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

William DeMeo
https://williamdemeo.org

Jacques Carette
McMaster University

1 Introduction

The Agda Universal Algebra Library (agda-algebras) is a collection of types and programs (theorems and proofs) formalizing the foundations of universal algebra in dependent type theory using the Agda programming language and proof assistant. The agda-algebras library now includes a substantial collection of definitions, theorems, and proofs from universal algebra and equational logic and as such provides many examples that exhibit the power of inductive and dependent types for representing and reasoning about general algebraic and relational structures.

The first major milestone of the agda-algebras project is a new formal proof of *Birkhoff's variety theorem* (also known as the *HSP theorem*), the first version of which was completed in January of 2021. To the best of our knowledge, this was the first time Birkhoff's theorem had been formulated and proved in dependent type theory and verified with a proof assistant.

In this paper, we present a single Agda module called Demos.HSP. This module extracts only those parts of the library needed to prove Birkhoff's variety theorem. In order to meet page limit guidelines, and to reduce strain on the reader, we omit proofs of some routine or technical lemmas that do not provide much insight into the overall development. However, a long version of this paper, which includes all code in the Demos.HSPmodule, is available on the arXiv. [reference needed]

In the course of our exposition of the proof of the HSP theorem, we discuss some of the more challenging aspects of formalizing *universal algebra* in type theory and the issues that arise when attempting to constructively prove some of the basic results in this area. We demonstrate that dependent type theory and Agda, despite the demands they place on the user, are accessible to working mathematicians who have sufficient patience and a strong enough desire to constructively codify their work and formally verify the correctness of their results. Perhpas our presentation will be viewed as a sobering glimpse of the painstaking process of doing mathematics in the languages of dependent type theory using the Agda proof assistant. Nonetheless we hope to make a compelling case for investing in these technologies. Indeed, we are excited to share the gratifying rewards that come with some mastery of type theory and interactive theorem proving.

1.1 Prior art

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

1. In [2], Capretta formalized the basics of universal algebra in the Calculus of Inductive Constructions using the Coq proof assistant;
2. In [4], Spitters and van der Weegen formalized the basics of universal algebra and some classical algebraic structures, also in the Calculus of Inductive Constructions using the Coq proof assistant and promoting the use of type classes;

3. In [3] Gunther, et al developed what was (prior to the agda-algebras library) the most extensive library of formalized universal algebra to date; like agda-algebras, that work is based on dependent type theory, is programmed in Agda, and goes beyond the Noether isomorphism theorems to include some basic equational logic; although the coverage is less extensive than that of agda-algebras, Gunther et al do treat *multisorted* algebras, whereas agda-algebras is currently limited to single sorted structures.
4. Lynge and Spitters [@Lynge:2019] (2019) formalize basic, mutisorted universal algebra, up to the Noether isomorphism theorems, in homotopy type theory; in this setting, the authors can avoid using setoids by postulating a strong extensionality axiom called *univalence*.

Some other projects aimed at formalizing mathematics generally, and algebra in particular, have developed into very extensive libraries that include definitions, theorems, and proofs about algebraic structures, such as groups, rings, modules, etc. However, the goals of these efforts seem to be the formalization of special classical algebraic structures, as opposed to the general theory of (universal) algebras. Moreover, the part of universal algebra and equational logic formalized in the agda-algebras library extends beyond the scope of prior efforts.

2 Preliminaries

2.1 Logical foundations

An Agda program typically begins by setting some language options and by importing types from existing Agda libraries. The language options are specified using the OPTIONS *pragma* which affect control the way Agda behaves by controlling the deduction rules that are available to us and the logical axioms that are assumed when the program is type-checked by Agda to verify its correctness. Every Agda program in the agda-algebras library, including the present module (Demos.HSP), begins with the following line.

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

We give only very terse descriptions of these options, and refer the reader to the accompanying links for more details.

- *without-K* disables Streicher's K axiom; see also the section on axiom K in the Agda Language Reference Manual.
- *exact-split* makes Agda accept only those definitions that behave like so-called *judgmental* equalities. 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 the cmdoption-safe section of the Agda Tools documentation and the Safe Agda section of the Agda Language Reference.

The OPTIONS pragma is usually followed by the start of a module and a list of import directives. For example, the collection of imports required for the present (Demos.HSP) module is relatively modest and is shown below.

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

open import Algebras.Basic using ( 𝓞 ; 𝓥 ; Signature )
```

```
module Demos.HSPnew {𝑆 : Signature 𝓞 𝓥} where

open import Agda.Primitive                  using     ( _⊔_ ; lsuc )
                                            renaming ( Set to Type )
open import Data.Product                    using     ( _×_ ; Σ-syntax ; _,_ ; Σ )
                                            renaming ( proj₁ to fst ; proj₂ to snd )
open import Function                        using     ( id ; Surjection ; flip ; Injection ; _∘_ )
                                            renaming ( Func to _⟶_ )
open import Level                           using     ( Level )
open import Relation.Binary                 using     ( Setoid ; IsEquivalence ; Rel )
open import Relation.Binary.Definitions using     ( Sym ; Symmetric ; Trans ; Transitive ; Reflexive )
open import Relation.Binary.PropositionalEquality
                                            using     ( _≡_ )
open import Relation.Unary                  using     ( Pred ; _⊆_ ; _∈_ )

import Function.Definitions                         as FD
import Relation.Binary.PropositionalEquality as ≡
import Relation.Binary.Reasoning.Setoid        as SetoidReasoning

open _⟶_ using ( cong ) renaming ( f to _⟨$⟩_ )
open Setoid using ( Carrier ; isEquivalence )
```

Note, in particular, we prefer to use Type to denote the built-in Set type, and the infix long arrow symbol _⟶_ to denote the Func type of the standard library. We use fst and snd in place of proj_1 and proj_2 for the first and second projections out of the product type _×_ and, when it improves readability of the code, we use the alternative notation |_| and ∥_∥ (resp.) for these projections.

2.2 Setoids

A *setoid* is a type packaged with an equivalence relation on the collection of inhabitants of that type. Setoids are useful for representing classical (set-theory-based) mathematics in a constructive, type-theoretic way because most mathematical structures are assumed to come equipped with some (often implicit) equivalence relation manifesting a notion of equality of elements, and therefore a type-theoretic representation of such a structure should also model its equality relation.

The agda-algebras library was first developed without the use of setoids, opting instead for specially constructed experimental quotient types. However, this approach resulted in code that was hard to comprehend and it became difficult to determine whether the resulting proofs were fully constructive. In particular, our initial proof of the Birkhoff variety theorem required postulating function extensionality, an axiom that is not provable in pure Martin-Löf type theory (MLTT). [reference needed]

In contrast, our current approach using setoids makes the equality relation of a given type explicit and this transparency can make it easier to determine the correctness and constructivity of the proofs. Using setiods we need no additional axioms beyond MLTT; in particular, no function extensionality axioms are postulated in our current formalization of Birkhoff's variety theorem.

2.3 Inverses of setoid functions

We define a data type that represent the semantic concept of the *image* of a function (cf. the Overture.Func.Inverses module of the agda-algebras library).

```
module _ {𝐀 : Setoid α ρᵃ}{𝐁 : Setoid β ρᵇ} where
 open Setoid 𝐁 using ( _≈_ ; sym ) renaming ( Carrier to B )

 data Image_∋_ (f : 𝐀 ⟶ 𝐁) : B → Type (α ⊔ β ⊔ ρᵇ) where
  eq : {b : B} → ∀ a → b ≈ (f ⟨$⟩ a) → Image f ∋ b
```

An inhabitant of Image f ∋ b is a dependent pair (a , p) , where a : A and p : b ≈ f a is a proof that f maps a to b. Since the proof that b belongs to the image of f is always accompanied by a witness a : A, we can actually *compute* a (pseudo)inverse of f. For convenience, we define this inverse function, which we call Inv, and which takes an arbitrary b : B and a (witness, proof)-pair, (a , p) : Image f ∋ b, and returns the witness a.

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

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

The latter (InvIsInverseʳ) proves that Inv f is the range-restricted right-inverse of the setoid function f.

2.4 Injective and surjective setoid functions

If f : $\mathbf{A}$ ⟶ $\mathbf{B}$ is a setoid function from $\mathbf{A}$ = (A , $\approx_0$) to $\mathbf{B}$ = (B , $\approx_1$), then we call f *injective* provided ∀ (a_0 a_1 : A), f ⟨\$⟩ a_0 $\approx_1$ f ⟨\$⟩ a_1 implies a_0 $\approx_0$ a_1; we call f *surjective* provided ∀ (b : B), ∃ (a : A) such that f ⟨\$⟩ a $\approx_1$ b. We codify these definitions in Agda and prove some of their properties inside the next submodule where we first set the stage by declaring two setoids $\mathbf{A}$ and $\mathbf{B}$, naming their equality relations, and making some definitions from the standard library available.

```
module _ {𝐀 : Setoid α ρᵃ}{𝐁 : Setoid β ρᵇ} where
 open Setoid 𝐀 using () renaming ( _≈_ to _≈₁_ )
 open Setoid 𝐁 using () renaming ( _≈_ to _≈₂_ )
 open FD _≈₁_ _≈₂_
```

The Agda Standard Library represents injective functions on bare types by the type Injective, which we now use to define IsInjective representing the property of being an injective setoid function. We then define the type IsSurjective to represent the property of being a surjective setoid function. Finally, we define SurjInv to represent the *right-inverse* of a surjective function. The definitions are as follows (cf. Overture.Func.Injective and Overture.Func.Surjective in the agda-algebras library).

```
IsInjective : (𝐀 ⟶ 𝐁) → Type (α ⊔ ρᵃ ⊔ ρᵇ)
IsInjective f = Injective (_⟨$⟩_ f)

IsSurjective : (𝐀 ⟶ 𝐁) → Type (α ⊔ β ⊔ ρᵇ)
IsSurjective F = ∀ {y} → Image F ∋ y
```

```
SurjInv : (f : A ⟶ B) → IsSurjective f → Carrier B → Carrier A
SurjInv f fonto b = Inv f (fonto {b})
```

2.4.1 Composition of injective and surjective setoid functions

Proving that the composition of injective setoid functions is again injective is simply a matter of composing the two assumed witnesses to injectivity. Proving that surjectivity is preserved under composition is only slightly more involved.

```
module _ {A : Setoid α ρᵃ}{B : Setoid β ρᵇ}{C : Setoid γ ρᶜ}
         (f : A ⟶ B)(g : B ⟶ C) where

 ∘-IsInjective : IsInjective f → IsInjective g → IsInjective (g ⟨∘⟩ f)
 ∘-IsInjective finj ginj = finj ∘ ginj

 ∘-IsSurjective : IsSurjective f → IsSurjective g → IsSurjective (g ⟨∘⟩ f)
 ∘-IsSurjective fonto gonto {y} = Goal
  where
  mp : Image g ∋ y → Image g ⟨∘⟩ f ∋ y
  mp (eq c p) = η fonto
   where
   open Setoid C using ( trans )
   η : Image f ∋ c → Image g ⟨∘⟩ f ∋ y
   η (eq a q) = eq a (trans p (cong g q))

  Goal : Image g ⟨∘⟩ f ∋ y
  Goal = mp gonto
```

2.5 Kernels

The *kernel* of a function f : A → B (where A and B are bare types) is defined informally by

$$\{(x , y) \in A \times A : f\, x = f\, y \}.$$

This can be represented in Agda in a number of ways, but for our purposes we find it most convenient to define the kernel as an inhabitant of a (unary) predicate over the square of the function's domain.

