# 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.

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

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

**Related Version** hosted on arXiv
*Part 2, Part 3*: `http://arxiv.org/a/demeo_w_1`

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

## Acknowledgements

The author thanks Martín Escardó for creating the Type Topology library and teaching us about it at the 2019 Midlands Graduate School in Computing Science [8].

## 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 [13], 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 [15] (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 [9] (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 [8]. 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 installment of a three-part series of papers describing the AgdaUALib. The second part is [6],

[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.

covering homomorphisms, terms, and subalgebras. The third part is [7], 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 [14] and Escardó's Introduction to Univalent Foundations of Mathematics with Agda [8].

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

- *Introduction to Univalent Foundations of Mathematics with Agda*, Escardó [8].
- *Dependent Types at Work*, Bove and Dybjer [2].
- *Dependently Typed Programming in Agda*, Norell and Chapman [12].
- *Formalization of Universal Algebra in Agda*, Gunther, Gadea, Pagano [9].
- *Programming Languages Foundations in Agda*, Philip Wadler [19].

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* [6]..
- *The Agda UALib, Part 3: free algebras, equational classes, and Birkhoff's theorem* [7]..

[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] This module defines (or imports) the most basic and important types of *Martin-Löf dependent type theory* (MLTT). Although this is standard, we take this opportunity to highlight aspects of the UALib syntax that may differ from that of "standard Agda."

### 2.1.1 Logical foundations

The UALib is based on a minimal version of MLTT that is the same or very close to the type theory on which Martín Escardó's Type Topology Agda library is based. 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 [14].

We begin by noting that only a very small collection of objects is assumed at the jumping-off point for MLTT. We have the *primitive types* ($\mathbb{0}$, $\mathbb{1}$, and $\mathbb{N}$, denoting the empty type, one-element type, and natural numbers), the *type formers* ($+$, $\Pi$, $\Sigma$, $\mathsf{Id}$, denoting *binary sum*, *product*, *sum*, and the *identity* type), and an infinite collection of *type universes* (types of types) and universe variables to denote them. Like Escardó's, our universe variables are typically upper-case caligraphic letters from the latter half of the English alphabet (e.g., $\mathcal{U}$, $\mathcal{V}$, $\mathcal{W}$, etc.).

#### 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.

```
{-# OPTIONS –without-K –exact-split –safe #-}
```
(1)

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 [16];
- *–exact-split* makes Agda accept only definitions that are *judgmental* equalities; see [18];
- *–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 [17] and [18].

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

[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`.

### 2.1.2 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[5]

```
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 a nonindented line of code appears.

### 2.1.3 Agda Universes

For the very small amount of background we require about the notion of *type universe* (or *level*), we refer the reader to the brief section on universe-levels in the Agda documentation.[6]

Throughout the AgdaUALib we use many of the nice tools that Martín Escardó has developed and made available in the Type Topology repository of Agda code for the *Univalent Foundations* of mathematics.[7] The first of these is the Universes module which we import as follows.

```
open import Universes public
```

Since we use the public directive, the Universes module will be available to all modules that import the present module (Prelude.Preliminaries).

The Universes module includes a number of symbols used to denote universes in Agda. In particular, following Escardó, we refer to universes using capitalized script letters from near the end of the alphabet, e.g., $\mathcal{U}$, $\mathcal{V}$, $\mathcal{W}$, $\mathcal{X}$, $\mathcal{Y}$, $\mathcal{Z}$, etc. To this list we add one more that we will use later to denote the universe level of operation symbol types (defined in the Algebras.Signatures module).

```
variable 𝓞 : Universe
```

The Universes module also provides elegant notation for the few primitive operations on universes that Agda supports. Specifically, the $\cdot$ operator maps a universe level $\mathcal{U}$ to the type Set $\mathcal{U}$, and the latter has type Set (lsuc $\mathcal{U}$). The Agda level lzero is renamed $\mathcal{U}_0$, so $\mathcal{U}_0\,\cdot$ is an

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

[6] See `https://agda.readthedocs.io/en/v2.6.1.3/language/universe-levels.html`.

[7] Escardó has written an outstanding set of notes called Introduction to Univalent Foundations of Mathematics with Agda, which we highly recommend to anyone looking for more details than we provide here about MLTT and Univalent Foundations/HoTT in Agda. [8].

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

To justify the introduction of this somewhat nonstandard notation for universe levels, Escardó points out that the Agda library uses Level for universes (so what we write as $\mathcal{U}\ \dot{}$ is written Set $\mathcal{U}$ in standard Agda), but in univalent mathematics the types in $\mathcal{U}\ \dot{}$ need not be sets, so the standard Agda notation can be a bit confusing, especially to newcomers.

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

### 2.1.4 Dependent types

#### Sigma types (dependent pairs)

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

The dependent product type is defined in the Type Topology in a standard way. For pedagogical purposes we repeat the definition here.[8]

```
record Σ {𝓤 𝓥} {A : 𝓤 ̇ } (B : A → 𝓥 ̇ ) : 𝓤 ⊔ 𝓥 ̇ where
  constructor _,_
  field
    pr₁ : A
    pr₂ : B pr₁
```

Agda's default syntax for this type is $\Sigma\ \lambda(x : A) \to B$, but we prefer the notation $\Sigma\ x : A\ ,\ B$, which is closer to the standard syntax described in the preceding paragraph. Fortunately, the Type Topology library makes the preferred notation available with the following type definition and syntax declaration (see [8, Σ types]).[9]

```
-Σ : {𝓤 𝓥 : Universe} (A : 𝓤 ̇ ) (B : A → 𝓥 ̇ ) → 𝓤 ⊔ 𝓥 ̇
-Σ A B = Σ B

syntax -Σ A (λ x → B) = Σ x : A , B
```

A special case of the Sigma type is the one in which the type $B$ doesn't depend on $A$. This is the usual Cartesian product, defined in Agda as follows.

