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 Cliff Bergman, Hyeyoung Shin, and Siva Somayyajula for supporting and contributing to this project, Adreas Abel for helpful corrections, and Martín Escardó for creating the Type Topology library and teaching us about it at the 2019 Midlands Graduate School in Computing Science [9]. Of course, the present work would not exist in its current form without the Agda 2 language by Ulf Norell.[1]

[1] Agda 2 is partially based on code from Agda 1 by Catarina Coquand and Makoto Takeyama, and from Agdalight by Ulf Norell and Andreas Abel.


This work and the Agda Universal Algebra Library by William DeMeo is licensed under a Creative Commons Attribution-ShareAlike 4.0 International License.


© 2021 William DeMeo. Based on work at https://gitlab.com/ualib/ualib.gitlab.io.
Compiled with xelatex on 26 Mar 2021 at 13:32.

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 [16], 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 [4] (1999) formalized the basics of universal algebra in the Calculus of Inductive Constructions using the Coq proof assistant;
- Spitters and van der Weegen [18] (2011) formalized the basics of universal algebra and some classical algebraic structures, also in the Calculus of Inductive Constructions using the Coq proof assistant, promoting the use of type classes as a preferable alternative to setoids;
- Gunther, et al [10] (2018) developed what seems to be (prior to the UALib) the most extensive library of formal universal algebra to date; in particular, this work includes a formalization of some basic equational logic; also (unlike the UALib) it handles *multisorted* algebraic structures; (like the UALib) it is based on dependent type theory and the Agda proof assistant.

Some other projects aimed at formalizing mathematics generally, and algebra in particular, have developed into very extensive libraries that include definitions, theorems, and proofs about algebraic structures, such as groups, rings, modules, etc. However, the goals of these efforts seem to be the formalization of special classical algebraic structures, as opposed to the general theory of (universal) algebras. Moreover, the part of universal algebra and equational logic formalized in the UALib extends beyond the scope of prior efforts. 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.[3]

1.3 Attributions and Contributions

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

Unless explicitly stated otherwise, the Agda source code described in this paper is due to the author, with the following caveat: the UALib depends on the Type Topology library of Martín Escardó [9]. Each dependency is carefully accounted for and mentioned in this paper. For the sake of completeness and clarity, and to keep the paper mostly self-contained, we repeat some definitions from the Type Topology library, but in each instance we cite the original source.[4]

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 [7],

[3] After completing the formal proof in Agda, we learned about a constructive version of Birkhoff's theorem proved by Carlström in [5]. 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.

[4] 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 [8], which will cover free algebras, equational classes of algebras (i.e., varieties), and Birkhoff's HSP theorem.

1.4 Organization of the paper

This present paper is organized into three parts as follows. The first part is §2 which introduces the basic concepts of type theory with special emphasis on the way such concepts are formalized in Agda. Specifically, §2.1 introduces *Sigma types* and Agda's hierarchy of *universes*. The important topics of *equality* and *function extensionality* are discussed in §2.2 and §2.3; §2.4 covers inverses and inverse images of functions. In §2.5 we describe a technical problem that one frequently encounters when working in a *noncumulative 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 [17] and Escardó's Introduction to Univalent Foundations of Mathematics with Agda [9].

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

- *Introduction to Univalent Foundations of Mathematics with Agda*, Escardó [9].
- *Dependent Types at Work*, Bove and Dybjer [2].
- *Dependently Typed Programming in Agda*, Norell and Chapman [15].
- *Formalization of Universal Algebra in Agda*, Gunther, Gadea, Pagano [10].
- *Programming Languages Foundations in Agda*, Philip Wadler [22].

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.[5]
- *The Agda UALib, Part 2: homomorphisms, terms, and subalgebras* [7].
- *The Agda UALib, Part 3: free algebras, equational classes, and Birkhoff's theorem* [8].