```
module _ {A : Type α}{B : Type β} where

 kernel : Rel B ρ → (A → B) → Pred (A × A) ρ
 kernel _≈_ f (x , y) = f x ≈ f y
```

The kernel of a setoid function f : $A \longrightarrow B$ is defined informally by

$$\{(x , y) \in A \times A : f \langle\$\rangle x \approx f \langle\$\rangle y\},$$

where _≈_ denotes the equality of B.

```
module _ {A : Setoid α ρᵃ}{B : Setoid β ρᵇ} where
 open Setoid A using () renaming ( Carrier to A )

 ker : (A ⟶ B) → Pred (A × A) ρᵇ
 ker g (x , y) = g ⟨$⟩ x ≈ g ⟨$⟩ y where open Setoid B using ( _≈_ )
```

3 Algebras

In this section we define the notion of an algebraic structure whose *domain* (or “carrier” or “universe”) is a setoid. Our first goal is to develop a working vocabulary and formal types for classical (single-sorted, set-based) universal algebra.

3.1 Signature of an algebra

In model theory, the *signature* $S = (C, F, R, \rho)$ of a structure consists of three (possibly empty) sets C, F, and R—called *constant symbols*, *function symbols*, and *relation symbols*, respectively—along with a function $\rho : C + F + R \to N$ that assigns an *arity* to each symbol. Often, but not always, N is taken to be the set of natural numbers.

As our focus here is universal algebra, we are more concerned with the restricted notion of an *algebraic signature* (or *signature* for algebraic structures), by which we mean a pair S = ((F , ρ)) consisting of a collection F of *operation symbols* and an *arity function* ρ : F → N that maps each operation symbol to its arity; here, N denotes the *arity type*. Heuristically, the arity ρ f of an operation symbol f ∈ F may be thought of as the “number of arguments” that f takes as “input.”

If the arity of f is n, then we call f an n-*ary* operation symbol. In case n is 0 (or 1 or 2 or 3, respectively) we call the function *nullary* (or *unary* or *binary* or *ternary*, respectively).

If A is a set and f is a (ρ f)-ary operation on A then the arguments of f form a (ρf)-tuple, say, (a 0, a 1, . . ., a (ρf - 1)), which may be viewed as the graph of a function, say, a : ρ f → A. When the codomain of ρ is ℕ, we may view ρ f as the finite set {0, 1, . . ., ρf - 1}. Thus, by identifying the ρf-th power of A with the type ρf → A of functions from {0, 1, . . ., ρf - 1} to A, we identify the collection of all tuples of arguments of f with the function type (ρf → A) → A.

3.1.1 Signature type

The agda-algebras library represents a *signature* as an inhabitant of the following dependent pair type.

```
Signature : (𝓞 𝓥 : Level) → Type (lsuc (𝓞 ⊔ 𝓥))
Signature 𝓞 𝓥 = Σ[ F ∈ Type 𝓞 ] (F → Type 𝓥)
```

Using special syntax for the first and second projections—∣_∣ and ∥_∥ (resp.)—if S : Signature 𝓞 𝓥 is a signature, then

- ∣ S ∣ denotes the set of operation symbols;
- ∥ S ∥ denotes the arity function.

Thus, if f : ∣ S ∣ is an operation symbol in the signature S, then ∥ S ∥ f is the arity of f.

We need to augment the ordinary Signature type so that it supports algebras over setoid domains. To do so, we define an operator ⟨_⟩ which translates an ordinary signature into a setoid signature, that is, a signature over a setoid domain. But first we must resolve a techinical issue involving dependent types that we now describe.

Suppose we are given two operations f and g and we have a tuple of arguments for f, say, u : ∥ S ∥ f → A, and a tuple of arguments for g, say, v : ∥ S ∥ g → A. If we know that f is identically equal to g—that is, f ≡ g (intensionally)—then we should be able to check whether u and v are pointwise equal. The problem here is that u and v ostensibly inhabit different types. To compare u and v we must convince Agda that, from f ≡ g we can deduce that u and v are actually of the same type. The type EqArgs (defined below, and adapted from Andreas Abel's development [ref needed]) resolves this technical issue nicely.

```
EqArgs : {𝑆 : Signature 𝓞 𝓥}{ξ : Setoid α ρᵃ}
 →      ∀ {f g} → f ≡ g → (∥ 𝑆 ∥ f → Carrier ξ) → (∥ 𝑆 ∥ g → Carrier ξ) → Type (𝓥 ⊔ ρᵃ)

EqArgs {ξ = ξ} ≡.refl u v = ∀ i → u i ≈ v i where open Setoid ξ using ( _≈_ )
```

Now we are ready to define the ⟨_⟩ operator, which translates an ordinary signature into a setoid signature.

```
module _ where
 open Setoid using ( _≈_ )
 open IsEquivalence using ( refl ; sym ; trans )

 ⟨_⟩ : Signature 𝓞 𝓥 → Setoid α ρᵃ → Setoid _ _

 Carrier (⟨ 𝑆 ⟩ ξ)              = Σ[ f ∈ ∣ 𝑆 ∣ ] ((∥ 𝑆 ∥ f) → ξ .Carrier)

 _≈_ (⟨ 𝑆 ⟩ ξ) (f , u) (g , v) = Σ[ eqv ∈ f ≡ g ] EqArgs{ξ = ξ} eqv u v

 refl  (isEquivalence (⟨ 𝑆 ⟩ ξ))                          = ≡.refl , λ i → Setoid.refl  ξ
 sym   (isEquivalence (⟨ 𝑆 ⟩ ξ)) (≡.refl , g)             = ≡.refl , λ i → Setoid.sym   ξ (g i)
 trans (isEquivalence (⟨ 𝑆 ⟩ ξ)) (≡.refl , g)(≡.refl , h) = ≡.refl , λ i → Setoid.trans ξ (g i) (h i)
```

3.2 Algebra type

Informally, an *algebraic structure in the signature* $S = (\mathsf{F}, \rho)$ (or S-*algebra*) is typically denoted by $\mathbf{A} = (\mathsf{A}, \mathsf{F}^{A})$ and consists of

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

We represent an algebra in Agda using a record type with two fields:

- Domain is a setoid denoting the underlying *universe* of the algebra (informally, the set of elements of the algebra);
- Interp represents the *interpretation* in the algebra of each operation symbol of the given signature. The record type Func from the Agda Standard Library provides what we need for an operation on the domain setoid.

Let us present the definition of the Algebra type and then discuss the definition of the Func type that provides the interpretation of each operation symbol.

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

The Interp field actually has type Func (⟨ 𝑆 ⟩ Domain) Domain (recall we renamed Func as the infix long-arrow symbol). The Func type is from the standard library and is defined as a record type with two fields. In the present instance, the fields are

- a function f : Carrier (⟨ 𝑆 ⟩ Domain) → Carrier Domain

- a proof cong : f Preserves _≈₁_ ⟶ _≈₂_ that f preserves the underlying setoid equalities.

Thus Interp gives, for each operation symbol in the signature S, a setoid function f—namely, a function where the domain is a power of Domain and the codomain is Domain—along with a proof that all operations so interpreted respect the underlying setoid equality on Domain.

Next we define some syntactic sugar that will make our Agda code easier to read and comprehend. If **A** is an algebra, then

- 𝔻[**A**] will denote the setoid Domain **A**, and
- 𝕌[**A**] will be the underlying carrier or "universe" of the algebra **A**.
- f ̂ **A** will denote the interpretation in the algebra **A** of the operation symbol f,

```
open Algebra

𝔻[_] : Algebra α ρᵃ → Setoid α ρᵃ
𝔻[ 𝑨 ] = Domain 𝑨

𝕌[_] : Algebra α ρᵃ → Type α
𝕌[ 𝑨 ] = Carrier (Domain 𝑨)

_̂_ : (f : ∣ 𝑆 ∣)(𝑨 : Algebra α ρᵃ) → (∥ 𝑆 ∥ f → 𝕌[ 𝑨 ]) → 𝕌[ 𝑨 ]

f ̂ 𝑨 = λ a → (Interp 𝑨) ⟨$⟩ (f , a)
```

3.3 Universe lifting of algebra types

A technical aspect of dealing with the noncumulativity of the hierarchy of type levels in Agda...

```
module _ (𝑨 : Algebra α ρᵃ) where
  open Setoid 𝔻[ 𝑨 ] using ( _≈_ ; refl ; sym ; trans )
  open Level

  Lift-Algˡ : (ℓ : Level) → Algebra (α ⊔ ℓ) ρᵃ
  Domain (Lift-Algˡ ℓ) =
    record { Carrier = Lift ℓ 𝕌[ 𝑨 ]
           ; _≈_ = λ x y → lower x ≈ lower y
           ; isEquivalence = record { refl = refl ; sym = sym ; trans = trans }}
  Interp (Lift-Algˡ ℓ) ⟨$⟩ (f , la) = lift ((f ̂ 𝑨) (lower ∘ la))
  cong (Interp (Lift-Algˡ ℓ)) (≡.refl , la=lb) = cong (Interp 𝑨) ((≡.refl , la=lb))

  Lift-Algʳ : (ℓ : Level) → Algebra α (ρᵃ ⊔ ℓ)
  Domain (Lift-Algʳ ℓ) =
    record { Carrier = 𝕌[ 𝑨 ]
           ; _≈_ = λ x y → Lift ℓ (x ≈ y)
           ; isEquivalence = record { refl = lift refl
                                    ; sym = λ x → lift (sym (lower x))
                                    ; trans = λ x y → lift (trans (lower x) (lower y)) }}
  Interp (Lift-Algʳ ℓ ) ⟨$⟩ (f , la) = (f ̂ 𝑨) la
  cong (Interp (Lift-Algʳ ℓ))(≡.refl , la≡lb) = lift(cong(Interp 𝑨)(≡.refl , λ i → lower (la≡lb i)))

Lift-Alg : (𝑨 : Algebra α ρᵃ)(ℓ₀ ℓ₁ : Level) → Algebra (α ⊔ ℓ₀) (ρᵃ ⊔ ℓ₁)
Lift-Alg 𝑨 ℓ₀ ℓ₁ = Lift-Algʳ (Lift-Algˡ 𝑨 ℓ₀) ℓ₁
```

3.4 Product Algebras

(cf. the Algebras.Func.Products module of the agda-algebras library.)

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

 ⨅ : (𝒜 : I → Algebra α ρᵃ) → Algebra (α ⊔ ι) (ρᵃ ⊔ ι)
 Domain (⨅ 𝒜) =
  record { Carrier = ∀ i → 𝕌[ 𝒜 i ]
         ; _≈_ = λ a b → ∀ i → (Setoid._≈_ 𝔻[ 𝒜 i ]) (a i)(b i)
         ; isEquivalence =
           record { refl  = λ i →      IsEquivalence.refl  (isEquivalence 𝔻[ 𝒜 i ])
                  ; sym   = λ x i →    IsEquivalence.sym   (isEquivalence 𝔻[ 𝒜 i ])(x i)
                  ; trans = λ x y i → IsEquivalence.trans (isEquivalence 𝔻[ 𝒜 i ])(x i)(y i) }}
 Interp (⨅ 𝒜) ⟨$⟩ (f , a) = λ i → (f ̂ (𝒜 i)) (flip a i)
 cong (Interp (⨅ 𝒜)) (≡.refl , f=g ) = λ i → cong (Interp (𝒜 i)) (≡.refl , flip f=g i )
```

