The Agda Universal Algebra Library Part 2: Structure

Homomorphisms, terms, classes of algebras, subalgebras, and homomorphic images

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 dependent type theory using the Agda programming language and proof assistant. The UALib includes a substantial collection of definitions, theorems, and proofs from universal algebra, equational logic, and model theory, and as such provides many examples that exhibit the power of inductive and dependent types for representing and reasoning about mathematical structures and equational theories. In this paper, we describe the types and proofs of the UALib that concern homomorphisms, terms, and subalgebras.

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

Keywords and phrases Agda, constructive mathematics, dependent types, equational logic, extensionality, formalization of mathematics, model theory, type theory, universal algebra

Related Version hosted on arXiv
Part 1, Part 3: http://arxiv.org/a/demeo_w_1

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

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 (AgdaUALib), a software library containing formal statements and proofs of the core definitions and results of universal algebra. The UALib is written in Agda [8], a programming language and proof assistant based on Martin-Löf Type Theory (MLTT) that supports dependent and inductive types.

1.1 Motivation

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

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

1.2 Attributions and Contributions

The mathematical results described in this paper have well known *informal* proofs. Our main contribution is the formalization, mechanization, and verification of the statements and proofs of these results in dependent type theory using Agda.

Unless explicitly stated otherwise, the Agda source code described in this paper is due to the author, with the following caveat: the UALib depends on the Type Topology library of Martín Escardó [5]. For convenience, we refer to Escardó's library as TypeTopo throughout the paper. For the sake of completeness and clarity, and to keep the paper mostly self-contained, we repeat some definitions from TypeTopo, but in each instance we cite the original source.[1]

1.3 Organization of the paper

In this paper we limit ourselves to the presentation of the middle third of the UALib, which includes types for representing *homomorphisms*, *terms*, and *subalgebras*. This limitation will give us the space required to discuss some of the more interesting type theoretic and foundational issues that arise when developing a library of this kind and when attempting to represent advanced mathematical notions in type theory and formalize them in Agda.

This is the second installment of a three-part series of papers describing the AgdaUALib. The first part ([3]) covers the logical foundations of Martin-Löf type theory (including *Sigma* and *Pi types*, *equality*, *extensionality*, *truncation*) and develops dependent types for representing *relations*, *algebras*, *congruences*, and *quotients*. The third part covers *free algebras*, *equational classes* of algebras (i.e., *varieties*), and *Birkhoff's HSP theorem*.

[1] In the UALib, such instances occur only inside hidden modules that are never actually used, followed immediately by a statement that imports the code in question from its original source.

The present paper is divided into three main parts (§2, §3, §4). The first of these introduces types representing *homomorphisms* from one algebra to another, and presents a formal statement and proof of the first fundamental theorem about homomorphisms, known as the *First Isomorphism Theorem*, as well as a version of the so-called *Second Isomorphism Theorem*. This is followed by dependent type definitions for representing *isomorphisms* and *homomorphic images* of algebraic structures.

In Section 3 we define inductive types to represent *terms* and the *term algebra* in a given signature. We prove the *universal property* of the term algebra which is the fact that term algebra is *free* (or *initial*) in the class of all algebras in the given signature. We define types that denote the interpretation of a term in an algebra type, called a *term operation*, including the interpretation of terms in *arbitrary products* of algebras (§3.2.1). We conclude § 3 with a subsection on the compatibility of terms with basic operations and congruence relations.

Section 4 presents inductive and dependent types for representing subuniverses and subalgebras of a given algebra. Here we define an inductive type that represents the *subuniverse generated by* X, for a given predicate $X : \mathsf{Pred} \mid \mathbf{A} \mid \_$,[2] and we use this type to formalize a few basic subuniverse lemmas. We also define types that pertain to arbitrary classes of algebras. In particular, in Subsection 4.2 on subalgebras, we define a type that represents the assertion that a given algebra is a subalgebra of some member of a class of algebras.

1.4 Resources

We conclude this introduction with some pointers to helpful reference materials. For the required background in Universal Algebra, we recommend the textbook by Clifford Bergman [1]. For the type theory background, we recommend the HoTT Book [9] and Escardó's Introduction to Univalent Foundations of Mathematics with Agda [5].

The following are informed the development of the UALib and are highly recommended.

- *Introduction to Univalent Foundations of Mathematics with Agda*, Escardó [5].
- *Dependent Types at Work*, Bove and Dybjer [2].
- *Dependently Typed Programming in Agda*, Norell and Chapman [7].
- *Formalization of Universal Algebra in Agda*, Gunther, Gadea, Pagano [6].
- *Programming Languages Foundations in Agda*, Philip Wadler [13].

More information about AgdaUALib can be obtained from the following official sources.

- `ualib.org` (the web site) documents every line of code in the library.
- `gitlab.com/ualib/ualib.gitlab.io` (the source code) AgdaUALib is open source.[3]
- *The Agda UALib, Part 1: equality, extensionality, truncation, and dependent types for relations and algebras* [3].
- *The Agda UALib, Part 3: free algebras, equational classes, and Birkhoff's theorem* [4].

The first item links to the official UALib html documentation which includes complete proofs of every theorem we mention here, and much more, including the Agda modules covered in the first and third installments of this series of papers on the UALib.

Finally, readers will get much more out of reading the paper if they download the AgdaUALib from `https://gitlab.com/ualib/ualib.gitlab.io`, install the library, and try it out for themselves.

---

[2] As we learned in [3], such X represents a subset of the domain of the algebra $\mathbf{A}$.

[3] License: Creative Commons Attribution-ShareAlike 4.0 International License.

2 Homomorphism Types

In this section we define types that represent some of the most important concepts from general (universal) algebra. Note that the Agda modules we describe here, and in succeeding sections depend on and import the modules that were presented in Part 1 of this series of three papers describing the AgdaUALib (see [3]).

We begin in Subsection 2.1 with the basic definition of *homomorphism*. In §2.2 we formalize the statement and proof of the first fundamental theorem about homomorphisms, which is sometimes referred to as the *First Isomorphism Theorem*. This is followed by §2.3, in which we define the type of *isomorphisms* between algebraic structures. Finally, in §2.4, we define types that manifest the notion of *homomorphic image*.

2.1 Basic definitions

This section presents the Homomorphisms.Basic module of the AgdaUALib, slightly abridged.[4] Since this is the first module we introduce in the installment of our series of papers documenting the AgdaUALib, we will begin by showing the start of the module in full. In later modules, we will leave such details implicit. Here is how the file `Homomorphisms/Basic.lagda` of the UALib begins.

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

open import Algebras.Signatures using (Signature; 𝓞; 𝓥)
open import MGS-Subsingleton-Theorems using (global-dfunext)

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

open import Algebras.Congruences{S = S} public
open import MGS-MLTT using (_≡⟨_⟩_; _■) public
```

The OPTIONS *pragma* sets some parameters that specify the type theoretic foundations that we assume. These are discussed in [3, §2.1], but let's briefly review: *–without-K* disables Streicher's K axiom (see [10]); *–exact-split* makes Agda accept only definitions that use *definitional* equalities (see [12]); *–safe* ensures that nothing is postulated outright, so that every non-MLTT axiom has to be an explicit assumption (see [11] and [12]).[5]

2.1.1 Homomorphisms

If $\mathbf{A}$ and $\mathbf{B}$ are S-algebras, then a *homomorphism* is a function $h : |\,\mathbf{A}\,| \to |\,\mathbf{B}\,|$ from the domain of $\mathbf{A}$ to the domain of $\mathbf{B}$ that is *compatible* (or *commutes*) with all of the basic operations of the signature; that is, for all operation symbols $f : |\,S\,|$ and all tuples $a : \|\,S\,\| \, f \to |\,\mathbf{A}\,|$, the following equality holds:[6]

$$h\,((f\,\hat{}\,\mathbf{A})\,a) \equiv (f\,\hat{}\,\mathbf{B})\,(h \circ a).$$

Instead of "homomorphism," we sometimes use the nickname "hom" to refer to such a map.

---

[4] For unabridged docs and source code see `https://ualib.gitlab.io/Homomorphisms.Basic.html` and `https://gitlab.com/ualib/ualib.gitlab.io/-/blob/master/UALib/Homomorphisms/Basic.lagda`.

[5] As in [3], MLTT stands for Martin-Löf Type Theory.

[6] Recall, $h \circ a$ is the tuple whose i-th component is $h\,(a\,i)$.

To formalize this concept, we first define a type representing the assertion that a function $h : |\,\mathbf{A}\,| \to |\,\mathbf{B}\,|$ commutes with a single basic operation f. With Agda's extremely flexible syntax, the defining equation above can be expressed unadulterated.[7]

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

  compatible-op-map : ∣ S ∣ → (∣ 𝐀 ∣ → ∣ 𝐁 ∣) → 𝓤 ⊔ 𝓥 ⊔ 𝓦 ̇
  compatible-op-map f h = ∀ a → h ((f ̂ 𝐀) a) ≡ (f ̂ 𝐁) (h ∘ a)
```

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

We now define the type hom $\mathbf{A}$ $\mathbf{B}$ of homomorphisms from $\mathbf{A}$ to $\mathbf{B}$ by first defining the property is-homomorphism.

```
  is-homomorphism : (∣ 𝐀 ∣ → ∣ 𝐁 ∣) → 𝓞 ⊔ 𝓥 ⊔ 𝓤 ⊔ 𝓦 ̇
  is-homomorphism g = ∀ f → compatible-op-map f g

  hom : 𝓞 ⊔ 𝓥 ⊔ 𝓤 ⊔ 𝓦 ̇
  hom = Σ g ꞉ (∣ 𝐀 ∣ → ∣ 𝐁 ∣ ) , is-homomorphism g
```

2.1.2 Examples of homomorphisms

Here we give a few very special examples of homomorphisms. In each case, the function in question commutes with the basic operations of *all* algebras and so, no matter the algebras involved, is always a homomorphism (trivially).

The most obvious example of a homomorphism is the identity map, which is proved to be a homomorphism as follows.

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

  𝑖𝑑 : (A : Algebra 𝓤 S) → hom A A
  𝑖𝑑 _ = (λ x → x) , id-is-hom
```

Next, lift and lower, defined in the Prelude.Lifts module (see [3, §2.5]),[8] are (the maps of) homomorphisms. Again, the proof is trivial in each case.

```
  Lift-is-hom : {𝐀 : Algebra 𝓤 S}{𝓦 : Universe} → is-homomorphism 𝐀 (Lift-alg 𝐀 𝓦) lift
  Lift-is-hom _ _ = refl

  𝓁𝒾𝒻𝓉 : {𝐀 : Algebra 𝓤 S}{𝓦 : Universe} → hom 𝐀 (Lift-alg 𝐀 𝓦)
  𝓁𝒾𝒻𝓉 = (lift , Lift-is-hom)

  lower-is-hom : {𝐀 : Algebra 𝓤 S}{𝓦 : Universe} → is-homomorphism (Lift-alg 𝐀 𝓦) 𝐀 lower
  lower-is-hom _ _ = refl
```

---

[7] Here we put the definition inside an *anonymous module*, which starts with the module keyword followed by an underscore (instead of a module name). The purpose is simply to move the postulated typing judgments (the "parameters" of the module, e.g., $\mathcal{U}\ \mathcal{W}$: Universe) out of the way so they don't obfuscate the definitions inside the module. In descriptions of the UALib, such as the present paper, we usually don't show the module declarations unless we wish to emphasize the typing judgments that are postulated in the module declaration.

[8] or `https://ualib.gitlab.io/Prelude.Lifts.html`.

```
ℓ𝑜𝑤𝑒𝑟 : (𝐀 : Algebra 𝓤 S){𝓦 : Universe} → hom (Lift-alg 𝐀 𝓦) 𝐀
ℓ𝑜𝑤𝑒𝑟 𝐀 = (lower , lower-is-hom{𝐀})
```

2.1.3 Monomorphisms and epimorphisms

A *monomorphism* is an injective homomorphism and an *epimorphism* is a surjective homomorphism. These are represented in the UALib by the following types.

