The Agda Universal Algebra Library (UALib)

version of 16 October 2020, 11:58.

- William DeMeo, Department of Algebra, Faculty of Mathematics and Physics, Charles University, Czech Republic
- Hyeyoung Shin, Faculty of Information Technology, Czech Technical University, Czech Republic
- Siva Somayyajula, Department of Computer Science, Carnegie Mellon University, USA

Table of contents ⇓

Preface

To support formalization in type theory of research level mathematics in universal algebra and related fields, we are developing a software library, called the Agda Universal Algebra Library (UALib). Our library contains formal statements and proofs of some of the core, foundational definitions and results universal algebra and is written in Agda.

Agda is a programming language and proof assistant, or "interactive theorem prover" (ITP), that not only supports dependent and inductive types, but also provides powerful *proof tactics* for proving things about the objects that inhabit these types.

Vision and Goals

The idea for the the Agda Universal Algebra Library (UALib) originated with the observation that, on the one hand a number of basic and important constructs in universal algebra can be defined recursively, and theorems about them proved inductively, while on the other hand the *types* (of type theory —in particular, dependent types and inductive types) make possible elegant formal representations of recursively defined objects, and constructive (*computable*) proofs of their properties. These observations suggest that there is much to gain from implementing universal algebra in a language that facilitates working with dependent and inductive types.

Primary Goals

The first goal of the UALib project is to demonstrate that it is possible to express the foundations of universal algebra in type theory and to formalize (and formally verify) the foundations in the Agda programming language. We will formalize a substantial portion of the edifice on which our own mathematical research depends, and demonstrate that our research can also be expressed in type theory and formally implemented in such a way that we and other working mathematicians can understand and verify the results. The resulting library will also serve to educate our peers, and encourage and help them to formally verify their own mathematics research.

Our field is deep and wide and codifying all of its foundations may seem like a daunting task and possibly risky investment of time and resources. However, we believe our subject is well served by a new, modern, constructive presentation of its foundations. Our new presentation expresses the foundations of universal algebra in the language of type theory, and uses the Agda proof assistant to codify and formally verify everything.

Secondary Goals

We wish to emphasize that our ultimate objective is not merely to translate existing results into a more modern and formal language. Indeed, one important goal is to develop a system that is useful for conducting research in mathematics, and that is how we intend to use our library once we have achieved our immediate objective of implementing the basic foundational core of universal algebra in Agda.

To this end, our intermediate-term objectives include

- developing domain specific "proof tactics" to express the idioms of universal algebra,
- incorporating automated proof search for universal algebra, and
- formalizing theorems emerging from our own mathematics research,
- documenting the resulting software libraries so they are usable by other working mathematicians.

For our own mathematics research, we believe a proof assistant equipped with specialized libraries for universal algebra, as well as domain-specific tactics to automate proof idioms of our field, will be extremely useful. Thus, a secondary goal is to demonstrate (to ourselves and colleagues) the utility of such libraries and tactics for proving new theorems.

Intended audience

This document describes the Agda Universal Algebra Library (UALib) in enough detail so that working mathematicians (and possibly some normal people, too) might be able to learn enough about Agda and its libraries to put them to use when creating, formalizing, and verifying new mathematics.

While there are no strict prerequisites, we expect anyone with an interest in this work will have been motivated by prior exposure to universal algebra, as presented in, say, Bergman:2012 or McKenzie:1987, and to a lesser extent category theory, as presented in categorytheory.gitlab.io or Riehl:2017.

Some prior exposure to type theory and Agda would be helpful, but even without this background one might still be able to get something useful out of this by referring to the appendix and glossary, while simultaneously consulting one or more of the references mentioned in references to fill in gaps as needed.

Finally, it is assumed that while reading these materials the reader is actively experimenting with Agda using emacs with its agda2-mode extension installed. If not, follow the directions on the main Agda website to install them.

Installing the library

The main repository for the UALib is https://gitlab.com/ualib/ualib.gitlab.io.

There are installation instructions in the main README.md file in that repository, but really all you need to do is have a working Agda (and agda2-mode) installation and clone the UALib repository with, e.g.,

```
git clone git@gitlab.com:ualib/ualib.gitlab.io.git
```

OR

```
git clone https://gitlab.com/ualib/ualib.gitlab.io.git
```

Unicode hints

Information about unicode symbols is readily available in Emacs agda2-mode; simply place the cursor on the character of interest and enter the command `M-x describe-char` (or `M-m h d c`). To see a full list of available characters, enter `M-x describe-input-method` (or `C-h I`).

Acknowledgments

Besides the main authors and developers of UALib, a number of other people have contributed to the project in one way or another.

Special thanks go to Clifford Bergman, Venanzio Capretta, Andrej Bauer, Miklós Maróti, and Ralph Freese, for many helpful discussions, as well as the invaluable instruction, advice, and encouragement that they continue to lend to this project, often without even knowing it.

The first author would also like to thank his postdoctoral advisors and their institutions for supporting (sometimes without their knowledge) work on this project. These include Peter Mayr and University of Colorado in Boulder (Aug 2017–May 2019), Ralph Freese and the University of Hawaii in Honolulu (Aug 2016–May 2017), Cliff Bergman and Iowa State University in Ames (Aug 2014–May 2016).

Attributions and citations

Regarding the mathematical results that are implemented in the UALib library, as well as the presentation and informal statements of these results in the documentation, The Authors makes no claims to originality.

Regarding the Agda source code in the UALib library, this is mainly due to The Authors.

HOWEVER, we have benefited from the outstanding lecture notes on Univalent Foundations and Homotopy Type Theory and the Type Topology Agda Library, both by Martin Hötzel Escardo. The first author is greatly indebted to Martin for teaching him about type theory in Agda at the Midlands Graduate School in the Foundations of Computing Science in Birmingham in 2019.

The development of the UALib and its documentation is informed by and benefits from the references listed in the references section below.

References

The following Agda documentation and tutorials are excellent. They have been quite helpful to The Author of UALib, and have informed the development of the latter and its documentation.

- Altenkirk, Computer Aided Formal Reasoning
- Bove and Dybjer, Dependent Types at Work
- Escardo, Introduction to Univalent Foundations of Mathematics with Agda
- Gunther, Gadea, Pagano, Formalization of Universal Algebra in Agda
- János, Agda Tutorial
- Norell and Chapman, Dependently Typed Programming in Agda
- Wadler, Programming Language Foundations in Agda

Finally, the official Agda Wiki, Agda User's Manual, Agda Language Reference, and the (open source) Agda Standard Library source code are also quite useful.

Table of contents

Agda Preliminaries

Notation. Here are some acronyms that we use frequently.

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

All but the most trivial Agda programs begin by setting some options that effect how Agda behaves and importing from existing libraries (e.g., the Agda Standard Library or, in our case, MHE's Type Topology library). In particular, logical axioms and deduction rules can be specified according to what one wishes to assume.

For example, we begin our agda development with the line

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

module ualib where

module prelude where

 open import Universes public
```

This specifies Agda `OPTIONS` that we will use throughout the library.

- `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* or *definitional* equalities. MHE explains this by saying it "makes sure that pattern matching corresponds to Martin-Löf eliminators;" see also the Pattern matching and equality section of the Agda Tools documentation.
- `safe` ensures that nothing is postulated outright—every non-MLTT axiom has to be an explicit assumption (e.g., an argument to a function or module); see also this section of the Agda Tools documentation and the Safe Agda section of the Agda Language Reference.

Universes

We import the `Universes` module from MHE's Type Topology library.

```
 open import Universes public
```

This `Universes` module provides, among other things, an elegant notation for type universes that we have fully adopted and we use MHE's notation throughout the UALib.

MHE has authored an outstanding set of notes on HoTT-UF-in-Agda called Introduction to Univalent Foundations of Mathematics with Agda. We highly recommend these notes to anyone wanting more details than we provide here about MLTT and the Univalent Foundations/HoTT extensions thereof.

Following MHE, we refer to universes using capitalized script letters $\mathcal{U},\mathcal{V},\mathcal{W},\mathcal{T}$. We add a few more to Martin's list.

```
variable 𝓘 𝓙 𝓚 𝓛 𝓜 𝓝 𝓞 𝓠 𝓡 𝓢 𝓧 : Universe
```

In the `Universes` module, MHE defines the ̇ operator which maps a universe $\mathcal{U}$ (i.e., a level) to `Set 𝓤`, and the latter has type `Set (lsuc 𝓤)`. The level `lzero` is renamed $\mathcal{U}_0$, so $\mathcal{U}_0$̇ is an alias for `Set lzero`.

Although it is nice and short, we won't show all of the `Universes` module here. Instead, we highlight the few lines of code from MHE's `Universes.lagda` file that makes available the notational devices that we just described and will adopt throughout the UALib.

Thus, $\mathcal{U}$̇ is simply an alias for `Set 𝓤`, and we have `Set 𝓤 : Set (lsuc 𝓤)`. Finally, `Set (lsuc lzero)` is denoted by `Set 𝓤₀ ⁺` which (MHE and) we denote by $\mathcal{U}_0{}^+$̇.

The following dictionary translates between standard Agda syntax and MHE/UALib.

```
Agda              MHE/UALib
====              ==========
Level             Universe
lzero             𝓤₀
𝓤 : Level         𝓤 : Universe
Set lzero         𝓤₀ ̇
Set 𝓤             𝓤 ̇
lsuc lzero        𝓤₀ ⁺
lsuc 𝓤            𝓤 ⁺
Set (lsuc lzero)  𝓤₀ ⁺ ̇
Set (lsuc 𝓤)      𝓤 ⁺ ̇
Setω              𝓤ω
```

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

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

Public imports

Next we import other parts of MHE's Type Topology library, using the Agda directive `public`, which means these imports will be available wherever the `prelude` module in imported. We describe some of these imports later, when making use of them, but we don't describe each one in detail. (The interested or confused reader should consult HoTT-UF-in-Agda to learn more.)

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

pattern refl x = 𝓻𝓮𝓯𝓵 {x = x}

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

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

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

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

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

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

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

```
open import MGS-Subsingleton-Truncation hiding (refl; _∈_; _⊆_) public
```

Dependent pair type

Our preferred notations for the first and second projections of a product are `|_|` and `‖_‖`, respectively; however, we will sometimes use the more standard $\mathtt{pr}_1$ and $\mathtt{pr}_2$, or even `fst` and `snd`, for emphasis, readability, or compatibility with other libraries.

```
|_| fst : {X : 𝓤 ̇ }{Y : X → 𝓥 ̇ } → Σ Y → X
| x , y | = x
fst (x , y) = x

‖_‖ snd : {X : 𝓤 ̇ }{Y : X → 𝓥 ̇ } → (z : Σ Y) → Y (pr₁ z)
‖ x , y ‖ = y
snd (x , y) = y
```

For the dependent pair type, we prefer the notation `Σ x ꞉ X , y`, which is more pleasing (and more standard in the literature) than Agda's default syntax (`Σ λ(x ꞉ X) → y`), and MHE has a useful trick that makes the preferred notation available by making index type explicit.

```
infixr -1 -Σ
-Σ : {𝓤 𝓥 : Universe} (X : 𝓤 ̇ ) (Y : X → 𝓥 ̇ ) → 𝓤 ⊔ 𝓥 ̇
-Σ X Y = Σ Y
syntax -Σ X (λ x → y) = Σ x ꞉ X , y -- type `:` as `\:4`
```

The symbol ꞉ is not the same as : despite how similar they may appear. The correct colon in the expression `Σ x ꞉ X , y` above is obtained by typing `\:4` in [agda2-mode](https://agda.readthedocs.io/en/v2.6.0.1/tools/emacs-mode.html).

MHE explains Sigma induction as follows: "To prove that `A z` holds for all `z : Σ Y`, for a given property `A`, we just prove that we have `A (x , y)` for all `x : X` and `y : Y x`. This is called `Σ` induction or `Σ` elimination (or `uncurry`).

```
Σ-induction : {X : 𝓤 ̇ }{Y : X → 𝓥 ̇ }{A : Σ Y → 𝓦 ̇ }
 →            ((x : X)(y : Y x) → A (x , y))
              ------------------------------
 →            ((x , y) : Σ Y) → A (x , y)
Σ-induction g (x , y) = g x y

curry : {X : 𝓤 ̇ }{Y : X → 𝓥 ̇ }{A : Σ Y → 𝓦 ̇ }
 →      (((x , y) : Σ Y ) → A (x , y))
       --------------------------------
 →      ((x : X) (y : Y x) → A (x , y))
curry f x y = f (x , y)
```

The special case in which the type `Y` doesn't depend on `X` is of course the usual Cartesian product.