4 Homomorphisms

4.1 Basic definitions

Here are some useful definitions and theorems extracted from the Homomorphisms.Func.Basic module of the agda-algebras library.

```
module _ (𝐀 : Algebra α ρᵃ)(𝐁 : Algebra β ρᵇ) where
 private ov = 𝓞 ⊔ 𝓥 ; a = α ⊔ ρᵃ ; b = β ⊔ ρᵇ ; c = 𝓞 ⊔ 𝓥 ⊔ α ⊔ ρᵃ ⊔ β ⊔ ρᵇ

 compatible-map-op : (𝔻[ 𝐀 ] ⟶ 𝔻[ 𝐁 ]) → ∣ 𝑆 ∣ → Type (𝓥 ⊔ α ⊔ ρᵇ)
 compatible-map-op h f = ∀ {a} → (h ⟨$⟩ ((f ̂ 𝐀) a)) ≈₂ ((f ̂ 𝐁) (λ x → (h ⟨$⟩ (a x))))
  where open Setoid 𝔻[ 𝐁 ] using () renaming ( _≈_ to _≈₂_ )

 compatible-map : (𝔻[ 𝐀 ] ⟶ 𝔻[ 𝐁 ]) → Type (ov ⊔ α ⊔ ρᵇ)
 compatible-map h = ∀ {f} → compatible-map-op h f

 -- The property of being a homomorphism.
 record IsHom (h : 𝔻[ 𝐀 ] ⟶ 𝔻[ 𝐁 ]) : Type (ov ⊔ α ⊔ ρᵇ) where
  constructor mkhom
  field compatible : compatible-map h

 -- The type of homomorphisms.
 hom : Type c
 hom = Σ (𝔻[ 𝐀 ] ⟶ 𝔻[ 𝐁 ]) IsHom
```

4.2 Monomorphisms and epimorphisms

```
 record IsMon (h : 𝔻[ 𝐀 ] ⟶ 𝔻[ 𝐁 ]) : Type (ov ⊔ a ⊔ ρᵇ) where
  field isHom : IsHom h
        isInjective : IsInjective h

  HomReduct : hom
  HomReduct = h , isHom

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

```
  record IsEpi (h : 𝔻[ 𝐀 ] ⟶ 𝔻[ 𝐁 ]) : Type (ov ⊔ α ⊔ b) where
    field isHom : IsHom h
          isSurjective : IsSurjective h

    HomReduct : hom
    HomReduct = h , isHom

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

open IsHom ; open IsMon ; open IsEpi

module _ (𝐀 : Algebra α ρᵃ)(𝐁 : Algebra β ρᵇ) where

  mon→intohom : mon 𝐀 𝐁 → Σ[ h ∈ hom 𝐀 𝐁 ] IsInjective ∣ h ∣
  mon→intohom (hh , hhM) = (hh , isHom hhM) , isInjective hhM

  epi→ontohom : epi 𝐀 𝐁 → Σ[ h ∈ hom 𝐀 𝐁 ] IsSurjective ∣ h ∣
  epi→ontohom (hh , hhE) = (hh , isHom hhE) , isSurjective hhE
```

4.3 Basic properties of homomorphisms

Here are some definitions and theorems extracted from the Homomorphisms.Func.Properties module of the agda-algebras library.

4.3.1 Composition of homomorphisms

The composition of homomorphisms is again a homomorphism. Similarly, the composition of epimorphisms is again an epimorphism.

```
module _ {𝐀 : Algebra α ρᵃ} {𝐁 : Algebra β ρᵇ} {𝐂 : Algebra γ ρᶜ}
         {g : 𝔻[ 𝐀 ] ⟶ 𝔻[ 𝐁 ]}{h : 𝔻[ 𝐁 ] ⟶ 𝔻[ 𝐂 ]} where

  open Setoid 𝔻[ 𝐂 ] using ( trans )

  ∘-is-hom : IsHom 𝐀 𝐁 g → IsHom 𝐁 𝐂 h → IsHom 𝐀 𝐂 (h ⟨∘⟩ g)
  ∘-is-hom ghom hhom = mkhom c
    where
    c : compatible-map 𝐀 𝐂 (h ⟨∘⟩ g)
    c = trans (cong h (compatible ghom)) (compatible hhom)

  ∘-is-epi : IsEpi 𝐀 𝐁 g → IsEpi 𝐁 𝐂 h → IsEpi 𝐀 𝐂 (h ⟨∘⟩ g)
  ∘-is-epi gE hE = record { isHom = ∘-is-hom (isHom gE) (isHom hE)
                          ; isSurjective = ∘-IsSurjective g h (isSurjective gE) (isSurjective hE) }


module _ {𝐀 : Algebra α ρᵃ} {𝐁 : Algebra β ρᵇ} {𝐂 : Algebra γ ρᶜ} where

  ∘-hom : hom 𝐀 𝐁 → hom 𝐁 𝐂 → hom 𝐀 𝐂
  ∘-hom (h , hhom) (g , ghom) = (g ⟨∘⟩ h) , ∘-is-hom hhom ghom

  ∘-epi : epi 𝐀 𝐁 → epi 𝐁 𝐂     → epi 𝐀 𝐂
  ∘-epi (h , hepi) (g , gepi) = (g ⟨∘⟩ h) , ∘-is-epi hepi gepi
```

4.3.2 Universe lifting of homomorphisms

First we define the identity homomorphism for setoid algebras and then we prove that the operations of lifting and lowering of a setoid algebra are homomorphisms.

4.4 Factorization of homomorphisms

If g : hom 𝑨 𝑩, h : hom 𝑨 𝑪, h is surjective, and ker h ⊆ ker g, then there exists φ : hom 𝑪 𝑩 such that g = φ ∘ h (cf. the Homomorphisms.Func.Factor module of the agda-algebras library).

```
module _ {𝑨 : Algebra α ρᵃ}(𝑩 : Algebra β ρᵇ){𝑪 : Algebra γ ρᶜ}
         (gh : hom 𝑨 𝑩)(hh : hom 𝑨 𝑪) where
 open Setoid 𝔻[ 𝑩 ] using () renaming ( _≈_ to _≈₂_ ; sym to sym₂ )
 open Setoid 𝔻[ 𝑪 ] using ( trans ) renaming ( _≈_ to _≈₃_ ; sym to sym₃ )
 open SetoidReasoning 𝔻[ 𝑩 ]
 private gfunc = ∣ gh ∣ ; g = _⟨$⟩_ gfunc ; hfunc = ∣ hh ∣ ; h = _⟨$⟩_ hfunc

 HomFactor : kernel _≈₃_ h ⊆ kernel _≈₂_ g → IsSurjective hfunc
  →          Σ[ φ ∈ hom 𝑪 𝑩 ] ∀ a → (g a) ≈₂ ∣ φ ∣ ⟨$⟩ (h a)

 HomFactor Khg hE = (φmap , φhom) , gφh
  where
  kerpres : ∀ a₀ a₁ → h a₀ ≈₃ h a₁ → g a₀ ≈₂ g a₁
  kerpres a₀ a₁ hyp = Khg hyp

  h⁻¹ : 𝕌[ 𝑪 ] → 𝕌[ 𝑨 ]
  h⁻¹ = SurjInv hfunc hE

  η : ∀ {c} → h (h⁻¹ c) ≈₃ c
  η = InvIsInverseʳ hE

  ζ : ∀{x y} → x ≈₃ y → h (h⁻¹ x) ≈₃ h (h⁻¹ y)
  ζ xy = trans η (trans xy (sym₃ η))

  φmap : 𝔻[ 𝑪 ] ⟶ 𝔻[ 𝑩 ]
  _⟨$⟩_ φmap = g ∘ h⁻¹
  cong φmap = Khg ∘ ζ
  open _⟶_ φmap using () renaming (cong to φcong)

  gφh : (a : 𝕌[ 𝑨 ]) → g a ≈₂ φmap ⟨$⟩ (h a)
  gφh a = Khg (sym₃ η)

  φcomp : compatible-map 𝑪 𝑩 φmap
  φcomp {f}{c} =
   begin
    φmap ⟨$⟩ ((f ̂ 𝑪) c)              ≈˘⟨ φcong (cong (Interp 𝑪) (≡.refl , λ _ → η)) ⟩
    g (h⁻¹ ((f ̂ 𝑪)(h ∘ h⁻¹ ∘ c)))   ≈˘⟨ φcong (compatible ∥ hh ∥)                    ⟩
    g (h⁻¹ (h ((f ̂ 𝑨)(h⁻¹ ∘ c))))  ≈˘⟨ gφh ((f ̂ 𝑨)(h⁻¹ ∘ c))                        ⟩
    g ((f ̂ 𝑨)(h⁻¹ ∘ c))              ≈⟨  compatible ∥ gh ∥                            ⟩
    (f ̂ 𝑩)(g ∘ (h⁻¹ ∘ c))            ∎

  φhom : IsHom 𝑪 𝑩 φmap
  compatible φhom = φcomp
```

4.5 Isomorphisms

(cf. the Homomorphisms.Func.Isomorphisms of the agda-algebras library.)

Two structures are *isomorphic* provided there are homomorphisms going back and forth between them which compose to the identity map.

```
module _ (𝐀 : Algebra α ρᵃ) (𝐁 : Algebra β ρᵇ) where
 open Setoid 𝔻[ 𝐀 ] using ( _≈_ ; sym ; trans )
 open Setoid 𝔻[ 𝐁 ] using () renaming ( _≈_ to _≈ᵇ_ ; sym to symᵇ ; trans to transᵇ)

 record _≅_ : Type (𝓞 ⊔ 𝓥 ⊔ α ⊔ β ⊔ ρᵃ ⊔ ρᵇ ) where
  constructor mkiso
  field
   to : hom 𝐀 𝐁
   from : hom 𝐁 𝐀
   to∼from : ∀ b → (∣ to ∣ ⟨$⟩ (∣ from ∣ ⟨$⟩ b)) ≈ᵇ b
   from∼to : ∀ a → (∣ from ∣ ⟨$⟩ (∣ to ∣ ⟨$⟩ a)) ≈ a

  toIsSurjective : IsSurjective ∣ to ∣
  toIsSurjective {y} = eq (∣ from ∣ ⟨$⟩ y) (symᵇ (to∼from y))

  toIsInjective : IsInjective ∣ to ∣
  toIsInjective {x} {y} xy = Goal
   where
   ξ : ∣ from ∣ ⟨$⟩ (∣ to ∣ ⟨$⟩ x) ≈ ∣ from ∣ ⟨$⟩ (∣ to ∣ ⟨$⟩ y)
   ξ = cong ∣ from ∣ xy
   Goal : x ≈ y
   Goal = trans (sym (from∼to x)) (trans ξ (from∼to y))

  fromIsSurjective : IsSurjective ∣ from ∣
  fromIsSurjective {y} = eq (∣ to ∣ ⟨$⟩ y) (sym (from∼to y))

  fromIsInjective : IsInjective ∣ from ∣
  fromIsInjective {x} {y} xy = Goal
   where
   ξ : ∣ to ∣ ⟨$⟩ (∣ from ∣ ⟨$⟩ x) ≈ᵇ ∣ to ∣ ⟨$⟩ (∣ from ∣ ⟨$⟩ y)
   ξ = cong ∣ to ∣ xy
   Goal : x ≈ᵇ y
   Goal = transᵇ (symᵇ (to∼from x)) (transᵇ ξ (to∼from y))

open _≅_
```

4.5.1 Properties of isomorphisms

```
≅-refl : Reflexive (_≅_ {α}{ρᵃ})
≅-refl {α}{ρᵃ}{𝐀} = mkiso 𝒾𝒹 𝒾𝒹 (λ b → refl) λ a → refl
 where open Setoid 𝔻[ 𝐀 ] using ( refl )

≅-sym : Sym (_≅_{β}{ρᵇ}) (_≅_{α}{ρᵃ})
≅-sym φ = mkiso (from φ) (to φ) (from∼to φ) (to∼from φ)

≅-trans : Trans (_≅_ {α}{ρᵃ})(_≅_{β}{ρᵇ})(_≅_{α}{ρᵃ}{γ}{ρᶜ})
≅-trans {ρᶜ = ρᶜ}{𝐀}{𝐁}{𝐂} ab bc = mkiso f g τ ν
 where
```

```
open Setoid 𝔻[ A ] using ( _≈_ ; trans )
open Setoid 𝔻[ C ] using () renaming ( _≈_ to _≈ᶜ_ ; trans to transᶜ )
f : hom A C
f = ∘-hom (to ab) (to bc)
g : hom C A
g = ∘-hom (from bc) (from ab)
τ : ∀ b → (| f | ⟨$⟩ (| g | ⟨$⟩ b)) ≈ᶜ b
τ b = transᶜ (cong | to bc | (to∼from ab (| from bc | ⟨$⟩ b))) (to∼from bc b)
ν : ∀ a → (| g | ⟨$⟩ (| f | ⟨$⟩ a)) ≈ a
ν a = trans (cong | from ab | (from∼to bc (| to ab | ⟨$⟩ a))) (from∼to ab a)
```

Fortunately, the lift operation preserves isomorphism (i.e., it's an *algebraic invariant*). As our focus is universal algebra, this is important and is what makes the lift operation a workable solution to the technical problems that arise from the noncumulativity of Agda's universe hierarchy.

4.6 Homomorphic Images

We begin with what for our purposes is the most useful way to represent the class of *homomorphic images* of an algebra in dependent type theory (cf. the Homomorphisms.Func.HomomorphicImages module of the agda-algebras library). (The first definition is merely a short-hand.)