```
is-monomorphism : (𝐀 : Algebra 𝓤 S)(𝐁 : Algebra 𝓦 S) → (| 𝐀 | → | 𝐁 |) → 𝓞 ⊔ 𝓥 ⊔ 𝓤 ⊔ 𝓦 ̇
is-monomorphism 𝐀 𝐁 g = is-homomorphism 𝐀 𝐁 g × Monic g

mon : Algebra 𝓤 S → Algebra 𝓦 S → 𝓞 ⊔ 𝓥 ⊔ 𝓤 ⊔ 𝓦 ̇
mon 𝐀 𝐁 = Σ g ꞉ (| 𝐀 | → | 𝐁 |) , is-monomorphism 𝐀 𝐁 g

is-epimorphism : (𝐀 : Algebra 𝓤 S)(𝐁 : Algebra 𝓦 S) → (| 𝐀 | → | 𝐁 |) → 𝓞 ⊔ 𝓥 ⊔ 𝓤 ⊔ 𝓦 ̇
is-epimorphism 𝐀 𝐁 g = is-homomorphism 𝐀 𝐁 g × Epic g

epi : Algebra 𝓤 S → Algebra 𝓦 S → 𝓞 ⊔ 𝓥 ⊔ 𝓤 ⊔ 𝓦 ̇
epi 𝐀 𝐁 = Σ g ꞉ (| 𝐀 | → | 𝐁 |) , is-epimorphism 𝐀 𝐁 g
```

It will be convenient to have a function that takes an inhabitant of mon (or epi) and extracts the homomorphism part (or *hom reduct*), which is the pair consisting of the map and a proof that the map is a homomorphism.

```
mon-to-hom : (𝐀 : Algebra 𝓤 S){𝐁 : Algebra 𝓦 S} → mon 𝐀 𝐁 → hom 𝐀 𝐁
mon-to-hom 𝐀 ϕ = | ϕ | , fst ‖ ϕ ‖

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

2.1.4 Equalizers in Agda

Recall, the equalizer of two functions $g\ h : A \to B$ 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 the UALib as follows.

```
module _ {𝓤 𝓦 : Universe}{𝐀 : Algebra 𝓤 S} {fe : dfunext 𝓥 𝓦} where

  E : {𝐁 : Algebra 𝓦 S}(g h : | 𝐀 | → | 𝐁 |) → Pred | 𝐀 | 𝓦
  E g h x = g x ≡ h x

  Ehom : (𝐁 : Algebra 𝓦 S) → hom 𝐀 𝐁 → hom 𝐀 𝐁 → Pred | 𝐀 | 𝓦
  Ehom _ g h x = | g | x ≡ | h | x
```

We will define subuniverses in the Algebras.Subuniverses module, but we note here that the equalizer of homomorphisms from $\mathbf{A}$ to $\mathbf{B}$ will turn out to be subuniverse of $\mathbf{A}$. Indeed, this is easily proved as follows.

```
Ehom-closed : (𝐁 : Algebra 𝓦 S)(g h : hom 𝐀 𝐁)
  →           ∀ f a → Π x ꞉ ∥ S ∥ f , (a x ∈ Ehom 𝐁 g h)
              ———————————————————————————————————————
  →           | g | ((f ̂ 𝐀) a) ≡ | h | ((f ̂ 𝐀) a)
```

```
Ehom-closed B g h f a p = ∣ g ∣ ((f ̂ A) a)  ≡⟨ ∥ g ∥ f a ⟩
                          (f ̂ B)(∣ g ∣ ∘ a) ≡⟨ ap (f ̂ B)(fe p) ⟩
                          (f ̂ B)(∣ h ∣ ∘ a) ≡⟨ (∥ h ∥ f a)⁻¹ ⟩
                          ∣ h ∣ ((f ̂ A) a)  ■
```

(Agda infers the types of the implicit arguments, $f : \mid S \mid$ and $a : \parallel S \parallel f \to \mid \mathbf{A} \mid$.)

2.1.5 Kernels of Homomorphisms

The *kernel* of a homomorphism is a congruence relation and conversely for every congruence relation θ, there exists a homomorphism with kernel θ (namely, that canonical projection onto the quotient modulo θ).

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

  homker-compatible : (B : Algebra 𝓦 S)(h : hom A B) → compatible A (ker ∣ h ∣)
  homker-compatible B h f {u}{v} Kerhab = γ
    where
    γ : ∣ h ∣ ((f ̂ A) u)  ≡ ∣ h ∣ ((f ̂ A) v)
    γ = ∣ h ∣ ((f ̂ A) u) ≡⟨ ∥ h ∥ f u ⟩
        (f ̂ B)(∣ h ∣ ∘ u) ≡⟨ ap (f ̂ B)(gfe λ x → Kerhab x) ⟩
        (f ̂ B)(∣ h ∣ ∘ v) ≡⟨ (∥ h ∥ f v)⁻¹ ⟩
        ∣ h ∣ ((f ̂ A) v) ■

  homker-equivalence : (B : Algebra 𝓦 S)(h : hom A B) → IsEquivalence (ker ∣ h ∣)
  homker-equivalence B 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 kercon.

```
kercon : (B : Algebra 𝓦 S) → hom A B → Congruence A
kercon B h = mkcon (ker ∣ h ∣)(homker-equivalence B h)(homker-compatible B h)
```

With this we define the corresponding quotient, along with some syntactic sugar to denote it.

```
kerquo : {A : Algebra 𝓤 S}(B : Algebra 𝓦 S)(h : hom A B) → Algebra (𝓤 ⊔ 𝓦 ⁺) S
kerquo {A} B h = A ╱ (kercon B h)

_[_]/ker_ : (A : Algebra 𝓤 S)(B : Algebra 𝓦 S)(h : hom A B) → Algebra (𝓤 ⊔ 𝓦 ⁺) S
A [ B ]/ker h = kerquo B h

infix 60 _[_]/ker_
```

Thus, given $h : \mathbf{hom}\ \mathbf{A}\ \mathbf{B}$ we can construct the quotient of $\mathbf{A}$ modulo the kernel of h with the shorthand $\mathbf{A}\ [\ \mathbf{B}\ ]/\mathbf{ker}\ h$.

2.1.6 The canonical projection

Given an algebra $\mathbf{A}$ and a congruence θ, the *natural* or *canonical projection* is a map from $\mathbf{A}$ onto $\mathbf{A} \mathbin{╱} \theta$ that is constructed (and proved) to be epimorphic, as follows.

```
πepi : {A : Algebra 𝓤 S} (θ : Congruence{𝓦} A) → epi A (A ╱ θ)
πepi {A} θ = cπ , cπ-is-hom , cπ-is-epic where
```

```
cπ : ∣ A ∣ → ∣ A ╱ θ ∣
cπ a = ⟪ a ⟫{⟨ θ ⟩}

cπ-is-hom : is-homomorphism A (A ╱ θ) cπ
cπ-is-hom _ _ = refl

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

It may happen that we don't care about the surjectivity of πepi, in which case we might prefer to work with the *homomorphic reduct* of πepi. This is obtained by applying epi-to-hom, like so.

```
πhom : {A : Algebra 𝓤 S}(θ : Congruence{𝓦} A) → hom A (A ╱ θ)
πhom {A} θ = epi-to-hom (A ╱ θ) (πepi θ)
```

We combine the foregoing to define a function that takes S-algebras $\mathbf{A}$ and $\mathbf{B}$, and a homomorphism h : hom $\mathbf{A}$ $\mathbf{B}$ and returns the canonical epimorphism from $\mathbf{A}$ onto the quotient algebra $\mathbf{A}$ [$\mathbf{B}$]/ ker h (which, recall, is $\mathbf{A}$ modulo the kernel of h).

```
πker : {A : Algebra 𝓤 S}(B : Algebra 𝓦 S)(h : hom A B) → epi A (A [ B ]/ker h)
πker {A} B h = πepi (kercon B h)
```

The kernel of the canonical projection of $\mathbf{A}$ onto $\mathbf{A}$ / θ is equal to θ, but since equality of inhabitants of certain types (like Congruence or Rel) can be a tricky business, we settle for proving the containment $\mathbf{A}$ / $\theta \subseteq \theta$. Of the two containments, this is the easier one to prove; luckily it is also the one we need later.

```
ker-in-con : (A : Algebra 𝓤 S)(θ : Congruence{𝓦} A)(x y : ∣ A ∣)
  →          ⟨ kercon (A ╱ θ) (πhom θ) ⟩ x y → ⟨ θ ⟩ x y

ker-in-con A θ x y hyp = ╱-≡ θ hyp
```

2.1.7 Product homomorphisms

Suppose we have an algebra $\mathbf{A}$: Algebra $\mathcal{U}$ S, a type $I : \mathcal{I}\ ^\cdot$, and a family $\mathcal{B}$: $I \to$ Algebra $\mathcal{W}$ S of algebras. We sometimes refer to the inhabitants of I as *indices*, and call $\mathcal{B}$ an *indexed family of algebras*. If in addition we have a family h : $(i : I) \to$ hom $\mathbf{A}$ $(\mathcal{B}\ i)$ of homomorphisms, then we can construct a homomorphism from $\mathbf{A}$ to the product $\prod \mathcal{B}$ in the natural way.

```
module _ {𝓤 𝓘 𝓦 : Universe} {fe : dfunext 𝓘 𝓦} where

  ⨅-hom-co : {A : Algebra 𝓤 S}{I : 𝓘 ̇}(ℬ : I → Algebra 𝓦 S)
    → Π i ꞉ I , hom A (ℬ i) → hom A (⨅ ℬ)

  ⨅-hom-co {A} ℬ h = ϕ , ϕhom
    where
    ϕ : ∣ A ∣ → ∣ ⨅ ℬ ∣
    ϕ a = λ i → ∣ h i ∣ a

    ϕhom : is-homomorphism A (⨅ ℬ) ϕ
    ϕhom f a = fe λ i → ∥ h i ∥ f a
```

The family h of homomorphisms inhabits the dependent type Π i ꞉ I , hom $\mathbf{A}$ $(\mathcal{B}\ i)$. The syntax we use to represent this type is available to us because of the way -Π is defined in TypeTopo. We

like this syntax because it is very close to the notation one finds in the standard type theory literature. However, we could equally well have used one of the following alternatives, which may be closer to "standard Agda" syntax:

```
Π λ i → hom A (ℬ i)   or   (i : I) → hom A (ℬ i)   or   ∀ i → hom A (ℬ i).
```

The foregoing generalizes easily to the case in which the domain is also a product of a family of algebras. That is, if we are given $\mathcal{A} : I \to \mathsf{Algebra}\ \mathcal{U}\ S$ and $\mathcal{B} : I \to \mathsf{Algebra}\ \mathcal{W}\ S$ (two families of S-algebras), and $h : \Pi\ i : I\ ,\ \mathsf{hom}\ (\mathcal{A}\ i)\ (\mathcal{B}\ i)$ (a family of homomorphisms), then we can construct a homomorphism from $\bigsqcap \mathcal{A}$ to $\bigsqcap \mathcal{B}$ in the following natural way.

