to string together these pieces to complete the proof of Birkhoff's celebrated HSP theorem asserting that every variety is defined by a set of identities (is an "equational class").

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

open import UALib.Algebras using (Signature; 𝓞; 𝓥; Algebra; _↠_)
open import UALib.Prelude.Preliminaries using (global-dfunext; Universe; _̇)


module UALib.Birkhoff.Theorem
  {𝑆 : Signature 𝓞 𝓥}{gfe : global-dfunext}
  {𝕏 : {𝓤 𝓧 : Universe}{X : 𝓧 ̇ }(𝑨 : Algebra 𝓤 𝑆) → X ↠ 𝑨}
  {𝓤 : Universe} {X : 𝓤 ̇}
  where


open import UALib.Birkhoff.Lemmata {𝑆 = 𝑆}{gfe}{𝕏}{𝓤}{X} public

open the-free-algebra {𝓤}{𝓤}{X}


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

  open the-relatively-free-algebra {𝓤}{𝓤}{X}{𝒦}
  open HSPLemmata {𝒦 = 𝒦}{hfe}{pe}{ssR}{ssA}

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

  birkhoff Cset {𝑨} MThVA = γ
    where
      T : Algebra (OV 𝓤) 𝑆
      T = 𝑻 X

      h₀ : X → ∣ 𝑨 ∣
      h₀ = fst (𝕏 𝑨)

      h₀E : Epic h₀
      h₀E = snd (𝕏 𝑨)

      ϕ : Σ h ꞉ (hom 𝔽 𝑨) , Epic ∣ h ∣
      ϕ = (𝔉-lift-hom 𝑨 h₀) , 𝔉-lift-of-epic-is-epic 𝑨 h₀ h₀E

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

$\gamma : \mathbb{A} \in \mathbb{V}$
$\gamma = \mathsf{vhimg}\ (\mathbb{F}{\in}\mathbb{V}\ \mathit{Cset})\ \mathsf{AiF}$

Some readers familiar with Birkhoff's theorem might worry that we haven't acheived our goal because they may be used to seeing it presented as an "if and only if" assertion. Those fears are quickly put to rest. Indeed, the converse of the result just proved is that every equational class is closed under HSP, but we already proved that, formally of course, in the closure module. Indeed, there it is proved that H, S, and P preserve identities.

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 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`.

6 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`.

7 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`.