```
ov : Level → Level
ov α = 𝓞 ⊔ 𝓥 ⊔ lsuc α

_IsHomImageOf_ : (B : Algebra β ρᵇ)(A : Algebra α ρᵃ) → Type (𝓞 ⊔ 𝓥 ⊔ α ⊔ β ⊔ ρᵃ ⊔ ρᵇ)
B IsHomImageOf A = Σ[ φ ∈ hom A B ] IsSurjective | φ |

HomImages : Algebra α ρᵃ → Type (α ⊔ ρᵃ ⊔ ov (β ⊔ ρᵇ))
HomImages {β = β}{ρᵇ = ρᵇ} A = Σ[ B ∈ Algebra β ρᵇ ] B IsHomImageOf A
```

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 **A** : Algebra α ρ, the type HomImages **A** denotes the class of algebras **B** : Algebra β ρ with a map φ : | **A** | → | **B** | such that φ is a surjective homomorphism.

5 Subalgebras

5.1 Basic definitions

```
_≤_ : Algebra α ρᵃ → Algebra β ρᵇ → Type (𝓞 ⊔ 𝓥 ⊔ α ⊔ ρᵃ ⊔ β ⊔ ρᵇ)
A ≤ B = Σ[ h ∈ hom A B ] IsInjective | h |
```

5.2 Basic properties

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

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

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

```
≤-trans : 𝐀 ≤ 𝐁 → 𝐁 ≤ 𝐂 → 𝐀 ≤ 𝐂
≤-trans ( f , finj ) ( g , ginj ) = (∘-hom f g ) , ∘-IsInjective ∣ f ∣ ∣ g ∣ finj ginj

≅-trans-≤ : 𝐀 ≅ 𝐁 → 𝐁 ≤ 𝐂 → 𝐀 ≤ 𝐂
≅-trans-≤ A≅B (h , hinj) = (∘-hom (to A≅B) h) , (∘-IsInjective ∣ to A≅B ∣ ∣ h ∣ (toIsInjective A≅B) hinj)
```

5.3 Products of subalgebras

```
module _ {ι : Level} {I : Type ι}{𝒜 : I → Algebra α ρᵃ}{ℬ : I → Algebra β ρᵇ} where

 ⨅-≤ : (∀ i → ℬ i ≤ 𝒜 i) → ⨅ ℬ ≤ ⨅ 𝒜
 ⨅-≤ B≤A = (hfunc , hhom) , hM
  where
  hi : ∀ i → hom (ℬ i) (𝒜 i)
  hi i = ∣ B≤A i ∣

  hfunc : 𝔻[ ⨅ ℬ ] ⟶ 𝔻[ ⨅ 𝒜 ]
  (hfunc ⟨$⟩ x) i = ∣ hi i ∣ ⟨$⟩ (x i)
  cong hfunc = λ xy i → cong ∣ hi i ∣ (xy i)

  hhom : IsHom (⨅ ℬ) (⨅ 𝒜) hfunc
  compatible hhom = λ i → compatible ∥ hi i ∥

  hM : IsInjective hfunc
  hM = λ xy i → ∥ B≤A i ∥ (xy i)
```

6 Terms

6.1 Basic definitions

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 ∩ ∣ S ∣ = ∅.

By a *word* in the language of S, we mean a nonempty, finite sequence of members of X ∪ ∣ S ∣. We denote the concatenation of such sequences by simple juxtaposition.

Let S_0 denote the set of nullary operation symbols of S. We define by induction on n the sets T_n of *words* over X ∪ ∣ S ∣ as follows (cf. [1, Def. 4.19]):

1. $T_0 := \mathsf{X} \cup \mathsf{S}_0$, and
2. $T_{n+1} := T_n \cup \mathcal{T}_n$.

where $\mathcal{T}_n$ is the collection of all f t such that f : ∣ S ∣ and t : ∥ S ∥ f → T_n. (Recall, ∥ S ∥ f is 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.

The definition of Term X is recursive, indicating that an inductive type could be used to represent the semantic notion of terms in type theory. Indeed, such a representation is given by the following inductive type.

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

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 (ℊ); hence the constructor names (ℊ for "generator" and node for "node").

Notation. As usual, the type X represents an arbitrary collection of variable symbols. Recall, ov χ is our shorthand notation for the universe level 𝓞 ⊔ 𝓥 ⊔ lsuc χ.

6.2 Equality of terms

We take a different approach here, using Setoids instead of quotient types. That is, we will define the collection of terms in a signature as a setoid with a particular equality-of-terms relation, which we must define. Ultimately we will use this to define the (absolutely free) term algebra as a Algebra whose carrier is the setoid of terms.

```
module _ {X : Type χ } where

 data _≐_ : Term X → Term X → Type (ov χ) where
  rfl : {x y : X} → x ≡ y → (ℊ x) ≐ (ℊ y)
  gnl : ∀ {f}{s t : ∥ 𝑆 ∥ f → Term X} → (∀ i → (s i) ≐ (t i)) → (node f s) ≐ (node f t)
```

It is easy to show that the equality-of-terms relation _≐_ is an equivalence relation, so we omit the formal proof. (See the Terms.Func.Basic module of the agda-algebras library for details.)

6.3 The term algebra

For a given signature S, if the type Term X is nonempty (equivalently, if X or ∣ S ∣ is nonempty), then we can define an algebraic structure, denoted by 𝐓 X and called the *term algebra in the signature S over* X. Terms are viewed as acting on other terms, so both the domain and basic operations of the algebra are the terms themselves.

- For each operation symbol f : ∣ S ∣, denote by f ̂ (𝐓 X) the operation on Term X that maps a tuple t : ∥ S ∥ f → ∣ 𝐓 X ∣ to the formal term f t.
- Define 𝐓 X to be the algebra with universe ∣ 𝐓 X ∣ := Term X and operations f ̂ (𝐓 X), one for each symbol f in ∣ S ∣.

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

```
TermSetoid : (X : Type χ) → Setoid (ov χ) (ov χ)
TermSetoid X = record { Carrier = Term X ; _≈_ = _≐_ ; isEquivalence = ≐-isEquiv }

𝐓 : (X : Type χ) → Algebra (ov χ) (ov χ)
Algebra.Domain (𝐓 X) = TermSetoid X
Algebra.Interp (𝐓 X) ⟨$⟩ (f , ts) = node f ts
cong (Algebra.Interp (𝐓 X)) (≡.refl , ss≐ts) = gnl ss≐ts
```

6.4 Interpretation of terms

The approach to terms and their interpretation in this module was inspired by Andreas Abel's formal proof of Birkhoff's completeness theorem.[1]

[1] See `http://www.cse.chalmers.se/~abela/agda/MultiSortedAlgebra.pdf`.

A substitution from X to Y associates a term in X with each variable in Y. The definition of Sub given here is essentially the same as the one given by Andreas Abel, as is the recursive definition of the syntax t [σ] , which denotes a term t applied to a substitution σ.

```
Sub : Type χ → Type χ → Type (ov χ)
Sub X Y = (y : Y) → Term X

_[_] : {X Y : Type χ}(t : Term Y) (σ : Sub X Y) → Term X
(ℊ x) [ σ ] = σ x
(node f ts) [ σ ] = node f (λ i → ts i [ σ ])
```

An environment for an algebra **A** in a context X is a map that assigns to each variable x : X an element in the domain of **A**, packaged together with an equality of environments, which is simply pointwise equality (relatively to the setoid equality of the underlying domain of **A**).

```
module Environment (A : Algebra α ℓ) where
 open Setoid 𝔻[ A ] using ( _≈_ ; refl ; sym ; trans )
 Env : Type χ → Setoid _ _
 Env X = record { Carrier = X → 𝕌[ A ]
                ; _≈_ = λ ρ ρ' → (x : X) → ρ x ≈ ρ' x
                ; isEquivalence = record { refl  = λ _ → refl
                                         ; sym   = λ h x → sym (h x)
                                         ; trans = λ g h x → trans (g x)(h x) }}
```

Next we define *evaluation of a term* in an environment ρ, interpreted in the algebra **A**.

```
⟦_⟧ : {X : Type χ}(t : Term X) → (Env X) ⟶ 𝔻[ A ]
⟦ ℊ x ⟧ ⟨$⟩ ρ = ρ x
⟦ node f args ⟧ ⟨$⟩ ρ = (Interp A) ⟨$⟩ (f , λ i → ⟦ args i ⟧ ⟨$⟩ ρ)
cong ⟦ ℊ x ⟧ u≈v = u≈v x
cong ⟦ node f args ⟧ x≈y = cong (Interp A)(≡.refl , λ i → cong ⟦ args i ⟧ x≈y )
```

An equality between two terms holds in a model if the two terms are equal under all valuations of their free variables (cf. Andreas Abel's formal proof of Birkhoff's completeness theorem).

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

≐→Equal : {X : Type χ}(s t : Term X) → s ≐ t → Equal s t
≐→Equal .(ℊ _) .(ℊ _) (rfl ≡.refl) = λ _ → refl
≐→Equal (node _ s)(node _ t)(gnl x) =
 λ ρ → cong (Interp A)(≡.refl , λ i → ≐→Equal(s i)(t i)(x i)ρ )
```

The proof that Equal is an equivalence relation is trivial, so we omit it. (See the Varieties.Func.SoundAndComplete module of the agda-algebras library for details.)

Evaluation of a substitution gives an environment (cf. [Andreas Abel's formal proof of Birkhoff's completeness theorem](http://www.cse.chalmers.se/ abela/agda/MultiSortedAlgebra.pdf))

```
⟦_⟧s : {X Y : Type χ} → Sub X Y → Carrier(Env X) → Carrier (Env Y)
⟦ σ ⟧s ρ x = ⟦ σ x ⟧ ⟨$⟩ ρ
```

Next we prove that ⟦ t [σ] ⟧ ρ ≃ ⟦ t ⟧ ⟦ σ ⟧ ρ (cf. Andreas Abel's formal proof of Birkhoff's completeness theorem).

```
substitution : {X Y : Type χ} → (t : Term Y) (σ : Sub X Y) (ρ : Carrier( Env X ) )
 → ⟦ t [ σ ] ⟧ ⟨$⟩ ρ ≈ ⟦ t ⟧ ⟨$⟩ (⟦ σ ⟧s ρ)

substitution (ℊ x) σ ρ = refl
substitution (node f ts) σ ρ = cong (Interp 𝐀)(≡.refl , λ i → substitution (ts i) σ ρ)
```

6.5 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.

