The Agda Universal Algebra Library
Part 1: Foundation

Equality, extensionality, truncation, and dependent types for relations and algebras

William DeMeo ✉ 🏠 ⓘ

Department of Algebra, Charles University in Prague

──── **Abstract** ───

The Agda Universal Algebra Library (UALib) is a library of types and programs (theorems and proofs) we developed to formalize the foundations of universal algebra in dependent type theory using the Agda programming language and proof assistant. The UALib includes a substantial collection of definitions, theorems, and proofs from general algebra and equational logic, including many examples that exhibit the power of inductive and dependent types for representing and reasoning about relations, algebraic structures, and equational theories. In this paper we describe several important aspects of the logical foundations on which the library is built. We also discuss (though sometimes only briefly) all of the types defined in the first 13 modules of the library, with special attention given to those details that seem most interesting or challenging from a type theory or mathematical foundations perspective.

Contents

1 Introduction

To support formalization in type theory of research level mathematics in universal algebra and related fields, we present the Agda Universal Algebra Library (AgdaUALib), a software library containing formal statements and proofs of the core definitions and results of universal algebra. The UALib is written in Agda [12], a programming language and proof assistant based on Martin-Löf Type Theory that not only supports dependent and inductive types, as well as proof tactics for proving things about the objects that inhabit these types.

1.1 Motivation

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

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

1.2 Prior art

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

- Capretta [3] (1999) formalized the basics of universal algebra in the Calculus of Inductive Constructions using the Coq proof assistant;
- Spitters and van der Weegen [14] (2011) formalized the basics of universal algebra and some classical algebraic structures, also in the Calculus of Inductive Constructions using the Coq proof assistant, promoting the use of type classes as a preferable alternative to setoids;
- Gunther, et al [10] (2018) developed what seems to be (prior to the UALib) the most extensive library of formal universal algebra to date; in particular, this work includes a formalization of some basic equational logic; also (unlike the UALib) it handles *multisorted* algebraic structures; (like the UALib) it is based on dependent type theory and the Agda proof assistant.

Some other projects aimed at formalizing mathematics generally, and algebra in particular, have developed into very extensive libraries that include definitions, theorems, and proofs about algebraic structures, such as groups, rings, modules, etc. However, the goals of these efforts seem to be the formalization of special classical algebraic structures, as opposed to the general theory of (universal) algebras. Moreover, the part of universal algebra and equational logic formalized in the UALib extends beyond the scope of prior efforts and. In particular, the library now includes a proof of Birkhoff's variety theorem. Most other proofs of this theorem that we know of are informal and nonconstructive.[1]

1.3 Attributions and Contributions

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

Unless explicitly stated otherwise, the Agda source code described in this paper is due to the author, with the following caveat: the UALib depends on the Type Topology library of Martín Escardó [9]. Each dependency is carefully accounted for and mentioned in this paper. For the sake of completeness and clarity, and to keep the paper mostly self-contained, we repeat some definitions from the Type Topology library, but in each instance we cite the original source.[2]

In this paper we limit ourselves to the presentation of the core foundational modules of the UALib so that we have space to discuss some of the more interesting type theoretic and foundational issues that arose when developing the library and attempting to represent advanced mathematical notions in type theory and formalize them in Agda. As such, this is only the first

---

[1] After completing the formal proof in Agda, we learned about a constructive version of Birkhoff's theorem proved by Carlström in [4]. The latter is presented in the informal style of standard mathematical writing, and as far as we know it was never formalized in type theory and type-checked with a proof assistant. Nonetheless, a comparison of Carlström's proof and the UALib proof would be interesting.

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

installment of a three-part series of papers describing the AgdaUALib. The second part is [7], covering homomorphisms, terms, and subalgebras. The third part is [8], which will cover free algebras, equational classes of algebras (i.e., varieties), and Birkhoff's HSP theorem.

1.4 Organization of the paper

This present paper is organized into three parts as follows. The first part is §2 which introduces the basic concepts of type theory with special emphasis on the way such concepts are formalized in Agda. Specifically, §2.1 introduces *Sigma types* and Agda's hierarchy of *universes*. The important topics of *equality* and *function extensionality* are discussed in §2.2 and §2.3; §2.4 covers inverses and inverse images of functions. In §2.5 we describe a technical problem that one frequently encounters when working in a *noncummulative universe hierarchy* and offer some tools for resolving the type-checking errors that arise from this.

The second part is §3 which covers *relation types* and *quotient types*. Specifically, §3.1 defines types that represent *unary* and *binary relations* as well as *function kernels*. These "discrete relation types," are all very standard. In §3.2 we introduce the (less standard) types that we use to represent *general* and *dependent relations*. We call these "continuous relations" because they can have arbitrary arity (general relations) and they can be defined over arbitrary families of types (dependent relations). In §3.3 we cover standard types for equivalence relations and quotients, and in §3.4 we discuss a family of concepts that are vital to the mechanization of mathematics using type theory; these are the closely related concepts of *truncation*, *sets*, *propositions*, and *proposition extensionality*.

The third part of the paper is §4 which covers the basic domain-specific types offered by the UALib. It is here that we finally get to see some types representing algebraic structures. Specifically, we describe types for *operations* and *signatures* (§4.1), *general algebras* (§4.2), and *product algebras* (§4.3), including types for representing *products over arbitrary classes of algebraic structures*. Finally, we define types for congruence relations and quotient algebras in §4.4.

1.5 Resources

We conclude this introduction with some pointers to helpful reference materials. For the required background in Universal Algebra, we recommend the textbook by Clifford Bergman [1]. For the type theory background, we recommend the HoTT Book [13] and Escardó's Introduction to Univalent Foundations of Mathematics with Agda [9].

The following are informed the development of the UALib and are highly recommended.
- *Introduction to Univalent Foundations of Mathematics with Agda*, Escardó [9]
- *Dependent Types at Work*, Bove and Dybjer [2]
- *Dependently Typed Programming in Agda*, Norell and Chapman [11]
- *Formalization of Universal Algebra in Agda*, Gunther, Gadea, Pagano [10]
- *Programming Languages Foundations in Agda* [18]

More information about AgdaUALib can be obtained from the following official sources.
- ualib.org (the web site) documents every line of code in the library.
- gitlab.com/ualib/ualib.gitlab.io (the source code) AgdaUALib is open source.[3]
- *The Agda UALib, Part 2: homomorphisms, terms, and subalgebras* [7].
- *The Agda UALib, Part 3: free algebras, equational classes, and Birkhoff's theorem* [8].

---

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

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

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

2   Agda Prelude

2.1   Preliminaries: logical foundations, universes, dependent types

This section presents the Prelude.Preliminaries module of the AgdaUALib, slightly abridged.[4] We begin by highlighting some of the key parts of the module, which lists everything we need from Martin Escardó's Type Topology library [9], defines some basic types, and proves some of their properties. We do not cover the entire Prelude.Preliminaries module here, but instead call attention to aspects that differ from standard Agda syntax.[5]

2.1.1   Logical foundations

The AgdaUALib is based on a minimal version of *Martin-Löf dependent type theory* (MLTT) that is the same or very close to the type theory on which Martín Escardó's Type Topology Agda library is based.[6] We won't go into great detail here because there are already other very nice resources available, such as the section A spartan Martin-Löf type theory of the lecture notes by Escardó just mentioned, the ncatlab entry on Martin-Löf dependent type theory, as well as the HoTT Book [13].

We will have much more to say about types and type theory as we progress. For now, suffice it to recall the handful of objects that are assumed at the jumping-off point for MLTT: "primitive" *types* ($\mathbb{0}$, $\mathbb{1}$, and $\mathbb{N}$, denoting the empty type, one-element type, and natural numbers), *type formers* ($+$, Π, Σ, Id, denoting *binary sum*, *product*, *sum*, and the *identity* type), and an infinite collection of *universes* (types of types) and universe variables to denote them (for which we will use upper-case caligraphic letters like $\mathcal{U}$, $\mathcal{V}$, $\mathcal{W}$, etc., typically from the latter half of the English alphabet).

2.1.2   Specifying logical foundations in Agda

An Agda program typically begins by setting some options and by importing types from existing Agda libraries. Options are specified with the OPTIONS *pragma* and control the way Agda behaves by, for example, specifying the logical axioms and deduction rules we wish to assume when the program is type-checked to verify its correctness. Every Agda program in the UALib begins with the following line.

$$\{\text{-\# OPTIONS } \text{--without-K --exact-split --safe \#-}\} \tag{1}$$

---

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

[5] For more details, see `https://ualib.gitlab.io/Prelude.Preliminaries.html`.

[6] This is also the type theory that Escardó taught us in the short course Introduction to Univalent Foundations of Mathematics with Agda at the Midlands Graduate School in the Foundations of Computing Science at University of Birmingham in 2019.

These options control certain foundational assumptions that Agda makes when type-checking the program to verify its correctness.

- *–without-K* disables Streicher's K axiom; see [15];
- *–exact-split* makes Agda accept only definitions that are *judgmental* equalities; see [17];
- *–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 [16] and [17].

Throughout this paper we take assumptions 1–3 for granted without mentioning them explicitly.

2.1.3 Agda Modules

The OPTIONS pragma is usually followed by the start of a module. For example, the Prelude.Preliminaries module begins with the following line.

    module Prelude.Preliminaries where

Sometimes we want to declare parameters that will be assumed throughout the module. For instance, when working with algebras, we often assume they come from a particular fixed signature, and this signature is something we could fix as a parameter at the start of a module. Thus, we might start an *anonymous submodule* of the main module with a line like[7]

    module _ {S : Signature 𝓞 𝓥} where

Such a module is called *anonymous* because an underscore appears in place of a module name. Agda determines where a submodule ends by indentation. This can take some getting used to, but after a short time it will feel very natural. The main module of a file must have the same name as the file, without the `.agda` or `.lagda` file extension. The code inside the main module is not indented. Submodules are declared inside the main module and code inside these submodules must be indented to a fixed column. As long as the code is indented, Agda considers it part of the submodule. A submodule is exited as soon as nonindented code appears.

2.1.4 Agda's universe hierarchy

The AgdaUALib adopts the notation of Martin Escardo's Type Topology library [9]. In particular, *universe levels*[8] are denoted by capitalized script letters from the second half of the alphabet, e.g., $\mathcal{U}$, $\mathcal{V}$, $\mathcal{W}$, etc. Also defined in Type Topology are the operators $\cdot$ and $^{+}$. These map a universe level $\mathcal{U}$ to the universe $\mathcal{U}\,\cdot := \mathsf{Set}\,\mathcal{U}$ and the level $\mathcal{U}\,^{+} := \mathsf{lsuc}\,\mathcal{U}$, respectively. Thus, $\mathcal{U}\,\cdot$ is simply an alias for the universe $\mathsf{Set}\,\mathcal{U}$, and we have $\mathcal{U}\,\cdot : \mathcal{U}\,^{+}\,\cdot$.

The hierarchy of universes in Agda is structured as $\mathcal{U}\,\cdot : \mathcal{U}\,^{+}\,\cdot$, $\mathcal{U}\,^{+}\,\cdot : \mathcal{U}\,^{++}\,\cdot$, etc. This means that the universe $\mathcal{U}\,\cdot$ has type $\mathcal{U}\,^{+}\,\cdot$, and $\mathcal{U}\,^{+}\,\cdot$ has type $\mathcal{U}\,^{++}\,\cdot$, and so on. It is important to note, however, this does *not* imply that $\mathcal{U}\,\cdot : \mathcal{U}\,^{++}\,\cdot$. In other words, Agda's universe hierarchy is *noncummulative*. This makes it possible to treat universe levels more generally and precisely, which is nice. On the other hand, a noncummulative hierarchy can sometimes make for a nonfun proof assistant. Luckily, there are ways to circumvent noncummulativity without introducing logical inconsistencies into the type theory. Section 4.2.4 describes some domain-specific tools that we developed for this purpose.

---

[7] The Signature type will be defined in Section 4.1.

[8] See https://agda.readthedocs.io/en/v2.6.1.2/language/universe-levels.html.

2.1.5 Sigma and Pi types

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

Agda's default syntax for a Sigma type is $\Sigma\,\lambda(x : X) \to Y$, but we prefer the notation $\Sigma\ x : X$, Y, which is closer to the standard syntax described in the preceding paragraph. Fortunately, this preferred notation is available in the Type Topology library (see [9, Σ types]).[9]

Convenient notations for the first and second projections out of a product are |_| and ‖_‖, respectively. However, to improve readability or to avoid notation clashes with other modules, we sometimes use more standard alternatives, such as pr_1 and pr_2, or fst and snd, or some combination of these. The definitions are standard so we omit them (see [6] for details).

2.2 Equality: definitional equality and transport

This section presents the Prelude.Equality module of the AgdaUALib, slightly abridged.[10] The type referred to as "reflexivity" or "refl" is a very basic but important type in MLTT. It represents *definitional equality*.[11] The refl type we use is a standard one. It is defined in the Identity-Type module of the Type Topology library, but (apart from slightly different syntax) it is equivalent to the identity type used in most other Agda libraries.