```
⨅-hom : {I : 𝓘 ̇ }(𝒜 : I → Algebra 𝒰 S)(ℬ : I → Algebra 𝒲 S)
  → Π i : I , hom (𝒜 i)(ℬ i) → hom (⨅ 𝒜)(⨅ ℬ)

⨅-hom 𝒜 ℬ h = ϕ , ϕhom
  where
  ϕ : ∣ ⨅ 𝒜 ∣ → ∣ ⨅ ℬ ∣
  ϕ = λ x i → ∣ h i ∣ (x i)

  ϕhom : is-homomorphism (⨅ 𝒜) (⨅ ℬ) ϕ
  ϕhom f 𝑎 = fe λ i → ∥ h i ∥ f (λ x → 𝑎 x i)
```

2.1.8 Projection homomorphisms

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

```
⨅-projection-hom : {I : 𝓘 ̇ }(ℬ : I → Algebra 𝒲 S) → Π i : I , hom (⨅ ℬ) (ℬ i)

⨅-projection-hom ℬ = λ i → ℎ i , ℎhom i
  where
  ℎ : ∀ i → ∣ ⨅ ℬ ∣ → ∣ ℬ i ∣
  ℎ i = λ x → x i

  ℎhom : ∀ i → is-homomorphism (⨅ ℬ) (ℬ i) (ℎ i)
  ℎhom _ _ _ = refl
```

Of course, we could prove a more general result involving projections onto multiple factors, but for many purposes the single-factor result suffices.

2.2 Homomorphism Theorems

This section presents the Homomorphisms.Noether module of the AgdaUALib, slightly abridged.[9]

2.2.1 The First Homomorphism Theorem

Here we formalize a version of the *first homomorphism theorem*, sometimes called *Noether's first homomorphism theorem*, after Emmy Noether who was among the first proponents of the abstract approach to the subject that we now call "modern algebra." Informally, the theorem

[9] For unabridged docs and source code see `https://ualib.gitlab.io/Homomorphisms.Noether.html` and `https://gitlab.com/ualib/ualib.gitlab.io/-/blob/master/UALib/Homomorphisms/Noether.lagda`.

states that every homomorphism from $\mathbf{A}$ to $\mathbf{B}$ (S-algebras) factors through the quotient algebra $\mathbf{A}\,╱\,\mathsf{ker}\ h$ ($\mathbf{A}$ modulo the kernel of the given homomorphism). In other terms, given $h : \mathsf{hom}\ \mathbf{A}\ \mathbf{B}$ there exists $\varphi : \mathsf{hom}\ (\mathbf{A}\,╱\,\mathsf{ker}h)\ \mathbf{B}$ which, when composed with the canonical projection $\pi\mathsf{ker} : \mathbf{A} \twoheadrightarrow \mathbf{A}\,╱\,\mathsf{ker}h$, is equal to h; that is, $h = \varphi \circ \pi\mathsf{ker}$. Moreover, φ is a *monomorphism* (injective homomorphism) and is unique.

Our formal proof of this theorem will require function extensionality as well as a couple of truncation assumptions. The function extensionality postulate (*fe*) will be clear enough (but see [3] for details). As for truncation, we require the following:[10]

- *uniqueness of kernel-membership/kernel-block-identity proofs*—proving that φ is monic requires (postulate *ssR*) that the kernel of h inhabits the type Pred_2 of *binary propositions* and (postulate *ssA*) that we are able to decide when two blocks of the kernel are equal;
- *uniqueness of codomain-identity proofs*—proving that φ is an embedding requires (postulate *Bset*) that the codomain $|\ \mathbf{B}\ |$ is a *set*, that is, has unique identity proofs.

Note that the classical, informal statement of the theorem does not demand that φ be an embedding (in our sense of having subsingleton fibers), and if we left this out of the consequent of the formal theorem statement below, then we could omit from the antecedent the assumption that $|\ \mathbf{B}\ |$ is a set.

Without further ado, we present our formalization of the first homomorphism theorem.[11]

```
module _ {𝓤 𝓦 : Universe}
         -- extensionality assumptions --
           (fe : dfunext 𝓥 𝓦)
           (pe : prop-ext 𝓤 𝓦)

         (𝐀 : Algebra 𝓤 S)(𝐁 : Algebra 𝓦 S)(h : hom 𝐀 𝐁)

         -- truncation assumptions --
           (Bset : is-set ∣ 𝐁 ∣)
           (ssR : ∀ a x → is-subsingleton (⟨ kercon 𝐁 h ⟩ a x))
           (ssA : ∀ C → is-subsingleton (𝒞 ⟨ kercon 𝐁 h ⟩ C))
  where
  FirstHomomorphismTheorem :

    Σ ϕ ꞉ hom (𝐀 [ 𝐁 ]/ker h) 𝐁 , (∣ h ∣ ≡ ∣ ϕ ∣ ∘ ∣ πker 𝐁 h ∣) × Monic ∣ ϕ ∣ × is-embedding ∣ ϕ ∣

  FirstHomomorphismTheorem = (ϕ , ϕhom) , ϕcom , ϕmon , ϕemb
    where
    θ : Congruence 𝐀
    θ = kercon 𝐁 h

    ϕ : ∣ 𝐀 [ 𝐁 ]/ker h ∣ → ∣ 𝐁 ∣
    ϕ a = ∣ h ∣ ⌜ a ⌝

    𝐑 : Pred₂ ∣ 𝐀 ∣ 𝓦
    𝐑 = ⟨ kercon 𝐁 h ⟩ , ssR
```

[10] See Relations.Truncation or [3] for a discussion of *truncation*, *sets*, and *uniqueness of proofs*.

[11] In this module we are already assuming *global* function extensionality (*gfe*), and we could just appeal to *gfe* (e.g., in the proof of FirstHomomorphismTheorem) instead of adding local function extensionality (*fe*) to the antecedent. However, adding the extra extensionality postulate here makes clear where and how the principle is applied.

```
ϕhom : is-homomorphism (𝐀 [ 𝐁 ]/ker h) 𝐁 ϕ
ϕhom f 𝒂 = ∣ h ∣ ( (f ̂ 𝐀) (λ x → ⌜ 𝒂 x ⌝) ) ≡⟨ ∥ h ∥ f (λ x → ⌜ 𝒂 x ⌝) ⟩
           (f ̂ 𝐁) (∣ h ∣ ∘ (λ x → ⌜ 𝒂 x ⌝)) ≡⟨ ap (f ̂ 𝐁) (fe λ x → refl) ⟩
           (f ̂ 𝐁) (λ x → ϕ (𝒂 x)) ■

ϕmon : Monic ϕ
ϕmon (.(⟨ θ ⟩ u) , u , refl) (.(⟨ θ ⟩ v) , v , refl) ϕuv =
  class-extensionality' {𝓡 = 𝐑} pe ssA (IsEquiv θ) ϕuv

ϕcom : ∣ h ∣ ≡ ϕ ∘ ∣ πker 𝐁 h ∣
ϕcom = refl

ϕemb : is-embedding ϕ
ϕemb = monic-is-embedding∣sets ϕ Bset ϕmon
```

Next we show that the homomorphism φ, whose existence we just proved, is unique.

```
NoetherHomUnique : (f g : hom (𝐀 [ 𝐁 ]/ker h) 𝐁)
  →                ∣ h ∣ ≡ ∣ f ∣ ∘ ∣ πker 𝐁 h ∣
  →                ∣ h ∣ ≡ ∣ g ∣ ∘ ∣ πker 𝐁 h ∣
                   ———————————————————————————
  →                ∀ a → ∣ f ∣ a ≡ ∣ g ∣ a

NoetherHomUnique f g hfk hgk (.(⟨ kercon 𝐁 h ⟩ a) , a , refl) =

  let θ = (⟨ kercon 𝐁 h ⟩ a , a , refl) in

  ∣ f ∣ θ ≡⟨ cong-app(hfk ⁻¹)a ⟩ ∣ h ∣ a ≡⟨ cong-app(hgk)a ⟩ ∣ g ∣ θ ■
```

If we postulate function extensionality, then we have[12]

```
fe-NoetherHomUnique : funext (𝓤 ⊔ 𝓦 ⁺) 𝓦 → (f g : hom (𝐀 [ 𝐁 ]/ker h) 𝐁)
  →                   ∣ h ∣ ≡ ∣ f ∣ ∘ ∣ πker 𝐁 h ∣ → ∣ h ∣ ≡ ∣ g ∣ ∘ ∣ πker 𝐁 h ∣ → ∣ f ∣ ≡ ∣ g ∣

fe-NoetherHomUnique fe f g hfk hgk = fe (NoetherHomUnique f g hfk hgk)
```

Assuming the hypotheses of the first homomorphism theorem, if we add the assumption that h is epic, then we get the so-called *first isomorphism theorem*.

```
FirstIsomorphismTheorem : dfunext 𝓦 𝓦 → Epic ∣ h ∣

  → Σ f ꞉ (epi (𝐀 [ 𝐁 ]/ker h) 𝐁) , (∣ h ∣ ≡ ∣ f ∣ ∘ ∣ πker 𝐁 h ∣) × is-embedding ∣ f ∣

FirstIsomorphismTheorem fev hE = (fmap , fhom , fepic) , refl , femb
  where
  θ : Congruence 𝐀
  θ = kercon 𝐁 h

  fmap : ∣ 𝐀 [ 𝐁 ]/ker h ∣ → ∣ 𝐁 ∣
```

[12] We already assumed *global* function extensionality in this module, so we could just appeal to that in this case. However, we make a local function extensionality assumption explicit here merely to highlight where and how the principle is applied.

```
fmap ⟪ a ⟫ = | h | ⌜ ⟪ a ⟫ ⌝

fhom : is-homomorphism (𝐀 [ 𝐁 ]/ker h) 𝐁 fmap
fhom f 𝒂 = | h |((f ̂ 𝐀) λ x → ⌜ 𝒂 x ⌝) ≡⟨ ∥ h ∥ f (λ x → ⌜ 𝒂 x ⌝) ⟩
           (f ̂ 𝐁)(| h | ∘ λ x → ⌜ 𝒂 x ⌝) ≡⟨ ap(f ̂ 𝐁)(fe λ _ → refl)⟩
           (f ̂ 𝐁) (fmap ∘ 𝒂) ■

fepic : Epic fmap
fepic b = γ where
  a : | 𝐀 |
  a = EpicInv | h | hE b

  bfa : b ≡ fmap ⟪ a ⟫
  bfa = (cong-app (EpicInvIsRightInv {fe = fev} | h | hE) b)⁻¹

  γ : Image fmap ∋ b
  γ = Image_∋_.eq b ⟪ a ⟫ bfa

fmon : Monic fmap
fmon (.(⟨ θ ⟩ u) , u , refl) (.(⟨ θ ⟩ v) , v , refl) fuv =
  class-extensionality' {𝐑 = ⟨ kercon 𝐁 h ⟩ , ssR} pe ssA (IsEquiv θ) fuv

femb : is-embedding fmap
femb = monic-is-embedding|sets fmap Bset fmon
```

The argument used above to prove NoetherHomUnique can also be used to prove uniqueness of the epimorphism f found in the isomorphism theorem.

```
NoetherIsoUnique : (f g : epi (𝐀 [ 𝐁 ]/ker h) 𝐁) → | h | ≡ | f | ∘ | πker 𝐁 h |
  →                          | h | ≡ | g | ∘ | πker 𝐁 h | → ∀ a → | f | a ≡ | g | a

NoetherIsoUnique f g hfk hgk (.(⟨ kercon 𝐁 h ⟩ a) , a , refl) =

  let θ = (⟨ kercon 𝐁 h ⟩ a , a , refl) in

    | f | θ ≡⟨ cong-app (hfk ⁻¹) a ⟩ | h | a ≡⟨ cong-app (hgk) a ⟩ | g | θ ■
```

2.2.2 Composition of homomorphisms

The composition of homomorphisms is again a homomorphism. There are a number of alternative ways to formalize this fact in Agda. The two representations included in the UALib are the following.

```
module _ {𝓧 𝓨 𝓩 : Universe} where

  ∘-hom : (𝐀 : Algebra 𝓧 S){𝐁 : Algebra 𝓨 S}(𝐂 : Algebra 𝓩 S)
    →          hom 𝐀 𝐁 → hom 𝐁 𝐂 → hom 𝐀 𝐂

  ∘-hom 𝐀 {𝐁} 𝐂 (g , ghom) (h , hhom) = h ∘ g , γ where

    γ : ∀ f a → (h ∘ g)((f ̂ 𝐀) a) ≡ (f ̂ 𝐂)(h ∘ g ∘ a)

    γ f a = (h ∘ g) ((f ̂ 𝐀) a) ≡⟨ ap h ( ghom f a ) ⟩
            h ((f ̂ 𝐁) (g ∘ a)) ≡⟨ hhom f ( g ∘ a ) ⟩
```

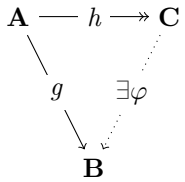
```
                (f ̂ C) (h ∘ g ∘ a) ■

∘-is-hom : (A : Algebra 𝓧 S){B : Algebra 𝓨 S}(C : Algebra 𝓩 S)
           {f : | A | → | B |} {g : | B | → | C |}
  →        is-homomorphism A B f → is-homomorphism B C g
  →        is-homomorphism A C (g ∘ f)

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

2.2.3 Homomorphism decomposition

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



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)
  →         kernel | h | ⊆ kernel | g |     → Epic | h |
            ---------------------------------------------
  →         Σ ϕ ꞉ (hom C B) , | g | ≡ | ϕ | ∘ | h |

homFactor fe{A}{B}{C}(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 → kernel h (x , hInv (h x))
  ξ x = (cong-app (EpicInvIsRightInv {fe = 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 = fe} h hEpi) (c x))⁻¹

  ι : (f : | S |)(c : ∥ S ∥ f → | C |) → c ≡ h ∘ (hInv ∘ c)
  ι f c = ap (λ - → - ∘ c)(EpicInvIsRightInv {fe = fe} h hEpi)⁻¹

  useker : ∀ f c → g(hInv (h((f ̂ A)(hInv ∘ c)))) ≡ g((f ̂ A)(hInv ∘ c))
  useker f c = Kh⊆Kg (cong-app (EpicInvIsRightInv{fe = fe} h hEpi)
```

```
                                  (h ((f ̂ A)(hInv ∘ c))) )

  ϕIsHomCB : (f : | S |)(c : ∥ S ∥ f → | C |) → ϕ((f ̂ C) c) ≡ (f ̂ B)(ϕ ∘ c)

  ϕIsHomCB f c = g (hInv ((f ̂ C) c))              ≡⟨ i   ⟩
                 g (hInv ((f ̂ C)(h ∘ (hInv ∘ c)))) ≡⟨ ii ⟩
                 g (hInv (h ((f ̂ A)(hInv ∘ c)))) ≡⟨ iii ⟩
                 g ((f ̂ A)(hInv ∘ c))             ≡⟨ iv  ⟩
                 (f ̂ B)(λ x → g (hInv (c x)))     ■
    where
    i   = ap (g ∘ hInv) (ap (f ̂ C) (ι f c))
    ii  = ap (g ∘ hInv) (hhom f (hInv ∘ c))⁻¹
    iii = useker f c
    iv  = ghom f (hInv ∘ c)
```

Here's a more general version.