[5] 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 Overture

2.1 Preliminaries: logical foundations, universes, dependent types

This section presents the Overture.Preliminaries module of the AgdaUALib, slightly abridged.[6] Here we define or import the basic types of *Martin-Löf type theory* (MLTT). Although this is standard stuff, 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 AgdaUALib is based on a type theory that is the same or very close to the one on which on which Martín Escardó's Type Topology Agda library is based. We don't discuss MLTT in great detail here because there are already nice and freely available resources covering the theory. (See, for example, the section A spartan Martin-Löf type theory of the lecture notes by Escardó [9], the ncatlab entry on Martin-Löf dependent type theory, or the HoTT Book [17].)

The objects and assumptions that form the foundation of MLTT are few. There are the *primitive types* ($\mathbb{0}$, $\mathbb{1}$, and $\mathbb{N}$, denoting the empty type, one-element type, and natural numbers), the *type formers* ($+$, Π, Σ, Id, denoting *binary sum*, *product*, *sum*, and the *identity* type). Each of these type formers is defined by a *type forming rule* which specifies how that type is constructed. Lastly, we have an infinite collection of *type universes* (types of types) and *universe variables* to denote them. Following Escardó, we denote universes in the UALib by upper-case calligraphic letters from the second half of the English alphabet; to be precise, these are $\mathcal{U}$, $\mathcal{V}$, $\mathcal{R}$, ..., $\mathcal{X}$, $\mathcal{Y}$, $\mathcal{Z}$.[7]

That's all. There are no further axioms or logical deduction (proof derivation) rules needed for the foundation of MLTT that we take as the starting point of the AgdaUALib. The logical semantics come from the propositions-as-types correspondence [14]: propositions and predicates are represented by types and the inhabitants of these types are the proofs of the propositions and predicates. As such, proofs are constructed using the type forming rules. In other words, the type forming rules *are* the proof derivation rules.

To this foundation, we add certain *extensionality principles* when and were we need them. These will be developed as we progress. However, classical axioms such as the *Axiom of Choice* or the *Law of the Excluded Middle* are not needed and are not assumed anywhere in the library. In that sense, all theorems and proofs in the UALib are *constructive* (as defined, e.g., in [12]).

A few specific instances (e.g., the proof of the Noether isomorphism theorems and Birkhoff's HSP theorem) require certain *truncation* assumptions. In such cases, the theory is not *predicative* (as defined, e.g., in [13]). These instances are always clearly identified.

---

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

[7] We avoid using $\mathcal{P}$ as a universe variable because in the Type Topology library $\mathcal{P}$ denotes a powerset type.

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 [19];
- *–exact-split* makes Agda accept only definitions that are *judgmental* equalities; see [21];
- *–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 [20] and [21].

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

2.1.2 Agda Modules

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

```
module Overture.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[8]

```
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 Universes in Agda

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.[9]

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 Founda-*

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

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

tions of mathematics.[10] 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 (Overture.Preliminaries). This module declares symbols used to denote universes. As mentioned, we adopt Escardó's convention of denoting universes by capital calligraphic letters, and most of the ones we use are already declared in Universes; those that are not are declared as follows.

```
variable 𝓸 𝓧 𝓨 𝓩 : Universe
```

The Universes module also provides alternative syntax for the 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 alias for Set lzero. Thus, $\mathcal{U}\ \cdot$ 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 ${\mathcal{U}_0}^+\ \cdot$.

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 $\mathcal{U}\ \cdot$ is written Set $\mathcal{U}$ in standard Agda), but in univalent mathematics the types in $\mathcal{U}\ \cdot$ 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 a universe at the level that is the least upper bound of two universe levels, say, $\mathcal{U}$ and $\mathcal{V}$. The universe level $\mathcal{U} \sqcup \mathcal{V}$ denotes this least upper bound. Here $\sqcup$ 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 $A : \mathcal{U}\ \cdot$, and a type family $B : A \to \mathcal{V}\ \cdot$, 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.[11]

```
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 $\Sigma\ x : A\ ,\ B$,

[10] 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. [9].

[11] 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.

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 [9, Σ types]).[12]

```
-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
```

Pi types (dependent functions)

Given universes $\mathcal{U}$ and $\mathcal{V}$, a type $A : \mathcal{U}\,\dot{}$, and a type family $B : A \to \mathcal{V}\,\dot{}$, 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 Π conform to the standard notation for Pi types, Escardó uses the same trick as the one used above for Sigma types.[12]

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

Projection notation

The definition of Σ (and thus ×) includes the fields pr_1 and 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: pr_1 and fst for the first projection, pr_2 and snd for the second. We define these alternative notations for projections as follows.