```
infixr 30 _×_
_×_ : 𝓤 ̇ → 𝓥 ̇ → 𝓤 ⊔ 𝓥 ̇
X × Y = Σ x ꞉ X , Y
```

Dependent function type

To make the syntax for `Π` conform to the standard notation for "Pi types" (or dependent function type), MHE uses the same trick as the one used above for "Sigma types."

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

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

Application

An important tool that we use often in Agda proofs is application of a function to an identification `p : x ≡ x'`. We apply the `ap` operator to obtain the identification `ap f p : f x ≡ f x'` when given `p : x ≡ x'` and `f : X → Y`.

Since `ap` is already defined in MHE's Type Topolgy library, we don't redefine it here. However, we do define some variations of `ap` that are sometimes useful.

```
ap-cong : {X : 𝓤 ̇ }{Y : 𝓥 ̇ }{f g : X → Y} {a b : X}
 →        f ≡ g  →  a ≡ b
```

```
          -----------------
 →        f a ≡ g b

ap-cong (refl _) (refl _) = refl _
```

Here is a related tool that we borrow from the Relation/Binary/PropositionalEquality.agda module of the Agda standard library.

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

cong-app (refl _) a = refl _
```

Function extensionality

Extensional equality of functions, or function extensionality, means that any two point-wise equal functions are equal. As MHE points out, this is known to be not provable or disprovable in Martin-Löf Type Theory (MLTT).

Nonetheless, we will mainly work with pointwise equality of functions, which MHE defines (in Type Topology) as follows:

```
_~_ : {X : 𝓤 ̇ } {A : X → 𝓥 ̇ } → Π A → Π A → 𝓤 ⊔ 𝓥 ̇
f ~ g = ∀ x → f x ≡ g x
infix 0 _~_
```

(The _~_ relation will be equivalent to equality of functions, once we have the principle of *univalence* at our disposal.)

Predicates, Subsets

We need a mechanism for implementing the notion of subsets in Agda. A typical one is called `Pred` (for predicate). More generally, `Pred A 𝓤` can be viewed as the type of a property that elements of type `A` might satisfy. We write `P : Pred A 𝓤` (read "`P` has type `Pred A 𝓤`") to represent the subset of elements of `A` that satisfy property `P`.

Here is the definition (which is similar to the one found in the `Relation/Unary.agda` file of Agda standard library).

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

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

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

The membership relation

We introduce notation so that we may indicate that `x` "belongs to" a "subset" `P`, or that `x` "has property" `P`, by writing either `x ∈ P` or `P x` (cf. `Relation/Unary.agda` in the Agda standard library).

```
infix 4 _∈_ _∉_
_∈_ : {A : 𝓤 ̇ } → A → Pred A 𝓦 → 𝓦 ̇
x ∈ P = P x

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

Subset relations and operations

The subset relation is then denoted, as usual, with the ⊆ symbol (cf. `Relation/Unary.agda` in the Agda standard library).

```
infix 4 _⊆_ _⊇_
_⊆_ : {A : 𝓤 ̇ } → Pred A 𝓦 → Pred A 𝓣 → 𝓤 ⊔ 𝓦 ⊔ 𝓣 ̇
P ⊆ Q = ∀ {x} → x ∈ P → x ∈ Q

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

infixr 1 _⊎_
```

```
-- Disjoint Union.
data _⊎_ (A : 𝓤 ̇) (B : 𝓥 ̇) : 𝓤 ⊔ 𝓥 ̇ where
  inj₁ : (x : A) → A ⊎ B
  inj₂ : (y : B) → A ⊎ B

-- Union.
infixr 6 _∪_
_∪_ : {A : 𝓤 ̇} → Pred A 𝓥 → Pred A 𝓦 → Pred A _
P ∪ Q = λ x → x ∈ P ⊎ x ∈ Q


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

Miscellany

Finally, we include the following list of "utilities" that will come in handy later. Most of these are self-explanatory, but we make a few remarks below when we feel there is something worth noting.

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

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

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

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

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

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

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

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


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

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

N.B. the assertion `Image f ∋ y` must come with a proof, which is of the form `∃a f a = y`, so we have a witness. Thus, the inverse can be "computed" in the following way:

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

The special case for Set (i.e., $\mathcal{U}_0$ ̇) is

```
inv : {A B : 𝓤₀ ̇ }(f : A → B)(b : B) → Image f ∋ b → A
inv {A} {B} = Inv {𝓤₀}{𝓤₀}{A}{B}

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

An epic (or surjective) function from $\mathcal{U}$ ̇ to $\mathcal{W}$ ̇ (and the special case for $\mathcal{U}_0$ ̇) is defined as follows.

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

epic : {A B : 𝓤₀ ̇ } (g : A → B) → 𝓤₀ ̇
epic = Epic {𝓤₀} {𝓤₀}
```

The (pseudo-)inverse of an epic function is

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

-- The (psudo-)inverse of an epic is the right inverse.
EInvIsRInv : funext 𝓦 𝓦 → {A : 𝓤 ̇ } {B : 𝓦 ̇ }
             (f : A → B)  (fEpic : Epic f)
            ---------------------------------
 →           f ∘ (EpicInv f fEpic) ≡ 𝑖𝑑 B
EInvIsRInv fe f fEpic = fe (λ x → InvIsInv f x (fEpic x))
```

Monics (or injective) functions are defined this way.

```
monic : {A : 𝓤 ̇ } {B : 𝓦 ̇ } (g : A → B) → 𝓤 ⊔ 𝓦 ̇
monic g = ∀ a₁ a₂ → g a₁ ≡ g a₂ → a₁ ≡ a₂
monic₀ : {A B : 𝓤₀ ̇ } (g : A → B) → 𝓤₀ ̇
monic₀ = monic {𝓤₀}{𝓤₀}

--The (pseudo-)inverse of a monic function
monic-inv : {A : 𝓤 ̇ } {B : 𝓦 ̇ } (f : A → B) → monic f
 →           (b : B) → Image f ∋ b → A
monic-inv f fmonic  = λ b Imf∋b → Inv f b Imf∋b

--The (psudo-)inverse of a monic is the left inverse.
monic-inv-is-linv : {A : 𝓤 ̇ }{B : 𝓦 ̇ }
                    (f : A → B) (fmonic : monic f)(x : A)
                   ---------------------------------------
  →                 (monic-inv f fmonic) (f x) (im x) ≡ x
monic-inv-is-linv f fmonic x = refl _
```

Finally, we define bijective functions as follows.

```
bijective : {A B : 𝓤₀ ̇ }(g : A → B) → 𝓤₀ ̇
bijective g = epic g × monic g

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

More extensionality

Here we collect miscellaneous definitions and proofs related to extensionality that will come in handy later.

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

--Opposite of function extensionality
intensionality : ∀ {𝓤 𝓦} {A : 𝓤 ̇ } {B : 𝓦 ̇ } {f g : A → B}
 →                  f ≡ g  →  (x : A)
                   ------------------
```

```
 →                    f x ≡ g x

intensionality  (refl _ ) _  = refl _

--Dependent intensionality
dep-intensionality : ∀ {𝓤 𝓦}{A : 𝓤 ̇ }{B : A → 𝓦 ̇ }
                     {f g : ∀(x : A) → B x}
 →                   f ≡ g  →  (x : A)
                    ------------------
 →                    f x ≡ g x

dep-intensionality (refl _ ) _ = refl _

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

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

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

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

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

Table of contents ⇑

Algebras in Agda

This chapter describes the basic module of the UALib , which begins our Agda formalization of the basic concepts and theorems of universal algebra. In this module we will codify such notions as operation, signature, and algebraic structure.

```
module basic where

 open prelude using (Universe; 𝓘; 𝓞; 𝓤; 𝓤₀;𝓥; 𝓦; 𝓣; 𝓧;
   _⁺; _̇;_⊔_; _,_; Σ; -Σ; ∣_∣; ‖_‖; 𝟘; 𝟚; _×_; Π; _≡_; Epic) public
```

Operation type

We define the type of **operations**, and give an example (the projections).

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

 --Example. the projections
 π : {I : 𝓥 ̇ } {A : 𝓤 ̇ } → I → Op I A
 π i x = x i
```

The type `Op` encodes the arity of an operation as an arbitrary type `I : 𝓥 ̇`, which gives us a very general way to represent an operation as a function type with domain `I → A` (the type of “tuples”) and codomain `A`.

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

Signature type

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

```
 --𝓞 is the universe in which operation symbols live
 --𝓥 is the universe in which arities live
 Signature : (𝓞 𝓥 : Universe) → 𝓞 ⁺ ⊔ 𝓥 ⁺ ̇
 Signature 𝓞 𝓥 = Σ F ꞉ 𝓞 ̇ , ( F → 𝓥 ̇ )
```

In the prelude module we defined the syntax ∣_∣ and ∥_∥ for the first and second projections, resp. Consequently, if `𝑆 : Signature 𝒪 𝒱` is a signature, then

∣ 𝑆 ∣ denotes the set of operation symbols (which is often called F), and

∥ 𝑆 ∥ denotes the arity function (which is often called ρ).

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

Algebra type

Finally, we are ready to define the type of algebras in the signature `S` (which we also call "S-algebras").

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

Algebra 𝓤 {𝒪}{𝒱} 𝑆 = Σ A ꞉ 𝓤 ̇ , ((𝑓 : ∣ 𝑆 ∣) → Op (∥ 𝑆 ∥ 𝑓) A)
```

Thus, algebras—in the signature S (or S-algebras) and with carrier types in the universe $\mathcal{U}$—inhabit the type `Algebra 𝓤 {𝒪} {𝒱} 𝑆`. (We may also write `Algebra 𝓤 𝑆` since $\mathcal{O}$ and $\mathcal{V}$ can be infered from the given signature S.)

In other words,

the type `Algebra 𝓤 𝑆` collects all the algebras of a particular signature S and carrier type $\mathcal{U}$, and this collection of algebras has type $\mathcal{O} ⊔ \mathcal{V} ⊔ \mathcal{U}^{+}$ ̇.

Recall, $\mathcal{O} ⊔ \mathcal{V} ⊔ \mathcal{U}^{+}$ denotes the smallest universe containing $\mathcal{O}$, $\mathcal{V}$, and the successor of $\mathcal{U}$.

N.B. The type `Algebra 𝓤 𝑆` doesn't define what an algebra *is* as a property. It defines a type of algebras; certain algebras inhabit this type—namely, the algebras consisting of a universe (say, `A`) of type $\mathcal{U}$ ̇, and a collection `(𝑓 : ∣ 𝑆 ∣) → Op (∥ 𝑆 ∥ 𝑓) A` of operations on `A`.

Here's an alternative syntax that might seem more familiar to readers of the standard universal algebra literature.

```
Algebra 𝓤 (F , ρ) = Σ A ꞉ 𝓤 ̇ ,  ((𝑓 : F )  → Op (ρ 𝑓) A )
```

Here `𝑆 = (F , ρ)` is the signature, `F` the type of operation symbols, and ρ the arity function.

Although this syntax would work equally well, we mention it merely for comparison and to demonstrate the flexibility of Agda. Throughout the library we stick to the syntax $f : ∣ S ∣$ for an operation symbol of the signature S, and ∥ 𝑆 ∥ f for the arity of that symbol. We find these conventions a bit more convenient for programming.

Example

A monoid signature has two operation symbols, say, `e` and `·`, of arities 0 and 2 (thus, of types `(𝟘 → A) → A` and `(𝟚 → A) → A`), resp.

```
data monoid-op : 𝓤₀ ̇ where
 e : monoid-op
 · : monoid-op

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

The types indicate that `e` is nullary (i.e., takes no arguments, equivalently, takes args of type `𝟘 → A`), while `·` is binary (as indicated by argument type `𝟚 → A`).

We will have more to say about the type of algebras later. For now, we continue defining the syntax used in the `agda-ualib` to represent the basic objects of universal algebra.

Syntactic sugar for operation interpretation

Before proceding, we define syntax that allows us to replace ∥ 𝐴 ∥ f with the slightly more standard-looking f ̂ A, where f is an operation symbol of the signature S of $\boldsymbol{A}$.