```
module _ {𝓧 𝓨 𝓩 : Universe} where

  HomFactor : (A : Algebra 𝓧 S){B : Algebra 𝓨 S}{C : Algebra 𝓩 S}
              (β : hom A B) (γ : hom A C)
    →         Epic | γ | → (kernel | γ |) ⊆ (kernel | β |)
              ————————————————————————————
    →         Σ ϕ ꞉ (hom C B) , | β | ≡ | ϕ | ∘ | γ |

  HomFactor A {B}{C} β γ γE Kγβ = (ϕ , ϕIsHomCB) , βϕγ
    where
    γInv : | C | → | A |
    γInv = λ y → (EpicInv | γ | γE) y

    ϕ : | C | → | B |
    ϕ = λ y → | β | ( γInv y )

    ξ : (x : | A |) → kernel | γ | (x , γInv (| γ | x))
    ξ x = ( cong-app (EpicInvIsRightInv{fe = gfe} | γ | γE) ( | γ | x ) )⁻¹

    βϕγ : | β | ≡ ϕ ∘ | γ |
    βϕγ = gfe λ x → Kγβ (ξ x)

    ι : (f : | S |)(c : ∥ S ∥ f → | C |) → c ≡ | γ | ∘ (γInv ∘ c)
    ι f c = ap (λ - → - ∘ c) (EpicInvIsRightInv{fe = gfe} | γ | γE)⁻¹

    useker : ∀ f c → | β | (γInv (| γ | ((f ̂ A) (γInv ∘ c)))) ≡ | β |((f ̂ A) (γInv ∘ c))
    useker f c = Kγβ (cong-app (EpicInvIsRightInv {fe = gfe} | γ | γE)
                               (| γ | ((f ̂ A)(γInv ∘ c))))

    ϕIsHomCB : ∀ f c → ϕ ((f ̂ C) c) ≡ ((f ̂ B)(ϕ ∘ c))

    ϕIsHomCB f c = | β | (γInv ((f ̂ C) c)) ≡⟨ i ⟩
                   | β | (γInv ((f ̂ C)(| γ | ∘ (γInv ∘ c)))) ≡⟨ ii ⟩
                   | β | (γInv (| γ | ((f ̂ A)(γInv ∘ c)))) ≡⟨ iii ⟩
                   | β | ((f ̂ A)(γInv ∘ c))                 ≡⟨ iv  ⟩
```

```
                ((f ̂ B)(λ x → ∣ β ∣ (γInv (c x)))) ■
  where
  i  = ap (∣ β ∣ ∘ γInv) (ap (f ̂ C) (ι f c))
  ii = ap (∣ β ∣ ∘ γInv) (∥ γ ∥ f (γInv ∘ c))⁻¹
  iii = useker f c
  iv = ∥ β ∥ f (γInv ∘ c)
```

If, in addition, both β and γ are epic, then so is φ.

```
HomFactorEpi : (A : Algebra 𝓧 S){B : Algebra 𝓨 S}{C : Algebra 𝓩 S}
               (β : hom A B) (βe : Epic ∣ β ∣)
               (ξ : hom A C) (ξe : Epic ∣ ξ ∣)
  →            (kernel ∣ ξ ∣) ⊆ (kernel ∣ β ∣)
               ———————————————————————————
  →            Σ ϕ ꞉ (epi C B) , ∣ β ∣ ≡ ∣ ϕ ∣ ∘ ∣ ξ ∣

HomFactorEpi A {B}{C} β βe ξ ξe kerincl = (fst ∣ ϕF ∣ , (snd ∣ ϕF ∣ , ϕE)) , ∥ ϕF ∥
  where
  ϕF : Σ ϕ ꞉ (hom C B) , ∣ β ∣ ≡ ∣ ϕ ∣ ∘ ∣ ξ ∣
  ϕF = HomFactor A {B}{C} β ξ ξe kerincl

  ξinv : ∣ C ∣ → ∣ A ∣
  ξinv = λ c → (EpicInv ∣ ξ ∣ ξe) c

  βinv : ∣ B ∣ → ∣ A ∣
  βinv = λ b → (EpicInv ∣ β ∣ βe) b

  ϕ : ∣ C ∣ → ∣ B ∣
  ϕ = λ c → ∣ β ∣ ( ξinv c )

  ϕE : Epic ϕ
  ϕE = epic-factor {fe = gfe} ∣ β ∣ ∣ ξ ∣ ϕ ∥ ϕF ∥ βe
```

2.3 Isomorphisms

This section presents the Homomorphisms.Isomorphisms module of the AgdaUALib, slightly abridged.[13] Here we formalize the notion of *isomorphism* between algebraic structures.

2.3.1 Definition of isomorphism

Recall, $f \sim g$ means f and g are *extensionally* (or *point-wise*) *equal*; i.e., $\forall\ x$, $f\ x \equiv g\ x$. We use this notion of equality of functions in the following definition of *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 ∣)
```

That is, two structures are *isomorphic* provided there are homomorphisms going back and forth between them which compose to the identity.

---

[13] For unabridged docs and source code see `https://ualib.gitlab.io/Homomorphisms.Isomorphisms.html` and `https://gitlab.com/ualib/ualib.gitlab.io/-/blob/master/UALib/Homomorphisms/Isomorphisms.lagda`.

2.3.2 Isomorphism is an equivalence relation

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

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

≅-trans : {A : Algebra 𝓧 S}{B : Algebra 𝓨 S}{C : Algebra 𝓩 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 = ∘-hom A C f1 f2

  g1 : hom C B
  g1 = fst ∥ bc ∥
  g2 : hom B A
  g2 = fst ∥ ab ∥
  g : hom C A
  g = ∘-hom C A g1 g2

  α : ∣ f ∣ ∘ ∣ g ∣ ∼ ∣ id C ∣
  α x = (ap ∣ f2 ∣(∣ snd ∥ ab ∥ ∣ (∣ g1 ∣ x)))·(∣ snd ∥ bc ∥ ∣) x

  β : ∣ g ∣ ∘ ∣ f ∣ ∼ ∣ id A ∣
  β x = (ap ∣ g2 ∣(∥ snd ∥ bc ∥ ∥ (∣ f1 ∣ x)))·(∥ snd ∥ ab ∥ ∥) x
```

To make trans-≅ easier to apply in certain situations, we define a couple of alternatives where the only difference is which arguments are implicit.

```
TRANS-≅ : {A : Algebra 𝓧 S}{B : Algebra 𝓨 S}{C : Algebra 𝓩 S}
  →       A ≅ B → B ≅ C → A ≅ C
TRANS-≅ {A}{B}{C} = trans-≅ A B C

Trans-≅ : (A : Algebra 𝓧 S){B : Algebra 𝓨 S}(C : Algebra 𝓩 S)
  →       A ≅ B → B ≅ C → A ≅ C
Trans-≅ A {B} C = trans-≅ A B C
```

2.3.3 Lift is an algebraic invariant

Fortunately, the lift operation preserves isomorphism (i.e., it's an *algebraic invariant*). As algebra is our main focus, this invariance of the lift operation is what makes it a workable solution to the technical problems that arise from the noncumulativity of the universe hierarchy discussed in Prelude.Lifts [3, §2.5].

```
open Lift
```

```
lift-alg-≅ : {A : Algebra 𝓤 S} → A ≅ (lift-alg A 𝓦)
lift-alg-≅ {A} = ℓift , ℓower A , extfun lift∼lower , extfun (lower∼lift{𝓦})


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

lift-alg-hom 𝓧 𝓨 {A} B (f , fhom) = lift ∘ f ∘ lower , γ
  where
  lABh : is-homomorphism (lift-alg A 𝓧) B (f ∘ lower)
  lABh = ∘-is-hom (lift-alg A 𝓧) B {lower}{f} (λ _ _ → reflℓ) fhom

  γ : is-homomorphism(lift-alg A 𝓧)(lift-alg B 𝓨) (lift ∘ (f ∘ lower))
  γ = ∘-is-hom (lift-alg A 𝓧) (lift-alg B 𝓨){f ∘ lower}{lift} lABh λ _ _ → reflℓ

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

lift-alg-iso A≅B = ≅-trans (≅-trans (≅-sym lift-alg-≅) A≅B) lift-alg-≅
```

2.3.4 Lift associativity

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

```
lift-alg-assoc : {A : Algebra 𝓤 S} → lift-alg A (𝓦 ⊔ 𝓕) ≅ (lift-alg (lift-alg A 𝓦) 𝓕)
lift-alg-assoc {A} = ≅-trans (≅-trans γ lift-alg-≅) lift-alg-≅
  where
  γ : lift-alg A (𝓦 ⊔ 𝓕) ≅ A
  γ = ≅-sym lift-alg-≅

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

2.3.5 Products preserve isomorphisms

Products of isomorphic families of algebras are themselves isomorphic. The proof looks a bit technical, but it is as straightforward as it ought to be.

```
⨅≅ : {𝒜 : I → Algebra 𝓤 S}{ℬ : I → Algebra 𝓦 S} → Π i ꞉ I , 𝒜 i ≅ ℬ i → ⨅ 𝒜 ≅ ⨅ ℬ

⨅≅ {𝒜}{ℬ} AB = γ
  where
  ϕ : ∣ ⨅ 𝒜 ∣ → ∣ ⨅ ℬ ∣
  ϕ a i = ∣ fst (AB i) ∣ (a i)

  ϕhom : is-homomorphism (⨅ 𝒜) (⨅ ℬ) ϕ
  ϕhom f a = gfe (λ i → ∥ fst (AB 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 → snd ∣ snd (AB i) ∣ f (λ x → 𝒃 x i))
```

```
ϕ~ψ : ϕ ∘ ψ ∼ | 𝒾𝒹 (⨅ ℬ) |
ϕ~ψ b = gfe λ i → fst ∥ snd (AB i) ∥ (b i)

ψ~ϕ : ψ ∘ ϕ ∼ | 𝒾𝒹 (⨅ 𝒜) |
ψ~ϕ a = gfe λ i → snd ∥ snd (AB 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 (though, just for fun, we use the universal quantifier syntax here to express the dependent function type in the statement of the lemma, instead of the Pi notation we used in the statement of the previous lemma; that is, $\forall\ i \to \mathcal{A}\ i \cong \mathcal{B}\ (\mathsf{lift}\ i)$ instead of $\Pi\ i : I\ ,\ \mathcal{A}\ i \cong \mathcal{B}\ (\mathsf{lift}\ i)$).