```
module _ {𝓤 : Universe}{A : 𝓤 ̇ }{B : A → 𝓥 ̇ } where
```

[12] **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.

```
|_| fst : Σ B → A
| x , y | = x
fst (x , y) = x

‖_‖ snd : (z : Σ B) → B (pr₁ z)
‖ x , y ‖ = y
snd (x , y) = y
```

Remarks.

- We place these definitions (of |_|, fst, ‖_‖ and snd) inside an *anonymous module*, which is a module that begins with the module keyword followed by an underscore character (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. In library documentation, such as the present paper, we often omit such module directives. In contrast, the collection of html pages at `ualib.org`, which is the most current and comprehensive documentation of the UALib, omits nothing.
- As the four definitions above make clear, multiple inhabitants of a single type (e.g., |_| and fst) may be declared on the same line.

2.2 Equality: definitional equality and transport

This section presents the Overture.Equality module of the AgdaUALib, slightly abridged.[13]

2.2.1 Definitional equality

Here we discuss what is probably the most important type in MLTT. It is called *definitional equality*. This concept is most understood, at least heuristically, with the following slogan: "Definitional equality is the substitution-preserving equivalence relation generated by definitions." We will make this precise below, but first let us quote from a primary source. Per Martin-Löf offers the following definition in [11, §1.11] (italics added):[14]

> *Definitional equality* is defined to be the equivalence relation, that is, reflexive, symmetric and transitive relation, which is generated by the principles that a definiendum is always definitionally equal to its definiens and that definitional equality is preserved under substitution.

To be sure we understand what this means, let := denote the relation with respect to which x is related to y (denoted $x := y$) if and only if y *is the definition of* x. Then the definitional equality relation $\equiv$ is the reflexive, symmetric, transitive, substitutive closure of :=. By *subsitutive closure* we mean closure under the following *substitution rule*.

$$\frac{\{A : \mathcal{U}\ \cdot\}\ \{B : A \to \mathcal{W}\ \cdot\}\ \{x\ y : A\} \quad x \equiv y}{B\ x \equiv B\ y}\ (subst)$$

---

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

[14] The *definiendum* is the left-hand side of a defining equation, the *definiens* is the right-hand side. For readers who have never generated an equivalence relation: the *reflexive closure* of $R \subseteq A \times A$ is the union of R and all pairs of the form $(a\ ,\ a)$; the *symmetric closure* is the union of R and its inverse $\{(y\ ,\ x) : (x\ ,\ y) \in R\}$; we leave it to the reader to come up with the correct definition of transitive closure.

The datatype used in the UALib to represent definitional equality is imported from the Identity-Type module of the Type Topology library, but apart from superficial syntactic differences, it is equivalent to the identity type used in all other Agda libraries we know of. We repeat the definition here for easy reference.

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

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 explicit the implicit argument x, then we use refl $\{x = x\}$.

Assumed module contexts

Before proceeding, a word about a special convention we adopt in the sequel is in order. To reduce reader strain, we will often omit easily inferred typing judgments which would normally appear in the list of parameters of a module or at the start of a type definition, and we sometimes make an announcement like the following (which applies to the present section):

Unless otherwise indicated, the prevailing context in this section is given by

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

Definitional equality is an equivalence

The relation ≡ just defined is naturally an equivalence relation, and the formal proof of this fact is trivial. Indeed, we don't need to prove reflexivity, since that is the defining property of ≡, and the proofs of symmetry and transitivity are also immediate.

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

≡-sym : {x y : A} → x ≡ y → y ≡ x
≡-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.

We prove that ≡ obeys the substitution rule (subst) in the next section (see ap §2.2.2), but first we define some syntactic sugar that will make it easier to apply symmetry and transitivity of ≡ in proofs.[15]