```
open basic

module _ {𝑆 : Signature 𝒪 𝒱}  where

 _̂_ : (𝑓 : ∣ 𝑆 ∣)
```

```
  →    (A : Algebra 𝓤 S)
  →    (∥ S ∥ f  →  | A |) → | A |

f ̂ A = λ x → (∥ A ∥ f) x

infix 40 _̂_
```

Now we can use `f ̂ A` to represent the interpretation of the basic operation symbol f in the algebra $\mathbf{A}$.

Below, we will need slightly different notation, namely, $t ̇ \mathbf{A}$, to represent the interpretation of a term t in the algebra $\mathbf{A}$. (In future releases of the UALib we may reconsider making it possible to use the same notation interpretations of operation symbols and terms.)

Products of algebras

The (indexed) product of a collection of algebras is also an algebra if we define such a product as follows:

```
⨅ : {I : 𝓘 ̇ }(𝒜 : I → Algebra 𝓤 S ) → Algebra (𝓤 ⊔ 𝓘) S
⨅ 𝒜 =  ((i : _) → | 𝒜 i |) ,  λ f x i → (f ̂ 𝒜 i) λ 𝓥 → x 𝓥 i

infixr -1 ⨅
```

(In `agda2-mode` ⨅ is typed as `\Glb`.)

Arbitrarily many variable symbols

Finally, since we typically want to assume that we have an arbitrary large collection `X` of variable symbols at our disposal (so that, in particular, given an algebra $\mathbf{A}$ we can always find a surjective map $h_0 : X \to |\mathbf{A}|$ from X to the universe of $\mathbf{A}$), we define a type for use when making this assumption.

```
_↠_ : 𝓧 ̇ → Algebra 𝓤 S → 𝓧 ⊔ 𝓤 ̇
X ↠ A = Σ h ꞉ (X → | A |) , Epic h
```

Unicode Hints 1

Table of some special characters used in the basic module.

To get	Type
$\mathcal{I}$	`\MCI`
$\mathcal{U}_0$	`\MCU\_0`
⊔	`\sqcup`
$\mathbb{0}$, $\mathbb{2}$	`\b0`, `\b2`
a, b	`\Mia`, `\Mib`
$\boldsymbol{a}$, $\boldsymbol{b}$	`\MIa`, `\MIb`
$\mathcal{A}$	`\McA`
$f ̂ \mathbf{A}$	`\Mif \^ \MIA`
≅	≅ or `\cong`
∘	`\comp` or `\circ`
$\mathcal{id}$	`\Mci\Mcd`
$\mathcal{L}\mathcal{K}$	`\McL\McK`
ϕ	`\phi`
⨅	`\Glb`

Emacs commands providing information about characters or input method:

- `M-x describe-char` (or `M-m h d c`) with the cursor on the character of interest
- `M-x describe-input-method` (or `C-h I`)

Table of contents ⇑

Congruences in Agda

This chapter describes the congruences module of the UALib.

N.B. Some of the code in this first part of this chapter pertaining to relations is borrowed from similar code in the Agda standard library (in the file [Relation/`Binary/Core.agda][]) that we translate into our notation for consistency.

```
open basic

module congruences where

 open prelude using (Pred; 𝓡; 𝓢; is-prop; 𝟙; _≡⟨_⟩_; _∎; refl; _⁻¹; funext; ap) public
```

Binary relation type

Heterogeneous binary relations.

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

Homogeneous binary relations.

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

Kernels

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

```
 KER : {A : 𝓤 ̇ } {B : 𝓦 ̇ } → (A → B) → 𝓤 ⊔ 𝓦 ̇
 KER {𝓤}{𝓦}{A} f = Σ x ꞉ A , Σ y ꞉ A , f x ≡ f y

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

or as a relation…

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

 -- (in the special case 𝓦 ≡ 𝓤)
 ker-rel : {A B : 𝓤 ̇ } → (A → B) → Rel A 𝓤
 ker-rel {𝓤} = KER-rel {𝓤} {𝓤}
```

or a binary predicate…

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

 -- (in the special case 𝓦 ≡ 𝓤)
 ker-pred : {A : 𝓤 ̇ }{B : 𝓤 ̇ } → (A → B) → Pred (A × A) 𝓤
 ker-pred {𝓤} = KER-pred {𝓤} {𝓤}
```

Implication

We denote and define implication or containment (which could also be written _⊆_) as follows.

```
 _⇒_ : {A : 𝓤 ̇ } {B : 𝓥 ̇ }
  →    REL A B 𝓡 → REL A B 𝓢
  →    𝓤 ⊔ 𝓥 ⊔ 𝓡 ⊔ 𝓢 ̇

 P ⇒ Q = ∀ {i j} → P i j → Q i j

 infixr 4 _⇒_

 _on_ : {A : 𝓤 ̇ } {B : 𝓥 ̇ } {C : 𝓦 ̇ }
  →     (B → B → C) → (A → B) → (A → A → C)
 _*_ on f = λ x y → f x * f y
```

Here is a more general version of implication

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

```
P =[ f ]⇒ Q = P ⇒ (Q on f)

infixr 4 _=[_]⇒_
```

Properties of binary relations

Reflexivity of a binary relation (say, ≈) on X, can be defined without an underlying equality as follows.

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

Similarly, we have the usual notion of symmetric (resp., transitive) binary relation.

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

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

For a binary relation ≈ on A, denote a single ≈-class (containing a) by [a] ≈,

```
[_]_ :  {A : 𝓤 ̇ } →  (a : A) → Rel A 𝓡 → 𝓤 ⊔ 𝓡 ̇
[ a ] _≈_ = Σ x ꞉ _ ,  a ≈ x
```

and denote the collection of all ≈-classes of A by A // ≈.

```
_//_ :  (A : 𝓤 ̇ ) → Rel A 𝓡 → (𝓤 ⊔ 𝓡) ⁺ ̇
A // ≈ = Σ C ꞉ _ ,   Σ a ꞉ A ,  C ≡ ( [ a ] ≈ )

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

The “trivial” or “diagonal” or “identity” relation is,

```
𝟎 : {A : 𝓤 ̇ } → 𝓤 ̇
𝟎{𝓤} {A} = Σ a ꞉ A , Σ b ꞉ A , a ≡ b

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

or, in various other guises,

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

--...as a binary predicate:
𝟎'' : {A : 𝓤 ̇ } → 𝓤 ̇
𝟎'' {𝓤} {A} = Σ p ꞉ (A × A) , ∣ p ∣ ≡ ∥ p ∥
```

The “universal” or “total” or “all” relation.

```
𝟏 : {A : 𝓤 ̇ } → Rel A 𝓤₀
𝟏 a b = 𝟙
```

Types for equivalences

Here are two ways to define an equivalence relation in Agda.

First, we use a record.

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

Here’s an alternative.

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

Of course, $\mathbf{0}$ is an equivalence relation, a fact we can prove as follows.

```
0-IsEquivalence : {A : 𝓤 ̇ } → IsEquivalence {𝓤}{𝓤}{A} 0-rel
0-IsEquivalence = record { rfl = ρ ; sym = σ ; trans = τ }
 where
  ρ : reflexive 0-rel
  ρ x =  x ≡⟨ refl _ ⟩ x ∎

  σ : symmetric 0-rel
  σ x y x≡y = x≡y ⁻¹

  τ : transitive 0-rel
  τ x y z x≡y y≡z = x ≡⟨ x≡y ⟩ y ≡⟨ y≡z ⟩ z ∎
```

We define the **lift** of a binary relation from pairs to pairs of tuples as follows:

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

We define **compatibility** of a given function-relation pair as follows:

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

Types for congruences

Finally, we come to the definition of a congruence, which we define in a module so we have an ambient signature S available.

```
open congruences

module _ {𝑆 : Signature 𝓞 𝓥}  where

 -- relation compatible with an operation
 compatible-op : {𝑨 : Algebra 𝓤 𝑆}
  →              ∣ 𝑆 ∣ → Rel ∣ 𝑨 ∣ 𝓤
  →              𝓥 ⊔ 𝓤 ̇
 compatible-op {𝓤} {𝑨} f r = (lift-rel r) =[ (∥ 𝑨 ∥ f) ]⇒ r

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

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

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

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

 con : (𝑨 : Algebra 𝓤 𝑆)  →  Pred (Rel ∣ 𝑨 ∣ 𝓤) _
 con 𝑨 = λ θ → IsEquivalence θ × compatible 𝑨 θ

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

The trivial congruence

We construct the "trivial" or "diagonal" or "identity" relation and prove it is a congruence as follows.

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

_╱_ : (𝑨 : Algebra 𝓤 𝑆) → Congruence 𝑨
      ---------------------------------
 →    Algebra (𝓤 ⁺) 𝑆

𝑨 ╱ θ = (( | 𝑨 | // ⟨ θ ⟩ ) , -- carrier
          (λ 𝑓 args         -- operations
            → ([ (𝑓 ̂ 𝑨) (λ i₁ → | ‖ args i₁ ‖ |) ] ⟨ θ ⟩) ,
              ((𝑓 ̂ 𝑨)(λ i₁ → | ‖ args i₁ ‖ |) , refl _ )
          )
         )
```

We would like to round out this chapter with a formalization of the trivial congruence of the free algebra $\mathbb{F}(\mathscr{K}, X)$, which we called $\Psi(\mathscr{K}, \mathbf{T}(X))$ in free algebras.

Unfortunately, this will have to wait until we have formalized the concepts of subalgebra and closure on which this congruence depends. Thus, our Agda definition of $\Psi(\mathscr{K}, \mathbf{T}(X))$ will appear in the closure module described in Chapter %s equational logic in agda.

Unicode Hints 2

Table of some special characters used in the congruences module.

To get	Type
≈	`\~~` or `\approx`
⇒	`\r2` or `\=>`
𝟎, 𝟏	`\B0`, `\B1`
θ, Δ	`\theta`, `\Delta`
⟨_⟩	`\<_\>`
╱	`\---` then right arrow a number of times

Emacs commands providing information about special characters/input methods:

- `M-x describe-char` (or `M-m h d c`) with the cursor on the character of interest
- `M-x describe-input-method` (or `C-h I`)

Table of contents ⇑

Homomorphisms in Agda

This chapter describes the homomorphisms module of the UALib.

Types for homomorphisms

Our implementation of the notion of homomorphisms in the UALib is an extensional one. What this means will become clear once we have presented the definitions (cf. Homomorphisms intensionally <homomorphisms intensionally>).

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

```
module homomorphisms {𝑆 : Signature 𝓞 𝓥} where

 open prelude using (_∘_; _∈_; _⊆_; EpicInv; cong-app; EInvIsRInv; Image_∋_) public

 op_interpreted-in_and_commutes-with :
  (𝑓 : | 𝑆 |) (𝑨 : Algebra 𝓤 𝑆) (𝑩 : Algebra 𝓦 𝑆)
  (g : | 𝑨 |  → | 𝑩 |) → 𝓥 ⊔ 𝓤 ⊔ 𝓦 ̇

 op 𝑓 interpreted-in 𝑨 and 𝑩 commutes-with g =
  ∀( 𝒂 : ‖ 𝑆 ‖ 𝑓 → | 𝑨 | ) → g ((𝑓 ̂ 𝑨) 𝒂) ≡ (𝑓 ̂ 𝑩) (g ∘ 𝒂)
```

```
all-ops-in_and_commute-with :
 (𝑨 : Algebra 𝓤 𝑆) (𝑩 : Algebra 𝓦 𝑆)
  →   (| 𝑨 |  → | 𝑩 | ) → 𝓞 ⊔ 𝓥 ⊔ 𝓤 ⊔ 𝓦 ̇

all-ops-in 𝑨 and 𝑩 commute-with g = ∀ (𝑓 : | 𝑆 |)
 → op 𝑓 interpreted-in 𝑨 and 𝑩 commutes-with g

is-homomorphism : (𝑨 : Algebra 𝓤 𝑆) (𝑩 : Algebra 𝓦 𝑆)
 →                (| 𝑨 | → | 𝑩 |) → 𝓞 ⊔ 𝓥 ⊔ 𝓤 ⊔ 𝓦 ̇

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

And now we define the type of homomorphisms.

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

An example of such a homomorphism is the identity map.

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

Composition

The composition of homomorphisms is again a homomorphism.

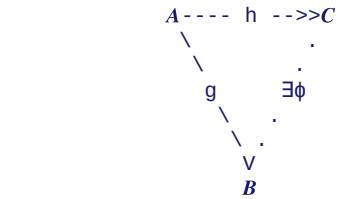
```
HCompClosed : {𝑨 : Algebra 𝓤 𝑆}{𝑩 : Algebra 𝓦 𝑆}{𝑪 : Algebra 𝓣 𝑆}
 →            hom 𝑨 𝑩   →   hom 𝑩 𝑪
              ---------------------
 →                  hom 𝑨 𝑪

HCompClosed {𝑨 = 𝑨}{𝑩 = 𝑩}{𝑪 = 𝑪}
 (g , ghom) (h , hhom) = h ∘ g , γ
  where
   γ : (𝑓 : | 𝑆 |) (𝒂 : ∥ 𝑆 ∥ 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)  ∎
```

Factorization

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`, that is, such that the following diagram commutes;



We now formalize the statement and proof of this basic fact. (Notice that the proof is fully constructive.)