We make refl available by importing it from the Identity-Type module. However, we first repeat the definition here (inside a hidden submodule) for clarity.[12]

```
module hide-refl {𝒰 : Universe} where

  data _≡_ {𝒰} {X : 𝒰 ˙ } : X → X → 𝒰 ˙ where refl : {x : X} → x ≡ x

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

Thus, whenever we need to complete a proof by simply asserting that x is definitionally equal to itself, we invoke refl. If we need to make x explicit, we use refl $\{x = x\}$.

Of course $\equiv$ is an equivalence relation and the formal proof of this is trivial. We don't need to prove reflexivity since it is the defining property of $\equiv$. Here are the (trivial) proofs of symmetry and transitivity of $\equiv$.

```
≡-symmetric : (x y : X) → x ≡ y → y ≡ x
≡-symmetric _ _ refl = refl
```

---

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

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

[11] For the definition and a discussion of definitional equality, see `https://ncatlab.org/nlab/show/equality#definitional_equality`.

[12] To hide code from the rest of the development, we enclose it in a named module. For example, the code inside the hide-refl module will not conflict with the original definitions from the Type Topology library, even though we import the latter right after presenting the definition. As long as we don't invoke open hide-refl, the code inside the hide-refl module remains essentially hidden (though Agda *will* type-check the code). It may seem odd to both define things in the hidden module only to immediately import the definition that we actually use, but we do this in an attempt to exhibit all of the types on which the UALib depends, in a clear and self-contained way, while also ensuring that this cannot be misinterpreted as a claim to originality.

```
≡-sym : {x y : X} → x ≡ y → y ≡ x
≡-sym refl = refl

≡-transitive : (x y z : X) → x ≡ y → y ≡ z → x ≡ z
≡-transitive _ _ _ refl refl = refl

≡-trans : {x y z : X} → x ≡ y → y ≡ z → x ≡ z
≡-trans refl refl = refl
```

The only difference between ≡-symmetric and ≡-sym (respectively, ≡-transitive and ≡-trans) is that the latter has fewer explicit arguments, which is sometimes convenient.

Many proofs make abundant use of the symmetry of ≡, and the following syntactic sugar can improve the readability of such proofs.[13]

```
module hide-sym-trans {𝒰 : Universe} where

    _⁻¹ : {X : 𝒰 · } → {x y : X} → x ≡ y → y ≡ x
    p ⁻¹ = ≡-sym p
```

If we have a proof $p : x \equiv y$, and we need a proof of $y \equiv x$, then instead of ≡-sym p we can use the more intuitive p^{-1}. Similarly, the following syntactic sugar makes abundant appeals to transitivity easier to stomach.

```
    _·_ : {X : 𝒰 · } {x y z : X} → x ≡ y → y ≡ z → x ≡ z
    p · q = ≡-trans p q

open import MGS-MLTT using (_⁻¹; _·_) public
```

2.2.1   Transport

Alonzo Church characterized equality by declaring two things equal iff no property (predicate) can distinguish them (see [5]). In other terms, x and y are equal iff for all P we have $P\ x \to P\ y$. One direction of this implication is sometimes called *substitution* or *transport* or *transport along an identity*. It asserts the following: if two objects are equal and one of them satisfies a given predicate, then so does the other. A type representing this notion is defined, along with the (polymorphic) identity function, in the MGS-MLTT module of the Type Topology library.

```
    id : {𝒳 : Universe} (X : 𝒳 · ) → X → X
    id X = λ x → x

    transport : {X : 𝒰 · } (A : X → 𝒲 · ) {x y : X} → x ≡ y → A x → A y
    transport A (refl {x = x}) = id (A x)
```

See [9] for a discussion of transport.[14]

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

---

[13] **Unicode Hints.** In agda2-mode, one types $^{-1}$ as `\^-\^1`, and *id* as `\Mii\Mid`, and type · as `\..`. In general, to get information about a given unicode character (e.g., how to type it) place the cursor over that character and type `M-x describe-char` (or `M-x h d c`).

[14] cf. transport in HoTT-Agda: `https://github.com/HoTT/HoTT-Agda/blob/master/core/lib/Base.agda`.

$\equiv$ f x' by simply applying the ap function like so, ap $f\ p$: $f\ x \equiv f\ x'$. Escardó defines ap in the Type Topology library as follows.

```
module hide-ap {𝒰 : Universe} where

  ap : {X : 𝒰 ˙}{Y : 𝒱 ˙}(f : X → Y){a b : X} → a ≡ b → f a ≡ f b
  ap f {a} p = transport (λ - → f a ≡ f -) p (refl {x = f a})

open import MGS-MLTT using (ap) public
```

Here are some variations of ap that are sometimes useful.

```
ap-cong : {A : 𝒰 ˙}{B : 𝒲 ˙}{f g : A → B} {a b : A} → f ≡ g → a ≡ b → f a ≡ g b
ap-cong refl refl = refl
```

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

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

2.2.2 $\equiv$-intro and $\equiv$-elim for nondependent pairs

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

```
≡-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

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

2.3 Extensionality: types for postulating function extensionality

This section presents the Prelude.Extensionality module of the AgdaUALib, slightly abridged.[15]

2.3.1 Background and motivation

This brief introduction to the basics of *function extensionality* is intended for novices. If you're already familiar with the concept, you may want to skip to the next subsection.

What does it mean to say that two functions $f\ g\colon X \to Y$ are equal? Suppose f and g are defined on $X = \mathbb{Z}$ (the integers) as follows: $fx := x + 2$ and $gx := ((2 * x) - 8)/2 + 6$. Would you say that f and g are equal? Are they the "same" function? What does that even mean?

---

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

If you know a little bit of basic algebra, then you probably can't resist the urge to reduce g to the form $x+2$ and proclaim that f and g are, indeed, equal. And you would be right, at least in middle school, and the discussion would end there. In the science of computing, however, more attention is paid to equality, and with good reason.

We can probably all agree that the functions f and g above, while not syntactically equal, do produce the same output when given the same input so it seems fine to think of the functions as the same, for all intents and purposes. But we should ask ourselves, at what point do we notice or care about the difference in the way functions are defined?

What if we had started out this discussion with two functions f and g both of which take a list as input and produce as output a correctly sorted version of that list? Are the functions the same? What if f was defined using the merge sort algorithm, while g used quick sort? Probably most of us wouldn't think of f and g as the same function in that case.

In the examples above, it is common to say that the two functions f and g are *extensionally equal*, since they produce the same (external) output when given the same input, but they are not *intensionally equal*, since their (internal) definitions differ.

In this subsection, we describe types that manifest this idea of *extensional equality of functions*, or *function extensionality*. (Most of these types are already defined in the Type Topology library, so the UALib merely imports the definitions from there.)

2.3.2   Definition of function extensionality

As alluded to above, a natural notion of function equality, sometimes called *pointwise equality*, is defined as follows: f and g are said to be *pointwise equal* provided $\forall x \to f x \equiv g x$. Here is how this notion of equality is expressed as a type in the Type Topology library.

$$\_\sim\_ : \{\mathcal{U} \; \mathcal{V} : \mathsf{Universe}\}\{X : \mathcal{U} \; ^{\textperiodcentered} \} \; \{A : X \to \mathcal{V} \; ^{\textperiodcentered} \} \to \Pi \, A \to \Pi \, A \to \mathcal{U} \sqcup \mathcal{V} \; ^{\textperiodcentered}$$
$$f \sim g = \forall \; x \to f \, x \equiv g \, x$$

Function extensionality is the assertion that pointwise equal functions are *definitionally equal*; that is, $\forall \, f \, g \, (f \sim g \to f \equiv g)$. In the Type Topology library, the types that represent this notion are funext (for nondependent functions) and dfunext (for dependent functions). They are defined as follows.

$$\mathsf{funext} : \forall \, \mathcal{U} \; \mathcal{W} \to (\mathcal{U} \sqcup \mathcal{W})^{\,+} \; ^{\textperiodcentered}$$
$$\mathsf{funext} \; \mathcal{U} \; \mathcal{W} = \{A : \mathcal{U} \; ^{\textperiodcentered} \} \; \{B : \mathcal{W} \; ^{\textperiodcentered} \} \; \{f \, g : \, A \to B\} \to f \sim g \to f \equiv g$$

$$\mathsf{dfunext} : \forall \, \mathcal{U} \; \mathcal{W} \to (\mathcal{U} \sqcup \mathcal{W})^{\,+} \; ^{\textperiodcentered}$$
$$\mathsf{dfunext} \; \mathcal{U} \; \mathcal{W} = \{A : \mathcal{U} \; ^{\textperiodcentered} \}\{B : A \to \mathcal{W} \; ^{\textperiodcentered} \}\{f \, g : \forall(x : \, A) \to B \, x\} \to f \sim g \to f \equiv g$$

In informal settings, this so-called "pointwise equality of functions" is typically what one means when one asserts that two functions are "equal."[16] However, it is important to keep in mind the following fact: *function extensionality is known to be neither provable nor disprovable in Martin-Löf type theory. It is an independent statement* ([9]).

---

[16] If one assumes the *univalence axiom* of Homotopy Type Theory, then the $\_\sim\_$ relation is equivalent to equality of functions. See the section "Function extensionality from univalence" of Escardó's notes [9].

2.3.3   Global function extensionality

An assumption that we adopt throughout much of the current version of the UALib is a *global function extensionality principle.* This asserts that function extensionality holds at all universe levels. Agda is capable of expressing types representing global principles as the language has a special universe level for such types. Following Escardó, we denote this universe by $\mathcal{U}\omega$ (which is just an alias for Agda's Setω universe).[17]   The types global-funext and global-dfunext are defined in the Type Topology library as follows.

> global-funext : $\mathcal{U}\omega$
> global-funext $= \forall \{\mathcal{U} \ \mathcal{V}\} \rightarrow$ funext $\mathcal{U} \ \mathcal{V}$
>
> global-dfunext : $\mathcal{U}\omega$
> global-dfunext $= \forall \{\mathcal{U} \ \mathcal{V}\} \rightarrow$ funext $\mathcal{U} \ \mathcal{V}$

The next two types define the converse of function extensionality.[18]

> extfun : $\{A : \mathcal{U} \ \cdot\}\{B : \mathcal{W} \ \cdot\}\{f \ g : A \rightarrow B\} \rightarrow f \equiv g \rightarrow f \sim g$
> extfun refl $\_$ = refl
>
> extdfun : $\{A : \mathcal{U} \ \cdot \}\{B : A \rightarrow \mathcal{W} \ \cdot \}(f \ g : \Pi \ B) \rightarrow f \equiv g \rightarrow f \sim g$
> extdfun $\_ \ \_$ refl $\_$ = refl

Though it may seem obvious to some readers, we wish to emphasize the important conceptual distinction between two different forms of type definitions we have seen so far. We do so by comparing the definitions of funext and extfun. In the definition of funext, the codomain is a generic type (namely, $(\mathcal{U} \sqcup \mathcal{V})^+ \ \cdot$). In the definition of extfun, the codomain is an assertion (namely, $f \sim g$). Also, the defining equation of funext is an assertion, while the defining equation of extfun is a proof. As such, extfun is a *proof object*; it proves (inhabits the type that represents) the proposition asserting that definitionally equivalent functions are pointwise equal. In contrast, funext is a type, and we may or may not wish to assume we have a proof for this type. That is, we could postulate that function extensionality holds and assume we have a witness, say, $fe :$ funext $\mathcal{U} \ \mathcal{V}$ (i.e., a proof that pointwise equal functions are equal), but as noted above the existence of such a witness cannot be proved in Martin-Löf type theory.

2.3.4   Alternative extensionality type

Finally, a useful alternative for expressing dependent function extensionality, which is essentially equivalent to dfunext, is to assert that extdfun is actually an *equivalence.* This requires a few definitions from the MGS-Equivalences module of the Type Topology library, which we now describe.

First, a type is a *singleton* if it has exactly one inhabitant and a *subsingleton* if it has at most one inhabitant. These properties are represented in the Type Topology library by the following type.

---

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

[18] In previous versions of the UALib this function was called intensionality, indicating that it represented the concept of *function intensionality*, but we realized this isn't quite right and changed the name to the less controvertial extfun. Also, we later realized that a function called happly, which is nearly identical to extdfun, was already defined in the MGS-FunExt-from-Univalence module of the Type Topology library.

```
is-center : (X : 𝒰 ˙) → X → 𝒰 ˙
is-center X c = (x : X) → c ≡ x

is-singleton : 𝒰 ˙ → 𝒰 ˙
is-singleton X = Σ c : X , is-center X c

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

Next, we consider the type is-equiv which is used to assert that a function is an equivalence in the sense that we now describe. First we need the concept of a fiber of a function. In the Type Topology library, fiber is defined as a Sigma type whose inhabitants represent inverse images of points in the codomain of the given function.