```
_⁻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

[15] **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`).

to transitivity easier to stomach.

```
_·_ : {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 [6]). 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.[16]

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

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

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. It is equivalent to the substitution rule (subst) we defined in the last section. If we have a function $f : A \to B$, two elements $x\ y : A$ of the domain, and an identity proof $p : x \equiv y$, then we obtain a proof of $f\ x \equiv f\ y$ by simply applying the ap function like so, ap $f\ p : f\ x \equiv f\ y$. Escardó defines ap in the 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})
```

This establishes that our definitional equality satisfies the substitution rule (subst).

Here's a useful variation of ap that we borrow from the `Relation/Binary/Core.agda` module of the Agda Standard Library (transcribed into TypeTopology/UALib 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 Overture.Extensionality module of the AgdaUALib, slightly abridged.[17]

2.3.1 Background and motivation

This brief introduction to *function extensionality* is intended for novices. Those already familiar with the concept might wish to skip to the next subsection.

---

[16] Including every line of code of the AgdaUALib in this paper would result in an unbearable reading experience. We include all significant sections of code from the first 13 modules, but we omit lines indicating that redundant definitions of functions (e.g., transport and ap) occur inside named "hidden" modules. We also omit lines importing the original definitions of such duplicate definitions from the Type Topology library.

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

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 $gx := ((2 * x) - 8)/2 + 6$. Should we call f and g equal? Are they the "same" function? What does that even mean?

It's hard to resist the urge to reduce g to $x+2$ and proclaim that f and g are equal. Indeed, this is often an acceptable answer and the discussion normally ends 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 argument and produce as output a correctly sorted version of the input list? Suppose f is defined using the merge sort algorithm, while g uses quick sort. Probably few of us would call f and g the "same" in this case.

In the examples above, it is common to say that the two functions 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 the next subsection we describe types that manifest this idea of *extensional equality of functions*, or *function extensionality*.[18]

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.

```
_∼_ : {A : 𝓤 ̇ } {B : A → 𝓦 ̇ } → Π B → Π B → 𝓤 ⊔ 𝓦 ̇
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 *point-wise equality of functions* is typically what one means when one asserts that two functions are "equal."[19] However, it is important to keep in mind the following fact: *function extensionality is known to be neither provable nor disprovable in Martin-Löf type theory. It is an independent statement.* [9]

---

[18] Most of these types are already defined in the Type Topology library, so the UALib imports the definitions from there; as usual, we redefine some of these types here for the purpose of explication.

[19] In fact, if one assumes the *univalence axiom* of Homotopy Type Theory [17], then point-wise equality of functions is equivalent to definitional equality of functions. See the section "Function extensionality from univalence" of [9].

2.3.3 Global function extensionality

An assumption that we adopt throughout much of the current version of the UALib is a *global function extensionality principle*. This asserts that function extensionality holds at all universe levels. Agda is capable of expressing types representing global principles as the language has a special universe level for such types. Following Escardó, we denote this universe by $\mathcal{U}\omega$ (which is just an alias for Agda's Setω universe).[20] 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.

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

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 postulate an inhabitant of this type. That is, we could postulate that function extensionality holds by assuming we have a witness, say, *fe* : funext $\mathcal{U}$ $\mathcal{V}$ (i.e., a proof that pointwise equal functions are equal), but as noted above the existence of such a witness cannot be proved in Martin-Löf type theory.

2.3.4 An 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* in a sense that we now describe. This requires a few definitions from the MGS-Equivalences module of the Type Topology library. First, a type is a *singleton* if it has exactly one inhabitant and a *subsingleton* if it has at most one inhabitant.

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

is-singleton : 𝓤 ̇ → 𝓤 ̇
is-singleton A = Σ c ꞉ A , is-center A c
```

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

```
is-subsingleton : 𝓤 ̇ → 𝓤 ̇
is-subsingleton A = (x y : A) → 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. This requires the concept of a *fiber* of a function, which can be represented as a Sigma type whose inhabitants denote inverse images of points in the codomain of the given function.

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

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

```
is-equiv : {A : 𝓤 ̇ } {B : 𝓦 ̇ } → (A → B) → 𝓤 ⊔ 𝓦 ̇
is-equiv f = ∀ y → is-singleton (fiber f y)
```

Finally we are ready to fulfill the promise of a type that provides an alternative means of postulating function extensionality.