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

 comm-hom-term : (t : Term X) (a : X → 𝕌[ 𝐀 ]) → h (⟦ t ⟧ ⟨$⟩ a) ≈ ⟦ t ⟧ᴮ ⟨$⟩ (h ∘ a)
 comm-hom-term (ℊ x) a = refl
 comm-hom-term (node f t) a = goal
  where
  goal : h (⟦ node f t ⟧ ⟨$⟩ a) ≈ (⟦ node f t ⟧ᴮ ⟨$⟩ (h ∘ a))
  goal = begin
       h (⟦ node f t ⟧ ⟨$⟩ a)         ≈⟨ compatible ∥ hh ∥ ⟩
       (f ̂ 𝐁)(λ i → h (⟦ t i ⟧ ⟨$⟩ a))    ≈⟨ cong(Interp 𝐁)(≡.refl , λ i → comm-hom-term (t i) a) ⟩
       (f ̂ 𝐁)(λ i → ⟦ t i ⟧ᴮ ⟨$⟩ (h ∘ a)) ≈⟨ refl                                ⟩
       (⟦ node f t ⟧ᴮ ⟨$⟩ (h ∘ a)) ∎
```

7 Model Theory and Equational Logic

(cf. the Varieties.Func.SoundAndComplete module of the agda-algebras library)

7.1 Basic definitions

Let S be a signature. By an *identity* or *equation* in S we mean an ordered pair of terms in a given context. For instance, if the context happens to be the type X : Type χ, then an equation will be a pair of inhabitants of the domain of term algebra **T** X.

We define an equation in Agda using the following record type with fields denoting the left-hand and right-hand sides of the equation, along with an equation "context" representing the underlying collection of variable symbols (cf. Andreas Abel's formal proof of Birkhoff's completeness theorem).

```
record Eq : Type (ov χ) where
 constructor _≈̇_
```

```
  field {cxt} : Type χ
        lhs   : Term cxt
        rhs   : Term cxt

open Eq public
```

We now define a type representing the notion of an equation p ≈̇ q holding (when p and q are interpreted) in algebra **A**.

If **A** is an S-algebra we say that **A** *satisfies* p ≈ q provided for all environments ρ : X → | **A** | (assigning values in the domain of **A** to variable symbols in X) we have ⟦ p ⟧ ⟨$⟩ ρ ≈ ⟦ q ⟧ ⟨$⟩ ρ. In this situation, we write **A** ⊨ (p ≈̇ q) and say that **A** *models* the identity p ≈ q.

If 𝒦 is a class of algebras, all of the same signature, we write 𝒦 |⊨ (p ≈̇ q) if, for every **A** ∈ 𝒦, we have **A** ⊨ (p ≈̇ q).

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 𝒦 ⊨ p ≈ q, we have settled on 𝒦 |⊨ (p ≈̇ q) to denote this relation. To reiterate, if 𝒦 is a class of S-algebras, we write 𝒦 |⊨ (p ≈̇ q) provided every **A** ∈ 𝒦 satisfies **A** ⊨ (p ≈̇ q).

```
_⊨_ : (A : Algebra α ρᵃ)(term-identity : Eq{χ}) → Type _
A ⊨ (p ≈̇ q) = Equal p q where open Environment A

_|⊨_ : Pred (Algebra α ρᵃ) ℓ → Eq{χ} → Type (ℓ ⊔ χ ⊔ ov(α ⊔ ρᵃ))
𝒦 |⊨ equ = ∀ A → 𝒦 A → A ⊨ equ
```

We denote by **A** ⊧ ℰ the assertion that the algebra **A** models every equation in a collection ℰ of equations.

```
_⊧_ : (A : Algebra α ρᵃ) → {ι : Level}{I : Type ι} → (I → Eq{χ}) → Type _
A ⊧ ℰ = ∀ i → Equal (lhs (ℰ i))(rhs (ℰ i)) where open Environment A
```

7.2 Equational theories and models

If 𝒦 denotes a class of structures, then Th 𝒦 represents the set of identities modeled by the members of 𝒦.

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

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

7.3 The entailment relation

Based on Andreas Abel's Agda formalization of Birkhoff's completeness theorem.

```
module _ {χ ι : Level} where

 data _⊢_▹_≈_ {I : Type ι}(ℰ : I → Eq) : (X : Type χ)(p q : Term X) → Type (ι ⊔ ov χ) where
  hyp : ∀ i → let p ≈̇ q = ℰ i in ℰ ⊢ _ ▹ p ≈ q
  app : ∀ {ps qs} → (∀ i → ℰ ⊢ Γ ▹ ps i ≈ qs i) → ℰ ⊢ Γ ▹ (node f ps) ≈ (node f qs)
  sub : ∀ {p q} → ℰ ⊢ Δ ▹ p ≈ q → ∀ (σ : Sub Γ Δ) → ℰ ⊢ Γ ▹ (p [ σ ]) ≈ (q [ σ ])
```

```
    ⊢refl  : ∀ {p}             → 𝓔 ⊢ Γ ▹ p ≈ p
    ⊢sym   : ∀ {p q : Term Γ}  → 𝓔 ⊢ Γ ▹ p ≈ q → 𝓔 ⊢ Γ ▹ q ≈ p
    ⊢trans : ∀ {p q r : Term Γ} → 𝓔 ⊢ Γ ▹ p ≈ q → 𝓔 ⊢ Γ ▹ q ≈ r → 𝓔 ⊢ Γ ▹ p ≈ r

  ⊢▹≈IsEquiv : {X : Type χ}{I : Type ι}{𝓔 : I → Eq} → IsEquivalence (𝓔 ⊢ X ▹_≈_)
  ⊢▹≈IsEquiv = record { refl = ⊢refl ; sym = ⊢sym ; trans = ⊢trans }
```

7.4 Soundness

In any model **A** of the equations 𝓔 derived equality is actual equality (cf. Andreas Abel's Agda formalization of Birkhoff's completeness theorem).

```
module Soundness {χ α ι : Level}{I : Type ι} (𝓔 : I → Eq{χ})
                 (𝐀 : Algebra α ρᵃ)    -- We assume an algebra 𝐀
                 (V : 𝐀 ⊨ 𝓔)           -- that models all equations in 𝓔.
                 where

  open SetoidReasoning 𝔻[ 𝐀 ]
  open Environment 𝐀
  open IsEquivalence using ( refl ; sym ; trans )

  sound : ∀ {p q} → 𝓔 ⊢ Γ ▹ p ≈ q → 𝐀 ⊨ (p ≈̇ q)
  sound (hyp i) = V i
  sound (app es) ρ = cong (Interp 𝐀) (≡.refl , λ i → sound (es i) ρ)
  sound (sub {p = p}{q} Epq σ) ρ =
    begin
      ⟦ p [ σ ] ⟧ ⟨$⟩   ρ ≈⟨  substitution p σ ρ    ⟩
      ⟦ p ⟧ ⟨$⟩ ⟦ σ ⟧s ρ ≈⟨  sound Epq (⟦ σ ⟧s ρ) ⟩
      ⟦ q ⟧ ⟨$⟩ ⟦ σ ⟧s ρ ≈˘⟨ substitution q σ ρ    ⟩
      ⟦ q [ σ ] ⟧ ⟨$⟩   ρ ∎
  sound ( ⊢refl  {p = p}                  ) = refl  EqualIsEquiv {x = p}
  sound ( ⊢sym   {p = p}{q}      Epq      ) = sym   EqualIsEquiv {x = p}{q}    (sound Epq)
  sound ( ⊢trans {p = p}{q}{r} Epq Eqr ) = trans EqualIsEquiv {i = p}{q}{r} (sound Epq)(sound Eqr)
```

7.5 The Closure Operators H, S, P and V

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

- H $\mathcal{K}$ = algebras isomorphic to a homomorphic image of a member 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 $\mathcal{K}$ ⊆ $\mathcal{K}$. Similarly, $\mathcal{K}$ is *closed under the taking of subalgebras* (resp., *arbitrary products*) provided S $\mathcal{K}$ ⊆ $\mathcal{K}$ (resp., P $\mathcal{K}$ ⊆ $\mathcal{K}$). The operators H, S, and P can be composed with one another repeatedly, forming yet more closure operators.

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.

A *variety* is a class of S-algebras that is closed under the taking of homomorphic images, subalgebras, and arbitrary products. To represent varieties we define types for the closure operators H, S, and P that are composable. Separately, we define a type V which represents closure under all three operators, H, S, and P. Thus, if $\mathcal{K}$ is a class of S-algebras', then V 𝒦 := H (S (P 𝒦)), and $\mathcal{K}$ is a variety iff V 𝒦 ⊆ 𝒦.

We now define the 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—the type S to represent classes of algebras that closed under the taking of subalgebras, and the type P to represent classes of algebras closed under the taking of arbitrary products.

```
module _ {α ρᵃ β ρᵇ : Level} where
 private a = α ⊔ ρᵃ ; b = β ⊔ ρᵇ

 H : ∀ ℓ → Pred(Algebra α ρᵃ) (a ⊔ ov ℓ) → Pred(Algebra β ρᵇ) (b ⊔ ov(a ⊔ ℓ))
 H _ 𝒦 𝑩 = Σ[ 𝑨 ∈ Algebra α ρᵃ ] 𝑨 ∈ 𝒦 × 𝑩 IsHomImageOf 𝑨

 S : ∀ ℓ → Pred(Algebra α ρᵃ) (a ⊔ ov ℓ) → Pred(Algebra β ρᵇ) (b ⊔ ov(a ⊔ ℓ))
 S _ 𝒦 𝑩 = Σ[ 𝑨 ∈ Algebra α ρᵃ ] 𝑨 ∈ 𝒦 × 𝑩 ≤ 𝑨

 P : ∀ ℓ ι → Pred(Algebra α ρᵃ) (a ⊔ ov ℓ) → Pred(Algebra β ρᵇ) (b ⊔ ov(a ⊔ ℓ ⊔ ι))
 P _ ι 𝒦 𝑩 = Σ[ I ∈ Type ι ] (Σ[ 𝒜 ∈ (I → Algebra α ρᵃ) ] (∀ i → 𝒜 i ∈ 𝒦) × (𝑩 ≅ ⨅ 𝒜))

module _ {α ρᵃ β ρᵇ γ ρᶜ δ ρᵈ : Level} where
 private a = α ⊔ ρᵃ ; b = β ⊔ ρᵇ ; c = γ ⊔ ρᶜ ; d = δ ⊔ ρᵈ

 V : ∀ ℓ ι → Pred(Algebra α ρᵃ) (a ⊔ ov ℓ) → Pred(Algebra δ ρᵈ) (d ⊔ ov(a ⊔ b ⊔ c ⊔ ℓ ⊔ ι))
 V ℓ ι 𝒦 = H{γ}{ρᶜ}{δ}{ρᵈ} (a ⊔ b ⊔ ℓ ⊔ ι) (S{β}{ρᵇ} (a ⊔ ℓ ⊔ ι) (P ℓ ι 𝒦))
```

7.5.1 Idempotence of S

S is a closure operator. The facts that S is monotone and expansive won't be needed, so we omit the proof of these facts. However, we will make use of idempotence of S, so we prove that property as follows.