```
fiber : {X : 𝒰 ˙} {Y : 𝒲 ˙} (f : X → Y) → Y → 𝒰 ⊔ 𝒲 ˙
fiber {X} f y = Σ x : X , f x ≡ y
```

A function is called an *equivalence* if all of its fibers are singletons.

```
is-equiv : {X : 𝒰 ˙} {Y : 𝒲 ˙} → (X → Y) → 𝒰 ⊔ 𝒲 ˙
is-equiv f = ∀ y → is-singleton (fiber f y)
```

Now we are finally ready to define the type hfunext that gives an alternative means of postulating function extensionality.

```
hfunext : (𝒰 𝒲 : Universe) → (𝒰 ⊔ 𝒲)⁺ ˙
hfunext 𝒰 𝒲 = {A : 𝒰 ˙}{B : A → 𝒲 ˙} (f g : Π B) → is-equiv (extdfun f g)
```

2.4 Inverses: epics, monics, embeddings, inverse images

This section presents the Prelude.Inverses module of the AgdaUALib, slightly abridged.[19] We begin by defining an inductive type that represents the semantic concept of *inverse image* of a function.

```
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
```

Next we verify that the type just defined is what we expect.

```
ImageIsImage : {A : 𝒰 ˙}{B : 𝒲 ˙}(f : A → B)(b : B)(a : A)
               ——————————————————————
    →          b ≡ f a → Image f ∋ b

ImageIsImage f b a b≡fa = eq b a b≡fa
```

Note that an inhabitant of Image $f \ni b$ is a pair (a , p), where $a : A$, and p is a proof that f maps a to b; that is, $p : b \equiv f a$. Since the proof that b belongs to the image of f is always accompanied by a "witness" $a : A$, we can actually *compute* a *pseudoinverse* of f. This function takes an arbitrary $b : B$ and a (*witness*, *proof*)-pair, $(a , p) :$ Image $f \ni b$, and returns a.

---

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

Inv : $\{A : \mathcal{U}^{\;\cdot}\}\{B : \mathcal{W}^{\;\cdot}\}(f : A \to B)\{b : B\} \to$ Image $f \ni b \to A$
Inv $f \{.(f\;a)\}$ (im a) $= a$
Inv f (eq _ a _) $= a$

We can prove that Inv f is the *right-inverse* of f, as follows.

InvIsInv : $\{A : \mathcal{U}^{\;\cdot}\}\{B : \mathcal{W}^{\;\cdot}\}(f : A \to B)\{b : B\}(q :$ Image $f \ni b) \to f($Inv $f\;q) \equiv b$
InvIsInv $f \{.(f\;a)\}$ (im a) $=$ refl
InvIsInv f (eq _ _ p) $= p^{\;-1}$

2.4.1 Surjective functions

An *epic* (or *surjective*) function from type $A : \mathcal{U}^{\;\cdot}$ to type $B : \mathcal{W}^{\;\cdot}$ is as an inhabitant of the Epic type, which we define as follows.

Epic : $\{A : \mathcal{U}^{\;\cdot}\}\;\{B : \mathcal{W}^{\;\cdot}\}\;(g : A \to B) \to \mathcal{U} \sqcup \mathcal{W}^{\;\cdot}$
Epic $g = \forall\;y \to$ Image $g \ni y$

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

EpicInv : $\{A : \mathcal{U}^{\;\cdot}\}\;\{B : \mathcal{W}^{\;\cdot}\}\;(f : A \to B) \to$ Epic $f \to B \to A$
EpicInv f *fepi* $b =$ Inv f (*fepi* b)

The function defined by EpicInv f *fepi* is indeed the right-inverse of f. To state this, we'll use the function composition operation $\circ$, which is already defined in the Type Topology library, as follows.

$\circ$: $\{X : \mathcal{U}^{\;\cdot}\}\;\{Y : \mathcal{W}^{\;\cdot}\}\{Z : Y \to \mathcal{W}^{\;\cdot}\} \to \Pi\;Z \to (f : X \to Y) \to (x : X) \to Z\;(f\;x)$
$g \circ f = \lambda\;x \to g\;(f\;x)$

EpicInvIsRightInv : funext $\mathcal{W}\;\mathcal{W} \to \{A : \mathcal{U}^{\;\cdot}\}\;\{B : \mathcal{W}^{\;\cdot}\}$
$\qquad\qquad\qquad (f : A \to B)\;(fepi :$ Epic $f)$
$\qquad\qquad\qquad$ ————————————
$\to \qquad\qquad\qquad f \circ ($EpicInv f *fepi*$) \equiv id\;B$

EpicInvIsRightInv *fe f fepi* $= fe\;(\lambda\;x \to$ InvIsInv f (*fepi* x))

2.4.2 Injective functions

We say that a function $g : A \to B$ is **monic** (or **injective** or **one-to-one**) if it doesn't map distinct elements to a common point. This property is formalized quite naturally using the Monic type, which we now define.

Monic : $\{A : \mathcal{U}^{\;\cdot}\}\;\{B : \mathcal{W}^{\;\cdot}\}(g : A \to B) \to \mathcal{U} \sqcup \mathcal{W}^{\;\cdot}$
Monic $g = \forall\;a_1\;a_2 \to g\;a_1 \equiv g\;a_2 \to a_1 \equiv a_2$

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

MonicInv : $\{A : \mathcal{U}^{\;\cdot}\}\{B : \mathcal{W}^{\;\cdot}\}(f : A \to B) \to$ Monic $f \to (b : B) \to$ Image $f \ni b \to A$
MonicInv f _ $= \lambda\;b\;Imf{\ni}b \to$ Inv $f\;Imf{\ni}b$

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

MonicInvIsLeftInv : $\{A : \mathcal{U}^{\ \boldsymbol{\cdot}}\}\{B : \mathcal{W}^{\ \boldsymbol{\cdot}}\}(f : A \to B)(fmonic : $ Monic $f)(x : A)$
   $\to$ (MonicInv f $fmonic$)$(f$ x)(im x) $\equiv x$
MonicInvIsLeftInv f $fmonic$ $x = $ refl

2.4.3 Embeddings

The type is-embedding f denotes the assertion that f is a function all of whose fibers are sub-singletons.

is-embedding : $\{X : \mathcal{U}^{\ \boldsymbol{\cdot}}\}\ \{Y : \mathcal{W}^{\ \boldsymbol{\cdot}}\} \to (X \to Y) \to \mathcal{U} \sqcup \mathcal{W}^{\ \boldsymbol{\cdot}}$
is-embedding $f = \forall\ y \to$ is-subsingleton (fiber f y)

This is a natural way to represent what we usually mean in mathematics by embedding. Observe that an embedding does not simply correspond to an injective map. However, if we assume that the codomain B has unique identity proofs (i.e., B is a *set*), then we can prove that a monic function into B is an embedding. We postpone this until we arrive at the Relations.Truncation module and take up the topic of sets.

It should be clear that embeddings are monic; from a proof $p :$ is-embedding f that f is an embedding we can construct a proof of Monic f. We verify this as follows.

embedding-is-monic : $\{\mathcal{X}\ \mathcal{Y} :$ Universe$\}\ \{X : \mathcal{X}^{\ \boldsymbol{\cdot}}\}\{Y : \mathcal{Y}^{\ \boldsymbol{\cdot}}\}$
                   $(f : X \to Y) \to$ is-embedding $f \to$ Monic f

embedding-is-monic f $femb$ x x' $fxfx' = $ ap pr$_1$ $((femb\ (f\ x))$ fa fb$)$
   where
   fa : fiber f $(f\ x)$
   fa $= x$, refl

   fb : fiber f $(f\ x)$
   fb $= x'$, $(fxfx'^{\ -1})$

Finally, one way to show that a function is an embedding is to first prove it is invertible and then invoke the following theorem.

invertibles-are-embeddings : $\{\mathcal{X}\ \mathcal{Y} :$ Universe$\}\ \{X : \mathcal{X}^{\ \boldsymbol{\cdot}}\}\ \{Y : \mathcal{Y}^{\ \boldsymbol{\cdot}}\}\ (f : X \to Y)$
   $\to$ invertible $f \to$ is-embedding f

invertibles-are-embeddings f $fi = $ equivs-are-embeddings f (invertibles-are-equivs f fi)

2.5 Lifts: making peace with a noncumulative universe hierarchy

This section presents the Prelude.Lifts module of the AgdaUALib, slightly abridged.[20]

2.5.1 The noncumulative universe hierarchy

The hierarchy of universe levels in Agda looks like this:

---

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

$$\mathcal{U}_0 : \mathcal{U}_1 \;\cdot\;, \qquad \mathcal{U}_1 : \mathcal{U}_2 \;\cdot\;, \qquad \mathcal{U}_2 : \mathcal{U}_3 \;\cdot\;, \ldots$$

This means that the type level of $\mathcal{U}_0$ is $\mathcal{U}_1\;\cdot$, and for each n the type level of $\mathcal{U}_n$ is $\mathcal{U}_{n+1}\;\cdot$. It is important to note, however, this does *not* imply that $\mathcal{U}_0 : \mathcal{U}_2\;\cdot$ and $\mathcal{U}_0 : \mathcal{U}_3\;\cdot$, and so on. In other words, Agda's universe hierarchy is *noncummulative*. This makes it possible to treat universe levels more generally and precisely, which is nice. On the other hand (in this author's experience) a noncummulative hierarchy can sometimes make for a nonfun proof assistant.

Luckily, there are ways to overcome this technical issue. We describe general lifting and lowering functions below, and then later, in §4.2.4, we'll see the domain-specific analogs of these tools which turn out to have some nice properties that make them very effective for resolving universe level problems when working with algebra types.

2.5.2 Lifting and lowering

Let us be more concrete about what is at issue here by giving an example. Agda frequently encounters errors during the type-checking process and responds by printing an error message. Often the message has the following form.

<pre style="color:red">
Algebras.lagda:498,20-23 𝒰 != 𝓞 ⊔ 𝒱 ⊔ 𝒰 ⁺ when checking that... has type...
</pre>

This error message means that Agda encountered the universe $\mathcal{U}$ on line 498 (columns 20–23) of the file Algebras.lagda, but was expecting to find the universe $\mathcal{O} \sqcup \mathcal{V} \sqcup \mathcal{U}^+$ instead.

To make these situations easier to deal with, we have developed some domain specific tools for the lifting and lowering of universe levels inhabited by some of the key algebraic types of the UALib. These tools must be applied with some care to avoid making the type theory inconsistent. In particular, we cannot lower the level of a type unless it was previously lifted to a (higher than necessary) universe level.

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

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

The point of having a ramified hierarchy of universes is to avoid Russell's paradox, and this would be subverted if we were to lower the universe of a type that wasn't previously lifted. However, we can prove that if an application of lower is immediately followed by an application of lift, then the result is the identity transformation. Similarly, lift followed by lower is the identity.

```
lower∼lift : {𝒲 𝒳 : Universe}{X : 𝒳 ˙} → lower{𝒲}{𝒳} ∘ lift ≡ id X
lower∼lift = refl

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

Evidently, the proofs are trivial. Nonetheless, we'll find a few holes that these observations can fill.

3   Relation Types

Here we present the submodules of the AgdaUALib's Relations module. In §3.1 we cover *unary* and *binary relations*, which we refer to as "discrete" to contrast them with the ("continuous") *general* and *dependent relations* that we take up in §3.2. We call the latter "continuous relations" because they can have arbitrary arity (general relations) and they can be defined over arbitrary families of types (dependent relations).

3.1   Discrete: predicates, axiom of extensionality, compatibility

This section presents the Relations.Discrete module of the AgdaUALib, slightly abridged.[21]

3.1.1   Unary relations

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

> Pred : $\mathcal{U}$ ˙ → ($\mathcal{W}$: Universe) → $\mathcal{U} \sqcup \mathcal{W}$ ⁺ ˙
> Pred A $\mathcal{W}$ = $A → \mathcal{W}$ ˙

If we are given a type $A : \mathcal{U}$ ˙ , then we think of Pred A $\mathcal{W}$ as the type of a property that inhabitants of A may or may not satisfy. If $P :$ Pred A $\mathcal{W}$, then we can view P as a collection of inhabitants of type A that "satisfy property P," or that "belong to the subset P of A."

Below we will often consider predicates over the class of all algebras of a particular type. Soon we will define the type Algebra $\mathcal{U}$ S of algebras (for some universe level $\mathcal{U}$), and the type Pred (Algebra $\mathcal{U}$ S) $\mathcal{U}$ will be inhabited by maps of the form $\mathbf{A} → \mathcal{U}$ ˙ ; more precisely, given an algebra $\mathbf{A}$: Algebra $\mathcal{U}$ S, we will consider the type Pred $\mathbf{A}$ $\mathcal{U}$ = $\mathbf{A} → \mathcal{U}$ ˙ . We will use predicates over algebra types to specify subclasses of algebras with certain properties.