```
hfunext : ∀ 𝓤 𝓦 → (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 Overture.Inverses module of the AgdaUALib, slightly abridged.[21] We begin by defining an inductive type that represents the *inverse image* of a function.

```
data Image_∋_ (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 : (f : A → B)(b : B)(a : A) → b ≡ f a → Image f ∋ b
ImageIsImage f b a b≡fa = eq b a b≡fa
```

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 : (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 : (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
```

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

2.4.1 Epics (surjective functions)

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

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

We obtain the right-inverse 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 : (f : A → B) → Epic f → B → A
EpicInv f fE b = Inv f (fE 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, as follows.

```
_∘_ : {C : B → 𝓦 ̇ } → Π C → (f : A → B) → (x : A) → C (f x)
g ∘ f = λ x → g (f x)
```

We can now express the assertion that EpicInv f does, indeed, give the right-inverse of f. Note that the proof requires function extensionality.

```
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))
```

Here we break with our convention of hiding anonymous module declarations in order to emphasize that function extensionality is required. Also, this gives us a chance to demonstrating how one postulates function extensionality in a module declaration. We will see many more such examples later.

2.4.2 Monics (injective functions)

We say that a function $g : A \to B$ is *monic* (or *injective*) if it does not map distinct elements to a common point. The Monic type, which we now define, manifests this property.

```
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 imfb → Inv f imfb
```

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 is-embedding type is defined in the Type Topology library in the following way.

```
is-embedding : (A → B) → 𝓤 ⊔ 𝓦 ̇
is-embedding f = ∀ b → is-subsingleton (fiber f b)
```

Thus, is-embedding f asserts that f is a function all of whose fibers are subsingletons. 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 : (f : A → B) → 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 : (f : A → B) → is-embedding f → Monic f
embedding-is-monic f femb x y fxfy = ap pr₁ ((femb (f x)) fx fy)
  where
  fx : fiber f (f x)
  fx = x , refl

  fy : fiber f (f x)
  fy = y , (fxfy ⁻¹)
```

2.5 Lifts: making peace with a noncumulative universe hierarchy

This section presents the Overture.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.[23] 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 *noncumulative*. This makes it possible to treat universe levels more generally and precisely, which is nice. On the other hand, a noncumulative hierarchy can sometimes make for a nonfun proof assistant. Specifically, in certain situations, the noncumulativity can make it unduly difficult to convince Agda that a program or proof is correct.

2.5.2 Lifting and lowering

Here we describe general lifting and lowering functions that help us overcome the technical issue described in the previous subsection. In Section 4.2.5 we will define a couple of domain-specific

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

[23] Recall, from the Overture.Preliminaries module (§2.1.3), the special notation we use to denote Agda's *levels* and *universes*.

analogs of these tools. Later, in the modules presented in [7, 8], we prove various properties that make these effective mechanisms for resolving universe level problems when working with algebra types.

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.

The general Lift record type that we now describe makes these situations easier to deal with. It takes a type inhabiting some universe and embeds it into a higher universe and, apart from syntax and notation, it is equivalent to the Lift type one finds in the Level module of the Agda Standard Library.

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

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

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

lower∼lift : {𝓦 𝓤 : Universe}{A : 𝓤 ̇} → lower{𝓦}{𝓤} ∘ lift ≡ 𝑖𝑑 A
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 relations: predicates, axiom of extensionality, compatibility

This section presents the Relations.Discrete module of the AgdaUALib, slightly abridged.[24] Here we present the submodules of the AgdaUALib's Relations module. In §3.1 we define types that represent *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

[24] 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.

typical approach is to use a *predicate* type, denoted by Pred.