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

homFactor fe {𝑨 = 𝑨}{𝑩 = 𝑩}{𝑪 = 𝑪}
 (g , ghom) (h , hhom) Kh⊆Kg hEpic = (ϕ , ϕIsHomCB) , g≡ϕ∘h
  where
```

```
hInv : | C | → | A |
hInv = λ c → (EpicInv h hEpic) c

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

ξ : (x : | A |) → ker-pred h (x , hInv (h x))
ξ x =  ( cong-app (EInvIsRInv fe h hEpic) ( 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 (EInvIsRInv fe h hEpic) (c x))⁻¹

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

ι f c = ap (λ - → - ∘ c)(EInvIsRInv fe h hEpic)⁻¹

useker : (f : | S |)  (c : ∥ S ∥ f → | C |)
 → g (hInv (h ((f ̂ A)(hInv ∘ c)))) ≡ g ((f ̂ A) (hInv ∘ c))

useker = λ f c
 → Kh⊆Kg (cong-app
          (EInvIsRInv fe h hEpic)
          (h ((f ̂ A)(hInv ∘ c))))


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

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 (λ - → g (hInv -)) (hhom f (hInv ∘ c))⁻¹
  iii = useker f c
  iv  = ghom f (hInv ∘ c)
```

Isomorphism

```
_≅_ : (A B : Algebra 𝓤 S) → 𝓤 ⊔ 𝓞 ⊔ 𝓥 ̇
A ≅ B =  Σ f ꞉ (hom A B) , Σ g ꞉ (hom B A) ,
           (| f | ∘ | g | ≡ | 𝒾𝒹 B |) × (| g | ∘ | f | ≡ | 𝒾𝒹 A |)
```

Homomorphic images

The following seem to be (for our purposes) the two most useful types for representing homomomrphic images of an algebra.

```
HomImage : {A : Algebra 𝓤 S} (B : Algebra 𝓤 S)(ϕ : hom A B) → | B | → 𝓤 ̇

HomImage B ϕ = λ b → Image | ϕ | ∋ b


HomImagesOf : {𝓤 : Universe} → Algebra 𝓤 S → 𝓞 ⊔ 𝓥 ⊔ 𝓤 ⁺ ̇

HomImagesOf {𝓤} A = Σ B ꞉ (Algebra 𝓤 S) , Σ ϕ ꞉ (| A | → | B |) ,
                      is-homomorphism A B ϕ × Epic ϕ
```

Here are some further definitions, derived from the one above, that will come in handy later.

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

B is-hom-image-of A = Σ Cϕ ꞉ (HomImagesOf A) , B ≅ | Cϕ |

_is-hom-image-of-class_ : {𝓤 : Universe}
```

```
  →                      Algebra 𝓤 𝑆
  →                      Pred (Algebra 𝓤 𝑆) (𝓤 ⁺)
  →                      𝓞 ⊔ 𝓥 ⊔ 𝓤 ⁺ ˙

_is-hom-image-of-class_ {𝓤} 𝑩 𝒦 = Σ 𝑨 ꞉ (Algebra 𝓤 𝑆) ,
                                  (𝑨 ∈ 𝒦) × (𝑩 is-hom-image-of 𝑨)

HomImagesOfClass : Pred (Algebra 𝓤 𝑆) (𝓤 ⁺) → 𝓞 ⊔ 𝓥 ⊔ 𝓤 ⁺ ˙

HomImagesOfClass 𝒦 = Σ 𝑩 ꞉ (Algebra _ 𝑆) , (𝑩 is-hom-image-of-class 𝒦)

H : Pred (Algebra 𝓤 𝑆) (𝓤 ⁺) → 𝓞 ⊔ 𝓥 ⊔ 𝓤 ⁺ ˙
H 𝒦 = HomImagesOfClass 𝒦
```

In the following definition $\mathscr{LK}$ represents a (universe-indexed) collection of classes.

```
H-closed : (ℒ𝒦 : (𝓤 : Universe) → Pred (Algebra 𝓤 𝑆) (𝓤 ⁺))
 →         (𝓤 : Universe) → Algebra 𝓤 𝑆
 →          𝓞 ⊔ 𝓥 ⊔ 𝓤 ⁺ ˙

H-closed ℒ𝒦 =
 λ 𝓤 𝑩 → _is-hom-image-of-class_ {𝓤 = 𝓤} 𝑩 (ℒ𝒦 𝓤) → 𝑩 ∈ (ℒ𝒦 𝓤)
```

Unicode Hints 3

Table of some special characters used in the homomorphisms module.

To get	Type
$\boldsymbol{a}$, $\boldsymbol{b}$	`\MIa`, `\MIb`
$f\hat{}\boldsymbol{A}$	`\Mif \^ \MIA`
≅	`\~=` or `\cong`
∘	`\comp` or `\circ`
id	`\Mci\Mcd`
$\mathscr{LK}$	`\McL\McK`
ϕ	`\phi`

Emacs commands providing information about special characters/input methods:

- `M-x describe-char` (or `M-m h d c`) with the cursor on the character of interest
- `M-x describe-input-method` (or `C-h I`)

Table of contents ⇑

Terms in Agda

This chapter describes the terms module of the UALib.

Types for terms

We start by declaring the module and importing the required dependencies.

```
open basic
open congruences
open prelude using (global-dfunext)

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

 open homomorphisms {𝑆 = 𝑆}

 open prelude using (intensionality; pr₂; Inv; InvIsInv; eq; fst; snd; 𝓇𝑒𝒻𝓁; _⁻¹∙_) public
```

Next, we define a datatype called `Term` which, not surprisingly, represents the type of terms. The type `X : 𝓧 ˙` represents an arbitrary collection of "variables."

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

open Term
```

The term algebra

The term algebra was described informally in terms. We denote this important algebra by $\boldsymbol{T}(X)$ and we implement it in Agda as follows.

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

term-op : {𝓤 : Universe}{X : 𝓤 ̇}(f : ∣ 𝑆 ∣)(args : ∥ 𝑆 ∥ f → Term{𝓤}{X} ) → Term
term-op f args = node f args
```

The universal property

We prove

1. every map `h : X → ∣ 𝑨 ∣` lifts to a homomorphism from $\boldsymbol{T}(X)$ to $\boldsymbol{A}$, and
2. the induced homomorphism is unique.

First, every map `X → ∣ 𝑨 ∣` lifts to a homomorphism.

```
--1.a. Every map (X → 𝑨) lifts.
free-lift : {𝓤 𝓦 : Universe}{X : 𝓤 ̇}{𝑨 : Algebra 𝓦 𝑆} (h : X → ∣ 𝑨 ∣)
 →          ∣ (𝑻{𝓤}{X}) ∣ → ∣ 𝑨 ∣

free-lift {X = X} h (generator x) = h x
free-lift {𝑨 = 𝑨} h (node f args) = (f ̂ 𝑨) λ i → free-lift{𝑨 = 𝑨} h (args i)

--1.b. The lift is (extensionally) a hom
lift-hom : {𝓤 𝓦 : Universe}{X : 𝓤 ̇}{𝑨 : Algebra 𝓦 𝑆}(h : X → ∣ 𝑨 ∣)
 →         hom (𝑻{𝓤}{X}) 𝑨

lift-hom {𝑨 = 𝑨} h = free-lift{𝑨 = 𝑨} h , λ f a → ap (_ ̂ 𝑨) 𝓻𝓮𝓯𝓵
```

Next, the lift to $(\boldsymbol{T}\ X \to \boldsymbol{A})$ is unique.

```
--2. The lift to (free → 𝑨) is (extensionally) unique.
free-unique : {𝓤 𝓦 : Universe}{X : 𝓤 ̇} → funext 𝒱 𝓦
 →            {𝑨 : Algebra 𝓦 𝑆}(g h : hom (𝑻{𝓤}{X}) 𝑨)
 →            (∀ x → ∣ g ∣ (generator x) ≡ ∣ h ∣ (generator x))
 →            (t : Term{𝓤}{X})
              ---------------------------
 →            ∣ g ∣ t ≡ ∣ h ∣ t

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

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

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

lift-agrees-on-X h₀ x = 𝓻𝓮𝓯𝓵

--Of course, the lift of a surjective map is surjective.
lift-of-epic-is-epic : {𝓤 : Universe}{X : 𝓤 ̇}{𝑨 : Algebra 𝓤 𝑆}(h₀ : X → ∣ 𝑨 ∣)
 →                     Epic h₀
                      ----------------------
 →                     Epic ∣ lift-hom{𝑨 = 𝑨} h₀ ∣

lift-of-epic-is-epic{X = X}{𝑨 = 𝑨} h₀ hE y = γ
 where
```

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

  h₀⁻¹y : X
  h₀⁻¹y = Inv h₀ y (hE y)

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

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

Finally, we prove that for every S-algebra $\mathbf{C}$, there exists an epimorphism from $\mathbf{T}$ onto $\mathbf{C}$.

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

  hE : Epic h₀
  hE = snd (𝕏 𝑪)

  h : hom 𝑻 𝑪
  h = lift-hom{𝑨 = 𝑪} h₀
```

Interpretation

Let `t : Term` be a term and $\mathbf{A}$ an S-algebra. We define the n-ary operation `t ̇ 𝑨` on $\mathbf{A}$ by structural recursion on `t`.

1. if `t = x ∈ X` (a variable) and `a : X → ∣ 𝑨 ∣` is a tuple of elements of `∣ 𝑨 ∣`, then `(t ̇ 𝑨) a = a x`.
2. if `t = f args`, where `f ∈ ∣ 𝑆 ∣` is an op symbol and `args : ∥ 𝑆 ∥ f → Term` is an `(∥ 𝑆 ∥ f)`-tuple of terms and `a : X → ∣ 𝑨 ∣` is a tuple from $\mathbf{A}$, then `(t ̇ 𝑨) a = ((f args) ̇ 𝑨) a = (f ̂ 𝑨) λ{ i → ((args i) ̇ 𝑨) a }`

```
_̇_ : {𝓤 𝓦 : Universe}{X : 𝓤 ̇ } → Term{𝓤}{X}
 →   (𝑨 : Algebra 𝓦 𝑆) → (X → ∣ 𝑨 ∣) → ∣ 𝑨 ∣

((generator x) ̇ 𝑨) a = a x

((node f args) ̇ 𝑨) a = (f ̂ 𝑨) λ i → (args i ̇ 𝑨) a
```

Next we show that if `p : ∣ 𝑻(X) ∣` is a term, then there exists f `: ∣ 𝑻(X) ∣` and t `: X → ∣ 𝑻(X) ∣` such that

`p ≡ (`f ̇ `𝑻(X)) `t.

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

term-op-interp1 = λ f args → 𝓇ℯ𝒻𝓁

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

term-op-interp2 f a1≡a2 = ap (node f) a1≡a2


term-op-interp3 : {𝓤 : Universe}{X : 𝓤 ̇ }(f : ∣ 𝑆 ∣){a1 a2 : ∥ 𝑆 ∥ f → Term{𝓤}{X}}
                  ---------------------------------------------------------------
 →                 a1 ≡ a2   →   node f a1 ≡ (f ̂ 𝑻) a2

term-op-interp3 f {a1}{a2} a1≡a2 =
 node f a1     ≡⟨ term-op-interp2 f a1≡a2 ⟩
 node f a2     ≡⟨ term-op-interp1 f a2 ⟩
 (f ̂ 𝑻) a2     ∎

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

term-gen (generator x) = (generator x) , 𝓇ℯ𝒻𝓁
```

```
term-gen (node f args) =
  node f (λ i → ∣ term-gen (args i) ∣ ) ,
    term-op-interp3 f (gfe λ i → ∥ term-gen (args i) ∥)

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

term-gen-agreement : {𝓤 : Universe}{X : 𝓤 ̇}(p : ∣ T{𝓤}{X} ∣)
 →                   (p ̇ 𝑻)generator ≡ (∣ term-gen p ∣ ̇ 𝑻)generator

term-gen-agreement (generator x) = refl
term-gen-agreement (node f args) = ap (f ̂ 𝑻) (gfe λ x → term-gen-agreement (args x))

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

Here are some definitions that are useful when dealing with the interpretations of terms in a product structure.

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

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

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

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

interp-prod2 fe (generator x₁) 𝒜 = refl

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

Compatibility of terms

In this section we present the formal proof of the fact that homomorphisms commute with terms. More precisely, if $\boldsymbol{A}$ and $\boldsymbol{B}$ are S-algebras, $h : \boldsymbol{A} \to \boldsymbol{B}$ a homomorphism, and t a term in the language of S, then for all $a : X \to |\, \boldsymbol{A} \,|$ we have

$h\,(t^{\boldsymbol{A}}\, a) = t^{\boldsymbol{B}}\,(h \circ a)$.

Homomorphisms commute with terms

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

comm-hom-term {𝓤}{𝓦}{𝓧}{X} fe 𝑨 𝑩 h (generator x) a = refl

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

Congruences commute with terms

Rounding out this chapter is an formal proof of the fact that terms respect congruences. More precisely, we show that for every term `t`, every `θ ∈ Con(A)`, and all tuples `a, b : X → A`, we have

(∀ x, a(x) θ b(x)) → (t ̇ **A**) a θ (t ̇ **A**) b.

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

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

compatible-term 𝑨 (node f args) θ p = pr₂( ∥ θ ∥ ) f λ{x → (compatible-term 𝑨 (args x) θ) p}
```

Table of contents ⇑

Subalgebras in Agda

This chapter describes the subuniverses module of the UALib.