```
_×_ : 𝓤 ̇ → 𝓥 ̇ → 𝓤 ⊔ 𝓥 ̇
A × B = Σ x : A , B
```

---

[8] In the UALib we put such redundant definitions inside "hidden" modules so that they doesn't conflict with the original definitions which we import and use. It may seem odd to define something in a hidden module only to import and use an alternative definition, but we do this in order to exhibit all of the types on which the UALib depends while ensuring that this cannot be misinterpreted as a claim to originality.

[9] **Attention!** The symbol : that appears in the special syntax defined here for the Σ type, and below for the Π type, is not the ordinary colon; rather, it is the symbol obtained by typing `\:4` in agda2-mode.

### Pi types (dependent functions)

Given universes $\mathcal{U}$ and $\mathcal{V}$, a type $A : \mathcal{U}\ ^\cdot$, and a type family $B : A \to \mathcal{V}\ ^\cdot$, the *Pi type* (or *dependent function type*) is denoted by $\Pi\ x : A\ ,\ B\ x$ and generalizes the function type $A \to B$ by letting the type $B\ x$ of the codomain depend on the value $x$ of the domain type. The dependent function type is defined in the Type Topology in a standard way. For the reader's benefit, however, we repeat the definition here. (In the UALib this definition is included in a named or "hidden" module.)

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

To make the syntax for $\Pi$ conform to the standard notation for Pi types, Escardó uses the same trick as the one used above for Sigma types.[9]

```
-Π : (A : 𝒰 ̇ )(B : A → 𝒲 ̇ ) → 𝒰 ⊔ 𝒲 ̇
-Π A B = Π B

syntax -Π A (λ x → b) = Π x : A , b
```

Once we have studied the types, defined in the Type Topology library and repeated here for illustration purposes, the original definitions are imported like so.

```
open import Sigma-Type
open import MGS-MLTT using (pr₁; pr₂; _×_; -Σ; Π; -Π)
```

We use the public directive so that the types are available to all modules that import the present module.

### Notation for the first and second projections

The definition of $\Sigma$ (and thus $\times$) includes the fields $\mathsf{pr}_1$ and $\mathsf{pr}_2$ representing the first and second projections out of the product. Sometimes we prefer to denote these projections by $|\_|$ and $\|\_\|$, respectively. However, for emphasis or readability we alternate between these and the following standard notations: $\mathsf{pr}_1$ and fst for the first projection, $\mathsf{pr}_2$ and snd for the second. We define these alternative notations for projections as follows.[10]

```
module _ {𝒰 : Universe} where

  |_| fst : {A : 𝒰 ̇ }{B : A → 𝒱 ̇ } → Σ B → A
  | x , y | = x
  fst (x , y) = x

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

[10] We put these definitions inside an *anonymous module*, which starts with the module keyword followed by an underscore (instead of a module name). The purpose is to move some of the postulated typing judgments—the "parameters" of the module (e.g., $\mathcal{U}$ : Universe)—out of the way so they don't obfuscate the definitions inside the module. N.B. In library documentation like the present paper we often omit such module directives, while the collection of html pages at `ualib.org`—most current and comprehensive documentation of the UALib—omits nothing.

## 2.2 Equality: definitional equality and transport

This section presents the Prelude.Equality module of the AgdaUALib, slightly abridged.[11]

### 2.2.1 Definitional equality

Here we discuss what is probably the most important type in MLTT. It is called *definitional equality*. As long as we know how to generate equivalence relations, this concept is easily understood, at least heuristically, as the following slogan.

> *Definitional equality is the equivalence relation generated by definitions.*

For readers unfamiliar with generating equivalence relations, let's make this precise. Start with the binary relation :=, which relates $x$ and $y$ if and only if $y$ *is the definition of* $x$, in which case we naturally write $x := y$. Now take the reflexive, symmetric, transitive closure of :=.[12] The result is the *definitional equality* relation.

In [10, §1.11, page 85], Per Martin-Löf describes definitional equality as follows:

> "Definitional equality is intensional equality, or equality of meaning (synonymy)... Definitional equality ≡ is a relation between linguistic expressions; it should not be confused with equality between objects (sets, elements of a set etc.)... Definitional equality is the equivalence relation generated by abbreviatory definitions, changes of bound variables and the principle of substituting equals for equals. Therefore it is decidable, but not in the sense that a ≡ b ∨ ¬(a ≡ b) holds, simply because a ≡ b is not a proposition in the sense of the present theory."
>
> Per Martin-Löf,
> *Padua Lecture Notes* [11]

The datatype we use to represent this equality is a standard one and is defined in the Identity-Type module of the Type Topology library. Apart from superficial syntactic differences it is equivalent to the identity type used in other Agda libraries. In the UALib we make the ≡ relation available by importing it from the Identity-Type module, but we repeat the definition here for easy reference.

```
data _≡_ {𝓤} {A : 𝓤 ̇ } : A → A → 𝓤 ̇ where refl : {x : A} → x ≡ x
```

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 ≡ 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 ≡. Here are the (trivial) proofs of symmetry and transitivity of ≡.[13]

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

≡-sym : {x y : A} → x ≡ y → y ≡ x
```

[11] 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`.

[12] To obtain the reflexive closure of a binary relation $R \subseteq A \times A$, add to $R$ all pairs $(x , x)$ in $A \times A$. Obtain the symmetric closure of a relation R by taking the union of R and its inverse {(y , x) : (x , y) ∈ R}. Readers should now try to infer the meaning of transitive closure.

[13] To reduce reader strain, we omit easily inferred typing judgments—in this case, 𝓤 : Universe and $A : \mathcal{U}$ ̇—which would normally be parameters in the module context or in the type definition itself.