Given two universes $\mathcal{U}$ $\mathcal{W}$ and a type $A : \mathcal{U}\ ^{\cdot}$, 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.[25]

```
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 $\mathcal{U}\ ^{\cdot}$, 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.[25]

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

The *subset* relation is denoted, as usual, with the $\subseteq$ symbol and is defined as follows.[25]

```
_⊆_ : Pred A 𝒲 → Pred A 𝒳 → 𝒰 ⊔ 𝒲 ⊔ 𝒳 ̇
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 $\dot{=}$ and define it as follows.[26]

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

A proof of $P \dot{=} Q$ 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 $P \dot{=} Q$ also holds, as we now confirm.

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

The converse is not provable in MLTT. However, we can postulate that it holds as an axiom if we wish. This is called the *axiom of extensionality* and a type that represents it is the following.

---

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

[26] **Unicode Hints**. In agda2-mode, `\.=` ⇝ $\dot{=}$, `\u+` ⇝ $\uplus$, `\b0` ⇝ $\mathbb{0}$, `\B0` ⇝ $\mathbf{0}$, `\b1` ⇝ $\mathbb{1}$, `\B1` ⇝ $\mathbf{1}$.

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

Note that ext-axiom does not itself postulate the axiom of extensionality; it merely says what it is. If we want to postulate the axiom, we must assume we have an inhabitant (or "witness") 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.[27]

```
module ext-axiom-postulated {𝓤 𝓦 : Universe}{A : 𝓤 ̇} {ea : ext-axiom A 𝓦} where
```

Other notions of extensionality come up often in the UALib; see, for example, §2.3 and §3.4.

Predicates toolbox

Here is a small collection of tools that will come in handy later. The first is an inductive type that represents *disjoint union*.[26]

```
data _⊎_ (A : 𝓤 ̇) (B : 𝓦 ̇) : 𝓤 ⊔ 𝓦 ̇ where
  inj₁ : (x : A) → A ⊎ B
  inj₂ : (y : B) → A ⊎ B
```

And this can be used to define a type representing *union*, as follows.

```
_∪_ : Pred A 𝓦 → Pred A 𝓧 → Pred A (𝓦 ⊔ 𝓧)
P ∪ Q = λ x → x ∈ P ⊎ x ∈ Q
```

Next we define convenient notation for asserting that the image of a function (the first argument) is contained in a predicate (the second argument).

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

The *empty set* is naturally represented by the *empty type*, 𝟘, and the latter is defined in Type Topology's Empty-Type module.[26,28]

```
∅ : Pred A 𝓤₀
∅ _ = 𝟘
```

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

```
⁅_⁆ : 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 $A \to A \to \mathcal{W}\ \dot{}$ (for some universe $\mathcal{W}$). Note, however,

[27] 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.

[28] 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)*.

this is not the same as a unary predicate over the function type $A \to A$ since the latter has type $(A \to A) \to \mathcal{W}\ ^{\cdot}$, while a binary relation should have type $A \to (A \to \mathcal{W}\ ^{\cdot})$.

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 ̇ → 𝓦 ̇ → (𝓧 : Universe) → 𝓤 ⊔ 𝓦 ⊔ 𝓧 ⁺ ̇
REL A B 𝓧 = A → B → 𝓧 ̇

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

The kernel of a function

The *kernel* of a function $f : A \to B$ 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 alternatives are defined in the UALib as follows.

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

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

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

ker-sigma' : (A → B) → 𝓤 ⊔ 𝓦 ̇
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.[26]

```
𝟎 : Rel A 𝓤
𝟎 x y = x ≡ y

𝟎-pred : Pred (A × A) 𝓤
𝟎-pred (x , y) = x ≡ y

𝟎-sigma : 𝓤 ̇
𝟎-sigma = Σ x ꞉ A , Σ y ꞉ A , x ≡ y

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.[26,29]

```
𝟏 : Rel A 𝓤₀
𝟏 a b = 𝟙
```

[29] 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.