```
lift-alg-⨅≅ : {I : 𝓘 ̇}{𝒜 : I → Algebra 𝓤 S}{ℬ : (Lift{𝓧} I) → Algebra 𝓦 S}
  →           (∀ i → 𝒜 i ≅ ℬ (lift i)) → lift-alg (⨅ 𝒜) 𝓧 ≅ ⨅ ℬ

lift-alg-⨅≅ {I}{𝒜}{ℬ} AB = γ
  where
  ϕ : | ⨅ 𝒜 | → | ⨅ ℬ |
  ϕ a i = | fst (AB (lower i)) | (a (lower i))

  ϕhom : is-homomorphism (⨅ 𝒜) (⨅ ℬ) ϕ
  ϕhom f a = gfe (λ i → (∥ fst (AB (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 → (snd | snd (AB i) |) f (λ x → b x (lift i)))

  ϕ~ψ : ϕ ∘ ψ ∼ | 𝒾𝒹 (⨅ ℬ) |
  ϕ~ψ b = gfe λ i → fst ∥ snd (AB (lower i)) ∥ (b i)

  ψ~ϕ : ψ ∘ ϕ ∼ | 𝒾𝒹 (⨅ 𝒜) |
  ψ~ϕ a = gfe λ i → snd ∥ snd (AB 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 (≅-sym lift-alg-≅) A≅B
```

2.3.6 Embedding tools

Finally, we prove some useful facts about embeddings that occasionally come in handy.

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

```
embedding-lift-nat hfu hfw h hem = NatΠ-is-embedding hfu hfw h hem

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

embedding-lift-nat' hfu hfw h hem = NatΠ-is-embedding hfu hfw h hem

embedding-lift : hfunext 𝓘 𝓤 → hfunext 𝓘 𝓦
  →              {I : 𝓘 ̇} {𝒜 : I → Algebra 𝓤 S}{ℬ : I → Algebra 𝓦 S}
  →              (h : ∀ i → ∣ 𝒜 i ∣ → ∣ ℬ i ∣) → (∀ i → is-embedding (h i))
                 ———————————————————————————-
  →              is-embedding(λ (x : ∣ ⨅ 𝒜 ∣) (i : I) → (h i)(x i))

embedding-lift hfu hfw {I}{𝒜}{ℬ} h hem = embedding-lift-nat' hfu hfw {I}{𝒜}{ℬ} h hem

iso→embedding : {𝓤 𝓦 : Universe}{𝐀 : Algebra 𝓤 S}{𝐁 : Algebra 𝓦 S}
  →             (ϕ : 𝐀 ≅ 𝐁) → is-embedding (fst ∣ ϕ ∣)

iso→embedding ϕ =equivs-are-embeddings (fst ∣ ϕ ∣) (invertibles-are-equivs (fst ∣ ϕ ∣) finv)
  where
  finv : invertible (fst ∣ ϕ ∣)
  finv = ∣ fst ∥ ϕ ∥ ∣ , (snd ∥ snd ϕ ∥ , fst ∥ snd ϕ ∥)
```

2.4 Homomorphic Images

This section presents the Homomorphisms.HomomorphicImages module of the AgdaUALib, slightly abridged.[14]

We begin with what seems, for our purposes, the most useful way to represent the class of *homomorphic images* of an algebra in dependent type theory.

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

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

These types should be self-explanatory, but just to be sure, let's describe the Sigma type appearing in the second definition. Given an S-algebra $\mathbf{A}$, the type HomImagesOf $\mathbf{A}$ denotes the class of algebras $\mathbf{B}$: Algebra $\mathcal{W}$ S with a map φ : $|\,\mathbf{A}\,| \to |\,\mathbf{B}\,|$ such that φ is a epimorphism.

The standard (informal) notion of the class of homomorphic images of an algebra assumes closure under isomorphism. Thus, we consider $\mathbf{B}$ to be a homomorphic image of $\mathbf{A}$ if (and only

[14] For unabridged docs and source code see `https://ualib.gitlab.io/Homomorphisms.HomomorphicImages.html` and `https://gitlab.com/ualib/ualib.gitlab.io/-/blob/master/UALib/Homomorphisms/HomomorphicImages.lagda`.

if) there exists an algebra $\mathbf{C}$ which is a homomorphic image of $\mathbf{A}$ and isomorphic to $\mathbf{B}$. In the UALib we express this notion with the following type.

```
_is-hom-image-of_ : (B : Algebra 𝓦 S)(A : Algebra 𝓤 S) → ov 𝓦 ⊔ 𝓤 ̇
B is-hom-image-of A = Σ Cϕ ꞉ (HomImagesOf A) , ∣ Cϕ ∣ ≅ B
```

2.4.1 Images of a class of algebras

Given a class $\mathscr{K}$ of S-algebras, we need a type that expresses the assertion that a given algebra is a *homomorphic image* of some algebra in the class, as well as a type that represents all such homomorphic images.

```
_is-hom-image-of-class_ : Algebra 𝓤 S → Pred (Algebra 𝓤 S)(𝓤 ⁺) → ov 𝓤 ̇
B is-hom-image-of-class 𝒦 = Σ A ꞉ (Algebra 𝓤 S) , (A ∈ 𝒦) × (B is-hom-image-of A)

HomImagesOfClass : Pred (Algebra 𝓤 S) (𝓤 ⁺) → ov 𝓤 ̇
HomImagesOfClass 𝒦 = Σ B ꞉ (Algebra 𝓤 S) , (B is-hom-image-of-class 𝒦)
```

2.4.2 Lifting tools

Here are some tools that have been useful (e.g., in the road to the proof of Birkhoff's HSP theorem). The first states and proves the simple fact that the lift of an epimorphism is an epimorphism.

```
lift-of-alg-epic-is-epic : (𝓧 : Universe){𝓦 : Universe}
                           {A : Algebra 𝓧 S}(B : Algebra 𝓨 S)(h : hom A B)
                           ————————————————————————————————
  →                        Epic ∣ h ∣ → Epic ∣ lift-alg-hom 𝓧 𝓦 B h ∣

lift-of-alg-epic-is-epic 𝓧 {𝓦} {A} B h hepi y = eq y (lift a) η
  where
  lh : hom (lift-alg A 𝓧) (lift-alg B 𝓦)
  lh = lift-alg-hom 𝓧 𝓦 B h

  ζ : Image ∣ h ∣ ∋ (lower y)
  ζ = hepi (lower y)

  a : ∣ A ∣
  a = Inv ∣ h ∣ ζ

  β : lift (∣ h ∣ a) ≡ (lift ∘ ∣ h ∣ ∘ lower{𝓦}) (lift a)
  β = ap (λ - → lift (∣ h ∣ ( - a))) (lower∼lift {𝓦} )

  η : y ≡ ∣ lh ∣ (lift a)
  η = y                ≡⟨ (extfun lift∼lower) y ⟩
      lift (lower y)   ≡⟨ ap lift (InvIsInv ∣ h ∣ ζ)⁻¹ ⟩
      lift (∣ h ∣ a)   ≡⟨ β ⟩
      ∣ lh ∣ (lift a) ■
```

```
lift-alg-hom-image : {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}{𝓦}{𝐀}{𝐁} ((𝐂 , ϕ , ϕhom , ϕepic) , C≅B) =
  (lift-alg 𝐂 𝓦 , ∣ lϕ ∣ , ∥ lϕ ∥ , lϕepic) , lift-alg-iso C≅B
    where
    lϕ : hom (lift-alg 𝐀 𝓧) (lift-alg 𝐂 𝓦)
    lϕ = (lift-alg-hom 𝓧 𝓦 𝐂) (ϕ , ϕhom)

    lϕepic : Epic ∣ lϕ ∣
    lϕepic = lift-of-alg-epic-is-epic 𝓧 𝐂 (ϕ , ϕhom) ϕepic
```

3 Types for Terms

3.1 Basic Definitions

This section presents the Terms.Basic module of the AgdaUALib, slightly abridged.[15] The theoretical background that begins each subsection below is based on Section 4.3 of Cliff Bergman's excellent textbook on universal algebra, [1, §4.3]. Apart from notation, our presentation is similar to Bergman's, but we will try to be concise, omitting some details and examples, in order to more quickly arrive at our objective, which is to use type theory to express the concepts and formalize them in the Agda language. We refer the reader to [1] for a more complete exposition of classical (informal) universal algebra.

3.1.1 The type of terms

Fix a signature S and let X denote an arbitrary nonempty collection of variable symbols. Assume the symbols in X are distinct from the operation symbols of S, that is $X \cap \mid S \mid = \emptyset$. By a *word* in the language of S, we mean a nonempty, finite sequence of members of $X \cup \mid S \mid$. We denote the concatenation of such sequences by simple juxtaposition. Let S_0 denote the set of nullary operation symbols of S. We define by induction on n the sets T_n of *words* over $X \cup \mid S \mid$ as follows (cf. [1, Def. 4.19]):

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

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

We define the collection of *terms* in the signature S over X by Term $X := \bigcup_n T_n$. By an *S-term* we mean a term in the language of S. Since the definition of Term X is recursive, it would seem that an inductive type could be used to represent the semantic notion of terms in type theory. Indeed, the *inductive type of terms* in the UALib is one such representation; it is defined as follows.

[15] For unabridged docs and source code see https://ualib.gitlab.io/Terms.Basic.html and https://gitlab.com/ualib/ualib.gitlab.io/-/blob/master/UALib/Terms/Basic.lagda.

```
data Term {𝔛 : Universe}(X : 𝔛 ̇ ) : ov 𝔛 ̇ where
  generator : X → Term X
  node : (f : ∣ S ∣)(t : ∥ S ∥ f → Term X) → Term X
```

This is a very basic inductive type that represents each term as a tree with an operation symbol at each node and a variable symbol at each leaf (generator).

Notation. As usual, the type X represents an arbitrary collection of variable symbols. Recall, ov $\mathfrak{X}$ ̇ is our shorthand notation for the universe $\mathcal{O} \sqcup \mathcal{V} \sqcup \mathfrak{X}^{+}$ ̇. Throughout this module the name of the first constructor of the Term type will remain generator. However, in all of the modules that follow, we will use the shorthand g to denote the generator constructor.

3.1.2 The term algebra

For a given signature S, if the type Term X is nonempty (equivalently, if X or $\mid S \mid$ is nonempty), then we can define an algebraic structure, denoted by $\mathbf{T}\ X$, called the *term algebra in the signature S over X*. Since terms take other terms as arguments they do double-duty, serving as both the elements of the domain and the basic operations of the algebra.

- For each operation symbol $f : \mid S \mid$, denote by $f\ \hat{}\ (\mathbf{T}\ X)$ the operation on Term X which maps each tuple $t : \| S \| f \to \mid \mathbf{T}\ X \mid$ of terms to the formal term $f\ t$.
- Define $\mathbf{T}\ X$ to be the algebra with universe $\mid \mathbf{T}\ X \mid :=$ Term X and operations $f\ \hat{}\ (\mathbf{T}\ X)$, one for each symbol f in $\mid S \mid$.

In Agda the term algebra can be defined as simply as one could hope.

```
𝐓 : {𝔛 : Universe}(X : 𝔛 ̇ ) → Algebra (ov 𝔛) S
𝐓 X = Term X , node
```

3.1.3 The universal property

The term algebra $\mathbf{T}\ X$ is *absolutely free* (or *universal* or *initial*) for algebras in the signature S. That is, for every S-algebra $\mathbf{A}$, the following hold.

1. Every function from X to $\mid \mathbf{A} \mid$ lifts to a homomorphism from $\mathbf{T}\ X$ to $\mathbf{A}$.
2. The homomorphism that exists by item 1 is unique.

We now prove this in Agda, starting with the fact that every map from X to $\mid \mathbf{A} \mid$ lifts to a map from $\mid \mathbf{T}\ X \mid$ to $\mid \mathbf{A} \mid$ in a natural way, by induction on the structure of a given term.

```
free-lift : (𝐀 : Algebra 𝓤 S)(h : X → ∣ 𝐀 ∣) → ∣ 𝐓 X ∣ → ∣ 𝐀 ∣
free-lift _ h (generator x) = h x
free-lift 𝐀 h (node f t) = (f ̂ 𝐀) (λ i → free-lift 𝐀 h (t i))
```

Naturally, at the base step of the induction, when the term has the form generator x, the free lift of h agrees with h. For the inductive step, when the given term has the form node $f\ t$, the free lift is defined as follows: Assuming (the induction hypothesis) that we know the image of each subterm $t\ i$ under the free lift of h, define the free lift at the full term by applying $f\ \hat{}\ \mathbf{A}$ to the images of the subterms.

The free lift so defined is a homomorphism by construction. Indeed, here is the formal statement and (trivial) proof of this fact.

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

Finally, we prove that the homomorphism is unique. This requires funext $\mathcal{V}$ $\mathcal{U}$ (i.e., *function extensionality* at universe levels $\mathcal{V}$ and $\mathcal{U}$) which we postulate by making it part of the premise in the following function type definition.

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

```
free-unique _ _ _ _ p (generator x) = p x
free-unique fe 𝐀 g h p (node f t) = ∣ g ∣ (node f t) ≡⟨ ∥ g ∥ f t ⟩
                                    (f ̂ 𝐀)(∣ g ∣ ∘ t) ≡⟨ α ⟩
                                    (f ̂ 𝐀)(∣ h ∣ ∘ t) ≡⟨ (∥ h ∥ f t)⁻¹ ⟩
                                    ∣ h ∣ (node f t) ■
  where
  α : (f ̂ 𝐀) (∣ g ∣ ∘ t) ≡ (f ̂ 𝐀) (∣ h ∣ ∘ t)
  α = ap (f ̂ 𝐀) (fe λ i → free-unique fe 𝐀 g h p (t i))
```

Let's account for what we have proved thus far about the term algebra. If we postulate a type $X : \mathcal{X}$ ̇ (representing an arbitrary collection of variable symbols) such that for each S-algebra $\mathbf{A}$ there is a map from X to the domain of $\mathbf{A}$, then it follows that for every S-algebra $\mathbf{A}$ there is a homomorphism from $\mathbf{T}\ X$ to $\vert\ \mathbf{A}\ \vert$ that "agrees with the original map on X," by which we mean that for all $x : X$ the lift evaluated at generator x is equal to the original function evaluated at x.

If we further assume that each of the mappings from X to $\vert\ \mathbf{A}\ \vert$ is *surjective*, then the homomorphisms constructed with free-lift and lift-hom are *epimorphisms*, as we now prove.