We define subuniverses and subalgebras and prove some basic facts about them in this, the subuniverses.lagda.rst file of the UALib.

Preliminaries

The subuniverses.lagda.rst file starts, as usual, by fixing a signature *S* and satisfying some dependencies.

```
open basic
open congruences
open prelude using (global-dfunext)

module subuniverses {𝑆 : Signature 𝓞 𝓥}
                    {𝕏 : {𝓤 𝓧 : Universe}{X : 𝓧 ̇}(𝑨 : Algebra 𝓤 𝑆) → X ↠ 𝑨}
                    {fe : global-dfunext} where

 open homomorphisms {𝑆 = 𝑆}

 open terms {𝑆 = 𝑆}{𝕏 = 𝕏}{gfe = fe} renaming (generator to ℊ)

 open import Relation.Unary using (⋂)

 open prelude using (Im_⊆_; Univalence; embeddings-are-lc; univalence-gives-global-dfunext;
  𝒫; _∈₀_; _⊆₀_; pr₁; domain; is-subsingleton; Π-is-subsingleton;is-equiv; lr-implication;
  ×-is-subsingleton; ∈-is-subsingleton; is-embedding; pr₁-embedding; rl-implication; inverse;
  embedding-gives-ap-is-equiv; is-set;_⇔_;transport; subset-extensionality'; equiv-to-subsingleton;
  powersets-are-sets'; _≃_; id; _∙_; logically-equivalent-subsingletons-are-equivalent) public
```

Types for subuniverses

We begin the subuniverses module with a straightforward definition of the collection of subuniverses of an algebra A. Since a subuniverse is a subset of the domain of A, it is defined as a predicate on ∣ A ∣. Thus, the collection of subuniverses is a predicate on predicates on ∣ A ∣.

```
 Subuniverses : (𝑨 : Algebra 𝓤 𝑆) → Pred (Pred ∣ 𝑨 ∣ 𝓣) (𝓞 ⊔ 𝓥 ⊔ 𝓤 ⊔ 𝓣)

 Subuniverses 𝑨 B = (f : ∣ 𝑆 ∣)(a : ∥ 𝑆 ∥ f → ∣ 𝑨 ∣) → Im a ⊆ B → (f ̂ 𝑨) a ∈ B
```

Subuniverse generation

Next we formalize the important theorem about subuniverse generation. Recall, if $\mathbf{A} = \langle A, \ldots \rangle$ is an S-algebra, if $\emptyset \neq A_0 \subseteq A$, and if we define by recursion the sets $A_{n+1} = A_n \cup \{f\, a \mid f : |S|, a : \|S\|\, f \to A_n\}$, then the subuniverse of A generated by A_0 is $\mathrm{Sg}^{\mathbf{A}}(A_0) = \bigcup_n A_n$.

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

```
data Sg (𝑨 : Algebra 𝓤 𝑆) (X : Pred ∣ 𝑨 ∣ 𝓣) : Pred ∣ 𝑨 ∣ (𝓞 ⊔ 𝓥 ⊔ 𝓤 ⊔ 𝓣) 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
```

Of course, we should be able to prove that Sg $\boldsymbol{A}$ X is indeed a subuniverse of $\boldsymbol{A}$.

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

And, as the subuniverse *generated by X*, it had better be the smallest subuniverse of $\boldsymbol{A}$ containing X. We prove this by induction, as follows:

```
sgIsSmallest : {𝑨 : Algebra 𝓤 𝑆}{X : Pred ∣ 𝑨 ∣ 𝓡} {Y : Pred ∣ 𝑨 ∣ 𝓢}
 →             Y ∈ Subuniverses 𝑨  →   X ⊆ Y
              ------------------------------
 →                    Sg 𝑨 X ⊆ Y

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

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

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

Closure under intersection

Recall that the intersection $\bigcap_i A_i$ of a collection $\{A_i \mid A_i \leq \boldsymbol{A}\}$ of subuniverses of an algebra $\boldsymbol{A}$ is again a subuniverse of $\boldsymbol{A}$. We formalize the statement and proof of this easy fact in Agda as follows.

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

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

Generation with terms

Recall that subuniverse can be generated using the images of terms: If Y is a subset of A, then

$\mathrm{Sg}^{\boldsymbol{A}}(Y) = \{t^{\boldsymbol{A}}\, a : t \in \boldsymbol{T}(X), a : X \to Y\}$.

To formalize this idea in Agda, we first prove that subuniverses are closed under the action of term operations.

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

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

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

This proves $\mathrm{Sg}^{\boldsymbol{A}}(Y) \supseteq \{t^{\boldsymbol{A}}\, a : t \in T(X), a : X \to Y\}$.

Next we prove $\mathrm{Sg}^{\boldsymbol{A}}(Y) \subseteq \{t^{\boldsymbol{A}}\, a : t \in T(X), a : X \to Y\}$ by the following steps:

1. The image of Y under all terms, which we call TermImage Y, is a subuniverse of $\mathbf{A}$; i.e.,

   TermImage $Y = \{t^{\mathbf{A}}\ a : t \in T(X), a : X \to Y\} \leq \mathbf{A}$.

2. $Y \subseteq$ TermImage Y (obvious)

3. $\mathrm{Sg}^{\mathbf{A}}(Y)$ is the smallest subuniverse containing Y (see sgIsSmallest) so

   $\mathrm{Sg}^{\mathbf{A}}(Y) \subseteq$ TermImage Y.

(The last item was already proved above; see `sgIsSmallest`.)

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

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

TermImageIsSub = λ f a x → app f a x

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

Y⊆TermImageY {a} a∈Y = var a∈Y

-- 3. Sg^A(Y) is the smallest subuniverse containing Y. (Proof: see `sgIsSmallest`)
```

Finally, we can prove the desired inclusion.

```
SgY⊆TermImageY : {𝑨 : Algebra 𝓤 𝑆}{Y : Pred ∣ 𝑨 ∣ 𝓤}
 →               Sg 𝑨 Y ⊆ TermImage 𝑨 Y
SgY⊆TermImageY = sgIsSmallest TermImageIsSub Y⊆TermImageY
```

Homomorphic images are subuniverses

In this subsection we show that the image of an (extensional) homomorphism is a subuniverse. Before implementing the result formally in Agda, let us recall the steps of the informal proof.

Let f be an operation symbol, let $b : \| S \| f \to | B |$ be a $(\| S \| f)$-tuple of elements of $| \mathbf{B} |$, and assume the image Im b of b belongs to the image `Image h` of h. We must show that $f^{\mathbf{B}}\ b \in$ Image h. The assumption Im $b \subseteq$ Image h implies that there is a $(\| S \| f)$-tuple $a : \| S \| f \to | \mathbf{A} |$ such that $h \circ a = b$. Since h is a homomorphism, we have $f^{\mathbf{B}}\ b = f^{\mathbf{B}}\ (h \circ a) = h\ (f^{\mathbf{A}}\ a) \in$ Image h.

Recall the definition of `HomImage` from the homomorphisms module,

```
HomImage : ∣ B ∣ → 𝓤 ̇
HomImage = λ b → Image ∣ h ∣ ∋ b
```

We are now ready to formalize the proof that homomorphic images are subuniverses.

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

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

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

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

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

Types for subalgebras

Finally, we define, once and for all, the type of subalgebras of an algebra (resp., subalgebras of algebras in a class of algebras) that we will use in the sequel.

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

SubalgebrasOfClass : {𝓢 : Universe} → Pred (Algebra 𝓢 𝓢)(𝓢 ⁺) → 𝓞 ⊔ 𝓥 ⊔ 𝓢 ⁺ ̇
SubalgebrasOfClass 𝒦 = Σ 𝑨 ꞉ (Algebra _ 𝓢) , (𝑨 ∈ 𝒦) × SubalgebrasOf 𝑨

SubalgebrasOfClass' : {𝓢 : Universe} → Pred (Algebra 𝓢 𝓢)(𝓞 ⊔ 𝓥 ⊔ 𝓢 ⁺) → 𝓞 ⊔ 𝓥 ⊔ 𝓢 ⁺ ̇
SubalgebrasOfClass' 𝒦 = Σ 𝑨 ꞉ (Algebra _ 𝓢) , (𝑨 ∈ 𝒦) × SubalgebrasOf 𝑨
```

Unicode Hints 4

Table of some special characters used in the subuniverses module.

To get	Type
$\mathcal{I}$, $\mathcal{T}$	`\MCI`, `\MCT`
_⊧_≈_	`_\models_\~~_`
_⊧_≋_	`_\models_\~~~_`
⊆	`\subseteq` or `\sub=`
⋂	`\bigcap` or `\I`
ξ	`\xi`

Emacs commands providing information about special characters/input methods:

- `M-x describe-char` (or `M-m h d c`) with the cursor on the character of interest
- `M-x describe-input-method` (or `C-h I`)

Table of contents ⇑

Equational Logic in Agda

This chapter describes the closure module of the agda-ualib.

Closure operators and varieties

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

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

As a straight-forward verification confirms, H, S, and P are closure operators. A class $\mathscr{K}$ of S-algebras is said to be *closed under the formation of homomorphic images* if H($\mathscr{K}$) $\subseteq \mathscr{K}$. Similarly, $\mathscr{K}$ is *closed under the formation of subalgebras* (resp., *products*) provided S($\mathscr{K}$) $\subseteq \mathscr{K}$ (resp., P($\mathscr{K}$) $\subseteq \mathscr{K}$).

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

The operators H, S, and P can be composed with one another repeatedly, forming yet more closure operators. If C_1 and C_2 are closure operators on classes of structures, let us say that $C_1 \leq C_2$ if for every class $\mathscr{K}$ we have $C_1(\mathscr{K}) \subseteq C_2(\mathscr{K})$.

A class $\mathscr{K}$ of S-algebras is called a **variety** if it is closed under each of the closure operators H, S, and P introduced above; the corresponding closure operator is often denoted $\mathbb{V}$. Thus, if $\mathscr{K}$ is a class of similar algebras, then the **variety generated by** $\mathscr{K}$ is denoted by $\mathbb{V}(\mathscr{K})$ and defined to be the smallest class that contains $\mathscr{K}$ and is closed under H, S, and P.

We would like to know how to construct $\mathbb{V}(\mathscr{K})$ directly from $\mathscr{K}$, but it's not immediately obvious how many times we would have to apply the operators H, S, P before the result stabilizes to form a variety—the **variety generated by** $\mathscr{K}$. Fortunately, Garrett Birkhoff proved that if we apply the operators in the correct order, then it suffices to apply each one only once.

Types for identities

In his treatment of Birhoff's HSP theorem, Cliff Bergman (at the start of Section 4.4 of his universal algebra textbook Bergman:2012) proclaims, "Now, finally, we can formalize the idea we have been using since the first page of this text." He then proceeds to define **identities of terms** as follows (paraphrasing for notational consistency):

Let S be a signature. An **identity** or **equation** in S is an ordered pair of terms, written $p \approx q$, from the term algebra $\boldsymbol{T}(X)$. If A is an S-algebra we say that A **satisfies** $p \approx q$ if $p^{\cdot} A \equiv q^{\cdot} A$. In this situation, we write $A \models p \approx q$.

If $\mathscr{K}$ is a class of S-algebras, we write $\mathscr{K} \models p \approxeq q$ if, for every $A \in \mathscr{K}$, $A \models p \approx q$. Finally, if $\mathscr{E}$ is a set of equations, we write $\mathscr{K} \models \mathscr{E}$ if every member of $\mathscr{K}$ satisfies every member of $\mathscr{E}$.

We formalize these notions in Agda in the closure module, which begins as follows. (Note the imports that were postponed until after the start of the closure module so that the imports share the same signature S with the closure module.