The implication relation[30]

The following types represent *implication* for binary relations.

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

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

These combine to give a nice, general implication operation.

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

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.

First, let us review the informal definition of compatibility. Suppose A and I are types and fix $f : (I \to A) \to A$ and $R : \mathsf{Rel}\ A\ \mathcal{W}$ (an I-ary operation and a binary relation on A, respectively). We say that f and R are *compatible* and we write[31] $f\ |:\ R$ just in case $\forall\ u\ v : I \to A$,

$$\Pi\ i : I\ ,\ R\ (u\ i)\ (v\ i) \quad \to \quad R\ (f\ u)\ (f\ v).$$

Here is how we implement this in the UALib.

```
eval-rel : Rel A 𝓦 → Rel (I → A)(𝓥 ⊔ 𝓦)
eval-rel R u v = Π i ꞉ I , R (u i) (v i)

_|:_ : ((I → A) → A) → Rel A 𝓦 → 𝓤 ⊔ 𝓥 ⊔ 𝓦 ̇
f |: R = (eval-rel R) =[ f ]⇒ R
```

The function eval-rel "lifts" a binary relation to the corresponding I-ary relation.[32]

3.2 Continuous relations: arbitrary-sorted relations of arbitrary arity*[33]

This section presents the Relations.Continuous module of the AgdaUALib.

Unless otherwise indicated, the prevailing context in this section is given by

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

---

[30] The definitions here are from the Agda Standard Library, translated into Type Topology/UALib notation.

[31] The symbol |: denoting compatibility comes from Cliff Bergman's universal algebra textbook [1].

[32] Initially we called the first function lift-rel because it "lifts" a binary relation on A to a binary relation on tuples of type $I \to A$. However, we renamed it eval-rel to avoid confusion with the universe level Lift type defined in the Overture.Lifts module, or with free-lift (Terms.Basic) which lifts a map defined on generators to a map on the thing being generated. Also, observe that we silently added the type $I : \mathcal{V}\ ̇$, representing *relation arity*, to the context; we will have more to say about relation arities in the next section (§3.2).

[33] Sections marked with an asterisk include new types that are more abstract and general (and frankly more interesting) than the ones defined in other sections. Moreover, the types defined in starred sections are used in only a few other places in the AgdaUALib, so they may be safely skimmed or even skipped.

3.2.1 Motivation

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 $A \to A \to \cdots \to A \to \mathcal{W}\ ^{\cdot}$ (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. It's easier and more general to instead define an arity type $I : \mathcal{V}\ ^{\cdot}$, and define the type representing I-ary relations on A as the function type $(I \to A) \to \mathcal{W}\ ^{\cdot}$. 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 ContRel to be the type $(I \to A) \to \mathcal{W}\ ^{\cdot}$ and we will call this the type of *continuous relations*. This generalizes the discrete relations we defined in Relations.Discrete (unary, binary, etc.) since continuous relations can be of arbitrary arity. They are not completely general, however, since they are defined over a single type. Said another way, these are "single-sorted" relations. We will remove this limitation 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 ContRel I A $\mathcal{W}$ be the single-sorted version of dependent continuous relations. The latter will represent relations that not only have arbitrary arities, but also are defined over arbitrary families of types.

To be more concrete, given an arbitrary family $A : I \to \mathcal{U}\ ^{\cdot}$ of types, we may have a relation from $A\ i$ to $A\ j$ to $A\ k$ to ..., where the collection represented by the indexing type I might not even be enumerable.[34] 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.