```
lift-of-epi-is-epi : {𝐀 : Algebra 𝓤 S}{h₀ : X → ∣ 𝐀 ∣}
                     ——————————————————————————————-
  →                  Epic h₀ → Epic ∣ lift-hom 𝐀 h₀ ∣
```

```
lift-of-epi-is-epi {𝐀}{h₀} hE y = γ
  where
  h₀⁻¹y = Inv h₀ (hE y)

  η : y ≡ ∣ lift-hom 𝐀 h₀ ∣ (generator h₀⁻¹y)
  η = (InvIsInv h₀ (hE y))⁻¹

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

The lift-hom and lift-of-epi-is-epi types will be called to action when such epimorphisms are needed later (e.g., in the Varieties module).

3.2 Term Operations

This section presents the Terms.Operations module of the AgdaUALib, slightly abridged.[16] Here we define *term operations* which are simply terms interpreted in a particular algebra, and we prove some compatibility properties of term operations.[17]

When we interpret a term in an algebra we call the resulting function a *term operation*. Given a term p and an algebra $\mathbf{A}$, we denote by $p\ ̇\ \mathbf{A}$ the *interpretation* of p in $\mathbf{A}$. This is defined inductively as follows.

1. If $p = x$ (a variable symbol) and $a : X \to |\ \mathbf{A}\ |$ a tuple of elements from the domain of $\mathbf{A}$, then $(p\ ̇\ \mathbf{A})\ a := a\ x$.
2. If $p = f\ t$ (where f is an operation symbol and t is a tuple of terms) and if $a : X \to |\ \mathbf{A}\ |$ is a tuple from $\mathbf{A}$, then we define $(p\ ̇\ \mathbf{A})\ a = (f\ t\ ̇\ \mathbf{A})\ a := (f\ ̂\ \mathbf{A})\ (\lambda\ i \to (t\ i\ ̇\ \mathbf{A})\ a)$.

Thus the interpretation of a term is defined by induction on the structure of the term, and the definition is formally implemented in the UALib as follows.

```
_ ̇_ : Term X → (𝐀 : Algebra 𝓤 S) → (X → | 𝐀 |) → | 𝐀 |
(ℊ x ̇ 𝐀) a = a x
(node f t ̇ 𝐀) a = (f ̂ 𝐀) λ i → (t i ̇ 𝐀) a
```

It turns out that the interpretation of a term is the same as the free-lift (modulo argument order and assuming function extensionality).

```
free-lift-interp : dfunext 𝓥 𝓤 → (𝐀 : Algebra 𝓤 S)(h : X → | 𝐀 |)(p : Term X)
  →                (p ̇ 𝐀) h ≡ (free-lift 𝐀 h) p
free-lift-interp _ 𝐀 h (ℊ x) = ρεφℓ
free-lift-interp fe 𝐀 h (node f t) = ap (f ̂ 𝐀) (fe λ i → free-lift-interp fe 𝐀 h (t i))
```

If the algebra $\mathbf{A}$ happens to be $\mathbf{T}\ X$, then we expect that $\forall\ s$ we have $(p\ ̇\ \mathbf{T}\ X)\ s \equiv p\ s$. But what is $(p\ ̇\ \mathbf{T}\ X)\ s$ exactly? By definition, it depends on the form of p as follows:

- if $p \equiv ℊ\ x$, then $(p\ ̇\ \mathbf{T}\ X)\ s := (ℊ\ x\ ̇\ \mathbf{T}\ \mathrm{X})\ s \equiv s\ x$;
- if $p = \mathsf{node}\ f\ t$, then $(p\ ̇\ \mathbf{T}\ X)\ s := (\mathsf{node}\ f\ t\ ̇\ \mathbf{T}\ X)\ s \equiv (f\ ̂\ \mathbf{T}\ X)\ \lambda\ i \to (t\ i\ ̇\ \mathbf{T}\ X)\ s$.

Now, assume $\phi : \mathsf{hom}\ \mathbf{T}\ \mathbf{A}$. Then by comm-hom-term, we have $|\ \phi\ |\ (p\ ̇\ \mathbf{T}\ X)\ s \equiv (p\ ̇\ \mathbf{A})\ |\ \phi\ | \circ s$.

- if $p \equiv ℊ\ x$ (and $t : X \to |\ \mathbf{T}\ X\ |$), then
  $|\ \phi\ |\ p := |\ \phi\ |\ (ℊ\ x) \equiv |\ \phi\ |\ (\lambda\ t \to t\ x) \equiv \lambda\ t \to (|\ \phi\ | \circ t)\ x$;
- if $p \equiv \mathsf{node}\ f\ t$, then
  $|\ \phi\ |\ \mathrm{p} := |\ \phi\ |\ (p\ ̇\ \mathbf{T}\ X)\ s = (\mathsf{node}\ f\ t\ ̇\ \mathbf{T}\ \mathrm{X})\ s\ (f\ ̂\ \mathbf{T}\ X)\ \lambda\ i \to (t\ i\ ̇\ \mathbf{T}\ X)\ s$.

We claim that for all $p : \mathsf{Term}\ X$ there exist $q : \mathsf{Term}\ X$ and $t : X \to |\ \mathbf{T}\ X\ |$ such that $p \equiv (q\ ̇\ \mathbf{T}\ X)\ t$. We prove this fact as follows.

```
term-interp : {𝔛 : Universe}{X : 𝔛 ̇} (f : | S |){s t : ∥ S ∥ f → Term X}
  →           s ≡ t → node f s ≡ (f ̂ 𝐓 X) t
term-interp f {s}{t} st = ap (node f) st
```

---

[16] For unabridged docs and source code see https://ualib.gitlab.io/Terms.Operations.html and https://gitlab.com/ualib/ualib.gitlab.io/-/blob/master/UALib/Terms/Operations.lagda.

[17] **Notation**. At the start of the TermsOperations module, for notational convenience we rename the generator constructor of the Term type, so that from now on we can use ℊ in place of generator.

```
module _ {𝓧 : Universe}{X : 𝓧 ̇}{fe : dfunext 𝓥 (ov 𝓧)} where

  term-gen : (p : ∣ 𝐓 X ∣) → Σ q ꞉ ∣ 𝐓 X ∣ , p ≡ (q ̇ 𝐓 X) ℊ
  term-gen (ℊ x) = (ℊ x) , refl
  term-gen (node f t) = node f(λ i → ∣ term-gen (t i)∣) , term-interp f (fe λ i → ∥ term-gen (t i)∥)

  term-gen-agreement : (p : ∣ 𝐓 X ∣) → (p ̇ 𝐓 X) ℊ ≡ (∣ term-gen p ∣ ̇ 𝐓 X) ℊ
  term-gen-agreement (ℊ x) = refl
  term-gen-agreement (node f t) = ap (f ̂ 𝐓 X) (fe λ x → term-gen-agreement (t x))

  term-agreement : (p : ∣ 𝐓 X ∣) → p ≡ (p ̇ 𝐓 X) ℊ
  term-agreement p = snd (term-gen p) · (term-gen-agreement p)⁻¹
```

3.2.1 Interpretation of terms in product algebras

Note that while in the previous section it sufficed to postulate a local version of function extensionality, in the present section we will assume the full global version (global-dfunext) is in force. (We are not sure whether this is necessary or if, with some effort, we could get a more moderate invocation of function extensionality to work here.)[18]

```
  interp-prod : {𝓦 : Universe}(p : Term X){I : 𝓦 ̇}
                (𝒜 : I → Algebra 𝓤 S)(a : X → ∀ i → ∣ 𝒜 i ∣)
                ———————————————————————————————————
    →           (p ̇ (⨅ 𝒜)) a ≡ (λ i → (p ̇ 𝒜 i) (λ j → a j i))
  interp-prod (ℊ x₁) 𝒜 a = refl

  interp-prod (node f t) 𝒜 a = let IH = λ x → interp-prod (t x) 𝒜 a
    in
    (f ̂ ⨅ 𝒜)(λ x → (t x ̇ ⨅ 𝒜) a)                    ≡⟨ ap (f ̂ ⨅ 𝒜)(gfe IH) ⟩
    (f ̂ ⨅ 𝒜)(λ x → (λ i → (t x ̇ 𝒜 i)(λ j → a j i))) ≡⟨ refl ⟩
    (λ i → (f ̂ 𝒜 i) (λ x → (t x ̇ 𝒜 i)(λ j → a j i))) ■


  interp-prod2 : (p : Term X){I : 𝓤 ̇}(𝒜 : I → Algebra 𝓤 S)
                 ———————————————————————————————————————
    →            (p ̇ ⨅ 𝒜) ≡ λ(t : X → ∣ ⨅ 𝒜 ∣) → (λ i → (p ̇ 𝒜 i)(λ x → t x i))

  interp-prod2 (ℊ x₁) 𝒜 = refl
  interp-prod2 (node f t) 𝒜 = gfe λ (tup : X → ∣ ⨅ 𝒜 ∣) →
    let IH = λ x → interp-prod (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))) ■
```

3.2.2 Compatibility of terms

We now prove two important facts about term operations. The first of these, which is used very often in the sequel, asserts that every term commutes with every homomorphism.

[18] We plan to resolve this, and if possible improve upon our treatment of function extensionality, before the next major release of the AgdaUALib.

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

comm-hom-term 𝐁 h (ℊ x) a = ℜℯ𝓁
comm-hom-term {𝐀} 𝐁 h (node f t) a = ∣ h ∣((f ̂ 𝐀)λ i → (t i ̇ 𝐀) a)  ≡⟨ i ⟩
                                      (f ̂ 𝐁)(λ i → ∣ h ∣((t i ̇ 𝐀) a)) ≡⟨ ii ⟩
                                      (f ̂ 𝐁)(λ r → (t r ̇ 𝐁)(∣ h ∣ ∘ a)) ■
  where
  i = ∥ h ∥ f(λ r → (t r ̇ 𝐀) a)
  ii = ap (f ̂ 𝐁)(gfe (λ i → comm-hom-term 𝐁 h (t i) a))
```

To conclude the Terms module, we prove that every term is compatible with every congruence relation. That is, if t : Term X and θ : Con $\mathbf{A}$, then $a\ \theta\ b \to t(a)\ \theta\ t(b)$.

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

compatible-term (ℊ x) θ p = p x
compatible-term (node f t) θ p = snd ∥ θ ∥ f λ x → (compatible-term (t x) θ) p
```

4 Subalgebra Types

4.1 Subuniverses

This section presents the Subalgebras.Subuniverses module of the AgdaUALib, slightly abridged.[19] We start by defining a type that represents the important concept of *subuniverse*. Suppose $\mathbf{A}$ is an algebra. A subset $B \subseteq |\mathbf{A}|$ is said to be *closed under the operations of* $\mathbf{A}$ if for each $f \in |S|$ and all tuples $b : \|S\| f \to B$ the element $(f\ \hat{}\ \mathbf{A})\ b$ belongs to B. If a subset $B \subseteq A$ is closed under the operations of $\mathbf{A}$, then we call B a *subuniverse* of $\mathbf{A}$.

We first show how to represent in Agda the collection of subuniverses of an algebra $\mathbf{A}$. Since a subuniverse is viewed as a subset of the domain of $\mathbf{A}$, we define it as a predicate on $|\mathbf{A}|$. Thus, the collection of subuniverses is a predicate on predicates on $|\mathbf{A}|$.