```
≡-sym refl = refl

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

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

The only difference between ≡-symmetric and ≡-sym (resp., ≡-transitive and ≡-trans) is that ≡-sym (resp., ≡-trans) 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.[14]

```
_⁻¹ : {x y : A} → 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.

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

### 2.2.2 Transport

Alonzo Church characterized equality by declaring two things equal if and only if no property (predicate) can distinguish them (see [5]). In other terms, $x$ and $y$ are equal if and only if 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, as follows.[15]

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

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

See [8] for a discussion of transport.[16]

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 $f : A \to B$, two elements $x\ x' : A$ of the domain, and an identity proof $p : x \equiv x'$, then we obtain a proof of $f\ x \equiv f\ x'$ by simply applying the ap function like so, ap $f\ p : f\ x \equiv f\ x'$. Escardó defines *ap* in the

[14] **Unicode Hints** (agda2-mode): `\^-\^1` ⇝ `⁻¹`; `\Mii\Mid` ⇝ *id*; `\.` ⇝ ·. In general, for information about a character, place the cursor on the character and type `M-x describe-char` (or `M-x h d c`).

[15] We can't show every line of code from the UALib in this paper, and we have chosen to omit the lines indicating that our redundant definitions of some functions (e.g., transport and ap) occur inside named "hidden" modules before the original definitions are then imported from the Type Topology library. As mentioned above, we do this in order to facilitate presentation of the UALib library in a clear and reasonably self-contained way, without claiming credit for type definitions that are not our own.

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

Type Topology library as follows.

```
ap : {A : 𝓤 ̇ }{B : 𝓥 ̇ }(f : A → B){a b : A} → a ≡ b → f a ≡ f b
ap f {a} p = transport (λ - → f a ≡ f -) p (refl {x = f a})
```

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
```

## 2.3 Extensionality: types for postulating function extensionality

This section presents the Prelude.Extensionality module of the AgdaUALib, slightly abridged.[17]

### 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?

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.)

[17] 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`.

### 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 $\forall x \to fx \equiv gx$. Here is how this notion of equality is expressed as a type in the Type Topology library.

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

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

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

In informal settings, this so-called "pointwise equality of functions" is typically what one means when one asserts that two functions are "equal."[18] 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.* [8]

### 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 Set$\omega$ universe).[19] The types global-funext and global-dfunext are defined in the Type Topology library as follows.

```
global-funext : 𝓤ω
global-funext = ∀ {𝓤 𝓥} → funext 𝓤 𝓥

global-dfunext : 𝓤ω
global-dfunext = ∀ {𝓤 𝓥} → dfunext 𝓤 𝓥
```

The next two types define the converse of function extensionality.[20]

```
extfun : {A : 𝓤 ̇}{B : 𝓦 ̇}{f g : A → B} → f ≡ g → f ∼ g
extfun refl _ = refl
```

---

[18] If one assumes the *univalence axiom* of Homotopy Type Theory, then the _∼_ relation is equivalent to equality of functions. See the section "Function extensionality from univalence" of Escardó's notes [8].

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

[20] 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. Later we realized that a function called happly, which is nearly identical to extdfun, is already defined in the MGS-FunExt-from-Univalence module of the Type Topology library.

```
extdfun : {A : 𝓤 ̇ }{B : A → 𝓦 ̇ }(f g : Π B) → f ≡ g → f ∼ g
extdfun _ _ refl _ = refl
```

Though it may seem obvious to some readers, we wish to emphasize the important conceptual distinction between two flavors of type definition. We do so by comparing the definitions of funext and extfun.

In the definition of funext, the codomain is a generic type (namely, $(\mathcal{U} \sqcup \mathcal{V})^+ \,\dot{}$), and the right-hand side of the defining equation of funext is an assertion (which may or may not hold). In the definition of extfun, the codomain is an assertion (namely, $f \sim g$), and the right-hand side of the defining equation is a proof of this assertion.

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, $fe : \mathsf{funext}\ \mathcal{U}\ \mathcal{V}$ (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.

```
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)
```

We are finally ready to fulfill our promise of a type that provides 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.[21] 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 $a$.

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

### 2.4.1 Epics (surjective functions)

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

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

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

```
EpicInv : {A : 𝓤 ̇} {B : 𝓦 ̇} (f : A → B) → Epic f → B → 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 ∘, which is already defined in the Type Topology library,

[21] 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.

as follows.

```
_∘_ : {X : 𝓤 ̇ } {Y : 𝓦 ̇ }{Z : Y → 𝓦 ̇ } → Π Z → (f : X → Y) → (x : X) → Z (f x)
g ∘ f = λ x → g (f x)

module _ {𝓤 𝓦 : Universe}{fe : funext 𝓦 𝓦}{A : 𝓤 ̇ }{B : 𝓦 ̇ } where

  EpicInvIsRightInv : (f : A → B)(fE : Epic f) → f ∘ (EpicInv f fE) ≡ id B
  EpicInvIsRightInv f fE = fe (λ x → InvIsInv f (fE x))
```

### 2.4.2 Monics (injective functions)

We say that a function $g : A \to B$ is *monic* (or *injective*) if it doesn't map distinct elements to a common point. The Monic type, which we now define, manifests this property. (To reduce reader strain, we omit some easily inferred typing judgments like $A : \mathcal{U}\ ̇$ and $B : \mathcal{W}\ ̇$.)

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

We obtain the (left-)inverse by applying the function MonicInv to $g$ and a proof that $g$ is monic.

```
MonicInv : (f : A → B) → Monic f → (b : B) → Image f ∋ b → A
MonicInv f _ = λ b Imf∋b → Inv f Imf∋b
```

The function defined by MonicInv $f$ $fM$ is a (left-)pseudo-inverse of $f$, and the proof is trivial.

```
MonicInvIsLeftInv : {f : A → B}{fM : Monic f}{x : A} → (MonicInv f fM)(f x)(im x) ≡ x
MonicInvIsLeftInv = refl
```

### 2.4.3 Embeddings

The type is-embedding $f$ asserts that $f$ is a function all of whose fibers are subsingletons.