```
S-idem : {𝒦 : Pred (Algebra α ρᵃ)(α ⊔ ρᵃ ⊔ ov ℓ)}
 → S{β = γ}{ρᶜ} (α ⊔ ρᵃ ⊔ ℓ) (S{β = β}{ρᵇ} ℓ 𝒦) ⊆ S{β = γ}{ρᶜ} ℓ 𝒦

S-idem (𝑨 , (𝑩 , sB , A≤B) , x≤A) = 𝑩 , (sB , ≤-trans x≤A A≤B)
```

7.5.2 Algebraic invariance of ⊧

The binary relation ⊧ would be practically useless if it were not an *algebraic invariant* (i.e., invariant under isomorphism). Let us now verify that the models relation we defined above has this essential property.

```
module _ {X : Type χ}{𝑨 : Algebra α ρᵃ}(𝑩 : Algebra β ρᵇ)(p q : Term X) where

 ⊧-I-invar : 𝑨 ⊧ (p ≈̇ q) → 𝑨 ≅ 𝑩 → 𝑩 ⊧ (p ≈̇ q)
 ⊧-I-invar Apq (mkiso fh gh f∼g g∼f) ρ =
  begin
   ⟦ p ⟧₂ ⟨$⟩ ρ             ≈˘⟨ cong ⟦ p ⟧₂ (λ x → f∼g (ρ x)) ⟩
   ⟦ p ⟧₂ ⟨$⟩ (f ∘ (g ∘ ρ)) ≈˘⟨ comm-hom-term fh p (g ∘ ρ)   ⟩
   f (⟦ p ⟧₁ ⟨$⟩ (g ∘ ρ))   ≈⟨  cong ∣ fh ∣ (Apq (g ∘ ρ))      ⟩
   f (⟦ q ⟧₁ ⟨$⟩ (g ∘ ρ))   ≈⟨  comm-hom-term fh q (g ∘ ρ)    ⟩
   ⟦ q ⟧₂ ⟨$⟩ (f ∘ (g ∘ ρ)) ≈⟨  cong ⟦ q ⟧₂ (λ x → f∼g (ρ x)) ⟩
   ⟦ q ⟧₂ ⟨$⟩ ρ             ∎
```

```
    where
    open Environment 𝐀 using () renaming ( ⟦_⟧ to ⟦_⟧₁ )
    open Environment 𝐁 using () renaming ( ⟦_⟧ to ⟦_⟧₂ )
    open SetoidReasoning 𝔻[ 𝐁 ]
    private f = _⟨$⟩_ ∣ fh ∣ ; g = _⟨$⟩_ ∣ gh ∣
```

7.5.3 Subalgebraic invariance of ⊧

Identities modeled by an algebra **A** are also modeled by every subalgebra of **A**, which fact can be formalized as follows.

```
module _ {X : Type χ}{𝐀 : Algebra α ρᵃ}{𝐁 : Algebra β ρᵇ}{p q : Term X} where

  ⊧-S-invar : 𝐀 ⊧ (p ≈̇ q) → 𝐁 ≤ 𝐀 → 𝐁 ⊧ (p ≈̇ q)
  ⊧-S-invar Apq B≤A b = goal
    where
    open Setoid 𝔻[ 𝐀 ] using ( _≈_ )
    open Environment 𝐀 using () renaming ( ⟦_⟧ to ⟦_⟧ᵃ )
    open Setoid 𝔻[ 𝐁 ] using () renaming ( _≈_ to _≈ᵇ_ )
    open Environment 𝐁 using ( ⟦_⟧ )
    open SetoidReasoning 𝔻[ 𝐀 ]
    hh : hom 𝐁 𝐀
    hh = ∣ B≤A ∣
    h = _⟨$⟩_ ∣ hh ∣
    ξ : ∀ b → h (⟦ p ⟧ ⟨$⟩ b) ≈ h (⟦ q ⟧ ⟨$⟩ b)
    ξ b = begin
          h (⟦ p ⟧ ⟨$⟩ b)      ≈⟨ comm-hom-term hh p b ⟩
          ⟦ p ⟧ᵃ ⟨$⟩ (h ∘ b)   ≈⟨ Apq (h ∘ b)          ⟩
          ⟦ q ⟧ᵃ ⟨$⟩ (h ∘ b)   ≈˘⟨ comm-hom-term hh q b ⟩
          h (⟦ q ⟧ ⟨$⟩ b)      ∎
    goal : ⟦ p ⟧ ⟨$⟩ b ≈ᵇ ⟦ q ⟧ ⟨$⟩ b
    goal = ∥ B≤A ∥ (ξ b)
```

7.5.4 Product invariance of ⊧

An identity satisfied by all algebras in an indexed collection is also satisfied by the product of algebras in that collection.

```
module _ {X : Type χ}{I : Type ℓ}(𝒜 : I → Algebra α ρᵃ){p q : Term X} where

  ⊧-P-invar : (∀ i → 𝒜 i ⊧ (p ≈̇ q)) → ⨅ 𝒜 ⊧ (p ≈̇ q)
  ⊧-P-invar 𝒜pq a = goal
    where
    open Environment (⨅ 𝒜) using () renaming ( ⟦_⟧ to ⟦_⟧₁ )
    open Environment using ( ⟦_⟧ )
    open Setoid 𝔻[ ⨅ 𝒜 ] using ( _≈_ )
    open SetoidReasoning 𝔻[ ⨅ 𝒜 ]
    ξ : (λ i → (⟦ 𝒜 i ⟧ p) ⟨$⟩ (λ x → (a x) i)) ≈ (λ i → (⟦ 𝒜 i ⟧ q) ⟨$⟩ (λ x → (a x) i))
    ξ = λ i → 𝒜pq i (λ x → (a x) i)
    goal : ⟦ p ⟧₁ ⟨$⟩ a ≈ ⟦ q ⟧₁ ⟨$⟩ a
    goal = begin
           ⟦ p ⟧₁ ⟨$⟩ a                    ≈⟨ interp-prod 𝒜 p a ⟩
```

```
(λ i → (⟦ 𝒜 i ⟧ p) ⟨$⟩ (λ x → (a x) i)) ≈⟨ ξ              ⟩
(λ i → (⟦ 𝒜 i ⟧ q) ⟨$⟩ (λ x → (a x) i)) ≈˘⟨ interp-prod 𝒜 q a ⟩
⟦ q ⟧₁ ⟨$⟩ a                          ∎
```

7.5.5 PS ⊆ SP

Another important fact we will need about the operators S and P is that a product of subalgebras of algebras in a class 𝒦 is a subalgebra of a product of algebras in 𝒦. We denote this inclusion by PS⊆SP, which we state and prove as follows.

```
module _ {𝒦 : Pred(Algebra α ρᵃ) (α ⊔ ρᵃ ⊔ ov ℓ)} where
 private
  a = α ⊔ ρᵃ
  oaℓ = ov (a ⊔ ℓ)

 PS⊆SP : P (a ⊔ ℓ) oaℓ (S{β = α}{ρᵃ} ℓ 𝒦) ⊆ S oaℓ (P ℓ oaℓ 𝒦)
 PS⊆SP {𝐁} (I , ( 𝒜 , sA , B≅⨅A )) = Goal
  where
  ℬ : I → Algebra α ρᵃ
  ℬ i = ∣ sA i ∣
  kB : (i : I) → ℬ i ∈ 𝒦
  kB i = fst ∥ sA i ∥
  ⨅A≤⨅B : ⨅ 𝒜 ≤ ⨅ ℬ
  ⨅A≤⨅B = ⨅-≤ λ i → snd ∥ sA i ∥
  Goal : 𝐁 ∈ S{β = oaℓ}{oaℓ}oaℓ (P {β = oaℓ}{oaℓ} ℓ oaℓ 𝒦)
  Goal = ⨅ ℬ , (I , (ℬ , (kB , ≅-refl))) , (≅-trans-≤ B≅⨅A ⨅A≤⨅B)
```

7.5.6 Identity preservation

The classes H 𝒦, S 𝒦, P 𝒦, and V 𝒦 all satisfy the same set of equations. We will only use a subset of the inclusions used to prove this fact. (For a complete proof, see the Varieties.Func.Preservation module of the agda-algebras library.)

7.5.6.1 H preserves identities

First we prove that the closure operator H is compatible with identities that hold in the given class.

```
module _ {X : Type χ}{𝒦 : Pred(Algebra α ρᵃ) (α ⊔ ρᵃ ⊔ ov ℓ)}{p q : Term X} where

 H-id1 : 𝒦 ⊫ (p ≈̇ q) → (H {β = α}{ρᵃ}ℓ 𝒦) ⊫ (p ≈̇ q)
 H-id1 σ 𝐁 (𝐀 , kA , BimgOfA) ρ = B⊧pq
  where
  IH : 𝐀 ⊧ (p ≈̇ q)
  IH = σ 𝐀 kA
  open Environment 𝐀 using () renaming ( ⟦_⟧ to ⟦_⟧₁)
  open Environment 𝐁 using ( ⟦_⟧ )
  open Setoid 𝔻[ 𝐁 ] using ( _≈_ )
  open SetoidReasoning 𝔻[ 𝐁 ]

  φ : hom 𝐀 𝐁
  φ = ∣ BimgOfA ∣
```

```
φE : IsSurjective ∣ φ ∣
φE = ∥ BimgOfA ∥
φ⁻¹ : 𝕌[ 𝐁 ] → 𝕌[ 𝐀 ]
φ⁻¹ = SurjInv ∣ φ ∣ φE

ζ : ∀ x → (∣ φ ∣ ⟨$⟩ (φ⁻¹ ∘ ρ) x ) ≈ ρ x
ζ = λ _ → InvIsInverseʳ φE

B⊧pq : (⟦ p ⟧ ⟨$⟩ ρ) ≈ (⟦ q ⟧ ⟨$⟩ ρ)
B⊧pq = begin
         ⟦ p ⟧ ⟨$⟩ ρ                           ≈˘⟨ cong ⟦ p ⟧ ζ                  ⟩
         ⟦ p ⟧ ⟨$⟩ (λ x → (∣ φ ∣ ⟨$⟩ (φ⁻¹ ∘ ρ) x)) ≈˘⟨ comm-hom-term φ p (φ⁻¹ ∘ ρ) ⟩
         ∣ φ ∣ ⟨$⟩ (⟦ p ⟧₁ ⟨$⟩ (φ⁻¹ ∘ ρ))         ≈⟨ cong ∣ φ ∣ (IH (φ⁻¹ ∘ ρ))       ⟩
         ∣ φ ∣ ⟨$⟩ (⟦ q ⟧₁ ⟨$⟩ (φ⁻¹ ∘ ρ))         ≈⟨ comm-hom-term φ q (φ⁻¹ ∘ ρ)  ⟩
         ⟦ q ⟧ ⟨$⟩ (λ x → (∣ φ ∣ ⟨$⟩ (φ⁻¹ ∘ ρ) x)) ≈⟨ cong ⟦ q ⟧ ζ                  ⟩
         ⟦ q ⟧ ⟨$⟩ ρ                           ∎
```

7.5.6.2 S preserves identities

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

The obvious converse is barely worth the bits needed to formalize it, but we will use it below, so let's prove it now.

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

7.5.6.3 P preserves identities

```
P-id1 : ∀{ι} → 𝒦 ∣⊨ (p ≈̇ q) → P {β = α}{ρᵃ}ℓ ι 𝒦 ∣⊨ (p ≈̇ q)
P-id1 σ 𝐀 (I , 𝒜 , kA , A≅⨅A) = ⊨-I-invar 𝐀 p q IH (≅-sym A≅⨅A)
  where
  ih : ∀ i → 𝒜 i ⊨ (p ≈̇ q)
  ih i = σ (𝒜 i) (kA i)
  IH : ⨅ 𝒜 ⊨ (p ≈̇ q)
  IH = ⊨-P-invar 𝒜 {p}{q} ih
```

7.5.6.4 V preserves identities

Finally, we prove the analogous preservation lemmas for the closure operator V.