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

An algebra can be constructed out of a subuniverse in the following natural way.

```
SubunivAlg : (𝐀 : Algebra 𝓤 S)(B : Pred ∣ 𝐀 ∣ 𝓦) → B ∈ Subuniverses 𝐀 → Algebra (𝓤 ⊔ 𝓦) S
SubunivAlg 𝐀 B B∈SubA = Σ B , λ f b → (f ̂ 𝐀)(fst ∘ b) , B∈SubA f (fst ∘ b)(snd ∘ b)
```

[19] For unabridged docs and source code see https://ualib.gitlab.io/Subalgebras.Subuniverses.html and https://gitlab.com/ualib/ualib.gitlab.io/-/blob/master/UALib/Subalgebras/Subuniverses.lagda.

4.1.1 Subuniverses as records

Next we define a type to represent a single subuniverse of an algebra. If $\mathbf{A}$ is the algebra in question, then the subuniverse will be a subset of (i.e., predicate over) the domain '$\mid \mathbf{A} \mid$' that belongs to Subuniverses $\mathbf{A}$.

```
record Subuniverse {𝐀 : Algebra 𝓤 S} : ov (𝓤 ⊔ 𝓦) ˙ where
  constructor mksub
  field
    sset : Pred ∣ 𝐀 ∣ 𝓦
    isSub : sset ∈ Subuniverses 𝐀
```

As an example application, here is a formal proof that the equalizer of two homomorphisms with domain $\mathbf{A}$ is a subuniverse of $\mathbf{A}$.

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

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

4.1.2 Subuniverse Generation

If $\mathbf{A}$ is an algebra and $X \subseteq \mid \mathbf{A} \mid$ a subset of the domain of $\mathbf{A}$, then the *subuniverse of* $\mathbf{A}$ *generated by* X is typically denoted by $\mathrm{Sg}^{\mathbf{A}}\ (X)$ and defined to be the smallest subuniverse of $\mathbf{A}$ containing X. Equivalently,

$$\mathrm{Sg}^{\mathbf{A}}\ (X) = \bigcap \{U : U \text{ is a subuniverse of } \mathbf{A} \text{ and } B \subseteq U\}.$$

We define an inductive type, denoted by Sg, that represents the subuniverse generated by a given subset of the domain of a given algebra, as follows.

```
data Sg (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
```

Given an arbitrary S-algebra $\mathbf{A}$ and subset X of $\mid \mathbf{A} \mid$, the type Sg X does indeed represent a subuniverse of $\mathbf{A}$; proving this with the inductive type Sg is trivial, as we see here.

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

Next we prove by structural induction that Sg X is the smallest subuniverse of $\mathbf{A}$ containing X.

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

sgIsSmallest _ _ _ XinY (var Xv) = XinY Xv

sgIsSmallest 𝐀 Y YsubA XinY (app f a SgXa) = fa∈Y
  where
  IH : Im a ⊆ Y
  IH i = sgIsSmallest 𝐀 Y YsubA XinY (SgXa i)

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

When the inhabitant of Sg X is constructed as app $f\ a\ SgXa$, we may assume (the induction hypothesis) that the arguments in the tuple a belong to Y. Then the result of applying f to a also belongs to Y since Y is a subuniverse.

4.1.3 Subuniverse Lemmas

Here we formalize a few basic properties of subuniverses. First, the intersection of subuniverses is again a subuniverse.

```
sub-intersection : {A : Algebra 𝓤 S}{I : ℐ ̇}{𝒜 : I → Pred | A | 𝓦}
  →               (Π i ꞉ I , 𝒜 i ∈ Subuniverses A) → ⋂ I 𝒜 ∈ Subuniverses A

sub-intersection α f a β = λ i → α i f a λ j → β j i
```

In the proof above, we assume the following typing judgments:

α : $\forall\ i \to \mathscr{A}\ i \in$ Subuniverses $\mathbf{A}$
f : $|\ S\ |$
a : $\|\ S\ \|\ f \to |\ \mathbf{A}\ |$
β : Im $a \subseteq \bigcap I\ \mathscr{A}$

and we must prove $(f\ \hat{}\ \mathbf{A})\ a \in \bigcap I\ \mathscr{A}$. In this case, Agda will fill in the proof term $\lambda\ i \to \alpha\ i\ f\ a\ (\lambda\ x \to \beta\ x\ i)$ automatically with the command `C-c C-a` in agda2-mode.

Next we formalize the proof that subuniverses are closed under the action of term operations.

```
sub-term-closed : {𝔛 : Universe}{X : 𝔛 ̇}(A : Algebra 𝓤 S){B : Pred | A | 𝓦}
  →               (B ∈ Subuniverses A) → (t : Term X)(b : X → | A |)
  →               (∀ x → b x ∈ B) → ((t ̇ A) b) ∈ B
sub-term-closed A α (ℊ x) b Bb = Bb x
sub-term-closed A {B} α (node f t) b β =
  α f (λ z → (t z ̇ A) b) λ x → sub-term-closed A {B} α (t x) b β
```

In the induction step of the foregoing proof, the typing judgments of the premise are these:

$\mathbf{A}$: Algebra $\mathcal{U}\ S$ B : Pred | $\mathbf{A}$ | $\mathcal{W}$ α : $B \in$ Subuniverses $\mathbf{A}$
f : $|\ S\ |$ t : $\|\ S\ \|\ f \to$ Term X
b : $X \to |\ \mathbf{A}\ |$ β : $\forall\ x \to b\ x \in B$

This is another instance in which Agda will fill in the correct proof term if we invoke the command `C-c C-a` in agda2-mode.

Alternatively, we could express the preceding fact using an inductive type representing images of terms.

```
data TermImage (A : Algebra 𝓤 S)(Y : Pred | A | 𝓦) : Pred | A | (𝓞 ⊔ 𝓥 ⊔ 𝓤 ⊔ 𝓦)
  where
  var : ∀ {y : | A |} → y ∈ Y → y ∈ TermImage A Y
  app : ∀ f t → Π x ꞉ ‖ S ‖ f , t x ∈ TermImage A Y → (f ̂ A) t ∈ TermImage A Y
```

By what we proved above, it should come as no surprise that TermImage $\mathbf{A}\ Y$ is a subuniverse of $\mathbf{A}$ that contains Y. Indeed, the proof is trivial.

```
TermImageIsSub : {A : Algebra 𝓤 S}{Y : Pred | A | 𝓦} → TermImage A Y ∈ Subuniverses A
TermImageIsSub = app

Y-onlyif-TermImageY : {A : Algebra 𝓤 S}{Y : Pred | A | 𝓦} → Y ⊆ TermImage A Y
Y-onlyif-TermImageY {a} Ya = var Ya
```

Since Sg **A** Y is the smallest subuniverse containing Y, we obtain the following inclusion.

```
SgY-onlyif-TermImageY : (A : Algebra 𝓤 S)(Y : Pred ∣ A ∣ 𝓦) → Sg A Y ⊆ TermImage A Y
SgY-onlyif-TermImageY A Y = sgIsSmallest A(TermImage A Y) TermImageIsSub Y-onlyif-TermImageY
```

4.1.4 Homomorphic images are subuniverses

Now that we have developed the machinery of subuniverse generation, we can prove two basic facts that play an important role in many theorems about algebraic structures. First, the image of a homomorphism is a subuniverse of its codomain.

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

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

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

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

Next we prove the important fact that homomorphisms are uniquely determined by their values on a generating set.

```
hom-unique : funext 𝓥 𝓦 → {A : Algebra 𝓤 S}{B : Algebra 𝓦 S}
             (X : Pred ∣ A ∣ 𝓤) (g h : hom A B)
  →          Π x ꞉ ∣ A ∣ , (x ∈ X → ∣ g ∣ x ≡ ∣ h ∣ x)
             ---------------------------------------
  →          Π a ꞉ ∣ A ∣ , (a ∈ Sg A X → ∣ g ∣ a ≡ ∣ h ∣ a)

hom-unique _ _ _ _ α a (var x) = α a x

hom-unique fe {A}{B} X g h α fa (app f a β) = ∣ g ∣ ((f ̂ A) a)  ≡⟨ ∥ g ∥ f a ⟩
                                              (f ̂ B)(∣ g ∣ ∘ a ) ≡⟨ ap (f ̂ B)(fe IH) ⟩
                                              (f ̂ B)(∣ h ∣ ∘ a) ≡⟨ ( ∥ h ∥ f a )⁻¹ ⟩
                                              ∣ h ∣ ((f ̂ A) a ) ■
  where IH = λ x → hom-unique fe {A}{B} X g h α (a x) (β x)
```

In the induction step of the foregoing proof, the typing judgments of the premise are these:

fe :	funext $\mathcal{V}$ $\mathcal{W}$	**A** :	Algebra $\mathcal{U}$ S	**B** :	Algebra $\mathcal{W}$ S
X :	Pred ∣ **A** ∣ $\mathcal{U}$	g h :	hom **A** **B**	α :	Π x ꞉ ∣ **A** ∣ , ($x \in X \to$ ∣ g ∣ $x \equiv$ ∣ h ∣ x)
fa :	∣ **A** ∣	f :	∣ S ∣	a :	∥ S ∥ $f \to$ ∣ **A** ∣
β :	Im $a \subseteq$ Sg **A** X				

where $fa = (f \hat{} \mathbf{A})\ a$. Under these assumptions, we prove ∣ g ∣ $((f \hat{} \mathbf{A})\ a) \equiv$ ∣ h ∣ $((f \hat{} \mathbf{A})\ a)$.

4.2 Subalgebras

This section presents the Subalgebras.Subalgebras module of the AgdaUALib, slightly abridged.[20] Here we define the Subalgebra type, representing the subalgebra of a given algebra, as well as the collection of all subalgebras of a given class of algebras.

4.2.1 Subalgebra type

Given algebras $\mathbf{A}$: Algebra $\mathcal{U}$ S and $\mathbf{B}$: Algebra $\mathcal{W}$ S, we say that $\mathbf{B}$ is a *subalgebra* of $\mathbf{A}$ just in case $\mathbf{B}$ can be *homomorphically embedded* in $\mathbf{A}$; i.e., there exists a map h : $\mid \mathbf{B} \mid \to \mid \mathbf{A} \mid$ that is both a homomorphism and an embedding.[21]

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

Subalgebra : {𝓦 𝓤 : Universe} → Algebra 𝓤 S → ov 𝓦 ⊔ 𝓤 ̇
Subalgebra {𝓦} 𝐀 = Σ 𝐁 ꞉ (Algebra 𝓦 S) , 𝐁 IsSubalgebraOf 𝐀
```

Note the order of the arguments. The universe $\mathcal{W}$ comes first because in certain situations we have to explicitly specify this universe, whereas we can almost always leave the universe $\mathcal{U}$ implicit. See, for example, the definition of IsSubalgebraOfClass below (§4.2.3).

4.2.2 Consequences of First Homomorphism Theorem

We take this opportunity to prove an important lemma that makes use of the IsSubalgebraOf type defined above; it is the following: If $\mathbf{A}$ and $\mathbf{B}$ are S-algebras and h : hom $\mathbf{A}$ $\mathbf{B}$ a homomorphism from $\mathbf{A}$ to $\mathbf{B}$, then the quotient $\mathbf{A} \,/\, \ker h$ is (isomorphic to) a subalgebra of $\mathbf{B}$. This is an easy corollary of the First Homomorphism Theorem proved in the Homomorphisms.Noether module.

```
FirstHomCorollary : {𝓤 𝓦 : Universe}
                    -- extensionality assumptions --
  →                   dfunext 𝓥 𝓦 → prop-ext 𝓤 𝓦

  →                 (𝐀 : Algebra 𝓤 S)(𝐁 : Algebra 𝓦 S)(h : hom 𝐀 𝐁)

                    -- truncation assumptions --
  →                   is-set ∣ 𝐁 ∣
  →                   (∀ a x → is-subsingleton (⟨ kercon 𝐁 h ⟩ a x))
  →                   (∀ C → is-subsingleton (𝒞{A = ∣ 𝐀 ∣}{⟨ kercon 𝐁 h ⟩} C))
                    ---------------------------------------------------
  →                 (𝐀 [ 𝐁 ]/ker h) IsSubalgebraOf 𝐁

FirstHomCorollary fe pe 𝐀 𝐁 h Bset ssR ssA = ϕhom , ϕemb
  where
```

[20] For unabridged docs and source code see `https://ualib.gitlab.io/Subalgebras.Subalgebras.html` and `https://gitlab.com/ualib/ualib.gitlab.io/-/blob/master/UALib/Subalgebras/Subalgebras.lagda`.

[21] An alternative which could end up being simpler and easier to work with would be to proclaim $\mathbf{B}$ a subalgebra of $\mathbf{A}$ iff there is an *injective* homomorphism from B into $\mathbf{A}$. In preparation for the next major release of the UALib, we will investigate the consequences of taking that path instead of the stricter embedding requirement we chose for the definition of the type IsSubalgebraOf.