```
open basic
open congruences
open prelude using (global-dfunext; dfunext; im; _∪_; inj₁; inj₂)

module closure
 {𝑆 : Signature 𝓞 𝓥}
 {X : 𝓤 ̇ }
 {𝒦 : Pred (Algebra 𝓤 𝑆)(𝓞 ⊔ 𝓥 ⊔ 𝓤 ⁺)}
 {𝕏 : {𝓤 𝓧 : Universe}{X : 𝓧 ̇ }(𝑨 : Algebra 𝓤 𝑆) → X ↠ 𝑨}
 {gfe : global-dfunext}
 {dfe : dfunext 𝓤 𝓤}
 {fevu : dfunext 𝓥 𝓤} where

 open homomorphisms {𝑆 = 𝑆} public
 open terms {𝑆 = 𝑆}{𝕏 = 𝕏}{gfe = gfe} renaming (generator to 𝑔) public
 open subuniverses {𝑆 = 𝑆}{𝕏 = 𝕏}{fe = gfe} public
```

Our first definition in the closure module is notation that represents the satisfaction of equations.

The standard notation is `𝑨 ⊧ p ≈ q`, which means that the identity `p ≈ q` is satisfied in $\boldsymbol{A}$. In otherwords, for all assignments `a : X → ∣ 𝑨 ∣` of values to variables, we have `(p ̇ 𝑨) a ≡ (q ̇ 𝑨) a`.

If $\mathscr{K}$ is a class of structures, it is standard to write `𝒦 ⊧ p ≈ q` just in case all structures in the class $\mathscr{K}$ model the identity p ≈ q. However, because a class of structures has a different type than a single structure, we will need different notation, so we have settled on writing `𝒦 ⊧ p ≋ q` to denote this concept.

```
 _⊧_≈_ : Algebra 𝓤 𝑆 → Term{𝓤}{X} → Term → 𝓤 ̇
 𝑨 ⊧ p ≈ q = (p ̇ 𝑨) ≡ (q ̇ 𝑨)

 _⊧_≋_ : Pred (Algebra 𝓤 𝑆) (𝓞 ⊔ 𝓥 ⊔ 𝓤 ⁺) → Term → Term → 𝓞 ⊔ 𝓥 ⊔ 𝓤 ⁺ ̇
 _⊧_≋_ 𝒦 p q = {𝑨 : Algebra _ 𝑆} → 𝒦 𝑨 → 𝑨 ⊧ p ≈ q
```

Equational theories and classes

Here we define the notation `Th` for the identities satisfied by all structures in a given class, and `Mod` for all structures that satisfy a given collection of identities.

```
 Th :  Pred (Algebra 𝓤 𝑆) (𝓞 ⊔ 𝓥 ⊔ 𝓤 ⁺) → Pred (Term × Term) (𝓞 ⊔ 𝓥 ⊔ 𝓤 ⁺)
 Th 𝒦 = λ (p , q) → 𝒦 ⊧ p ≋ q

 Mod : Pred (Term × Term) (𝓞 ⊔ 𝓥 ⊔ 𝓤 ⁺) → Pred (Algebra 𝓤 𝑆) (𝓞 ⊔ 𝓥 ⊔ 𝓤 ⁺)
 Mod ℰ = λ A → ∀ p q → (p , q) ∈ ℰ → A ⊧ p ≈ q
```

Compatibility of identities

Identities are compatible with the formation of subalgebras, homomorphic images and products. More precisely, for every class $\mathscr{K}$ of structures, each of the classes S($\mathscr{K}$), H($\mathscr{K}$), P($\mathscr{K}$), $\mathbb{V}(\mathscr{K})$ satisfies the same set of identities as does $\mathscr{K}$.

Here we formalize the notion of closure under the taking of products, subalgebras, and homomorphic images, and we prove that each of these closures preserves identities.

First a data type that represents classes of algebraic structures that are closed under the taking of products of algebras in the class can be defined in Agda as follows.

```
 --Closure under products
 data PClo : Pred (Algebra 𝓤 𝑆) (𝓞 ⊔ 𝓥 ⊔ 𝓤 ⁺) where
```

```
 pbase : {𝑨 : Algebra _ 𝑆} → 𝑨 ∈ 𝒦 → 𝑨 ∈ PClo
 prod : {I : 𝓤 ̇ }{𝒜 : I → Algebra _ 𝑆} → (∀ i → 𝒜 i ∈ PClo) → ⨅ 𝒜 ∈ PClo
```

A datatype that represents classes of structures that are closed under the taking of subalgebras is

```
-- Subalgebra Closure
data SClo : Pred (Algebra 𝓤 𝑆) (𝓞 ⊔ 𝓥 ⊔ 𝓤 ⁺ ) where
 sbase : {𝑨 :  Algebra _ 𝑆} → 𝑨 ∈ 𝒦 → 𝑨 ∈ SClo
 sub : {𝑨 : Algebra _ 𝑆} → 𝑨 ∈ SClo → (sa : SubalgebrasOf 𝑨) → ∣ sa ∣ ∈ SClo
```

Next, a datatype representing classes of algebras that are closed under homomorphic images of algebras in the class,

```
--Closure under hom images
data HClo : Pred (Algebra 𝓤 𝑆) (𝓞 ⊔ 𝓥 ⊔ 𝓤 ⁺) where
 hbase : {𝑨 : Algebra _ 𝑆} → 𝑨 ∈ 𝒦 → 𝑨 ∈ HClo
 hhom : {𝑨 : Algebra _ 𝑆} → 𝑨 ∈ HClo → ((𝑩 , _ ) : HomImagesOf 𝑨) → 𝑩 ∈ HClo
```

And, finally, an inductive type representing classes that are closed under all three, H, S, and P,

```
-- Variety Closure
data VClo : Pred (Algebra 𝓤 𝑆)(𝓞 ⊔ 𝓥 ⊔ 𝓤 ⁺ ) where
 vbase : {𝑨 : Algebra 𝓤 𝑆} → 𝑨 ∈ 𝒦 → 𝑨 ∈ VClo
 vprod : {I : 𝓤 ̇ }{𝒜 : I → Algebra _ 𝑆} → (∀ i → 𝒜 i ∈ VClo) → ⨅ 𝒜 ∈ VClo
 vsub : {𝑨 : Algebra 𝓤 𝑆} → 𝑨 ∈ VClo → (sa : SubalgebrasOf 𝑨) → ∣ sa ∣ ∈ VClo
 vhom : {𝑨 : Algebra 𝓤 𝑆} → 𝑨 ∈ VClo → ((𝑩 , _ , _) : HomImagesOf 𝑨) → 𝑩 ∈ VClo
```

Products preserve identities

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

```
products-preserve-identities : (p q : Term{𝓤}{X}) (I : 𝓤 ̇ ) (𝒜 : I → Algebra 𝓤 𝑆)
 →                             ((i : I) → (𝒜 i) ⊨ p ≈ q)
                               ----------------------------------------------------
 →                             ⨅ 𝒜 ⊨ p ≈ q

products-preserve-identities p q I 𝒜 𝒜⊨p≈q = γ
  where
   γ : (p ̇ ⨅ 𝒜) ≡ (q ̇ ⨅ 𝒜)
   γ = gfe λ a →
    (p ̇ ⨅ 𝒜) a                              ≡⟨ interp-prod{𝓤 = 𝓤} fevu p 𝒜 a ⟩
    (λ i → ((p ̇ (𝒜 i)) (λ x → (a x) i))) ≡⟨ gfe (λ i → cong-app (𝒜⊨p≈q i) (λ x → (a x) i)) ⟩
    (λ i → ((q ̇ (𝒜 i)) (λ x → (a x) i))) ≡⟨ (interp-prod gfe q 𝒜 a)⁻¹ ⟩
    (q ̇ ⨅ 𝒜) a                              ∎

products-in-class-preserve-identities : (p q : Term{𝓤}{X}) (I : 𝓤 ̇ ) (𝒜 : I → Algebra 𝓤 𝑆)
 →                                      𝒦 ⊨ p ≋ q  →  ((i : I) → 𝒜 i ∈ 𝒦)
                                        ----------------------------------------------------
 →                                       ⨅ 𝒜 ⊨ p ≈ q

products-in-class-preserve-identities p q I 𝒜 𝒦⊨p≋q all𝒜i∈𝒦 = γ
  where
   𝒜⊨p≈q : ∀ i → (𝒜 i) ⊨ p ≈ q
   𝒜⊨p≈q i = 𝒦⊨p≋q (all𝒜i∈𝒦 i)

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

Subalgebras preserve identities

We now show that every term equation, `p ≈ q`, that is satisfied by all algebras in $\mathscr{K}$ is also satisfied by every subalgebra of every member of $\mathscr{K}$. In other words, the collection of identities modeled by a given class of algebras is also modeled by all of the subalgebras of algebras in that class.

```
subalgebras-preserve-identities : {𝒦 : Pred (Algebra 𝓤 𝑆)(𝓞 ⊔ 𝓥 ⊔ 𝓤 ⁺)}(p q : Term)
                                  ((_ , _ , (𝑩 , _ , _)) : SubalgebrasOfClass' 𝒦)
 →                                𝒦 ⊨ p ≋ q
                                  -------------
 →                                𝑩 ⊨ p ≈ q

subalgebras-preserve-identities {𝒦} p q (𝑨 , KA , (𝑩 , h , (hem , hhm))) Kpq = γ
 where
  β : 𝑨 ⊨ p ≈ q
  β = Kpq KA
```

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

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

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

Closure under hom images

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

$\mathcal{K} \models p \approx q \Leftrightarrow \forall \mathbf{A} \in \mathcal{K}, \forall h \in \mathrm{Hom}(\mathbf{T}(X), \mathbf{A}), h \circ p^{\mathbf{T}(X)} = h \circ q^{\mathbf{T}(X)}$.

We now formalize this result in Agda. We begin with the "only if" direction.

```
identities-compatible-with-homs : (p q : Term{𝓤}{X})
                                  (p≋q : 𝒦 ⊧ p ≋ q)
                                 ---------------------------------------------------
 →                                ∀ (𝑨 : Algebra 𝓤 𝑆)(KA : 𝑨 ∈ 𝒦)(h : hom (𝑻{𝓤}{X}) 𝑨)
                                  →  ∣ h ∣ ∘ (p ̂ 𝑻{𝓤}{X}) ≡ ∣ h ∣ ∘ (q ̂ 𝑻)

identities-compatible-with-homs p q p≋q 𝑨 KA h = γ
 where
  pA≡qA : p ̂ 𝑨 ≡ q ̂ 𝑨
  pA≡qA = p≋q KA

  pAh≡qAh : ∀(𝒂 : X → ∣ 𝑻 ∣ ) → (p ̂ 𝑨)(∣ h ∣ ∘ 𝒂) ≡ (q ̂ 𝑨)(∣ h ∣ ∘ 𝒂)
  pAh≡qAh 𝒂 = intensionality pA≡qA (∣ h ∣ ∘ 𝒂)

  hpa≡hqa : ∀(𝒂 : X → ∣ 𝑻 ∣ ) → ∣ h ∣ ((p ̂ 𝑻) 𝒂) ≡ ∣ h ∣ ((q ̂ 𝑻) 𝒂)
  hpa≡hqa 𝒂 =
   ∣ h ∣ ((p ̂ 𝑻) 𝒂)  ≡⟨ comm-hom-term{𝒪 ⊔ 𝒱 ⊔ 𝓤 ⁺} fevu (𝑻{𝓤}{X}) 𝑨 h p 𝒂 ⟩
   (p ̂ 𝑨)(∣ h ∣ ∘ 𝒂) ≡⟨ pAh≡qAh 𝒂 ⟩
   (q ̂ 𝑨)(∣ h ∣ ∘ 𝒂) ≡⟨ (comm-hom-term{𝒪 ⊔ 𝒱 ⊔ 𝓤 ⁺} fevu 𝑻 𝑨 h q 𝒂)⁻¹ ⟩
   ∣ h ∣ ((q ̂ 𝑻) 𝒂)  ∎

  γ : ∣ h ∣ ∘ (p ̂ 𝑻) ≡ ∣ h ∣ ∘ (q ̂ 𝑻)
  γ = gfe hpa≡hqa
```

And now for the "if" direction.

```
homs-compatible-with-identities : (p q : Term{𝓤}{X})
                                  (hp≡hq : ∀ (𝑨 : Algebra 𝓤 𝑆)(KA : 𝑨 ∈ 𝒦) (h : hom (𝑻{𝓤}{X}) 𝑨)
                                           → ∣ h ∣ ∘ (p ̂ 𝑻) ≡ ∣ h ∣ ∘ (q ̂ 𝑻))
                                 ------------------------------------------------------------
 →                                𝒦 ⊧ p ≋ q

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

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

Of course, we can easily combine the last two results into a single "iff" theorem.

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

```
                                                  →    | hh | ∘ (p ̂ 𝑇) ≡ | hh | ∘ (q ̂ 𝑇))

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

Next we prove a fact that might seem obvious or at least intuitive—namely, that identities modeled by an algebra are compatible with the interpretation of terms in that algebra.