```
is-embedding : {X : 𝓤 ̇ } {Y : 𝓦 ̇ } → (X → Y) → 𝓤 ⊔ 𝓦 ̇
is-embedding f = ∀ y → 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.

Finding a proof that a function is an embedding isn't always easy, but one path that is often straightforward is to first prove that the function is invertible and then invoke the following theorem.

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

Finally, embeddings are monic; from a proof $p$ : is-embedding $f$ that $f$ is an embedding we can construct a proof of Monic $f$. We confirm this as follows.

```
embedding-is-monic : {X : 𝓧 ̇}{Y : 𝓨 ̇}(f : X → Y) → is-embedding f → Monic f
embedding-is-monic f femb a b fafb = ap pr₁ ((femb (f a)) fa fb)
  where
  fa : fiber f (f a)
```

```
fa = a , refl

fb : fiber f (f a)
fb = b , (fafb ⁻¹)
```

## 2.5 Lifts: making peace with a noncumulative universe hierarchy

This section presents the Prelude.Lifts module of the AgdaUALib, slightly abridged.[22]

### 2.5.1 Agda's universe hierarchy

The hierarchy of universes in Agda is structured as follows: $\mathcal{U}\ \cdot : \mathcal{U}\ ^{+}\ \cdot$, $\mathcal{U}\ ^{+}\ \cdot : \mathcal{U}\ ^{++}\ \cdot$, 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. Specifically, in certain situations, the noncummulativity makes it unduly difficult to convince Agda that a program or proof is correct.

Presently (in §2.5.2) we will describe general lifting and lowering functions that help us overcome this technical issue. Later (in §4.2.4) we provide some domain-specific analogs of these tools. We will prove some nice properties that make these effective mechanisms 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 considering a typical example. Agda frequently encounters errors during the type-checking process and responds by printing an error message. Often the message has the following form.

```
Birkhoff.lagda:498,20-23
𝓤 != 𝓞 ⊔ 𝓥 ⊔ (𝓤 ⁺) when checking that... has type...
```

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

There are some general "lifting and lowering" tools that make these situations easier to deal with. These 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
```

[22] 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.

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.

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

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

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

# 3 Relation Types

## 3.1 Discrete: predicates, axiom of extensionality, compatibility

This section presents the Relations.Discrete module of the AgdaUALib, slightly abridged.[23] 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 relations" to contrast them with the ("continuous") *general* and *dependent relations* that we introduce 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).

### 3.1.1 Unary relations

In set theory, given two sets $A$ and $P$, we say that $P$ is a *subset* of $A$, and we write $P \subseteq A$, just in case $\forall\ x\ (x \in P \to x \in A)$. We need a mechanism for representing this notion in Agda. A typical approach is to use a *predicate* type, denoted by Pred.

Given two universes $\mathcal{U}\ \mathcal{W}$ and a type $A : \mathcal{U}\ ̇$, the type Pred $A\ \mathcal{W}$ represents *properties* that inhabitants of $A$ may or may not satisfy. We write $P$ : Pred $A\ \mathcal{U}$ to represent the semantic concept of the collection of inhabitants of $A$ that satisfy (or belong to) $P$. Here is the definition.[24]

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

Later we consider predicates over the class of algebras in a given signature. In the Algebras module we will define the type Algebra $\mathcal{U}\ S$ of $S$-algebras with domain type $\mathcal{U}\ ̇$, and the type Pred (Algebra $\mathcal{U}\ S$) $\mathcal{W}$ will represent classes of $S$-algebras with certain properties.

### 3.1.2 Membership and inclusion relations

Like the Agda Standard Library, the UALib includes types that represent the *element inclusion* and *subset inclusion* relations from set theory. 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. Nonetheless, here it is.[24]

---

[23] 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`.

[24] cf. `Relation/Unary.agda` in the Agda Standard Library.

```
_∈_ : A → Pred A 𝓨 → 𝓨 ̇
x ∈ P = P x
```

The *subset* relation is denoted, as usual, with the ⊆ symbol and is defined as follows.[24]

```
_⊆_ : Pred A 𝓨 → Pred A 𝓩 → 𝓧 ⊔ 𝓨 ⊔ 𝓩 ̇
P ⊆ Q = ∀ {x} → x ∈ P → x ∈ Q
```

### 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 ≐ and define it as follows.[25]

```
_≐_ : Pred A 𝓨 → Pred A 𝓩 → 𝓧 ⊔ 𝓨 ⊔ 𝓩 ̇
P ≐ Q = (P ⊆ Q) × (Q ⊆ P)
```

A proof of $P \doteq Q$ is a pair $(p\ ,\ q)$ where $p : P \subseteq Q$ and $q : Q \subseteq P$ are proofs of the first and second inclusions, respectively. If $P$ and $Q$ are definitionally equal (i.e., $P \equiv Q$), then both $P \subseteq Q$ and $Q \subseteq P$ hold, so $P \doteq Q$ also holds, as we now confirm.

```
Pred-≡ : {P Q : Pred A 𝓨} → P ≡ Q → P ≐ Q
Pred-≡ refl = (λ z → z) , (λ z → z)
```

The converse of Pred-≡ 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 this axiom is the following.