3.1.2   Membership and inclusion relations

Of course, the UALib includes types that represent element-wise inclusion and subset containment. For example, given a predicate P, we may represent that "x belongs to P" or that "x has property P," by writing either $x \in$ P or P x. The definition of $\in$ is standard (cf. `Relation/Unary.agda` in the Agda Standard Library). Nonetheless, here it is.

> _∈_ : $\{A : \mathcal{X}$ ˙ $\} → A →$ Pred A $\mathcal{Y} → \mathcal{Y}$ ˙
> $x \in P = P$ x

The *subset* relation is denoted, as usual, with the $\subseteq$ symbol and is defined as follows.

> _⊆_ : $\{A : \mathcal{X}$ ˙ $\} →$ Pred A $\mathcal{Y} →$ Pred A $\mathcal{Z} → \mathcal{X} \sqcup \mathcal{Y} \sqcup \mathcal{Z}$ ˙
> $P \subseteq Q = \forall \{x\} → x \in P → x \in Q$

---

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

3.1.3 The axiom of extensionality

In type theory everything is represented as a type and, as we have just seen, this includes subsets. Equality of types is a nontrivial matter, and thus so is equality of subsets when represented as unary predicates. Fortunately, it is straightforward to write down a type that represents what it typically means in informal mathematics to say that two subsets are (extensionally) equal—namely, they contain the same elements. In the UALib we denote this type by $\doteq$ and define it as follows.[22]

$$\_\doteq\_ : \{\mathcal{X}\ \mathcal{Y}\ \mathcal{Z} : \mathsf{Universe}\}\{A : \mathcal{X}\ ^{\cdot}\} \to \mathsf{Pred}\ A\ \mathcal{Y} \to \mathsf{Pred}\ A\ \mathcal{Z} \to \mathcal{X} \sqcup \mathcal{Y} \sqcup \mathcal{Z}\ ^{\cdot}$$
$$P \doteq Q = (P \subseteq Q) \times (Q \subseteq P)$$

A proof of $P \doteq Q$ is a pair $(p\ ,\ q)$ where p is a proof of the first inclusion (that is, $p : P \subseteq Q$), and q is a proof of the second.

If P and Q are definitionally equal (i.e., $P \equiv Q$), then of course both $P \subseteq Q$ and $Q \subseteq P$ will hold, so $P \doteq Q$ will also hold.

$$\mathsf{Pred}\text{-}\equiv\ :\ \{\mathcal{X}\ \mathcal{Y}\ :\ \mathsf{Universe}\}\{A : \mathcal{X}\ ^{\cdot}\}\{P\ Q : \mathsf{Pred}\ A\ \mathcal{Y}\} \to P \equiv Q \to P \doteq Q$$
$$\mathsf{Pred}\text{-}\equiv\ \mathsf{refl} = (\lambda\ z \to z)\ ,\ (\lambda\ z \to z)$$

The converse of Pred-$\equiv$ is not provable in Martin-Löf Type Theory. However, we can postulate it axiomatically if we wish. This is called the *axiom of extensionality* and a type that represents it can be defined in Agda as follows.

$$\mathsf{ext\text{-}axiom}\ :\ \{\mathcal{X}\ :\ \mathsf{Universe}\} \to \mathcal{X}\ ^{\cdot} \to (\mathcal{Y} : \mathsf{Universe}) \to \mathcal{X} \sqcup \mathcal{Y}\ ^{+\ \cdot}$$
$$\mathsf{ext\text{-}axiom}\ A\ \mathcal{Y} = \forall\ (P\ Q : \mathsf{Pred}\ A\ \mathcal{Y}) \to P \doteq Q \to P \equiv Q$$

We treat extensionality in greater detail and generality in §2.3 and §3.4.

3.1.4 Binary Relations

In set theory, a binary relation on a set A is simply a subset of the Cartesian product $A \times A$. As such, we could model these as predicates over the product type $A \times A$, or as relations of type $A \to A \to \mathcal{R}\ ^{\cdot}$ (for some universe $\mathcal{R}$). We define these below.

A generalization of the notion of binary relation is a *relation from A to B*, which we define first and treat binary relations on a single A as a special case.

$$\mathsf{REL} : \mathcal{U}\ ^{\cdot} \to \mathcal{R}\ ^{\cdot} \to (\mathcal{N} : \mathsf{Universe}) \to (\mathcal{U} \sqcup \mathcal{R} \sqcup \mathcal{N}\ ^{+})\ ^{\cdot}$$
$$\mathsf{REL}\ A\ B\ \mathcal{N} = A \to B \to \mathcal{N}\ ^{\cdot}$$

$$\mathsf{Rel} : \mathcal{U}\ ^{\cdot} \to (\mathcal{N} : \mathsf{Universe}) \to \mathcal{U} \sqcup \mathcal{N}\ ^{+\ \cdot}$$
$$\mathsf{Rel}\ A\ \mathcal{N} = \mathsf{REL}\ A\ A\ \mathcal{N}$$

The *kernel* of $f : A \to B$ is defined informally by $\{(x\ ,\ y) \in A \times A : f\,x = f\,y\}$. This can be represented in type theory and Agda in a number of ways, each of which may be useful in a particular context. For example, we could define the kernel as a Sigma type (omitted), or as a unary relation (predicate) over the Cartesian product $A \times A$,

$$\mathsf{KER\text{-}pred} : \{A : \mathcal{U}\ ^{\cdot}\}\{B : \mathcal{R}\ ^{\cdot}\} \to (A \to B) \to \mathsf{Pred}\ (A \times A)\ \mathcal{R}$$
$$\mathsf{KER\text{-}pred}\ g\ (x\ ,\ y) = g\,x \equiv g\,y$$

---

[22]**Unicode Hint**. In agda2-mode type \doteq or \.= to produce $\doteq$

or as a relation from A to A,

> KER-rel : $\{A : \mathcal{U}^{\ \cdot}\} \{B : \mathcal{R}^{\ \cdot}\} \to (A \to B) \to$ Rel $A\ \mathcal{R}$
> KER-rel $g\ x\ y = g\ x \equiv g\ y$

For example, the *identity relation* (the kernel of an injective function) could be represented as a Sigma type (omitted), or a relation, or a predicate.

> 0-rel : $\{A : \mathcal{U}^{\ \cdot}\} \to$ Rel $A\ \mathcal{U}$
> 0-rel $a\ b = a \equiv b$
>
> 0-pred : $\{A : \mathcal{U}^{\ \cdot}\} \to$ Pred $(A \times A)\ \mathcal{U}$
> 0-pred $(a\ ,\ a') = a \equiv a'$

3.1.5   The implication relation

We denote and define *implication* for binary predicates (relations) as follows.

> _on_ : $\{\mathcal{X}\ \mathcal{Y}\ \mathcal{Z} :$ Universe$\}\{A : \mathcal{X}^{\ \cdot}\}\{B : \mathcal{Y}^{\ \cdot}\}\{C : \mathcal{Z}^{\ \cdot}\}$
> $\quad \to \quad (B \to B \to C) \to (A \to B) \to (A \to A \to C)$
>
> R on $g = \lambda\ x\ y \to R\ (g\ x)\ (g\ y)$
>
>
> _⇒_ : $\{\mathcal{W}\ \mathcal{X}\ \mathcal{Y}\ \mathcal{Z} :$ Universe$\}\{A : \mathcal{W}^{\ \cdot}\}\{B : \mathcal{X}^{\ \cdot}\}$
> $\quad \to \quad$ REL $A\ B\ \mathcal{Y} \to$ REL $A\ B\ \mathcal{Z} \to \mathcal{W} \sqcup \mathcal{X} \sqcup \mathcal{Y} \sqcup \mathcal{Z}^{\ \cdot}$
>
> $P \Rightarrow Q = \forall\ \{i\ j\} \to P\ i\ j \to Q\ i\ j$

We can combine **_on_** and **_⇒_** to define a nice, general implication operation. (This is borrowed from the Agda Standard Library; we have merely translated into Type Topology/UALib notation.)

> _=[_]⇒_ : $\{\mathcal{W}\ \mathcal{X}\ \mathcal{Y}\ \mathcal{Z} :$ Universe$\}\{A : \mathcal{W}^{\ \cdot}\}\{B : \mathcal{X}^{\ \cdot}\}$
> $\quad \to \quad\quad$ Rel $A\ \mathcal{Y} \to (A \to B) \to$ Rel $B\ \mathcal{Z} \to \mathcal{W} \sqcup \mathcal{Y} \sqcup \mathcal{Z}^{\ \cdot}$
>
> $P =[\ g\ ]\Rightarrow Q = P \Rightarrow (Q$ on $g)$

3.1.6   Compatibility of functions and binary relations

Before discussing general and dependent relations, we pause to define some types that are useful for asserting and proving facts about *compatibility* of functions with binary relations. The first definition simply lifts a binary relation on A to a binary relation on tuples of type $I \to A$.[23]

> module _ $\{\mathcal{U}\ \mathcal{V}\ \mathcal{W} :$ Universe$\}\{I : \mathcal{V}^{\ \cdot}\}\{A : \mathcal{U}^{\ \cdot}\}$ where
>
> lift-rel : Rel $A\ \mathcal{W} \to (I \to A) \to (I \to A) \to \mathcal{V} \sqcup \mathcal{W}^{\ \cdot}$
> lift-rel $R\ a\ a' = \forall\ i \to R\ (a\ i)\ (a'\ i)$
>
> compatible-fun : $(f : (I \to A) \to A)(R :$ Rel $A\ \mathcal{W}) \to \mathcal{V} \sqcup \mathcal{U} \sqcup \mathcal{W}^{\ \cdot}$
> compatible-fun $f\ R = ($lift-rel $R) =[\ f\ ]\Rightarrow R$

---

[23] N.B. This *relation* lifting is not to be confused with the sort of *universe* lifting that we defined in the Prelude.Lifts module.

We used the slick implication notation in the definition of compatible-fun, but we could have defined it more explicitly, like so.

> compatible-fun' : $(f : (I \to A) \to A)(R : $ Rel $A\ \mathcal{W}) \to \mathcal{V} \sqcup \mathcal{U} \sqcup \mathcal{W}$ ˙
> compatible-fun' $f\ R = \forall\ a\ a' \to ($lift-rel $R)\ a\ a' \to R\ (f\ a)\ (f\ a')$

However, this is a rare case in which the more elegant syntax may result in simpler proofs when applying the definition. (See, for example, compatible-term in the Terms.Operations module.)

3.2 Continuous: arbitrary-sorted relations of arbitrary arity

This section presents the Relations.Continuous module of the AgdaUALib, slightly abridged.[24] In set theory, an n-ary relation on a set A is simply a subset of the n-fold product $A \times A \times \cdots \times A$. As such, we could model these as predicates over the type $A \times \cdots \times A$, or as relations of type $A \to A \to \cdots \to A \to \mathcal{W}$ ˙ (for some universe $\mathcal{W}$). To implement such a relation in type theory, we would need to know the arity in advance, and then somehow form an n-fold arrow $\to$.

A more general and straightforward approach is to instead define an arity type $I : \mathcal{V}$ ˙ , and define the type representing I-ary relations on A as the function type $(I \to A) \to \mathcal{W}$ ˙. Then, if we are specifically interested in an n-ary relation for some natural number n, we could take I to be a finite set (e.g., of type Fin n).

Below we will define ConRel to be the type $(I \to A) \to \mathcal{W}$ ˙ and we will call ConRel the type of *continuous relations.* This generalizes the discrete relations we defined in [Relations.Discrete] (unary, binary, ternary, etc.) since continuous relations can be of arbitrary arity. Still, they are not completely general since they are defined over a single type—said another way, these are "single-sorted" relations—but we will remove this limitation as well when we define the type of *dependent continuous relations*.

Just as Rel $A\ \mathcal{W}$' was the single-sorted special case of the multisorted REL $A\ B\ \mathcal{W}$' type, so too will ConRel $I\ A\ \mathcal{W}$' be the single-sorted version of a completely general type of relations. The latter will represent relations that not only have arbitrary arities, but also are defined over arbitrary families of types.

To be more concrete, given an arbitrary family $A : I \to \mathcal{U}$ ˙ of types, we may have a relation from $A\ i$ to $A\ j$ to $A\ k$ to ..., *ad infinitum*, where the collection represented by the "indexing" type I might not even be enumerable.[25]

We will refer to such relations as *dependent continuous relations* (or *dependent relations*) because the definition of a type that represents them requires dependent types. The DepRel type that we define below manifests this completely general notion of relation.

3.2.1 Continuous relation types

We now define the type *ConRel* that represents predicates (or relations) of arbitrary arity over a single type A. We call this the type of **continuous relations**.[26]

---

[24] For unabridged docs and source code see https://ualib.gitlab.io/Relations.Continuous.html and https://gitlab.com/ualib/ualib.gitlab.io/-/blob/master/UALib/Relations/Continuous.lagda.

[25] Because the collection represented by the indexing type I might not even be enumerable, technically speaking, instead of "$A\ i$ to $A\ j$ to $A\ k$ to ...," we should have written something like "TO $(i : I)$, $A\ i$."