```
module _ {X : Type χ}{ι : Level}{𝒦 : Pred(Algebra α ρᵃ)(α ⊔ ρᵃ ⊔ ov ℓ)}{p q : Term X} where
 private
  aℓι = α ⊔ ρᵃ ⊔ ℓ ⊔ ι

 V-id1 : 𝒦 ∣⊨ (p ≈̇ q) → V ℓ ι 𝒦 ∣⊨ (p ≈̇ q)
 V-id1 σ 𝐁 (𝐀 , (⨅A , p⨅A , A≤⨅A) , BimgA) =
  H-id1{ℓ = aℓι}{𝒦 = S aℓι (P {β = α}{ρᵃ}ℓ ι 𝒦)}{p = p}{q} spK⊧pq 𝐁 (𝐀 , (spA , BimgA))
   where
   spA : 𝐀 ∈ S aℓι (P {β = α}{ρᵃ}ℓ ι 𝒦)
  spA = ⨅A , (p⨅A , A≤⨅A)
  spK⊧pq : S aℓι (P ℓ ι 𝒦) |⊨ (p ≈ ̇ q)
  spK⊧pq = S-id1{ℓ = aℓι}{p = p}{q} (P-id1{ℓ = ℓ} {𝒦 = 𝒦}{p = p}{q} σ)
```

7.5.7 Th 𝒦 ⊆ Th (V 𝒦)

From V-id1 it follows that if 𝒦 is a class of algebras, then the set of identities modeled by the algebras in 𝒦 is contained in the set of identities modeled by the algebras in V 𝒦. In other terms, Th 𝒦 ⊆ Th (V 𝒦). We formalize this observation as follows.

```
classIds-⊆-VIds : 𝒦 |⊨ (p ≈ ̇ q) → (p , q) ∈ Th (V ℓ ι 𝒦)
classIds-⊆-VIds pKq 𝐀 = V-id1 pKq 𝐀
```

8 Free Algebras

8.1 The absolutely free algebra 𝐓 X

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

- Every function from X to ∣ **A** ∣ lifts to a homomorphism from 𝐓 X to **A**.
- 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 ∣ **A** ∣ lifts to a map from ∣ 𝐓 X ∣ to ∣ **A** ∣ in a natural way, by induction on the structure of the given term.

```
module _ {X : Type χ}{𝐀 : Algebra α ρᵃ}(h : X → 𝕌[ 𝐀 ]) where
  open Setoid 𝔻[ 𝐀 ] using ( _≈_ ; reflexive ; refl ; trans )

  free-lift : 𝕌[ 𝐓 X ] → 𝕌[ 𝐀 ]
  free-lift (ℊ x) = h x
  free-lift (node f t) = (f ̂ 𝐀) (λ i → free-lift (t i))

  free-lift-func : 𝔻[ 𝐓 X ] ⟶ 𝔻[ 𝐀 ]
  free-lift-func ⟨$⟩ x = free-lift x
  cong free-lift-func = flcong
    where
    flcong : ∀ {s t} → s ≐ t → free-lift s ≈ free-lift t
    flcong (_≐_.rfl x) = reflexive (≡.cong h x)
    flcong (_≐_.gnl x) = cong (Interp 𝐀) (≡.refl , (λ i → flcong (x i)))
```

Naturally, at the base step of the induction, when the term has the form ℊ 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 ̂ **A** to the images of the subterms.

The free lift so defined is a homomorphism by construction. Indeed, here is the trivial proof.

```
  lift-hom : hom (𝐓 X) 𝐀
  lift-hom = free-lift-func , hhom
    where
    hfunc : 𝔻[ 𝐓 X ] ⟶ 𝔻[ 𝐀 ]
    hfunc = free-lift-func

    hcomp : compatible-map (𝐓 X) 𝐀 free-lift-func
    hcomp {f}{a} = cong (Interp 𝐀) (≡.refl , (λ i → (cong free-lift-func){a i} ≐-isRefl))

    hhom : IsHom (𝐓 X) 𝐀 hfunc
    hhom = mkhom (λ{f}{a} → hcomp{f}{a})

module _ {X : Type χ}{𝐀 : Algebra α ρᵃ} where
  open Setoid 𝔻[ 𝐀 ] using ( _≈_ ; refl )
  open Environment 𝐀 using ( ⟦_⟧ )

  free-lift-interp : (η : X → 𝕌[ 𝐀 ])(p : Term X) → ⟦ p ⟧ ⟨$⟩ η ≈ (free-lift {𝐀 = 𝐀} η) p
  free-lift-interp η (ℊ x) = refl
  free-lift-interp η (node f t) = cong (Interp 𝐀) (≡.refl , (free-lift-interp η) ∘ t)
```

8.2 The relatively free algebra 𝔽[X]

We now define the algebra 𝔽[X], which represents the relatively free algebra. Here, as above, X plays the role of an arbitrary nonempty collection of variables. (It would suffice to takeX to be the cardinality of the largest algebra in $\mathscr{K}$, but since we don't know that cardinality, we leave X aribtrary for now.)

```
module FreeAlgebra {χ : Level}{ι : Level}{I : Type ι}(ℰ : I → Eq) where
  open Algebra

  FreeDomain : Type χ → Setoid _ _
  FreeDomain X = record { Carrier       = Term X
                        ; _≈_           = ℰ ⊢ X ▹_≈_
                        ; isEquivalence = ⊢▹≈IsEquiv }
```

The interpretation of an operation is simply the operation itself. This works since ℰ ⊢ X ▹_≈_ is a congruence.

```
FreeInterp : ∀ {X} → ⟨ 𝑆 ⟩ (FreeDomain X) ⟶ FreeDomain X
FreeInterp ⟨$⟩ (f , ts) = node f ts
cong FreeInterp (≡.refl , h) = app h

𝔽[_] : Type χ → Algebra (ov χ) (ι ⊔ ov χ)
Domain 𝔽[ X ] = FreeDomain X
Interp 𝔽[ X ] = FreeInterp
```

8.3 Basic properties of free algebras

```
module FreeHom (χ : Level) {𝒦 : Pred(Algebra α ρᵃ) (α ⊔ ρᵃ ⊔ ov ℓ)} where
  private ι = ov(χ ⊔ α ⊔ ρᵃ ⊔ ℓ)
  open Eq

  ℐ : Type ι -- indexes the collection of equations modeled by 𝒦
  ℐ = Σ[ eq ∈ Eq{χ} ] 𝒦 ⊫ ((lhs eq) ≈̇ (rhs eq))

  ℰ : ℐ → Eq
  ℰ (eqv , p) = eqv
  𝓔⊢[_]▹Th𝒦 : (X : Type χ) → ∀{p q} → 𝓔 ⊢ X ▹ p ≈ q → 𝒦 ⊫ (p ≈̇ q)
  𝓔⊢[ X ]▹Th𝒦 x 𝑨 kA = sound (λ i ρ → ∥ i ∥ 𝑨 kA ρ) x
    where open Soundness 𝓔 𝑨
  open FreeAlgebra {ι = ι}{I = ℐ} 𝓔 using ( 𝔽[_] )
```

8.3.1 The natural epimorphism from 𝑻 X to 𝔽[X]

We now define the natural epimorphism from 𝑻 X onto the relatively free algebra 𝔽[X] and prove that the kernel of this morphism is the congruence of 𝑻 X defined by the identities modeled by (S 𝒦, hence by) 𝒦.

```
epi𝔽[_] : (X : Type χ) → epi (𝑻 X) 𝔽[ X ]
epi𝔽[ X ] = h , hepi
  where
  open Algebra (𝑻 X) using () renaming (Domain to TX)
  open Setoid TX using () renaming ( _≈_ to _≈₀_ ; refl to refl₀ )
  open Algebra 𝔽[ X ] using () renaming ( Domain to F )
  open Setoid F using () renaming ( _≈_ to _≈₁_ ; refl to refl₁ )
  open _≐_

  c : ∀ {x y} → x ≈₀ y → x ≈₁ y
  c (rfl {x}{y} ≡.refl) = refl₁
  c (gnl {f}{s}{t} x) = cong (Interp 𝔽[ X ]) (≡.refl , c ∘ x)

  h : TX ⟶ F
  h = record { f = id ; cong = c }

  hepi : IsEpi (𝑻 X) 𝔽[ X ] h
  compatible (isHom hepi) = cong h refl₀
  isSurjective hepi {y} = eq y refl₁

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

hom𝔽[_]-is-epic : (X : Type χ) → IsSurjective ∣ hom𝔽[ X ] ∣
hom𝔽[ X ]-is-epic = IsEpi.isSurjective (snd (epi𝔽[ X ]))
```

As promised, we prove that the kernel of the natural epimorphism is the congruence defined by the identities modelled by 𝒦.

```
class-models-kernel : ∀{X p q} → (p , q) ∈ ker ∣ hom𝔽[ X ] ∣ → 𝒦 ⊫ (p ≈̇ q)
class-models-kernel {X = X}{p}{q} pKq = 𝓔⊢[ X ]▹Th𝒦 pKq

kernel-in-theory : {X : Type χ} → ker ∣ hom𝔽[ X ] ∣ ⊆ Th (V ℓ ι 𝒦)
kernel-in-theory {X = X} {p , q} pKq vkA x = classIds-⊆-VIds {ℓ = ℓ}{p = p}{q}
                                                (class-models-kernel pKq) vkA x

module _ {X : Type χ} {𝑨 : Algebra α ρᵃ}{sA : 𝑨 ∈ S {β = α}{ρᵃ} ℓ 𝒦} where
  open Environment 𝑨 using ( Equal )
  ker𝔽⊆Equal : ∀{p q} → (p , q) ∈ ker ∣ hom𝔽[ X ] ∣ → Equal p q
  ker𝔽⊆Equal{p = p}{q} x = S-id1{ℓ = ℓ}{p = p}{q} (𝓔⊢[ X ]▹Th𝒦 x) 𝑨 sA

𝒦⊫→𝓔⊢ : {X : Type χ} → ∀{p q} → 𝒦 ⊫ (p ≈̇ q) → 𝓔 ⊢ X ▹ p ≈ q
𝒦⊫→𝓔⊢ {p = p} {q} pKq = hyp ((p ≈̇ q) , pKq) where open _⊢_▹_≈_ using (hyp)
```

8.3.2 The universal property

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

 open FreeHom {ℓ = ℓ}(α ⊔ ρᵃ ⊔ ℓ) {𝒦}
 open FreeAlgebra {ι = ι}{I = ℐ} ℰ using ( 𝔽[_] )
 open Setoid 𝔻[ 𝑨 ]                  using ( trans ; sym ; refl ) renaming ( Carrier to A )


 𝔽-ModTh-epi : 𝑨 ∈ Mod (Th (V ℓ ι 𝒦))
  → epi 𝔽[ A ] 𝑨
 𝔽-ModTh-epi A∈ModThK = φ , isEpi
  where
   φ : 𝔻[ 𝔽[ A ] ] ⟶ 𝔻[ 𝑨 ]
   _⟨$⟩_ φ = free-lift{𝑨 = 𝑨} id
   cong φ {p} {q} pq = trans ( sym (free-lift-interp{𝑨 = 𝑨} id p) )
                       ( trans ( A∈ModThK{p = p}{q} (kernel-in-theory pq) id )
                               ( free-lift-interp{𝑨 = 𝑨} id q ) )
   isEpi : IsEpi 𝔽[ A ] 𝑨 φ
   compatible (isHom isEpi) = cong (Interp 𝑨) (≡.refl , (λ _ → refl))
   isSurjective isEpi {y} = eq (ℊ y) refl

 𝔽-ModTh-epi-lift : 𝑨 ∈ Mod (Th (V ℓ ι 𝒦)) → epi 𝔽[ A ] (Lift-Alg 𝑨 ι ι)
 𝔽-ModTh-epi-lift A∈ModThK = ∘-epi (𝔽-ModTh-epi (λ {p q} → A∈ModThK{p = p}{q})) ToLift-epi
```