```
ext-axiom : 𝓧 ̇ → (𝓨 : Universe) → 𝓧 ⊔ 𝓨 ⁺ ̇
ext-axiom A 𝓨 = ∀ (P Q : Pred A 𝓨) → P ≐ Q → P ≡ Q
```

Note that the type ext-axiom does not itself postulate the axiom of extensionality. It merely defines the axiom. If we want to postulate it, we must assume we have a witness, or inhabitant of the type. We could do this in Agda in a number of ways, but probably the easiest is to simply add the witness as a parameter to a module, like so.[26]

```
module ext-axiom-postulated {𝓧 𝓨 : Universe}{A : 𝓧 ̇} {ea : ext-axiom A 𝓨} where
```

We treat other notions of extensionality in §2.3 and §3.4.

#### Predicates toolbox

Here is a small collection of tools that will come in handy later. The first provides convenient notation for asserting that the image of a function (the first argument) is contained in a predicate (the second argument).

---

[25] **Unicode Hints**. In agda2-mode, `\.=` ⇝ ≐, `\u+` ⇝ ⊎, `\b0` ⇝ 𝟘, `\B0` ⇝ **0**.

[26] Agda also has a postulate mechanism that we could use, but this would require omitting the *–safe* pragma from the OPTIONS directive at the start of the module.

```
Im_⊆_ : (A → B) → Pred B 𝓔 → 𝓧 ⊔ 𝓔 ̇
Im f ⊆ S = ∀ x → f x ∈ S
```

The following inductive type represents *disjoint union*.[25]

```
data _∪̇_ {𝓧 𝓨 : Universe}(A : 𝓧 ̇) (B : 𝓨 ̇) : 𝓧 ⊔ 𝓨 ̇ where
  inj₁ : (x : A) → A ∪̇ B
  inj₂ : (y : B) → A ∪̇ B
```

And this can be used to represent *union*, as follows.

```
_∪_ : {𝓧 𝓨 𝓩 : Universe}{A : 𝓧 ̇} → Pred A 𝓨 → Pred A 𝓩 → Pred A _
P ∪ Q = λ x → x ∈ P ∪̇ x ∈ Q
```

The *empty set* is naturally represented by the *empty type*, $\mathbb{0}$, and the latter is defined in Type Topology's Empty-Type module.[25,27]

```
open import Empty-Type using (𝟘)

∅ : {𝓧 : Universe}{A : 𝓧 ̇} → Pred A 𝓤₀
∅ = λ _ → 𝟘
```

Before closing our little predicates toolbox, let's insert a type that provides a natural way to represent *singletons*.

```
⁅_⁆ : {𝓧 : Universe}{A : 𝓧 ̇} → A → Pred A _
⁅ x ⁆ = x ≡_
```

### 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 such a relation as a (unary) predicate over the product type $A \times A$, or as an inhabitant of the function type $A \to A \to \mathcal{R}\ ̇$ (for some universe $\mathcal{R}$). Note, however, this is not the same as a unary predicate over the function type $A \to A$ since the latter has type $(A \to A) \to \mathcal{R}\ ̇$, while a binary relation should have type $A \to (A \to \mathcal{R}\ ̇)$.

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.

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

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

#### The kernel of a function

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 in a number of ways, each of which may be useful in a particular context. For example, we could define the kernel to be an inhabitant of a (binary) relation type, a (unary) predicate type, a (curried) Sigma type, or an (uncurried) Sigma type. The

[27] The empty type is defined in Type Topology's Empty-Type module as an inductive type with no constructors; that is, `data 𝟘 {𝓤} : 𝓤 ̇ where – (empty body)`.

alternatives are defined in the UALib as follows.

```
ker : (A → B) → Rel A ℛ
ker g x y = g x ≡ g y

kernel : (A → B) → Pred (A × A) ℛ
kernel g (x , y) = g x ≡ g y

ker-sigma : (A → B) → 𝒰 ⊔ ℛ ̇
ker-sigma g = Σ x ꞉ A , Σ y ꞉ A , g x ≡ g y

ker-sigma' : (A → B) → 𝒰 ⊔ ℛ ̇
ker-sigma' g = Σ (x , y) ꞉ (A × A) , g x ≡ g y
```

Similarly, the *identity relation* (which is equivalent to the kernel of an injective function) can be represented using any one of the following types.[25]

```
𝟎 : Rel A 𝒰
𝟎 a b = a ≡ b

𝟎-pred : Pred (A × A) 𝒰
𝟎-pred (a , a') = a ≡ a'

𝟎-sigma : 𝒰 ̇
𝟎-sigma = Σ a ꞉ A , Σ b ꞉ A , a ≡ b

𝟎-sigma' : 𝒰 ̇
𝟎-sigma' = Σ (x , y) ꞉ (A × A) , x ≡ y
```

Finally, the *total relation* over $A$, which in set theory is the full Cartesian product $A \times A$, can be represented using the one-element type from Type Topology's Unit-Type module, as follows.[25,28]

```
open import Unit-Type using (𝟙)

𝟏 : Rel A 𝓤₀
𝟏 a b = 𝟙
```

### 3.1.5 The implication relation

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

```
_on_ : (B → B → C) → (A → B) → (A → A → C)
R on g = λ x y → R (g x) (g y)

_⇒_ : REL A B 𝒴 → REL A B 𝓧 → 𝒲 ⊔ 𝒳 ⊔ 𝒴 ⊔ 𝓧 ̇
P ⇒ Q = ∀ {i j} → P i j → 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.)

```
_=[_]⇒_ : Rel A 𝒴 → (A → B) → Rel B 𝓧 → 𝒲 ⊔ 𝒴 ⊔ 𝓧 ̇
P =[ g ]⇒ Q = P ⇒ (Q on g)
```

[28] The one-element type is defined in Type Topology's Unit-Type module as an inductive type with a single constructor, denoted ⋆, as follows: `data 𝟙 {𝓤} : 𝓤 ̇ where ⋆ : 𝟙`.