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

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

Identities for product closure

Next we prove

- `pclo-id1`: if every algebra in the class $\mathscr{K}$ satisfies a particular identity, say $p \approx q$, then every algebra in the closure PClo of $\mathscr{K}$ under the taking of arbitrary products also satisfies $p \approx q$, and, conversely,
- `pclo-id2`: if every algebra of the product closure PClo of $\mathscr{K}$ satisfies $p \approx q$, then so does every algebra in $\mathscr{K}$.

Here's proof of the first item.

```
pclo-id1 : ∀ {p q} → (𝒦 ⊧ p ≋ q) → (PClo ⊧ p ≋ q)
pclo-id1 {p} {q} α (pbase x) = α x
pclo-id1 {p} {q} α (prod{I}{𝒜} 𝒜-P𝒦 ) = γ
 where
  IH : (i : I)  → (p ̂ 𝒜 i) ≡ (q ̂ 𝒜 i)
  IH = λ i → pclo-id1{p}{q} α  ( 𝒜-P𝒦  i )

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

The second item is even easier to prove since $\mathscr{K} \subseteq$ PClo.

```
pclo-id2 : ∀{p q} → ((PClo) ⊧ p ≋ q ) → (𝒦 ⊧ p ≋ q)
pclo-id2 p A∈𝒦 = p (pbase A∈𝒦)
```

Identities for subalgebra closure

Here we prove

- `sclo-id1`: if every algebra in the class $\mathscr{K}$ satisfies $p \approx q$, then so does every algebra in the closure SClo of $\mathscr{K}$ under the taking of subalgebras; and, conversely,
- `sclo-id2`: if every algebra of the subalgebra closure SClo of $\mathscr{K}$ satisfies $p \approx q$, then so does every algebra in $\mathscr{K}$.

First we need to define a type that represents singletons containing exactly one algebra.

```
{_}  : Algebra 𝓤 𝑆 → Pred (Algebra 𝓤 𝑆)(𝓞 ⊔ 𝓥 ⊔ 𝓤 ⁺)
{ 𝑨 }  𝑩 = 𝑨 ≡ 𝑩
```

The formal statement and proof of the first item above is as follows.

```
sclo-id1 : ∀{p q} → (𝒦 ⊧ p ≋ q) → (SClo ⊧ p ≋ q)
sclo-id1 {p} {q} 𝒦⊧p≋q (sbase A∈𝒦) = 𝒦⊧p≋q A∈𝒦
sclo-id1 {p} {q} 𝒦⊧p≋q (sub {𝑨 = 𝑨} A∈SCl𝒦 sa) =

 -- (We apply subalgebras-preserve-identities to the class 𝒦 ∪ { 𝑨 } )
 subalgebras-preserve-identities p q (𝑨 , inj₂ 𝓇𝓮𝒻𝓁 , sa) 𝒦A⊧p≋q

  where
   A⊧p≈q : 𝑨 ⊧ p ≈ q
   A⊧p≈q = sclo-id1{p}{q} 𝒦⊧p≋q A∈SCl𝒦

   Asingleton⊧p≋q :  { 𝑨 }  ⊧ p ≋ q
```

```
  Asingleton⊨p≋q (refl _)  = A⊨p≈q

  𝒦A⊨p≋q : (𝒦 ∪  ｛ 𝑨 ｝ ) ⊨ p ≋ q
  𝒦A⊨p≋q {𝑩} (inj₁ x) = 𝒦⊨p≋q x
  𝒦A⊨p≋q {𝑩} (inj₂ y) = Asingleton⊨p≋q y
```

As with the analogous result for products, proving the second item from the list above is trivial.

```
sclo-id2 : ∀ {p q} → (SClo ⊨ p ≋ q) → (𝒦 ⊨ p ≋ q)
sclo-id2 p A∈𝒦 = p (sbase A∈𝒦)
```

Identities for hom image closure

We prove

- `hclo-id1`: if every algebra in the class $\mathscr{K}$ satisfies a $p \approx q$, then so does every algebra in the closure HClo of $\mathscr{K}$ under the taking of homomorphic images; and, conversely,
- `hclo-id2`: if every algebra of the homomorphic image closure HClo of $\mathscr{K}$ satisfies $p \approx q$, then so does every algebra in $\mathscr{K}$.

```
hclo-id1 : ∀{p q} → (𝒦 ⊨ p ≋ q) → (HClo ⊨ p ≋ q)
hclo-id1 {p}{q} α (hbase KA) = α KA
hclo-id1 {p}{q} α (hhom{𝑨} HCloA (𝑩 , ϕ , (ϕhom , ϕsur))) = γ
 where
  β : 𝑨 ⊨ p ≈ q
  β = (hclo-id1{p}{q} α) HCloA

  preim : (𝒃 : X → ∣ 𝑩 ∣)(x : X) → ∣ 𝑨 ∣
  preim 𝒃 x = (Inv ϕ (𝒃 x) (ϕsur (𝒃 x)))

  ζ : (𝒃 : X → ∣ 𝑩 ∣) → ϕ ∘ (preim 𝒃) ≡ 𝒃
  ζ 𝒃 = gfe λ x → InvIsInv ϕ (𝒃 x) (ϕsur (𝒃 x))

  γ : (p ̂ 𝑩) ≡ (q ̂ 𝑩)
  γ = gfe λ 𝒃 →
   (p ̂ 𝑩) 𝒃              ≡⟨ (ap (p ̂ 𝑩) (ζ 𝒃))⁻¹ ⟩
   (p ̂ 𝑩) (ϕ ∘ (preim 𝒃)) ≡⟨ (comm-hom-term gfe 𝑨 𝑩 (ϕ , ϕhom) p (preim 𝒃))⁻¹ ⟩
   ϕ((p ̂ 𝑨)(preim 𝒃))     ≡⟨ ap ϕ (intensionality β (preim 𝒃)) ⟩
   ϕ((q ̂ 𝑨)(preim 𝒃))     ≡⟨ comm-hom-term gfe 𝑨 𝑩 (ϕ , ϕhom) q (preim 𝒃) ⟩
   (q ̂ 𝑩)(ϕ ∘ (preim 𝒃))  ≡⟨ ap (q ̂ 𝑩) (ζ 𝒃) ⟩
   (q ̂ 𝑩) 𝒃               ∎

hclo-id2 : ∀ {p q} → (HClo ⊨ p ≋ q) → (𝒦 ⊨ p ≋ q)
hclo-id2 p KA = p (hbase KA)
```

Identities for HSP closure

Finally, we prove

- `vclo-id1`: if every algebra in the class $\mathscr{K}$ satisfies a $p \approx q$, then so does every algebra in the closure VClo of $\mathscr{K}$ under the taking of homomorphic images, subalgebras, and products; and, conversely,
- `vclo-id2`: if every algebra of the varietal closure VClo of $\mathscr{K}$ satisfies $p \approx q$, then so does every algebra in $\mathscr{K}$.

```
vclo-id1 : ∀ {p q} → (𝒦 ⊨ p ≋ q) → (VClo ⊨ p ≋ q)
vclo-id1 {p} {q} α (vbase A∈𝒦) = α A∈𝒦
vclo-id1 {p} {q} α (vprod{I = I}{𝒜 = 𝒜} 𝒜∈VClo) = γ
 where
  IH : (i : I) → 𝒜 i ⊨ p ≈ q
  IH i = vclo-id1{p}{q} α (𝒜∈VClo i)

  γ : p ̂ (⨅ 𝒜)  ≡ q ̂ (⨅ 𝒜)
  γ = products-preserve-identities p q I 𝒜 IH

vclo-id1 {p} {q} α ( vsub {𝑨 = 𝑨} A∈VClo sa ) =
 subalgebras-preserve-identities  p q (𝑨 , inj₂ refl , sa) 𝒦A⊨p≋q
  where
   A⊨p≈q : 𝑨 ⊨ p ≈ q
   A⊨p≈q = vclo-id1{p}{q} α A∈VClo

   Asingleton⊨p≋q :  ｛ 𝑨 ｝  ⊨ p ≋ q
   Asingleton⊨p≋q (refl _)  = A⊨p≈q

   𝒦A⊨p≋q : (𝒦 ∪  ｛ 𝑨 ｝ ) ⊨ p ≋ q
```

```
    𝒦⊧p≋q {B} (inj₁ x) = α x
    𝒦⊧p≋q {B} (inj₂ y) = Asingleton⊧p≋q y


vclo-id1 {p}{q} α (vhom{A = A} A∈VClo (B , ϕ , (ϕh , ϕE))) = γ
 where
  β : A ⊧ p ≈ q
  β = vclo-id1{p}{q} α A∈VClo

  preim : (b : X → ∣ B ∣)(x : X) → ∣ A ∣
  preim b x = (Inv ϕ (b x) (ϕE (b x)))

  ζ : (b : X → ∣ B ∣) → ϕ ∘ (preim b) ≡ b
  ζ b = gfe λ x → InvIsInv ϕ (b x) (ϕE (b x))

  γ : (p ̂ B) ≡ (q ̂ B)
  γ = gfe λ b →
   (p ̂ B) b               ≡⟨ (ap (p ̂ B) (ζ b))⁻¹ ⟩
   (p ̂ B) (ϕ ∘ (preim b)) ≡⟨ (comm-hom-term gfe A B (ϕ , ϕh) p (preim b))⁻¹ ⟩
   ϕ((p ̂ A)(preim b))     ≡⟨ ap ϕ (intensionality β (preim b)) ⟩
   ϕ((q ̂ A)(preim b))     ≡⟨ comm-hom-term gfe A B (ϕ , ϕh) q (preim b) ⟩
   (q ̂ B)(ϕ ∘ (preim b))   ≡⟨ ap (q ̂ B) (ζ b) ⟩
   (q ̂ B) b                ∎

vclo-id2 : ∀ {p q} → (VClo ⊧ p ≋ q) → (𝒦 ⊧ p ≋ q)
vclo-id2 p A∈𝒦 = p (vbase A∈𝒦)
```

Axiomatization of a class

We now prove that a class $\mathscr{K}$ of structures is axiomatized by Th(VClo($\mathscr{K}$)), which is the set of equations satisfied by all members of the varietal closure of $\mathscr{K}$.

```
-- Th (VClo 𝒦) is precisely the set of identities modeled by 𝒦
ThHSP-axiomatizes : (p q : ∣ T ∣)
       ---------------------------------------
 →     𝒦 ⊧ p ≋ q  ⇔  ((p , q) ∈ Th (VClo))

ThHSP-axiomatizes p q =
 (λ 𝒦⊧p≋q A∈VClo𝒦 → vclo-id1{p = p}{q = q} 𝒦⊧p≋q A∈VClo𝒦) ,
 λ pq∈Th A∈𝒦 → pq∈Th (vbase A∈𝒦)
```

The free algebra in Agda

Recall that term algebra $\mathbf{T}(X)$ is the absolutely free algebra in the class $\mathcal{K}(S)$ of all S-structures. In this section, we formalize, for a given class $\mathscr{K}$ of S-algebras, the (relatively) free algebra in SP($\mathscr{K}$) over X. Recall, this was defined above in free algebras as follows:

$\mathbb{F}(\mathscr{K}, X) := \mathbf{T}(X)/\Psi(\mathscr{K}, \mathbf{T}(X))$.

Thus, we must first formalize the congruence $\psi(\mathscr{K}, \mathbf{T}(X))$ which is defined by

$\Psi(\mathscr{K}, \mathbf{T}(X)) := \bigwedge \psi(\mathscr{K}, \mathbf{T}(X))$,

where $\psi(\mathscr{K}, \mathbf{T}(X)) := \{\theta \in \mathrm{Con}\, \mathbf{T}(X) : \mathbf{A}/\theta \in \mathrm{S}(\mathscr{K})\}$.

Strictly speaking, X is not a subset of $\mathbb{F}(\mathscr{K}, X)$ so it doesn't make sense to say that "X generates $\mathbb{F}(\mathscr{K}, X)$." But as long as $\mathscr{K}$ contains a nontrivial algebra, we will have $\Psi(\mathscr{K}, \mathbf{T}(X)) \cap X^2 \neq \emptyset$, and we can identify X with $X/\Psi(\mathscr{K}, \mathbf{T}(X))$ in $\mathbb{F}(\mathscr{K}, X)$.