8.4 Products of classes of algebras

We want to pair each $(\mathbf{A} , \mathsf{p})$ (where $\mathsf{p} : \mathbf{A} \in \mathsf{S}\ \mathcal{K}$) with an environment $\rho : \mathsf{X} \to |\ \mathbf{A}\ |$ so that we can quantify over all algebras *and* all assignments of values in the domain $|\ \mathbf{A}\ |$ to variables in X.

```
  module _ (𝒦 : Pred(Algebra α ρᵃ) (α ⊔ ρᵃ ⊔ ov ℓ)){X : Type (α ⊔ ρᵃ ⊔ ℓ)} where
   private ι = ov(α ⊔ ρᵃ ⊔ ℓ)
   open FreeHom {ℓ = ℓ} (α ⊔ ρᵃ ⊔ ℓ){𝒦}
   open FreeAlgebra {ι = ι}{I = ℐ} ℰ using ( 𝔽[_] )
   open Environment using ( Env )

   ℑ⁺ : Type ι
   ℑ⁺ = Σ[ 𝑨 ∈ (Algebra α ρᵃ) ] (𝑨 ∈ S ℓ 𝒦) × (Carrier (Env 𝑨 X))

   𝔄⁺ : ℑ⁺ → Algebra α ρᵃ
   𝔄⁺ i = | i |

   ℭ : Algebra ι ι
   ℭ = ⨅ 𝔄⁺
```

Next we define a useful type, skEqual, which we use to represent a term identity $\mathsf{p} \approx \mathsf{q}$ for any given $\mathsf{i} = (\mathbf{A} , \mathsf{sA} , \rho)$ (where $\mathbf{A}$ is an algebra, $\mathsf{sA} : \mathbf{A} \in \mathsf{S}\ \mathcal{K}$ is a proof that $\mathbf{A}$ belongs to $\mathsf{S}\ \mathcal{K}$, and ρ is a mapping from X to the domain of $\mathbf{A}$). Then we prove AllEqual⊆ker𝔽 which asserts that if the identity $\mathsf{p} \approx \mathsf{q}$ holds in all $\mathbf{A} \in \mathsf{S}\ \mathcal{K}$ (for all environments), then $\mathsf{p} \approx \mathsf{q}$ holds in the relatively free algebra 𝔽[X]; equivalently, the pair $(\mathsf{p} , \mathsf{q})$ belongs to the kernel of the natural homomorphism from $\mathbf{T}$ X onto 𝔽[X]. We will use this fact below to prove that there is a monomorphism from 𝔽[X] into ℭ, and thus 𝔽[X] is a subalgebra of ℭ, so belongs to S (P 𝒦).

```
skEqual : (i : ℑ⁺) → ∀{p q} → Type ρᵃ
skEqual i {p}{q} = ⟦ p ⟧ ⟨$⟩ snd ∥ i ∥ ≈ ⟦ q ⟧ ⟨$⟩ snd ∥ i ∥
  where
  open Setoid 𝔻[ 𝔄⁺ i ] using ( _≈_ )
  open Environment (𝔄⁺ i) using ( ⟦_⟧ )

AllEqual⊆ker𝔽 : ∀ {p q} → (∀ i → skEqual i {p}{q}) → (p , q) ∈ ker ∣ hom𝔽[ X ] ∣
AllEqual⊆ker𝔽 {p} {q} x = Goal
  where
  open Setoid 𝔻[ 𝔽[ X ] ] using ( _≈_ )
  S𝒦|⊨pq : S{β = α}{ρᵃ} ℓ 𝒦 |⊨ (p ≈̇ q)
  S𝒦|⊨pq 𝐀 sA ρ = x (𝐀 , sA , ρ)
  Goal : p ≈ q
  Goal = 𝒦|⊨→ℰ⊢ (S-id2{ℓ = ℓ}{p = p}{q} S𝒦|⊨pq)

homℭ : hom (𝐓 X) ℭ
homℭ = ⨅-hom-co 𝔄⁺ h
  where
  h : ∀ i → hom (𝐓 X) (𝔄⁺ i)
  h i = lift-hom (snd ∥ i ∥)

ker𝔽⊆kerℭ : ker ∣ hom𝔽[ X ] ∣ ⊆ ker ∣ homℭ ∣
ker𝔽⊆kerℭ {p , q} pKq (𝐀 , sA , ρ) = Goal
  where
  open Setoid 𝔻[ 𝐀 ] using ( _≈_ ; sym ; trans )
  open Environment 𝐀 using ( ⟦_⟧ )
  fl : ∀ t → ⟦ t ⟧ ⟨$⟩ ρ ≈ free-lift ρ t
  fl t = free-lift-interp {𝐀 = 𝐀} ρ t
  subgoal : ⟦ p ⟧ ⟨$⟩ ρ ≈ ⟦ q ⟧ ⟨$⟩ ρ
  subgoal = ker𝔽⊆Equal{𝐀 = 𝐀}{sA} pKq ρ
  Goal : (free-lift{𝐀 = 𝐀} ρ p) ≈ (free-lift{𝐀 = 𝐀} ρ q)
  Goal = trans (sym (fl p)) (trans subgoal (fl q))


hom𝔽ℭ : hom 𝔽[ X ] ℭ
hom𝔽ℭ = ∣ HomFactor ℭ homℭ hom𝔽[ X ] ker𝔽⊆kerℭ hom𝔽[ X ]-is-epic ∣

kerℭ⊆ker𝔽 : ∀{p q} → (p , q) ∈ ker ∣ homℭ ∣ → (p , q) ∈ ker ∣ hom𝔽[ X ] ∣
kerℭ⊆ker𝔽 {p}{q} pKq = E⊢pq
  where
  pqEqual : ∀ i → skEqual i {p}{q}
  pqEqual i = goal
    where
    open Environment (𝔄⁺ i) using ( ⟦_⟧ )
    open Setoid 𝔻[ 𝔄⁺ i ] using ( _≈_ ; sym ; trans )
    goal : ⟦ p ⟧ ⟨$⟩ snd ∥ i ∥ ≈ ⟦ q ⟧ ⟨$⟩ snd ∥ i ∥
    goal = trans (free-lift-interp{𝐀 = ∣ i ∣}(snd ∥ i ∥) p)
                 (trans (pKq i)(sym (free-lift-interp{𝐀 = ∣ i ∣} (snd ∥ i ∥) q)))
  E⊢pq : ℰ ⊢ X ▹ p ≈ q
  E⊢pq = AllEqual⊆ker𝔽 pqEqual
```

```
monF𝔈 : mon F[ X ] 𝔈
monF𝔈 = ∣ homF𝔈 ∣ , isMon
  where
  isMon : IsMon F[ X ] 𝔈 ∣ homF𝔈 ∣
  isHom isMon = ∥ homF𝔈 ∥
  isInjective isMon {p} {q} φpq = ker𝔈⊆kerF φpq
```

Now that we have proved the existence of a monomorphism from F[X] to 𝔈 we can prove that F[X] is a subalgebra of 𝔈, so belongs to S (P 𝒦).

```
F≤𝔈 : F[ X ] ≤ 𝔈
F≤𝔈 = mon→≤ monF𝔈

SPF : F[ X ] ∈ S ι (P ℓ ι 𝒦)
SPF = S-idem SSPF
  where
  PS𝔈 : 𝔈 ∈ P (α ⊔ ρᵃ ⊔ ℓ) ι (S ℓ 𝒦)
  PS𝔈 = ℑ⁺ , (𝔄⁺ , ((λ i → fst ∥ i ∥) , ≅-refl))
  SP𝔈 : 𝔈 ∈ S ι (P ℓ ι 𝒦)
  SP𝔈 = PS⊆SP {ℓ = ℓ} PS𝔈
  SSPF : F[ X ] ∈ S ι (S ι (P ℓ ι 𝒦))
  SSPF = 𝔈 , (SP𝔈 , F≤𝔈)
```

9 The HSP Theorem

Finally, we are in a position to prove Birkhoff's celebrated variety theorem.

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

  Birkhoff : ∀ 𝐀 → 𝐀 ∈ Mod (Th (V ℓ ι 𝒦)) → 𝐀 ∈ V ℓ ι 𝒦
  Birkhoff 𝐀 ModThA = V-≅-lc{α}{ρᵃ}{ℓ} 𝒦 𝐀 VlA
    where
    open Setoid 𝔻[ 𝐀 ] using () renaming ( Carrier to A )
    spFA : F[ A ] ∈ S{ι} ι (P ℓ ι 𝒦)
    spFA = SPF{ℓ = ℓ} 𝒦
    epiFlA : epi F[ A ] (Lift-Alg 𝐀 ι ι)
    epiFlA = F-ModTh-epi-lift{ℓ = ℓ} (λ {p q} → ModThA{p = p}{q})
    lAimgFA : Lift-Alg 𝐀 ι ι IsHomImageOf F[ A ]
    lAimgFA = epi→ontohom F[ A ] (Lift-Alg 𝐀 ι ι) epiFlA
    VlA : Lift-Alg 𝐀 ι ι ∈ V ℓ ι 𝒦
    VlA = F[ A ] , spFA , lAimgFA
```

The converse inclusion, V 𝒦 ⊆ Mod (Th (V 𝒦)), is a simple consequence of the fact that Mod Th is a closure operator. Nonetheless, completeness demands that we formalize this inclusion as well, however trivial the proof.

```
module _ {𝐀 : Algebra α ρᵃ} where
 Birkhoff-converse : 𝐀 ∈ V{α}{ρᵃ}{α}{ρᵃ}{α}{ρᵃ} ℓ ι 𝒦 → 𝐀 ∈ Mod{X = 𝕌[ 𝐀 ]} (Th (V ℓ ι 𝒦))
```

```
  Birkhoff-converse vA pThq = pThq 𝐀 vA
```

We have thus proved that every variety is an equational class.

Readers familiar with the classical formulation of the Birkhoff HSP theorem as an "if and only if" assertion might worry that the proof is still incomplete. However, recall that in the Varieties.Func.Preservation module we proved the following identity preservation lemma:

```
V-id1 : 𝒦 |⊫ p ≈̇ q → V 𝒦 |⊫ p ≈̇ q
```

Thus, if 𝒦 is an equational class—that is, if 𝒦 is the class of algebras satisfying all identities in some set—then V 𝒦 ⊆ 𝒦. On the other hand, we proved that V is expansive in the Varieties.Func.Closure module:

```
V-expa : 𝒦 ⊆ V 𝒦
```

so 𝒦 (= V 𝒦 = H S P 𝒦) is a variety.

Taken together, V-id1 and V-expa constitute formal proof that every equational class is a variety. This completes the formal proof of Birkhoff's variety theorem.

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 Venanzio Capretta. Universal algebra in type theory. In *Theorem proving in higher order logics (Nice, 1999)*, volume 1690 of *Lecture Notes in Comput. Sci.*, pages 131–148. Springer, Berlin, 1999. doi:10.1007/3-540-48256-3_10.

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

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