### 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 $I \to A$.[29]

```
module _ {𝓤 𝓥 𝓦 : Universe}{I : 𝓥 ̇}{A : 𝓤 ̇} where

  lift-rel : Rel A 𝓦 → (I → A) → (I → A) → 𝓥 ⊔ 𝓦 ̇
  lift-rel R a a' = ∀ i → R (a i) (a' i)

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

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 → A) → A)(R : Rel A 𝓦) → 𝓥 ⊔ 𝓤 ⊔ 𝓦 ̇
compatible-fun' f R = ∀ a a' → (lift-rel R) a a' → 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.[30] 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.

---

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

[30] 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`.

To be more concrete, given an arbitrary family $A : I \to \mathcal{U}\ ^{\cdot}$ 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.[31]

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**.[32]

```
ConRel : 𝒱 ̇ → 𝒰 ̇ → (𝒲 : Universe) → 𝒱 ⊔ 𝒰 ⊔ 𝒲 ⁺ ̇
ConRel I A 𝒲 = (I → A) → 𝒲 ̇
```

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

```
module _ {𝒰 𝒱 𝒲 : Universe} {I J : 𝒱 ̇} {A : 𝒰 ̇} where

  lift-con-rel : ConRel I A 𝒲 → (I → J → A) → 𝒱 ⊔ 𝒲 ̇
  lift-con-rel R 𝕔 = ∀ (j : J) → R λ i → (𝕔 i) j

  con-compatible-fun : (I → (J → A) → A) → ConRel I A 𝒲 → 𝒱 ⊔ 𝒰 ⊔ 𝒲 ̇
  con-compatible-fun 𝕗 R = ∀ 𝕔 → (lift-con-rel R) 𝕔 → R λ i → (𝕗 i) (𝕔 i)
```

In the definition of con-compatible-fun, we let Agda infer the type of $\mathbb{c}$, 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 : 𝒱 ̇)(A : I → 𝒰 ̇)(𝒲 : Universe) → 𝒱 ⊔ 𝒰 ⊔ 𝒲 ⁺ ̇
DepRel I A 𝒲 = Π A → 𝒲 ̇
```

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 _ {𝒰 𝒱 𝒲 : Universe} {I J : 𝒱 ̇} {A : I → 𝒰 ̇} where

  lift-dep-rel : DepRel I A 𝒲 → (∀ i → J → A i) → 𝒱 ⊔ 𝒲 ̇
```

[31] 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 $(i : I)$ , $A\ i$."

[32] For consistency and readability, throughout the UALib we treat two universe variables with special care. The first of these is $\mathcal{O}$ 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.

```
lift-dep-rel R 𝔠 = ∀ (j : J) → R (λ i → (𝔠 i) j)

dep-compatible-fun : (∀ i → (J → A i) → A i) → DepRel I A 𝒲 → 𝒱 ⊔ 𝒰 ⊔ 𝒲 ̇
dep-compatible-fun 𝔣 R = ∀ 𝔠 → (lift-dep-rel R) 𝔠 → R λ i → (𝔣 i)(𝔠 i)
```

In the definition of dep-compatible-fun, we let Agda infer the type of $\mathfrak{c}$, 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.[33]

### 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 ̇
```

[33] 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.

```
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 $[\ a\ ]\ R := \text{all } b : A \text{ such that } R\ a\ b$. 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.[34]

```
/-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}

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

  /-supset : {a b : A}{R : Rel A ℛ} → IsEquivalence R → R a b → [ a ] R ⊇ [ b ] R
  /-supset {a}{b} Req Rab {x} Rbx = (trans Req) a b x Rab Rbx

  /-≐ : {a b : A}{R : Rel A ℛ} → IsEquivalence R → R a b → [ a ] R ≐ [ b ] R
  /-≐ Req Rab = /-subset Req Rab , /-supset Req Rab
```

## 3.4 Truncation: continuous propositions, quotient extensionality

This section presents the Relations.Truncation module of the AgdaUALib, slightly abridged.[35] 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 [8], or Guillaume Brunerie, Truncations and truncated higher inductive types, or Section 7.1 of the HoTT book [14].[36]

### 3.4.1 Background and motivation[37]

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

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

[35] 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`.

[36] **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.

[37] 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.

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 [8]).

### 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 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.[38]

```
is-set : 𝒰 ̇ → 𝒰 ̇
is-set X = (x y : X) → is-subsingleton (x ≡ 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-≡, which we also import, is part of Escardó's characterization of equality in Sigma types ([8]). It is defined as follows.

```
module hide-to-Σ-≡ {𝒰 𝒲 : Universe} where

  to-Σ-≡ : {X : 𝒰 ̇} {A : X → 𝒲 ̇} {σ τ : Σ A}
    → Σ p ꞉ ∣ σ ∣ ≡ ∣ τ ∣ , (transport A p ‖ σ ‖) ≡ ‖ τ ‖
    → σ ≡ τ

  to-Σ-≡ (refl {x = x} , refl {x = a}) = refl {x = (x , a)}

open import MGS-Embeddings using (to-Σ-≡) public
```

We will use is-embedding, is-set, and to-Σ-≡ 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 _⟺_ with which to represent such equivalences.

[38] 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."

```
_⟺_ : {𝓤 𝓦 : Universe} → 𝓤 ̇ → 𝓦 ̇ → 𝓤 ⊔ 𝓦 ̇
X ⟺ Y = (X → Y) × (Y → X)

module _ {𝓤 𝓦 : Universe}{A : 𝓤 ̇}{B : 𝓦 ̇} where

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

  monic-is-embedding|sets f Bset fmon b (a , fa≡b) (a' , fa'≡b) = γ
    where
    faa' : f a ≡ f a'
    faa' = ≡-Trans (f a) (f a') fa≡b (fa'≡b ⁻¹)

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

    𝒜 : _ → _ ̇
    𝒜 a = f a ≡ b

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

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

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

  Pred₁ : 𝓤 ̇ → (𝓦 : Universe) → 𝓤 ⊔ 𝓦 ⁺ ̇
  Pred₁ A 𝓦 = Σ P ꞉ (Pred A 𝓦) , ∀ x → is-subsingleton (P x)
```

(Recall that Pred A 𝓦 is simply the function type $A \to \mathcal{W}\ \dot{}$.)