[26] For consistency and readability, throughout the UALib we treat two universe variables with special care. The first of these is 𝕆 which shall be reserved for types that represent *operation symbols* (see AlgebrasSignatures). The second is $\mathcal{V}$ which we reserve for types representing *arities* of relations or operations.

ConRel : $\mathcal{V}^{\,\cdot} \to \mathcal{U}^{\,\cdot} \to (\mathcal{W} : \mathsf{Universe}) \to \mathcal{V} \sqcup \mathcal{U} \sqcup \mathcal{W}^{\,+\,\cdot}$
ConRel $I\ A\ \mathcal{W} = (I \to A) \to \mathcal{W}^{\,\cdot}$

We now define types that are useful for asserting and proving facts about *compatibility* of functions with continuous relations.

module $\_$ $\{\mathcal{U}\ \mathcal{V}\ \mathcal{W} : \mathsf{Universe}\}\ \{I\ J : \mathcal{V}^{\,\cdot}\}\ \{A : \mathcal{U}^{\,\cdot}\}$ where

  lift-con-rel : ConRel $I\ A\ \mathcal{W} \to (I \to J \to A) \to \mathcal{V} \sqcup \mathcal{W}^{\,\cdot}$
  lift-con-rel R ⓐ $= \forall\ (j : J) \to R\ \lambda\ i \to (ⓐ\ i)\ j$

  con-compatible-fun : $(I \to (J \to A) \to A) \to$ ConRel $I\ A\ \mathcal{W} \to \mathcal{V} \sqcup \mathcal{U} \sqcup \mathcal{W}^{\,\cdot}$
  con-compatible-fun $𝕗\ R = \forall\ ⓐ \to$ (lift-con-rel R) ⓐ $\to R\ \lambda\ i \to (𝕗\ i)\ (ⓐ\ i)$

In the definition of con-compatible-fun, we let Agda infer the type of ⓐ, which is $I \to (J \to A)$.

3.2.2   Dependent relations

In this section we exploit the power of dependent types to define a completely general relation type. Specifically, we let the tuples inhabit a dependent function type, where the codomain may depend upon the input coordinate $i :\ I$ of the domain. Heuristically, think of the inhabitants of the following type as relations from $A\ i_1$ to $A\ i_2$ to $A\ i_3$ to .... (This is just for intuition since the domain I need not be enumerable.)

DepRel : $(I : \mathcal{V}^{\,\cdot})(A : I \to \mathcal{U}^{\,\cdot})(\mathcal{W} : \mathsf{Universe}) \to \mathcal{V} \sqcup \mathcal{U} \sqcup \mathcal{W}^{\,+\,\cdot}$
DepRel $I\ A\ \mathcal{W} = \Pi\ A \to \mathcal{W}^{\,\cdot}$

We call DepRel the type of *dependent relations.*

Above we saw lifts of continuous relations and what it means for such relations to be compatible with functions. We conclude this module by defining the (only slightly more complicated) lift of dependent relations, and the type that represents compatibility of a tuple of operations with a dependent relation.

module $\_$ $\{\mathcal{U}\ \mathcal{V}\ \mathcal{W} : \mathsf{Universe}\}\ \{I\ J : \mathcal{V}^{\,\cdot}\}\ \{A : I \to \mathcal{U}^{\,\cdot}\}$ where

  lift-dep-rel : DepRel $I\ A\ \mathcal{W} \to (\forall\ i \to J \to A\ i) \to \mathcal{V} \sqcup \mathcal{W}^{\,\cdot}$
  lift-dep-rel R ⓐ $= \forall\ (j : J) \to R\ (\lambda\ i \to (ⓐ\ i)\ j)$

  dep-compatible-fun : $(\forall\ i \to (J \to A\ i) \to A\ i) \to$ DepRel $I\ A\ \mathcal{W} \to \mathcal{V} \sqcup \mathcal{U} \sqcup \mathcal{W}^{\,\cdot}$
  dep-compatible-fun $𝕗\ R = \forall\ ⓐ \to$ (lift-dep-rel R) ⓐ $\to R\ \lambda\ i \to (𝕗\ i)(ⓐ\ i)$

In the definition of dep-compatible-fun, we let Agda infer the type of ⓐ, which is $(i :\ I) \to J \to A\ i$.

3.3   Quotients: equivalences, class representatives, quotient types

This section presents the Relations.Quotients module of the AgdaUALib, slightly abridged.[27]

---

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

3.3.1   Properties of binary relations

Let $\mathcal{U}$: Universe be a universe and $A : \mathcal{U}$ ˙ a type. In RelationsDiscrete we defined types for representing and reasoning about binary relations on A. In this module we will define types for binary relations that have special properties. The most important special properties of relations are the ones we now define.

```
module _ {𝒰 : Universe} where

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

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

  antisymmetric : {ℛ : Universe}{X : 𝒰 ˙ } → Rel X ℛ → 𝒰 ⊔ ℛ ˙
  antisymmetric _≈_ = ∀ x y → x ≈ y → y ≈ x → x ≡ y

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

The Type Topology library also defines the following *uniqueness-of-proofs* property that a binary relation may or may not possess.

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

Thus, if R : Rel A $\mathcal{R}$, then is-subsingleton-valued R is the assertion that for each pair x y : A there is *at most one proof* of R x y.

3.3.2   Equivalence classes

A binary relation is called a *preorder* if it is reflexive and transitive. An *equivalence relation* is a symmetric preorder. Here are the types we use to represent these concepts in the UALib.

```
module _ {𝒰 ℛ : Universe} where

  is-preorder : {X : 𝒰 ˙ } → Rel X ℛ → 𝒰 ⊔ ℛ ˙
  is-preorder _≈_ = is-subsingleton-valued _≈_ × reflexive _≈_ × transitive _≈_

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

  is-equivalence-relation : {X : 𝒰 ˙ } → Rel X ℛ → 𝒰 ⊔ ℛ ˙
  is-equivalence-relation _≈_ = is-preorder _≈_ × symmetric _≈_
```

An easy first example of an equivalence relation is the kernel of any function. Here is how we prove that the kernel of a function is, indeed, an equivalence relation on the domain of the function.

```
  map-kernel-IsEquivalence : {𝒰 𝒲 : Universe}{A : 𝒰 ˙}{B : 𝒲 ˙}
                             (f : A → B) → IsEquivalence (KER-rel{𝒰}{𝒲} f)
```

```
map-kernel-IsEquivalence {𝒰}{𝒲} f =
  record { rfl = λ x → refl
         ; sym = λ x y x₁ → ≡-sym{𝒲} (f x) (f y) x₁
         ; trans = λ x y z x₁ x₂ → ≡-trans (f x) (f y) (f z) x₁ x₂ }
```

3.3.3 Equivalence classes

If R is an equivalence relation on A, then for each $a : A$, there is an *equivalence class* containing a, which we denote and define by $[\,a\,]\ R :=$ all $b : A$ such that $R\ a\ b$. We often refer to $[\,a\,]\ R$ as the *R-class containing a*.

```
module _ {𝒰 ℛ : Universe} where

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

So, $x \in [\,a\,]\ R$ if and only if $R\ a\ x$, as desired.

We define the type of all R-classes of the relation R as follows.

```
  𝒞 : {A : 𝒰 ˙}{R : Rel A ℛ} → Pred A ℛ → (𝒰 ⊔ ℛ ⁺) ˙
  𝒞 {A} {R} C = Σ a : A , C ≡ ( [ a ] R)
```

If R is an equivalence relation on A, then the *quotient* of A modulo R is denoted by $A\ /\ R$ and is defined to be the collection $\{[\,a\,]\ R \mid a : A\}$ of equivalence classes of R. There are a few ways we could define the quotient with respect to a relation, but we find the following to be the most useful.

```
  _/_ : (A : 𝒰 ˙) → Rel A ℛ → 𝒰 ⊔ (ℛ ⁺) ˙
  A / R = Σ C : Pred A ℛ , 𝒞 {R = R} C
```

We define the following introduction rule for an R-class with a designated representative.

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

If the relation is reflexive, then we have the following elimination rules.[28]

```
  /-refl : {A : 𝒰 ˙}(a b : A){R : Rel A ℛ} → reflexive R → [ a ] R ≡ [ b ] R → R a b

  /-refl a b rfl x = cong-app-pred b (rfl b) (x ⁻¹)


  ⌜_⌝ : {A : 𝒰 ˙}{R : Rel A ℛ} → A / R → A

  ⌜ a ⌝ = | ∥ a ∥ |
```

Later we will need the following additional quotient tools.

```
module _ {𝒰 ℛ : Universe}{A : 𝒰 ˙} where

  open IsEquivalence{𝒰}{ℛ}
```

---

[28] **Unicode Hint**. Type ⌜ and ⌝ as \cul and \cur in agda2-mode.

$/\text{-subset} : \{a\ b : A\}\{R : \textsf{Rel}\ A\ \mathscr{R}\} \to \textsf{IsEquivalence}\ R \to R\ a\ b \to [\ a\ ]\ R \subseteq [\ b\ ]\ R$
$/\text{-subset}\ \{a\}\{b\}\ Req\ Rab\ \{x\}\ Rax = (\textsf{trans}\ Req)\ b\ a\ x\ (\textsf{sym}\ Req\ a\ b\ Rab)\ Rax$

$/\text{-supset} : \{a\ b : A\}\{R : \textsf{Rel}\ A\ \mathscr{R}\} \to \textsf{IsEquivalence}\ R \to R\ a\ b \to [\ a\ ]\ R \supseteq [\ b\ ]\ R$
$/\text{-supset}\ \{a\}\{b\}\ Req\ Rab\ \{x\}\ Rbx = (\textsf{trans}\ Req)\ a\ b\ x\ Rab\ Rbx$

$/\text{-}\doteq : \{a\ b : A\}\{R : \textsf{Rel}\ A\ \mathscr{R}\} \to \textsf{IsEquivalence}\ R \to R\ a\ b \to [\ a\ ]\ R \doteq [\ b\ ]\ R$
$/\text{-}\doteq\ Req\ Rab = /\text{-subset}\ Req\ Rab\ ,\ /\text{-supset}\ Req\ Rab$

3.4 Truncation: continuous propositions, quotient extensionality

This section presents the Relations.Truncation module of the AgdaUALib, slightly abridged.[29] Here we discuss *truncation* and *h-sets* (which we just call *sets*). We first give a brief discussion of standard notions of truncation, and then we describe a viewpoint which seems useful for formalizing mathematics in Agda. Readers wishing to learn more about truncation and proof-relevant mathematics should consult other sources, such as Section 34 and 35 of Martín Escardó's notes [9], or Guillaume Brunerie, Truncations and truncated higher inductive types, or Section 7.1 of the HoTT book [13].[30]

3.4.1 Background and motivation[31]

In general, we may have many inhabitants of a given type, hence (via Curry-Howard) many proofs of a given proposition. For instance, suppose we have a type X and an identity relation $\_\equiv_x\_$ on X so that, given two inhabitants of X, say, $a\ b : X$, we can form the type $a \equiv_x b$. Suppose p and q inhabit the type $a \equiv_x b$; that is, p and q are proofs of $a \equiv_x b$, in which case we write $p\ q : a \equiv_x b$. We might then wonder whether and in what sense are the two proofs p and q the equivalent.

We are asking about an identity type on the identity type $\equiv_x$, and whether there is some inhabitant, say, r of this type; i.e., whether there is a proof $r : p \equiv_{x1} q$ that the proofs of $a \equiv_x b$ are the same. If such a proof exists for all $p\ q : a \equiv_x b$, then the proof of $a \equiv_x b$ is unique; as a property of the types X and $\equiv_x$, this is sometimes called *uniqueness of identity proofs*.

Now, perhaps we have two proofs, say, $r\ s : p \equiv_{x1} q$ that the proofs p and q are equivalent. Then of course we wonder whether $r \equiv_{x2} s$ has a proof! But at some level we may decide that the potential to distinguish two proofs of an identity in a meaningful way (so-called *proof-relevance*) is not useful or desirable. At that point, say, at level k, we would be naturally inclined to assume that there is at most one proof of any identity of the form $p \equiv_{xk} q$. This is called truncation (at level k) (see, e.g., the Truncation section of Escardó's notes [9]).

3.4.2 Sets

In homotopy type theory, a type X with an identity relation $\equiv_x$ is called a *set* (or *0-groupoid*) if for every pair $x\ y : X$ there is at most one proof of $x \equiv_x y$. In other words, the type X, along

---

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

[30] **Remark**. Agda now has a built in type called Prop which may provide some or all of what we develop in this module. (See the Prop Section of agda.readthedocs.io.) However, we don't use it anywhere in the UALib; it seems we get along just fine without it.

[31] The remarks in this subsection serve to introduce novices to the basic notion of truncation. Readers already familiar with this notion may wish to skip to the next subsection.

with it's equality type $\equiv_x$, form a *set* if for all $x\ y : X$ there is at most one proof of $x \equiv_x y$.