```
THI = HomImagesOf (T{𝓤}{X})

Timg : 𝓞 ⊔ 𝓥 ⊔ 𝓤 ⁺ ̇
Timg = Σ A ꞉ (Algebra 𝓤 S) ,
        Σ ϕ ꞉ hom (T{𝓤}{X}) A , (A ∈ SClo) × Epic ∣ ϕ ∣

TA : (ti : Timg) → Algebra 𝓤 S
TA ti = ∣ ti ∣

TA∈SClo𝒦 : (ti : Timg) → (TA ti) ∈ SClo
TA∈SClo𝒦 ti = ∣ pr₂ ∥ ti ∥ ∣

Tϕ : (ti : Timg) → hom T (TA ti)
Tϕ ti = pr₁ ∥ ti ∥
```

```
TϕE : (ti : Timg) → Epic ∣ (Tϕ ti) ∣
TϕE ti = ∥ pr₂ ∥ ti ∥ ∥

TKER : 𝓞 ⊔ 𝓥 ⊔ 𝓤 ⁺ ̇
TKER = Σ (p , q) ꞉ (∣ T ∣ × ∣ T ∣) ,
   ∀ ti → (p , q) ∈ KER-pred{B = ∣ (TA ti) ∣} ∣ Tϕ ti ∣

Ψ : Pred (∣ T{𝓤}{X} ∣ × ∣ T ∣) (𝓞 ⊔ 𝓥 ⊔ 𝓤 ⁺)
Ψ (p , q) =
 ∀ ti → ∣ (Tϕ ti) ∣ ∘ (p ̇ T) ≡ ∣ (Tϕ ti) ∣ ∘ (q ̇ T)

Ψ' : Pred (∣ T ∣ × ∣ T ∣) (𝓞 ⊔ 𝓥 ⊔ 𝓤 ⁺)
Ψ' (p , q) = ∀ ti → ∣ (Tϕ ti) ∣ p ≡ ∣ (Tϕ ti) ∣ q
```

N.B. Ψ is the kernel of $\mathbf{T} \to \mathbb{F}(\mathscr{K}, \mathbf{T})$. Therefore, to prove $\mathbf{A}$ is a homomorphic image of $\mathbb{F}(\mathscr{K}, \mathbf{T})$, it suffices to show that the kernel of the lift $h : \mathbf{T} \to \mathbf{A}$ contains Ψ.

```
   T---- g --->>𝔽  (ker g = Ψ)
    \          .
     \        .
      h     ∃ϕ     (want: Ψ ⊆ ker h)
       \   .
        \ .
         V
         A
```

More tools for Birkhoff's theorem

Here are some of the key facts and identities we need to complete the proof of Birkhoff's HSP theorem.

```
SClo→Timg : (C : Algebra 𝓤 S) → C ∈ SClo → Timg

SClo→Timg C C∈SClo = C , (fst (Thom-gen C)) , (C∈SClo , (snd (Thom-gen C)))

Timg→T⊧ : ∀ p q → (p , q) ∈ Ψ' → (ti : Timg)
        ---------------------------------------------------
 →        ∣ (Tϕ ti) ∣ ((p ̇ T) 𝓰) ≡ ∣ (Tϕ ti) ∣ ((q ̇ T) 𝓰)

Timg→T⊧ p q pΨq ti = goal1
 where
  C : Algebra 𝓤 S
  C = ∣ ti ∣

  ϕ : hom T C
  ϕ = Tϕ ti

  pCq : ∣ ϕ ∣ p ≡ ∣ ϕ ∣ q
  pCq = pΨq ti

  𝓅 𝓆 : ∣ T ∣  -- Notation: 𝓅 = \Mcp
  𝓅 = ∣ tg p ∣
  𝓆 = ∣ tg q ∣

  p≡𝓅 : p ≡ (𝓅 ̇ T) 𝓰
  p≡𝓅 = ∥ tg p ∥

  q≡𝓆 : q ≡ (𝓆 ̇ T) 𝓰
  q≡𝓆 = ∥ tg q ∥

  ξ : ∣ ϕ ∣ ((𝓅 ̇ T) 𝓰) ≡ ∣ ϕ ∣ ((𝓆 ̇ T) 𝓰)
  ξ = (ap ∣ ϕ ∣ p≡𝓅)⁻¹ · pCq · (ap ∣ ϕ ∣ q≡𝓆)

  goal1 : ∣ ϕ ∣ ((p ̇ T) 𝓰) ≡ ∣ ϕ ∣ ((q ̇ T) 𝓰)
  goal1 = (ap ∣ ϕ ∣ (term-gen-agreement p)) · ξ · (ap ∣ ϕ ∣ (term-gen-agreement q))⁻¹

Ψ⊆ThSClo𝒦 : Ψ ⊆ (Th SClo)
Ψ⊆ThSClo𝒦 {p , q} pΨq {C} C∈SClo𝒦 = C⊧p≈q
 where
  ti : Timg
  ti = SClo→Timg C C∈SClo𝒦

  ϕ : hom T C
  ϕ = Tϕ ti

  ϕE : Epic ∣ ϕ ∣
  ϕE = TϕE ti
```

```
  ϕsur : (c : X → ∣ C ∣ )(x : X) → Image ∣ ϕ ∣ ∋ (c x)
  ϕsur c x = ϕE (c x)

  pre : (c : X → ∣ C ∣)(x : X) → ∣ T ∣
  pre c x = (Inv ∣ ϕ ∣ (c x) (ϕsur c x))

  ζ : (c : X → ∣ C ∣) → ∣ ϕ ∣ ∘ (pre c) ≡ c
  ζ c = gfe λ x → InvIsInv ∣ ϕ ∣ (c x) (ϕsur c x)

  γ : ∣ ϕ ∣ ∘ (p ̇ T) ≡ ∣ ϕ ∣ ∘ (q ̇ T)
  γ = pΨq ti

  C⊨p≈q : (p ̇ C) ≡ (q ̇ C)

  C⊨p≈q = gfe λ c →
   (p ̇ C) c                ≡⟨ (ap (p ̇ C) (ζ c))⁻¹ ⟩
   (p ̇ C)(∣ ϕ ∣ ∘ (pre c))  ≡⟨ (comm-hom-term gfe T C ϕ p (pre c))⁻¹ ⟩
   ∣ ϕ ∣ ((p ̇ T)(pre c))    ≡⟨ intensionality γ (pre c) ⟩
   ∣ ϕ ∣ ((q ̇ T)(pre c))    ≡⟨ comm-hom-term gfe T C ϕ q (pre c) ⟩
   (q ̇ C)(∣ ϕ ∣ ∘ (pre c))  ≡⟨ ap (q ̇ C) (ζ c) ⟩
   (q ̇ C) c                ∎

Ψ⊆Th𝒦 : ∀ p q → (p , q) ∈ Ψ → 𝒦 ⊨ p ≋ q
Ψ⊆Th𝒦 p q pΨq {A} KA = Ψ⊆ThSClo𝒦{p , q} pΨq (sbase KA)
```

Unicode Hints 5

Table of some special characters used in the closure module.

To get	Type
a, b	`\MIa`, `\MIb`
$f^{\frown}A$	`\Mif \^ \MIA`
≅	≅ or `\cong`
∘	`\comp` or `\circ`
$\dot{\mathcal{id}}$	`\Mci\Mcd`
$\mathscr{L}\mathscr{K}$	`\McL\McK`
ϕ	`\phi`

Emacs commands providing information about special characters/input methods:

- `M-x describe-char` (or `M-m h d c`) with the cursor on the character of interest
- `M-x describe-input-method` (or `C-h I`)

Table of contents ⇑

HSP Theorem in Agda

Here we give a formal proof in Agda of Birkhoff's theorem, which says that a variety is an equational class. In other words, if a class $\mathscr{K}$ of algebras is closed under the operators $\boldsymbol{H}$, $\boldsymbol{S}$, $\boldsymbol{P}$, then $\mathscr{K}$ is an equational class (i.e., $\mathscr{K}$ is the class of all algebras that model a particular set of identities).

In addition to the usual importing of dependencies, We start the birkhoff module with a fixed signature and a type `X`. As in the `terms` module, `X` represents an arbitrary (infinite) collection of "variables" (which will serve as the generators of the term algebra $\boldsymbol{T}$(X)).

```
open basic
open congruences
open prelude using (global-dfunext; dfunext; funext; Pred)

module birkhoff
 {S : Signature 𝓞 𝓥}
 {X : 𝓤 }
 {𝒦 : Pred (Algebra 𝓤 S)(𝓞 ⊔ 𝓥 ⊔ 𝓤 ⁺)}
 {𝕏 : {𝓤 𝓧 : Universe}{X : 𝓧 ̇ }(A : Algebra 𝓤 S) → X ↠ A}
 {gfe : global-dfunext}
 {dfe : dfunext 𝓤 𝓤}
```

```
 {fevu : dfunext 𝓥 𝓤} where

 open closure {𝑆 = 𝑆}{X = X}{𝒦 = 𝒦}{𝕏 = 𝕏}{gfe = gfe}{dfe = dfe}{fevu = fevu}
```

Equalizers in Agda

The equalizer of two functions (resp., homomorphisms) `g h : A → B` is the subset of `A` on which the values of the functions `g` and `h` agree. We formalize this notion in Agda as follows.

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

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

It turns out that the equalizer of two homomorphisms is closed under the operations of `A` and is therefore a subalgebra of the common domain, as we now prove.

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

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

Thus, `EH` is a subuniverse of `A`.

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

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

Homomorphism determination

The homomorphisms module formalizes the notion of homomorphism and proves some basic facts about them. Here we show that homomorphisms are determined by their values on a generating set. This is proved here, and not in the homomorphisms module because we need `Sg` from the `subuniverses` module.

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

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

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

A formal proof of Birkhoff's theorem

Here's the statement we wish to prove:

```
 birkhoff : (𝑨 : Algebra 𝓤 𝑆) → 𝑨 ∈ Mod (Th VClo)
            ---------------------------------------
  →                      𝑨 ∈ VClo
```

Here's the partial proof:

```
 birkhoff 𝑨 A∈ModThV = 𝑨∈VClo
  where
```

```
ℋ : X ↠ A
ℋ = 𝕏 A

h₀ : X → ∣ A ∣
h₀ = fst ℋ

h : hom T A
h = lift-hom{A = A} h₀

Ψ⊆ThVClo : Ψ ⊆ Th VClo
Ψ⊆ThVClo {p , q} pΨq =
 (lr-implication (ThHSP-axiomatizes p q)) (Ψ⊆Thℋ p q pΨq)

Ψ⊆A⊨ : ∀{p}{q} → (p , q) ∈ Ψ → A ⊨ p ≈ q
Ψ⊆A⊨ {p} {q} pΨq = A∈ModThV p q (Ψ⊆ThVClo {p , q} pΨq)

Ψ⊆Kerh : Ψ ⊆ KER-pred{B = ∣ A ∣} ∣ h ∣
Ψ⊆Kerh {p , q} pΨq = hp≡hq
 where
  hp≡hq : ∣ h ∣ p ≡ ∣ h ∣ q
  hp≡hq = hom-id-compatibility p q A h (Ψ⊆A⊨{p}{q} pΨq)

--We need to find C : Algebra 𝒰 S such that C ∈ VClo and ∃ ϕ : hom C A with ϕE : Epic ∣ ϕ ∣.
--Then we can prove A ∈ VClo ℋ by vhom C∈VClo (A , ∣ ϕ ∣ , (∥ ϕ ∥ , ϕE))
-- since vhom : {A : Algebra 𝒰 S} → A ∈ VClo ℋ → ((B , _ , _) : HomImagesOf A) → B ∈ VClo ℋ
C : Algebra 𝒰 S
C = {!!}

ϕ : Σ h ꞉ (hom C A) , Epic ∣ h ∣
ϕ = {!!}

hic : HomImagesOf C
hic = (A , ∣ fst ϕ ∣ , (∥ fst ϕ ∥ , snd ϕ) )

A∈VClo : A ∈ VClo
A∈VClo = vhom{C} {!!} hic
```