The principle of *proposition extensionality* asserts that logically equivalent propositions are equivalent. That is, if we have $P\ Q : \mathsf{Pred}_1$ and $\mid P \mid \subseteq \mid Q \mid$ and $\mid Q \mid \subseteq \mid P \mid$, then $P \equiv Q$.

This is formalized as follows (cf. the section "Prop extensionality and the powerset" of [8]).

```
prop-ext : (𝓤 𝓦 : Universe) → (𝓤 ⊔ 𝓦) ⁺ ̇
prop-ext 𝓤 𝓦 = ∀ {A : 𝓤 ̇}{P Q : Pred₁ A 𝓦 } → | P | ⊆ | Q | → | Q | ⊆ | P | → P ≡ Q
```

Recall, we defined the relation `_≐_` for predicates as follows: $P \doteq Q = (P \subseteq Q) \times (Q \subseteq P)$. Therefore, if we assume `PropExt` $A\ \mathcal{W}\ \{P\}\{Q\}$ holds, then it follows that $P \equiv Q$.

```
prop-ext' : (A : 𝓤 ̇)(𝓦 : Universe){P Q : Pred₁ 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 $P \subseteq Q \times Q \subseteq P \to P \equiv Q$, 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*.[39]

As above, we use the `is-subsingleton` type of the Type Topology library to impose this truncation assumption on a binary relation.[40]

```
Pred₂ : 𝓤 ̇ → (𝓦 : Universe) → 𝓤 ⊔ 𝓦 ⁺ ̇
Pred₂ A 𝓦 = Σ R ꞉ (Rel A 𝓦) , ∀ x y → is-subsingleton (R x y)
```

(Recall, `Rel A 𝓦` is simply the function type $A \to A \to \mathcal{W}\ \dot{}$.)

### 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 _ {𝓤 ℛ : Universe}{A : 𝓤 ̇}{R : Pred₂ A ℛ} where

  class-extensionality : prop-ext 𝓤 ℛ → {u v : A} → IsEquivalence | R |
    →                    | R | u v → [ u ] | R | ≡ [ v ] | R |

  class-extensionality pe {u}{v} Reqv Ruv = γ
    where
      P Q : Pred₁ A ℛ
      P = (λ a → | R | u a) , (λ a → ‖ R ‖ u a)
      Q = (λ a → | R | v a) , (λ a → ‖ R ‖ v a)

      α : [ u ] | R | ⊆ [ v ] | R |
```

---

[39] 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.

[40] Using the definition `is-subsingleton-valued` from § 3.3.1, we could have defined `Pred₂` by `Σ R ꞉ (Rel A 𝓦) , is-subsingleton-valued R`, but this seems less transparent than our explicit definition.

```
    α ua = fst (/-≐ Reqv Ruv) ua

    β : [ v ] | R | ⊆ [ u ] | R |
    β va = snd (/-≐ Reqv Ruv) va

    PQ : P ≡ Q
    PQ = (prop-ext' pe (α , β))

    γ : [ u ] | R | ≡ [ v ] | R |
    γ = ap fst PQ


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

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


class-extensionality' : 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 $I$ $A$ $\mathcal{W}$ holds for some $I$, $A$ and $\mathcal{W}$, then we can prove that logically equivalent continuous propositions of type ConProp $I$ $A$ $\mathcal{W}$ are equivalent.

```
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 (§2.1.4) 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.[41]

### 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 $I : \mathcal{V}$, which gives us a very general way to represent an operation as a function type with domain $I \to A$ (the type of "tuples") and codomain $A$. The last two lines of the code block above codify the $i$-th $I$-ary projection operation on $A$.

[41] 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 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, $\mathcal{O}$ will always denote the universe of *operation symbol* types, while $\mathcal{V}$ is the universe of *arity* types.

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

#### 4.1.2.1 Example

Here is how we might define the signature for monoids as a member of the type Signature $\mathcal{O}\ \mathcal{V}$.

```
data monoid-op : 𝓞₀ ̇ where
  e : monoid-op
  · : monoid-op

monoid-sig : Signature 𝓞₀ 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 $\lambda\ \{\ \mathsf{e} \to \mathbb{0}\ ;\ \cdot \to \mathbb{2}\ \}$ which maps `e` to the empty type $\mathbb{0}$ (since `e` is the nullary identity) and maps `·` to the two element type $\mathbb{2}$ (since `·` is binary).[42]

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

This section presents the Algebras.Algebras module of the AgdaUALib, slightly abridged.[43]

### 4.2.1 The Algebra type

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

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

Algebra 𝓤 S = Σ A ꞉ 𝓤 ̇ , ((f ꞉ ∣ S ∣) → Op (∥ S ∥ f) A)
```

We could refer to an inhabitant of this type as a "$\infty$-algebra" because its domain can be an arbitrary type, say, $A : \mathcal{U}$ 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

[42] The types $\mathbb{0}$ and $\mathbb{2}$ are defined in the MGS-MLTT module of the Type Topology library.

[43] 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`.

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 ($\infty$-)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.

```
_̂_ : (f : ∣ S ∣)(𝐀 : Algebra 𝓤 S) → (∥ S ∥ f → ∣ 𝐀 ∣) → ∣ 𝐀 ∣

f ̂ 𝐀 = λ a → (∥ 𝐀 ∥ f) a
```

So, if $f : \mid S \mid$ is an operation symbol in the signature $S$, and if $a : \parallel S \parallel f \to \mid \mathbf{A} \mid$ is a tuple of the appropriate arity, then $(f \mathbin{\hat{}} \mathbf{A})\ a$ denotes the operation $f$ interpreted in $\mathbf{A}$ 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 $\mathbf{A}$, no matter the type of its domain, we have a surjective map $h : X \to \mid \mathbf{A} \mid$ from variables onto the domain of $\mathbf{A}$. We may use the following definition to express this assumption when we need it.

```
_↠_ : {S : Signature 𝓞 𝓥}{𝓤 𝓧 : Universe} → 𝓧 ̇ → Algebra 𝓤 S → 𝓧 ⊔ 𝓤 ̇
X ↠ 𝐀 = Σ h ꞉ (X → ∣ 𝐀 ∣) , 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.