This notion is formalized in the Type Topology library using the types is-set which is defined using the is-subsingleton type that we saw earlier (§2.4) as follows.[32]

is-set : $\mathcal{U}$ ˙ $\to \mathcal{U}$ ˙
is-set $X = (x\ y : X) \to$ is-subsingleton $(x \equiv y)$

Thus, the pair $(X, \equiv_x)$ forms a set iff it satisfies $\forall\ x\ y : X \to$ is-subsingleton $(x \equiv_x y)$.

The function to-Σ-$\equiv$, which we also import, is part of Escardó's characterization of equality in Sigma types ([9]). It is defined as follows.

module hide-to-Σ-$\equiv$ {$\mathcal{U}\ \mathcal{W}$: Universe} where

to-Σ-$\equiv$: $\{X : \mathcal{U}$ ˙ $\}\ \{A : X \to \mathcal{W}$ ˙ $\}\ \{\sigma\ \tau : \Sigma\ A\}$
  $\to \Sigma\ p : |\ \sigma\ | \equiv |\ \tau\ |$, (transport $A\ p\ \|\ \sigma\ \|) \equiv \|\ \tau\ \|$
  $\to \sigma \equiv \tau$

to-Σ-$\equiv$ (refl $\{x = x\}$, refl $\{x = a\}) =$ refl $\{x = (x , a)\}$

open import MGS-Embeddings using (to-Σ-$\equiv$) public

We will use is-embedding, is-set, and to-Σ-$\equiv$ in the next subsection to prove that a monic function into a set is an embedding.

3.4.3 Injective functions are set embeddings

Before moving on to define propositions, we discharge an obligation mentioned but left unfulfilled in the embeddings section of the PreludeInverses module. Recall, we described and imported the is-embedding type, and we remarked that an embedding is not simply a monic function. However, if we assume that the codomain is truncated so as to have unique identity proofs (i.e., is a set), then we can prove that any monic function into that codomain will be an embedding. On the other hand, embeddings are always monic, so we will end up with an equivalence. To prepare for this, we define a type _$\Longleftrightarrow$_ with which to represent such equivalences.

$\Longleftrightarrow$: $\{\mathcal{U}\ \mathcal{W}$: Universe$\} \to \mathcal{U}$ ˙ $\to \mathcal{W}$ ˙ $\to \mathcal{U} \sqcup \mathcal{W}$ ˙
$X \Longleftrightarrow Y = (X \to Y) \times (Y \to X)$

module _ {$\mathcal{U}\ \mathcal{W}$: Universe}$\{A : \mathcal{U}$ ˙$\}\{B : \mathcal{W}$ ˙$\}$ where

monic-is-embedding|sets : $(f : A \to B) \to$ is-set $B \to$ Monic $f \to$ is-embedding f

monic-is-embedding|sets $f\ Bset\ fmon\ b\ (a , fa{\equiv}b)\ (a' , fa'{\equiv}b) = \gamma$
  where
  faa' : $f\ a \equiv f\ a'$
  faa' $=$ $\equiv$-Trans $(f\ a)\ (f\ a')\ fa{\equiv}b\ (fa'{\equiv}b\ ^{-1})$

  aa' : $a \equiv a'$
  aa' $= fmon\ a\ a'\ faa'$

  $\mathscr{A}$: _ $\to$ _ ˙
  $\mathscr{A}\ a = f\ a \equiv b$

---

[32] As Escardó explains, "at this point, with the definition of these notions, we are entering the realm of univalent mathematics, but not yet needing the univalence axiom."

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

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

In stating the previous result, we introduce a new convention to which we hope to adhere. Whenever a result holds only for sets, we will add the special suffix |sets, which hopefully calls to mind the standard mathematical notation for the restriction of a function to a subset of its domain.

Embeddings are always monic, so we conclude that when a function's codomain is a set, then that function is an embedding if and only if it is monic.

embedding-iff-monic|sets : (f : A → B) → is-set B
————————————————
→   is-embedding f ⟺ Monic f

embedding-iff-monic|sets f $Bset$ = (embedding-is-monic f), (monic-is-embedding|sets f $Bset$)

3.4.4 Propositions

Sometimes we will want to assume that a type X is a *set*. As we just learned, this means there is at most one proof that two inhabitants of X are the same. Analogously, for predicates on X, we may wish to assume that there is at most one proof that an inhabitant of X satisfies the given predicate. If a unary predicate satisfies this condition, then we call it a (unary) *proposition*. We now define a type that captures this concept.

module _ {𝒰 : Universe} where

Pred$_1$: 𝒰 ˙ → (𝒲 : Universe) → 𝒰 ⊔ 𝒲 $^+$ ˙
Pred$_1$ A 𝒲 = Σ P : (Pred A 𝒲) , ∀ x → is-subsingleton (P x)

(Recall that Pred A 𝒲 is simply the function type A → 𝒲 ˙.)

The principle of *proposition extensionality* asserts that logically equivalent propositions are equivalent. That is, if we have P Q : Pred$_1$ and | P | ⊆ | Q | and | Q | ⊆ | P |, then P ≡ Q. This is formalized as follows (cf. the section "Prop extensionality and the powerset" of [9]).

prop-ext : (𝒰 𝒲 : Universe) → (𝒰 ⊔ 𝒲) $^+$ ˙
prop-ext 𝒰 𝒲 = ∀ {A : 𝒰 ˙}{P Q : Pred$_1$ A 𝒲 } → | P | ⊆ | Q | → | Q | ⊆ | P | → P ≡ Q

Recall, we defined the relation _≐_ for predicates as follows: P ≐ Q = (P ⊆ Q) × (Q ⊆ P). Therefore, if we assume PropExt A 𝒲 {P}{Q} holds, then it follows that P ≡ Q.

prop-ext' : (A : 𝒰 ˙)(𝒲 : Universe){P Q : Pred$_1$ A 𝒲} → prop-ext A 𝒲
→ | P | ≐ | Q | → P ≡ Q

prop-ext' A 𝒲 pe hyp = pe (fst hyp) (snd hyp)

Thus, for truncated predicates P and Q, if PropExt holds, then P ⊆ Q × Q ⊆ P → P ≡ Q, which is a useful extensionality principle.

3.4.5 Binary propositions

Given a binary relation R, it may be necessary or desirable to assume that there is at most one way to prove that a given pair of elements is R-related. If this is true of R, then we call R a

binary proposition.[33]

As above, we use the is-subsingleton type of the Type Topology library to impose this truncation assumption on a binary relation.[34]

> Pred$_2$: $\mathcal{U}$ $^{\cdot}$ $\rightarrow$ ($\mathcal{W}$: Universe) $\rightarrow$ $\mathcal{U}$ $\sqcup$ $\mathcal{W}$ $^{+}$ $^{\cdot}$
> Pred$_2$ A $\mathcal{W}$ = Σ R : (Rel A $\mathcal{W}$) , $\forall$ x y $\rightarrow$ is-subsingleton (R x y)

(Recall, Rel A $\mathcal{W}$ is simply the function type $A \rightarrow A \rightarrow \mathcal{W}$ $^{\cdot}$.)

3.4.6 Quotient extensionality

We need a (subsingleton) identity type for congruence classes over sets so that we can equate two classes even when they are presented using different representatives. Proposition extensionality is precisely what we need to accomplish this. (Notice that we don't require *function extensionality* (§2.3) here.)

> module _ {$\mathcal{U}$ $\mathcal{R}$: Universe}{A : $\mathcal{U}$ $^{\cdot}$}{**R** : Pred$_2$ A $\mathcal{R}$} where
>
>   class-extensionality : prop-ext $\mathcal{U}$ $\mathcal{R}$ $\rightarrow$ {u v : A} $\rightarrow$ IsEquivalence | **R** |
> $\rightarrow$ | **R** | u v $\rightarrow$ [u] | **R** | $\equiv$ [v] | **R** |
>
>   class-extensionality *pe* {u}{v} *Reqv Ruv* = γ
>   where
>   P Q : Pred$_1$ A $\mathcal{R}$
>   P = (λ a $\rightarrow$ | **R** | u a) , (λ a $\rightarrow$ ‖ **R** ‖ u a)
>   Q = (λ a $\rightarrow$ | **R** | v a) , (λ a $\rightarrow$ ‖ **R** ‖ v a)
>
> α : [u] | **R** | $\subseteq$ [v] | **R** |
> α *ua* = fst (/-$\doteq$ *Reqv Ruv*) *ua*
>
> β : [v] | **R** | $\subseteq$ [u] | **R** |
> β *va* = snd (/-$\doteq$ *Reqv Ruv*) *va*
>
>   PQ : P $\equiv$ Q
>   PQ = (prop-ext' *pe* (α , β))
>
> γ : [u] | **R** | $\equiv$ [v] | **R** |
> γ = ap fst PQ
>
>
>   to-subtype-⟦⟧ : {C D : Pred A $\mathcal{R}$}{c : $\mathcal{C}$ C}{d : $\mathcal{C}$ D}
> $\rightarrow$ ($\forall$ C $\rightarrow$ is-subsingleton ($\mathcal{C}${R = | **R** |} C))
> $\rightarrow$ $C \equiv D \rightarrow (C$, $c) \equiv (D$, $d)$
>
>   to-subtype-⟦⟧ {$D = D$}{c}{d} *ssA CD* = to-Σ-$\equiv$ (*CD* , *ssA D* (transport $\mathcal{C}$ *CD c*) *d*)

---

[33] This is another example of *proof-irrelevance* since, if R is a binary proposition and we have two proofs of R x y, then we can assume that the proofs are indistinguishable or that any distinctions are irrelevant.

[34] Using the definition is-subsingleton-valued from § 3.3.1, we could have defined Pred$_2$ by Σ R : (Rel A $\mathcal{W}$) , is-subsingleton-valued R, but this seems less transparent than our explicit definition.

```
class-extensionality' : prop-ext 𝒰 ℛ → {u v : A} → (∀ C → is-subsingleton (𝒞 C))
   →                     IsEquivalence | R | → | R | u v → ⟦ u ⟧ ≡ ⟦ v ⟧

class-extensionality' pe {u}{v} ssA Reqv Ruv = γ
   where
      CD : [ u ] | R | ≡ [ v ] | R |
      CD = class-extensionality pe Reqv Ruv

      γ : ⟦ u ⟧ ≡ ⟦ v ⟧
      γ = to-subtype-⟦⟧ ssA CD
```

3.4.7 Continuous proposition types

We defined a type called ConRel in the RelationsContinuous module to represent relations of
arbitrary arity. Naturally, we define a type of *truncated continuous relations*, the inhabitants
of which we will call *continuous propositions*.

```
ConProp : 𝒱 ˙ → 𝒰 ˙ → (𝒲 : Universe) → 𝒱 ⊔ 𝒰 ⊔ 𝒲 ⁺ ˙
ConProp I A 𝒲 = Σ P : (ConRel I A 𝒲) , ∀ a → is-subsingleton (P a)

con-prop-ext : 𝒱 ˙ → 𝒰 ˙ → (𝒲 : Universe) → 𝒱 ⊔ 𝒰 ⊔ 𝒲 ⁺ ˙
con-prop-ext I A 𝒲 = {P Q : ConProp I A 𝒲} →| P | ⊆ | Q | → | Q | ⊆ | P | → P ≡ Q
```

The point of this is that if we assume con-prop-ext I A 𝒲 holds for some I, A and 𝒲, then we
can prove that logically equivalent continuous propositions of type ConProp I A 𝒲 are equival-
ent.

```
con-prop-ext' : (I : 𝒱 ˙)(A : 𝒰 ˙)(𝒲 : Universe){P Q : ConProp I A 𝒲}
   →              con-prop-ext I A 𝒲
                  ————————————————————
   →              | P | ≐ | Q | → P ≡ Q

con-prop-ext' I A 𝒲 pe hyp = pe | hyp | ∥ hyp ∥
```

While we're at it, we might as well take the abstraction one step further and define *truncated
dependent relations*, which we'll call *dependent propositions*.

```
DepProp : (I : 𝒱 ˙)(A : I → 𝒰 ˙)(𝒲 : Universe) → 𝒱 ⊔ 𝒰 ⊔ 𝒲 ⁺ ˙
DepProp I A 𝒲 = Σ P : (DepRel I A 𝒲) , ∀ a → is-subsingleton (P a)


dep-prop-ext : (I : 𝒱 ˙)(A : I → 𝒰 ˙)(𝒲 : Universe) → 𝒱 ⊔ 𝒰 ⊔ 𝒲 ⁺ ˙

dep-prop-ext I A 𝒲 = {P Q : DepProp I A 𝒲 } → | P | ⊆ | Q | → | Q | ⊆ | P | → P ≡ Q
```

Applying the extensionality principle for dependent relations is no harder than applying the
special cases of this principle defined earlier.

```
dep-prop-ext' : (I : 𝒱 ˙)(A : I → 𝒰 ˙)(𝒲 : Universe)
                 {P Q : DepProp I A 𝒲} → dep-prop-ext I A 𝒲
                 ————————————————————————-
   →              | P | ≐ | Q | → P ≡ Q

dep-prop-ext' I A 𝒲 pe hyp = pe | hyp | ∥ hyp ∥
```

4   Algebra Types

A standard way to define algebraic structures in type theory is using record types. However, we feel the dependent pair (or Sigma) type (§2.1.5) is more natural, as it corresponds semantically to the existential quantifier of logic. Therefore, many of the important types of the UALib are defined as Sigma types. In this section, we use function types and Sigma types to define the types of *operations* and *signatures* (§4.1), *algebras* (§4.2), and *product algebras* (§4.3), *congruence relations*4.4, and *quotient algebras*4.4.2.