```
FirstHomThm : Σ ϕ ꞉ hom (𝐀 [ 𝐁 ]/ker h) 𝐁 , (∣ h ∣ ≡ ∣ ϕ ∣ ∘ ∣ πker 𝐁 h ∣ )
                                              × Monic ∣ ϕ ∣ × is-embedding ∣ ϕ ∣
FirstHomThm = FirstHomomorphismTheorem fe pe 𝐀 𝐁 h Bset ssR ssA

ϕhom : hom (𝐀 [ 𝐁 ]/ker h) 𝐁
ϕhom = ∣ FirstHomThm ∣

ϕemb : is-embedding ∣ ϕhom ∣
ϕemb = snd (snd (snd FirstHomThm))
```

One special case to which we will apply this is where the algebra $\mathbf{A}$ is the term algebra $\mathbf{T}\ X$. We formalize this special case here so that it's readily available when we need it later.

```
free-quot-subalg : {𝓤 𝓧 : Universe}
                   -extensionality assumptions -
 →                 dfunext 𝓥 𝓤 → prop-ext (ov 𝓧) 𝓤

 →                 (X : 𝓧 ̇)(𝐁 : Algebra 𝓤 S)(h : hom (𝐓 X) 𝐁)

                   -truncation assumptions -
 →                 is-set ∣ 𝐁 ∣
 →                 (∀ p q → is-subsingleton (⟨ kercon 𝐁 h ⟩ p q))
 →                 (∀ C → is-subsingleton (𝒞{A = ∣ 𝐓 X ∣}{⟨ kercon 𝐁 h ⟩} C))
                   ———————————————————————————
 →                 ((𝐓 X) [ 𝐁 ]/ker h) IsSubalgebraOf 𝐁

free-quot-subalg fe pe X 𝐁 h Bset ssR ssB = FirstHomCorollary fe pe (𝐓 X) 𝐁 h Bset ssR ssB
```

Notation. For convenience, we define the following shorthand for the subalgebra relation.

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

From now on we will use $\mathbf{B} \leq \mathbf{A}$ to express the assertion that $\mathbf{B}$ is a subalgebra of $\mathbf{A}$.

4.2.3 Subalgebras of a class

One of our goals is to formally express and prove properties of classes of algebraic structures. Fixing a signature S and a universe $\mathcal{U}$, we define a class of S-algebras with domains of type $\mathcal{U}$ ̇ as a predicate over the Algebra $\mathcal{U}$ S type. In the syntax of the UALib, such predicates inhabit the type Pred (Algebra $\mathcal{U}$ S) $\mathcal{E}$, for some universe $\mathcal{E}$.

Suppose $\mathcal{K}$: Pred (Algebra $\mathcal{U}$ S) $\mathcal{E}$ denotes a class of S-algebras and $\mathbf{B}$: Algebra $\mathcal{W}$ S denotes an arbitrary S-algebra. Then we might wish to consider the assertion that $\mathbf{B}$ is a subalgebra of some algebra in the class $\mathcal{K}$. The next type we define allows us to express this assertion as $\mathbf{B}$ IsSubalgebraOfClass $\mathcal{K}$.

```
_IsSubalgebraOfClass_ : Algebra 𝓦 S → Pred (Algebra 𝓤 S) 𝓔 → ov (𝓤 ⊔ 𝓦) ⊔ 𝓔 ̇
𝐁 IsSubalgebraOfClass 𝒦 = Σ 𝐀 ꞉ Algebra 𝓤 S , Σ sa ꞉ Subalgebra{𝓦} 𝐀 , (𝐀 ∈ 𝒦) × (𝐁 ≅ ∣ sa ∣)
```

Using this type, we express the collection of all subalgebras of algebras in a class by the type SubalgebraOfClass, which we now define.

```
SubalgebraOfClass : {𝓦 𝓤 : Universe} → Pred (Algebra 𝓤 S)(ov 𝓤) → ov (𝓤 ⊔ 𝓦) ̇
SubalgebraOfClass {𝓦} 𝒦 = Σ 𝐁 ꞉ Algebra 𝓦 S , 𝐁 IsSubalgebraOfClass 𝒦
```

4.2.4 Subalgebra lemmas

We conclude this module by proving a number of useful facts about subalgebras. Some of the formal statements below may appear to be redundant, and indeed they are to some extent. However, each one differs slightly from the next, if only with respect to the explicitness or implicitness of their arguments. The aim is to make it as convenient as possible to apply the lemmas in different situations. (We're stocking the UALib utility closet now; elegance is not the priority.)

First we show that the subalgebra relation is a *preorder*. Recall, this means it is reflexive and transitive.[22]

```
≤-reflexive : {𝓤 : Universe}(A : Algebra 𝓤 S) → A ≤ A
≤-reflexive A = (id | A | , id-is-hom) , id-is-embedding

≤-refl : {𝓤 : Universe}{A : Algebra 𝓤 S} → A ≤ A
≤-refl {𝓤}{A} = ≤-reflexive A

≤-transitivity : (A : Algebra 𝔛 S)(B : Algebra 𝔜 S)(C : Algebra 𝔉 S)
  →              C ≤ B → B ≤ A → C ≤ A
≤-transitivity A B C CB BA = (∘-hom C A | CB | | BA |) , ∘-embedding ‖ BA ‖ ‖ CB ‖

≤-trans : (A : Algebra 𝔛 S){B : Algebra 𝔜 S}{C : Algebra 𝔉 S}
  →       C ≤ B → B ≤ A → C ≤ A
≤-trans A {B}{C} = ≤-transitivity A B C
```

Next we prove that if two algebras are isomorphic and one of them is a subalgebra of **A**, then so is the other.

```
≤-iso : (A : Algebra 𝔛 S){B : Algebra 𝔜 S}{C : Algebra 𝔉 S}
  →     C ≅ B → B ≤ A → C ≤ A
≤-iso A {B} {C} CB BA = (g ∘ f , gfhom) , gfemb
    where
      f : | C | → | B |
      f = fst | CB |
      g : | B | → | A |
      g = fst | BA |

      gfemb : is-embedding (g ∘ f)
      gfemb = ∘-embedding (‖ BA ‖) (iso→embedding CB)

      gfhom : is-homomorphism C A (g ∘ f)
      gfhom = ∘-is-hom C A {f}{g} (snd | CB |) (snd | BA |)
```

The following variations on this theme are sometimes useful.

```
≤-trans-≅ : (A : Algebra 𝔛 S){B : Algebra 𝔜 S}(C : Algebra 𝔉 S)
  →         A ≤ B → A ≅ C → C ≤ B
≤-trans-≅ A {B} C A≤B B≅C = ≤-iso B (≅-sym B≅C) A≤B
```

[22] In [3], in the Relations.Quotients module, we defined *preorder* for binary relation types. Here, however, we will content ourselves with merely proving reflexivity and transitivity of the subalgebra relation ≤, without worry about first defining it as an inhabitant of an honest-to-goodness binary relation type, of the sort introduced in the Relations.Discrete module. Perhaps we will address this matter in a future release of the UALib.

```
≤-TRANS-≅ : (𝐀 : Algebra 𝓧 S){𝐁 : Algebra 𝓨 S}(𝐂 : Algebra 𝓩 S)
  →                 𝐀 ≤ 𝐁 → 𝐁 ≅ 𝐂 → 𝐀 ≤ 𝐂
≤-TRANS-≅ 𝐀 𝐂 A≤B B≅C = ( ∘-hom 𝐀 𝐂 ∣ A≤B ∣ ∣ B≅C ∣) ,
                                  ∘-embedding (iso→embedding B≅C)(∥ A≤B ∥)
```

Next we prove a monotonicity property of $\leq$.

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

Later we will require a number of facts having to do with the relationship between the subalgebra relation and the lifting of algebras to higher universe levels. Here are the tools we need.

```
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 (𝐀 , (sa , (KA , B≅sa))) = 𝐀 , sa , KA , ≅-trans (≅-sym lift-alg-≅) B≅sa

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

≤-lift-alg : (𝐀 : Algebra 𝓧 S){𝐁 : Algebra 𝓨 S} → 𝐁 ≤ 𝐀 → 𝐁 ≤ lift-alg 𝐀 𝓩
≤-lift-alg 𝐀 {𝐁} B≤A = ≤-TRANS-≅ 𝐁 {𝐀} (lift-alg 𝐀 𝓩) B≤A lift-alg-≅

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

lift-alg-≤-lift {𝐀} 𝐁 A≤B = ≤-trans (lift-alg 𝐁 𝓦) (≤-trans 𝐁 lAA A≤B) B≤lB
  where

  lAA : (lift-alg 𝐀 𝓩) ≤ 𝐀
  lAA = lift-alg-≤ 𝐀 {𝐀} ≤-refl

  B≤lB : 𝐁 ≤ lift-alg 𝐁 𝓦
  B≤lB = ≤-lift-alg 𝐁 {𝐁} ≤-refl
```

5 Concluding Remarks

We've reached the end of the second installment in our three-part series describing the AgdaUALib. The next part [4] covers free algebras, equational classes of algebras (i.e., varieties), and Birkhoff's HSP theorem.

We conclude by noting that one of our goals is to make computer formalization of mathematics more accessible to mathematicians working in universal algebra and model theory. We welcome feedback from the community and are happy to field questions about the UALib, how it is installed, and how it can be used to prove theorems that are not yet part of the library. Merge requests submitted to the UALib's main gitlab repository are especially welcomed. Please visit the repository at `https://gitlab.com/ualib/ualib.gitlab.io/` and help us improve it.

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 Ana Bove and Peter Dybjer. *Dependent Types at Work*, pages 57–99. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009. doi:10.1007/978-3-642-03153-3_2.

3 William DeMeo. The Agda Universal Algebra Library, Part 1: Foundation. *CoRR*, abs/2103.05581, 2021. Source code: https://gitlab.com/ualib/ualib.gitlab.io. URL: https://arxiv.org/abs/2101.10166, arXiv:2103.05581.

4 William DeMeo. The Agda Universal Algebra Library, Part 3: Identity. *CoRR*, 2021. (to appear) Source code: https://gitlab.com/ualib/ualib.gitlab.io. URL: http://arxiv.org/a/demeo_w_1.

5 Martín Hötzel Escardó. Introduction to univalent foundations of mathematics with Agda. *CoRR*, abs/1911.00580, 2019. URL: http://arxiv.org/abs/1911.00580, arXiv:1911.00580.

6 Emmanuel Gunther, Alejandro Gadea, and Miguel Pagano. Formalization of universal algebra in Agda. *Electronic Notes in Theoretical Computer Science*, 338:147 – 166, 2018. The 12th Workshop on Logical and Semantic Frameworks, with Applications (LSFA 2017). URL: http://www.sciencedirect.com/science/article/pii/S1571066118300768, doi:https://doi.org/10.1016/j.entcs.2018.10.010.

7 Ulf Norell. Dependently typed programming in agda. In *International school on advanced functional programming*, pages 230–266. Springer, 2008.

8 Ulf Norell. Dependently typed programming in Agda. In *Proceedings of the 6th International Conference on Advanced Functional Programming*, AFP'08, pages 230–266, Berlin, Heidelberg, 2009. Springer-Verlag. URL: http://dl.acm.org/citation.cfm?id=1813347.1813352.

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

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

11 The Agda Team. Agda Language Reference section on Safe Agda, 2021. URL: https://agda.readthedocs.io/en/v2.6.1/language/safe-agda.html#safe-agda.

12 The Agda Team. Agda Tools Documentation section on Pattern matching and equality, 2021. URL: https://agda.readthedocs.io/en/v2.6.1/tools/command-line-options.html#pattern-matching-and-equality.

13 Philip Wadler, Wen Kokke, and Jeremy G. Siek. *Programming Language Foundations in Agda*. July 2020. URL: http://plfa.inf.ed.ac.uk/20.07/.