```
module _ {𝕏 : {𝓤 𝓧 : Universe}{X : 𝓧 ̇}(𝐀 : Algebra 𝓤 S) → X ↠ 𝐀} where
```

Then fst(𝕏 𝐀) will denote the surjective map $h : X \to \mid \mathbf{A} \mid$, and snd(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.

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

  lift-op : {𝓤 : Universe}{I : 𝓥 ̇}{A : 𝓤 ̇} → ((I → A) → A) → (𝓦 : Universe)
    → ((I → Lift{𝓦} A) → Lift {𝓦} A)

  lift-op f 𝓦 = λ x → lift (f (λ i → Lift.lower (x i)))

  open algebra

  lift-alg : {𝓤 : Universe} → Algebra 𝓤 S → (𝓦 : Universe) → Algebra (𝓤 ⊔ 𝓦) S
  lift-alg 𝐀 𝓦 = Lift ∣ 𝐀 ∣ , (λ (f : ∣ S ∣) → lift-op (f ̂ 𝐀) 𝓦)
```

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

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

### 4.2.5 Compatibility of binary relations

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

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

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 ̂ 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 ̂ 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 ̂ 𝐀) (𝔞 i))
```

where we have let Agda infer the type of $\mathfrak{a}$, which is $(i : I) \to \| S \| f \to \mid \mathbf{A} \mid$.

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.[44] We begin this module by assuming a signature $S$ : Signature $\mathcal{O}$ $\mathcal{V}$ which is then present and available throughout the module. Because of this, in contrast to our (highly abridged) descriptions

[44] 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`.

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.

```
⨅ : {𝓤 𝓘 : Universe}{I : 𝓘 ̇ }(𝒜 : I → Algebra 𝓤 S ) → Algebra (𝓘 ⊔ 𝓤) S

⨅ 𝒜 = (∀ i → ∣ 𝒜 i ∣) ,                    -- domain of the product algebra

      λ f a i → (f ̂ 𝒜 i) λ x → a x i       -- 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 Algebra $\mathcal{U}$ $S$, for some universe $\mathcal{U}$ and signature $S$. That is, $\mathscr{K}$ : Pred (Algebra $\mathcal{U}$ $S$) _.[45] Later we will formally state and prove that the product of all subalgebras of algebras in such a class belongs to SP($\mathscr{K}$) (subalgebras of products of algebras in $\mathscr{K}$). That is, ⨅ S($\mathscr{K}$) ∈ 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.[46] 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.[47]

```
module _ {𝓤 𝓧 : Universe}{X : 𝓧 ̇} where

  ℑ : Pred (Algebra 𝓤 S)(ov 𝓤) → (𝓧 ⊔ ov 𝓤) ̇

  ℑ 𝒦 = Σ 𝐀 ꞉ (Algebra 𝓤 S) , (𝐀 ∈ 𝒦) × (X → ∣ 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

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

[46] 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 Algebra $\mathcal{U}$ $S$, or even an arbitrary predicate.

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

do want to lay our hands on whatever context is ultimately assumed. Including the "context map" inside the index type $\mathcal{I}$ of the product turns out to be a convenient way to achieve this flexibility.

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

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

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

```
𝔄 : (𝒦 : Pred (Algebra 𝒰 S)(ov 𝒰)) → ℐ 𝒦 → Algebra 𝒰 S

𝔄 𝒦 = λ (i : (ℐ 𝒦)) → ∣ i ∣
```

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

```
class-product : Pred (Algebra 𝒰 S)(ov 𝒰) → Algebra (𝓧 ⊔ ov 𝒰) S

class-product 𝒦 = ⨅ ( 𝔄 𝒦 )
```

If $p : \mathbf{A} \in \mathcal{K}$ and $h : X \to |\ \mathbf{A}\ |$, then we can think of the triple $(\mathbf{A} , p , h) \in \mathcal{I}\ \mathcal{K}$ 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 $\bigsqcap\ (\ \mathfrak{A}\ \mathcal{K}\ )$ 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.[48] 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 : {𝒰 : Universe}(A : Algebra 𝒰 S) → ov 𝒰 ̇
Con {𝒰} A = Σ θ ꞉ ( Rel ∣ A ∣ 𝒰 ) , IsEquivalence θ × compatible A θ

record Congruence {𝒰 𝒲 : Universe} (A : Algebra 𝒰 S) : ov 𝒲 ⊔ 𝒰 ̇ where
  constructor mkcon
  field
    ⟨_⟩ : Rel ∣ A ∣ 𝒲
    Compatible : compatible 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.

---

[48] 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`.

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 _ {𝓤 : Universe} where

  𝟎-IsEquivalence : {A : 𝓤 ̇ } → IsEquivalence{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 _ {𝓤 : Universe} where

  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} 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 $\mathbf{A}$ with respect to a congruence relation $\theta$ of $\mathbf{A}$. This quotient is typically denote by $\mathbf{A} \,/\, \theta$ and Agda allows us to define and express quotients using the standard notation.[49]

```
_╱_ : {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) _
```

---

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

## 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.

## 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, 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.

7 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.

8 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.

9 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.

10 Per Martin-Löf. An intuitionistic theory of types: predicative part. In *Logic Colloquium '73 (Bristol, 1973)*, pages 73–118. Studies in Logic and the Foundations of Mathematics, Vol. 80. North-Holland, Amsterdam, 1975.

11 Per Martin-Löf. *Intuitionistic type theory*, volume 1 of *Studies in Proof Theory. Lecture Notes*. Bibliopolis, Naples, 1984. Notes by Giovanni Sambin.

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

13 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.

14 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.

15 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.

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

17 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.

18 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.

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