4.1   Signatures: types for operations and signatures

This section presents the Algebras.Signatures module of the AgdaUALib, slightly abridged.[35]

4.1.1   Operation type

We begin by defining the type of *operations*, and give an example (the projections).

```
module _ {𝒰 : Universe} where

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

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

The type Op encodes the arity of an operation as an arbitrary type $I : 𝒱$, 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.

4.1.2   Signature type

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

```
Signature : (𝒪 𝒱 : Universe) → (𝒪 ⊔ 𝒱) ⁺ ˙
Signature 𝒪 𝒱 = Σ F : 𝒪 ˙ , (F → 𝒱 ˙)
```

As mentioned in the section on Relations of arbitrary arity in the Relations.Continuous module, 𝒪 will always denote the universe of *operation symbol* types, while 𝒱 is the universe of *arity* types.

In the Prelude module we defined special syntax for the first and second projections—namely, $|\_|$ and $\|\_\|$, respectively. Consequently, if $\{S : \text{Signature } 𝒪 \, 𝒱\}$ is a signature, then $| S |$ denotes the set of operation symbols, and $\| S \|$ denotes the arity function. If $f : | S |$ is an operation symbol in the signature S, then $\| S \| \, f$ is the arity of f.

---

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

4.1.2.1 Example

Here is how we might define the signature for monoids as a member of the type Signature 𝟢 𝒱.

```
data monoid-op : 𝟢 ˙ where
  e : monoid-op
  · : monoid-op

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

As expected, the signature for a monoid consists of two operation symbols, e and ·, and a function λ { e → 𝟘 ; · → 𝟚 } which maps e to the empty type 𝟘 (since e is the nullary identity) and maps · to the two element type 𝟚 (since · is binary).[36]

4.2 Algebras: types for algebras, operation interpretation, and compatibility

This section presents the Algebras.Algebras module of the AgdaUALib, slightly abridged.[37]

4.2.1 The Algebra type

For a fixed signature S : Signature 𝟢 𝒱 and universe 𝒰, we define the type of *algebras in the signature S* (or *S-algebras*) and with *domain* (or *carrier* or *universe*) A : 𝒰 as follows.

```
Algebra : (𝒰 : Universe)(S : Signature 𝟢 𝒱) → 𝟢 ⊔ 𝒱 ⊔ 𝒰 ⁺ ˙

Algebra 𝒰 S = Σ A : 𝒰 ˙ , ((f : | S |) → Op (‖ S ‖ f) A)
```

We could refer to an inhabitant of this type as a "∞-algebra" because its domain can be an arbitrary type, say, A : 𝒰 and need not be truncated at some level; in particular, A need not be a *set*. (See the discussion in §3.4.2.)

We might take this opportunity to define the type of "0-algebras" (algebras whose domains are sets), which is probably closer to what most of us think of when doing informal universal algebra. However, below we will only need to know that the domains of our algebras are sets in a few places in the UALib, so it seems preferable to work with general (∞-)algebras throughout and then assume uniqueness of identity proofs explicitly and only where needed.

4.2.2 Operation interpretation syntax

We now define a convenient shorthand for the interpretation of an operation symbol. This looks more similar to the standard notation one finds in the literature as compared to the double bar notation we started with, so we will use this new notation almost exclusively in the remaining modules of the UALib.

```
_ˆ_ : (f : | S |)(𝐀 : Algebra 𝒰 S) → (‖ S ‖ f → | 𝐀 |) → | 𝐀 |

f ˆ 𝐀 = λ a → (‖ 𝐀 ‖ f) a
```

---

[36] The types 𝟘 and 𝟚 are defined in the MGS-MLTT module of the Type Topology library.

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

So, if $f : \mid S \mid$ is an operation symbol in the signature S, and if $a : \parallel S \parallel f \to \mid A \mid$ is a tuple of the appropriate arity, then $(f \mathbin{\hat{}} A)\, a$ denotes the operation f interpreted in A and evaluated at a.

4.2.3   Arbitrarily many variable symbols

We sometimes want to assume that we have at our disposal an arbitrary collection X of variable symbols such that, for every algebra A, no matter the type of its domain, we have a surjective map $h : X \to \mid A \mid$ from variables onto the domain of A. We may use the following definition to express this assumption when we need it.

> _↠_ : $\{S : \mathsf{Signature}\ \mathfrak{O}\ \mathscr{V}\}\{\mathcal{U}\ \mathfrak{X} : \mathsf{Universe}\} \to \mathfrak{X}\ ^{\textstyle\cdot} \to \mathsf{Algebra}\ \mathcal{U}\ S \to \mathfrak{X} \sqcup \mathcal{U}\ ^{\textstyle\cdot}$
> $X ↠ A = \Sigma\ h : (X \to \mid A \mid)\ ,\ \mathsf{Epic}\ h$

Now we can assert, in a specific module, the existence of the surjective map described above by including the following line in that module's declaration, like so.

> $\mathsf{module}\ \_\ \{\mathbb{X} : \{\mathcal{U}\ \mathfrak{X} : Universe\}\{X : \mathfrak{X}\}(A : \mathsf{Algebra}\ \mathcal{U}\ S) \to X ↠ A\}\ \mathsf{where}$

Then $\mathsf{fst}(\mathbb{X}\ A)$ will denote the surjective map $h : X \to \mid A \mid$, and $\mathsf{snd}(\mathbb{X}\ A)$ will be a proof that h is surjective.

4.2.4   Lifts of algebras

Here we define some domain-specific lifting tools for our operation and algebra types.

> $\mathsf{module}\ \_\ \{\mathfrak{O}\ \mathscr{V} : \mathsf{Universe}\}\{S : \mathsf{Signature}\ \mathfrak{O}\ \mathscr{V}\}\ \mathsf{where}\ \text{–}\ \Sigma\ F : \mathfrak{O}\ ^{\textstyle\cdot}\ ,\ (\ F \to \mathscr{V}\ ^{\textstyle\cdot}\ )\}\ where$
>
> $\mathsf{lift\text{-}op} : \{\mathcal{U} : \mathsf{Universe}\}\{I : \mathscr{V}\ ^{\textstyle\cdot}\}\{A : \mathcal{U}\ ^{\textstyle\cdot}\} \to ((I \to A) \to A) \to (\mathscr{W} : \mathsf{Universe})$
> $\qquad \to ((I \to \mathsf{Lift}\{\mathscr{W}\}\ A) \to \mathsf{Lift}\ \{\mathscr{W}\}\ A)$
>
> $\mathsf{lift\text{-}op}\ f\ \mathscr{W} = \lambda\ x \to \mathsf{lift}\ (f\ (\lambda\ i \to \mathsf{Lift.lower}\ (x\ i)))$
>
> $\mathsf{open\ algebra}$
>
> $\mathsf{lift\text{-}alg} : \{\mathcal{U} : \mathsf{Universe}\} \to \mathsf{Algebra}\ \mathcal{U}\ S \to (\mathscr{W} : \mathsf{Universe}) \to \mathsf{Algebra}\ (\mathcal{U} \sqcup \mathscr{W})\ S$
> $\mathsf{lift\text{-}alg}\ A\ \mathscr{W} = \mathsf{Lift}\ \mid A \mid\ ,\ (\lambda\ (f : \mid S \mid) \to \mathsf{lift\text{-}op}\ (f \mathbin{\hat{}} A)\ \mathscr{W})$
>
> $\mathsf{lift\text{-}alg\text{-}record\text{-}type} : \{\mathcal{U} : \mathsf{Universe}\} \to \mathsf{algebra}\ \mathcal{U}\ S \to (\mathscr{W} : \mathsf{Universe}) \to \mathsf{algebra}\ (\mathcal{U} \sqcup \mathscr{W})\ S$
> $\mathsf{lift\text{-}alg\text{-}record\text{-}type}\ A\ \mathscr{W} = \mathsf{mkalg}\ (\mathsf{Lift}\ (\mathsf{univ}\ A))\ (\lambda\ (f : \mid S \mid) \to \mathsf{lift\text{-}op}\ ((\mathsf{op}\ A)\ f)\ \mathscr{W})$

We use the function $\mathsf{lift\text{-}alg}$ to resolve errors that arise when working in Agda's noncummulative hierarchy of type universes. (See the discussion in $\mathsf{Prelude.Lifts}$.)

4.2.5   Compatibility of binary relations

If A is an algebra and R a binary relation, then $\mathsf{compatible}\ A\ R$ will represents the assertion that R is *compatible* with all basic operations of A. Recall, informally this means for every operation symbol $f : \mid S \mid$ and all pairs $a\ a' : \parallel S \parallel f \to \mid A \mid$ of tuples from the domain of A, the following implication holds:

if $R\ (a\ i)\ (a'\ i)$ for all i, then $R\ ((f \mathbin{\hat{}} A)\ a)\ ((f \mathbin{\hat{}} A)\ a')$.

The formal definition representing this notion of compatibility is easy to write down since we already have a type that does all the work.

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

    compatible : (**A** : Algebra 𝒰 S) → Rel | **A** | 𝒲 → 𝒪 ⊔ 𝒰 ⊔ 𝒱 ⊔ 𝒲 ˙
    compatible **A** R = ∀ f → compatible-fun (f ˆ **A**) R

Recall the compatible-fun type was defined in Relations.Discrete module.

4.2.6 Compatibility of continuous relations

Next we define a type that represents *compatibility of a continuous relation* with all operations of an algebra. Fist, we define compatibility with a single operation.

module _ {𝒰 𝒲 : Universe} {S : Signature 𝒪 𝒱} {**A** : Algebra 𝒰 S} {I : 𝒱 ˙} where

    con-compatible-op : | S | → ConRel I | **A** | 𝒲 → 𝒰 ⊔ 𝒱 ⊔ 𝒲 ˙
    con-compatible-op f R = con-compatible-fun (λ _ → (f ˆ **A**)) R

In case it helps the reader understand con-compatible-op, we redefine it explicitly without the help of con-compatible-fun.

con-compatible-op' : | S | → ConRel I | **A** | 𝒲 → 𝒰 ⊔ 𝒱 ⊔ 𝒲 ˙
con-compatible-op' f R = ∀ 𝕒 → (lift-con-rel R) 𝕒 → R (λ i → (f ˆ **A**) (𝕒 i))

where we have let Agda infer the type of 𝕒, which is $(i : I) → \| S \| f → | **A** |$.

With con-compatible-op in hand, it is a trivial matter to define a type that represents *compatibility of a continuous relation with an algebra*.

con-compatible : ConRel I | **A** | 𝒲 → 𝒪 ⊔ 𝒰 ⊔ 𝒱 ⊔ 𝒲 ˙
con-compatible R = ∀ (f : | S |) → con-compatible-op f R

4.3 Products: types for products over arbitrary classes of algebras

This section presents the Algebras.Products module of the AgdaUALib, slightly abridged.[38] We begin this module by assuming a signature S : Signature 𝒪 𝒱 which is then present and available throughout the module. Because of this, in contrast to our (highly abridged) descriptions of previous modules, we present the first few lines of the Algebras.Products module in full. They are as follows.

{-# OPTIONS –without-K –exact-split –safe #-}
open import Algebras.Signatures using (Signature; 𝒪; 𝒱)
module Algebras.Products {S : Signature 𝒪 𝒱} where
open import Algebras.Algebras hiding (𝒪; 𝒱) public

Notice that we import the Signature type from the Algebras.Signatures module first, before the module line, so that we may use it to declare the signature S as a parameter of the Algebras.Products module.

The product of S-algebras is defined as follows.

---

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

$$\prod : \{\mathscr{U}\ \mathscr{I} : \mathsf{Universe}\}\{I : \mathscr{I}\ ^{\textbf{.}}\}(\mathscr{A} : I \to \mathsf{Algebra}\ \mathscr{U}\ S\ ) \to \mathsf{Algebra}\ (\mathscr{I} \sqcup \mathscr{U})\ S$$

$$\prod \mathscr{A} = (\forall\ i \to |\ \mathscr{A}\ i\ |)\ ,\qquad\qquad\qquad \text{– domain of the product algebra}$$

$$\qquad \lambda\ f\ a\ i \to (f\ \hat{}\ \mathscr{A}\ i)\ \lambda\ x \to a\ x\ i \qquad\text{– basic operations of the product algebra}$$

4.3.1 Products of classes of algebras

An arbitrary class $\mathscr{K}$ of algebras is represented as a predicate over the type $\mathsf{Algebra}\ \mathscr{U}\ S$, for some universe $\mathscr{U}$ and signature S. That is, $\mathscr{K} : \mathsf{Pred}\ (\mathsf{Algebra}\ \mathscr{U}\ S)\ \_.$[39] Later we will formally state and prove that the product of all subalgebras of algebras in such a class belongs to $\mathsf{SP}(\mathscr{K})$ (subalgebras of products of algebras in $\mathscr{K}$). That is, $\prod \mathsf{S}(\mathscr{K}) \in \mathsf{SP}(\mathscr{K})$. This turns out to be a nontrivial exercise. In fact, it is not even clear (at least not to this author) how one should express the product of an entire class of algebras as a dependent type. However, if one ponders this for a while, the right type will eventually reveal itself, and will then seem obvious.[40] The solution is the class-product type whose construction is the main goal of this section.

First, we need a type that will serve to index the class, as well as the product of its members.[41]

    module _ $\{\mathscr{U}\ \mathscr{X} : \mathsf{Universe}\}\{X : \mathscr{X}\ ^{\textbf{.}}\}$ where

        $\mathfrak{I} : \mathsf{Pred}\ (\mathsf{Algebra}\ \mathscr{U}\ S)(\mathsf{ov}\ \mathscr{U}) \to (\mathscr{X} \sqcup \mathsf{ov}\ \mathscr{U})\ ^{\textbf{.}}$

        $\mathfrak{I}\ \mathscr{K} = \Sigma\ \mathbf{A} : (\mathsf{Algebra}\ \mathscr{U}\ S)\ ,\ (\mathbf{A} \in \mathscr{K}) \times (X \to |\ \mathbf{A}\ |)$

Notice that the second component of this dependent pair type is $(\mathbf{A} \in \mathscr{K}) \times (X \to |\ \mathbf{A}\ |)$. In previous versions of the UALib this second component was simply $\mathbf{A} \in \mathscr{K}$, until we realized that adding the type $X \to |\ \mathbf{A}\ |$ is quite useful. Later we will see exactly why, but for now suffice it to say that a map of type $X \to |\ \mathbf{A}\ |$ may be viewed abstractly as an *ambient context*, or more concretely, as an assignment of *values* in $|\ \mathbf{A}\ |$ to *variable symbols* in X. When computing with or reasoning about products, while we don't want to rigidly impose a context in advance, want do want to lay our hands on whatever context is ultimately assumed. Including the "context map" inside the index type $\mathfrak{I}$ of the product turns out to be a convenient way to achieve this flexibility.

Taking the product over the index type $\mathfrak{I}$ requires a function that maps an index $i : \mathfrak{I}$ to the corresponding algebra. Each index $i : \mathfrak{I}$ denotes a triple, say, $(\mathbf{A}\ ,\ p\ ,\ h)$, where

    $\mathbf{A} : \mathsf{Algebra}\ \mathscr{U}\ S,\qquad p : \mathbf{A} \in \mathscr{K},\qquad h : X \to |\ \mathbf{A}\ |,$

so the function mapping an index to the corresponding algebra is simply the first projection.

    $\mathfrak{A} : (\mathscr{K} : \mathsf{Pred}\ (\mathsf{Algebra}\ \mathscr{U}\ S)(\mathsf{ov}\ \mathscr{U})) \to \mathfrak{I}\ \mathscr{K} \to \mathsf{Algebra}\ \mathscr{U}\ S$

    $\mathfrak{A}\ \mathscr{K} = \lambda\ (i : (\mathfrak{I}\ \mathscr{K})) \to |\ i\ |$

---

[39] The underscore is merely a placeholder for the universe of the predicate type and doesn't concern us here.

[40] At least this was our experience, but readers are encouraged to try to come up with a type that represents the product of all members of an inhabitant of a predicate over $\mathsf{Algebra}\ \mathscr{U}\ S$, or even an arbitrary predicate.

[41] **Notation**. Given a signature $S : \mathsf{Signature}\ \mathbb{O}\ \mathscr{V}$, the type $\mathsf{Algebra}\ \mathscr{U}\ S$ has universe $\mathbb{O} \sqcup \mathscr{V} \sqcup \mathscr{U}\ ^{+}$. In the UALib, such universes abound, and $\mathbb{O}$ and $\mathscr{V}$ remain fixed throughout the library. So, for notational convenience, we define the following shorthand for universes of this form: $\mathsf{ov}\ \mathscr{U} = \mathbb{O} \sqcup \mathscr{V} \sqcup \mathscr{U}\ ^{+}$

Finally, we define class-product which represents the product of all members of $\mathcal{K}$.

> class-product : Pred (Algebra $\mathcal{U}$ S)(ov $\mathcal{U}$) → Algebra ($\mathcal{X}$ ⊔ ov $\mathcal{U}$) S
>
> class-product $\mathcal{K}$ = $\prod$ ($\mathfrak{A}$ $\mathcal{K}$)

If $p : \mathbf{A} \in \mathcal{K}$ and $h : X \to | \mathbf{A} |$, then we can think of the triple $(\mathbf{A} , p , h) \in \mathfrak{I} \mathcal{K}$ as an index over the class, and so we can think of $\mathfrak{A} (\mathbf{A} , p , h)$ (which is simply $\mathbf{A}$) as the projection of the product $\prod$ ($\mathfrak{A}$ $\mathcal{K}$) onto the $(\mathbf{A} , p , h)$-th component.

4.4   Congruences: types for congruences and quotient algebras

This section presents the Algebras.Congruences module of the AgdaUALib, slightly abridged.[42] A *congruence relation* of an algebra $\mathbf{A}$ is defined to be an equivalence relation that is compatible with the basic operations of $\mathbf{A}$. This concept can be represented in a number of different ways in type theory. For example, we define both a Sigma type Con and a record type Congruence, each of which captures the informal notion of congruence, and each one is useful in certain contexts. (We will see examples later.)

> Con : {$\mathcal{U}$: Universe}($\mathbf{A}$: Algebra $\mathcal{U}$ S) → ov $\mathcal{U}$ ˙
> Con {$\mathcal{U}$} $\mathbf{A}$ = Σ θ : (Rel | $\mathbf{A}$ | $\mathcal{U}$) , IsEquivalence θ × compatible $\mathbf{A}$ θ
>
> record Congruence {$\mathcal{U}$ $\mathcal{W}$: Universe} ($\mathbf{A}$: Algebra $\mathcal{U}$ S) : ov $\mathcal{W}$ ⊔ $\mathcal{U}$ ˙ where
>   constructor mkcon
>   field
>   ⟨_⟩ : Rel | $\mathbf{A}$ | $\mathcal{W}$
>   Compatible : compatible $\mathbf{A}$ ⟨_⟩
>   IsEquiv : IsEquivalence ⟨_⟩
>
> open Congruence

4.4.1   Example

We defined the zero relation 0-rel in the Relations.Discrete module, and we now demonstrate how to build the trivial congruence out of this relation.

The relation 0-rel is equivalent to the identity relation ≡ and these are obviously both equivalences. In fact, we already proved this of ≡ in the Prelude.Equality module, so we simply apply the corresponding proofs.

> module _ {$\mathcal{U}$: Universe} where
>
>   0-IsEquivalence : {A : $\mathcal{U}$ ˙ } → IsEquivalence{$\mathcal{U}$}{$A = A$} 0-rel
>   0-IsEquivalence = record { rfl = ≡-rfl; sym = ≡-sym; trans = ≡-trans }

Next we formally record another obvious fact—namely, that 0-rel is compatible with all operations of all algebras.

> module _ {$\mathcal{U}$: Universe} where

---

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

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

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

Finally, we have the ingredients need to construct the zero congruence of any algebra we like.

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

4.4.2 Quotient Algebras

An important construction in universal algebra is the quotient of an algebra **A** with respect to a congruence relation θ of **A**. This quotient is typically denote by **A** / θ and Agda allows us to define and express quotients using the standard notation.[43]

```
__/__ : {𝒰 𝓡 : Universe}(A : Algebra 𝒰 S) → Congruence{𝒰}{𝓡} A → Algebra (𝒰 ⊔ 𝓡 ⁺) S

A ╱ θ = ( | A | / ⟨ θ ⟩ ) ,   – the domain of the quotient algebra

      λ f 𝒂 → [[ (f ^ A) (λ i → | ‖ 𝒂 i ‖ |) ]]   – the basic operations of the quotient algebra
```

4.4.3 Examples

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

```
module _ {𝒰 𝓡 : Universe} where

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

  Zero╱ θ = λ x x₁ → x ≡ x₁
```

Finally, the following elimination rule is sometimes useful.

```
╱-refl : {A : Algebra 𝒰 S}(θ : Congruence{𝒰}{𝓡} A){a a' : | A |}
   → [[ a ]]{⟨ θ ⟩} ≡ [[ a' ]] → ⟨ θ ⟩ a a'

╱-refl θ refl = IsEquivalence.rfl (IsEquiv θ) _
```

5   Concluding Remarks

We've reached the end of Part 1 of our three-part series describing the AgdaUALib. Part 2 will cover homomorphism, terms, and subalgebras, and Part 3 will cover free algebras, equational classes of algebras (i.e., varieties), and Birkhoff's HSP theorem.

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

---

[43]**Unicode Hints**. Produce the ╱ symbol in agda2-mode by typing `\---` and then `C-f` a number of times.

──── **References** ────

1   Clifford Bergman. *Universal Algebra: fundamentals and selected topics*, volume 301 of *Pure and Applied Mathematics (Boca Raton)*. CRC Press, Boca Raton, FL, 2012.

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

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

4   Jesper Carlström. A constructive version of birkhoff's theorem. *Mathematical Logic Quarterly*, 54(1):27–34, 2008. URL: `https://onlinelibrary.wiley.com/doi/abs/10.1002/malq.200710023`, `arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/malq.200710023`, `doi:https://doi.org/10.1002/malq.200710023`.

5   Alonzo Church. A formulation of the simple theory of types. *The Journal of Symbolic Logic*, 5(2):56–68, 1940. URL: `http://www.jstor.org/stable/2266170`.

6   William DeMeo. The Agda Universal Algebra Library and Birkhoff's Theorem in Dependent Type Theory. *CoRR*, abs/2101.10166, 2021. source code: `https://gitlab.com/ualib/ualib.gitlab.io`. URL: `https://arxiv.org/abs/2101.10166`, `arXiv:2101.10166`.

7   William DeMeo. The Agda Universal Algebra Library, Part 2: Structure. *CoRR*, abs/2103.09092, 2021. Source code: `https://gitlab.com/ualib/ualib.gitlab.io`. URL: `https://arxiv.org/abs/2103.09092`, `arXiv:2103.09092`.

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

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

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

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

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

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

14 Bas Spitters and Eelis van der Weegen. Type classes for mathematics in type theory. *CoRR*, abs/1102.1323, 2011. URL: `http://arxiv.org/abs/1102.1323`, `arXiv:1102.1323`.

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

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

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

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

A   Imports from the Type Topology library

We have attempted to keep a careful accounting of every type definition that the AgdaUALib imports from Martín Escardó's Type Topology library and below we provide a list of these dependencies.

   Although we present these here as import statements, they are not imported in the UALib in a single file (though they could have been). Instead, since dependencies reveal the foundations upon which the library is built, and since we are interested in foundations, we try to keep all dependencies in the foreground, and resist the temptation to store them all in a single file that we never have to think about again.

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

open import Identity-Type renaming (_≡_ to infix 0 _≡_)

open import MGS-MLTT using (_∘_; domain; codomain; transport; _≡⟨_⟩_; _∎; − _×_; pr_1; pr_2; -Σ; Π;
   ¬; id; _∼_; _+_; 𝟘; 𝟙; 𝟚; _⇔_; lr-implication; rl-implication; id; _$^{-1}$; ap)

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

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

open import MGS-Powerset renaming (_∈_ to _∈$_0$_; _⊆_ to _⊆$_0$_; ∈-is-subsingleton to ∈$_0$-is-subsingleton)
   using (𝒫; equiv-to-subsingleton; powersets-are-sets'; subset-extensionality'; propext; _holds; Ω)

open import MGS-Embeddings using (Nat; NatΠ; NatΠ-is-embedding; is-embedding; pr_1-embedding; ∘-embedding;
   is-set; _↪_; embedding-gives-ap-is-equiv; embeddings-are-lc; ×-is-subsingleton; id-is-embedding)

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

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

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

   Notice that we carefully specify which definitions and results we want to import from each of Escardo's modules. This is not absolutely necessary, and we could have simply used, e.g., open import MGS-MLTT public, omitting the using (_∘_; domain; ... ap) However, being specific here has advantages. Besides helping us avoid naming conflicts, it makes explicit which components of the type theory we are using.