3.2.2 Continuous relation types

We now define the type ContRel which represents predicates of arbitrary arity over a single type A. We call this the type of *continuous relations*.[35]

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

Next we present types that are useful for asserting and proving facts about *compatibility* of functions with continuous relations. The first is an *evaluation* function which "lifts" an I-ary relation to an $(I \to J)$-ary relation. The lifted relation will relate an I-tuple of J-tuples when the "I-slices" (or "rows") of the J-tuples belong to the original relation.

```
eval-cont-rel : ContRel I A 𝒲 → (I → J → A) → 𝒱 ⊔ 𝒲 ̇
eval-cont-rel R 𝒂 = Π j ꞉ J , R λ i → 𝒂 i j

cont-compatible-fun : ((J → A) → A) → ContRel I A 𝒲 → 𝒱 ⊔ 𝒰 ⊔ 𝒲 ̇
cont-compatible-fun f R = Π 𝒂 ꞉ (I → J → A) , (eval-cont-rel R 𝒂 → R λ i → (f (𝒂 i)))
```

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

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

3.2.3 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 $A\ i$ to $A\ j$ to $A\ k$ to ....

```
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 _ {I J : 𝓥 ̇} {𝒜 : I → 𝓤 ̇} where

  eval-dep-rel : DepRel I 𝒜 𝓦 → (∀ i → J → 𝒜 i) → 𝓥 ⊔ 𝓦 ̇
  eval-dep-rel R 𝒂 = ∀ (j : J) → R (λ i → (𝒂 i) j)

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

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

3.3 Quotients: equivalences, class representatives, quotient types

This section presents the Relations.Quotients module of the AgdaUALib, slightly abridged.[36]

3.3.1 Properties of binary relations

In the Relations.Discrete module 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.

```
reflexive : Rel A 𝓦 → 𝓤 ⊔ 𝓦 ̇
reflexive _≈_ = ∀ x → x ≈ x

symmetric : Rel A 𝓦 → 𝓤 ⊔ 𝓦 ̇
symmetric _≈_ = ∀ x y → x ≈ y → y ≈ x

antisymmetric : Rel A 𝓦 → 𝓤 ⊔ 𝓦 ̇
antisymmetric _≈_ = ∀ x y → x ≈ y → y ≈ x → x ≡ y

transitive : Rel A 𝓦 → 𝓤 ⊔ 𝓦 ̇
transitive _≈_ = ∀ x y z → x ≈ y → y ≈ z → x ≈ z
```

The Type Topology library defines a *uniqueness-of-proofs* principle for binary relations.

[36] 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-subsingleton-valued : Rel A 𝒲 → 𝒰 ⊔ 𝒲 ·
is-subsingleton-valued _≈_ = ∀ x y → is-subsingleton (x ≈ y)
```

Thus, if R : Rel A $\mathcal{W}$, then is-subsingleton-valued R asserts that for each pair x y : A there is *at most one proof* of R x y. In the section on *Truncation* below (§ 3.4) we introduce a number of similar but more general types to represent uniqueness-of-proofs principles for relations of arbitrary arity, over arbitrary types.

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.

```
is-preorder : Rel A 𝒲 → 𝒰 ⊔ 𝒲 ·
is-preorder _≈_ = is-subsingleton-valued _≈_ × reflexive _≈_ × transitive _≈_

record IsEquivalence (_≈_ : Rel A 𝒲) : 𝒰 ⊔ 𝒲 · where
  field
    rfl   : reflexive _≈_
    sym   : symmetric _≈_
    trans : transitive _≈_

is-equivalence : Rel A 𝒲 → 𝒰 ⊔ 𝒲 ·
is-equivalence _≈_ = 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 : (f : A → B) → IsEquivalence (ker{𝒰}{𝒲} f)
map-kernel-IsEquivalence f = record { rfl = λ x → refl
                                    ; sym = λ x y x₁ → ≡-sym{𝒲} x₁
                                    ; trans = λ x y z x₁ x₂ → ≡-trans 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 $[\ a\ ]\ R := \{b : A \mid R\ a\ b\}$.

```
[_]_ : A → Rel A 𝒲 → Pred A 𝒲
[ a ] R = λ x → R a x
```

Thus, $x \in [\ a\ ]\ R$ if and only if $R\ a\ x$, as desired. We often refer to $[\ a\ ]\ R$ as the *R-class containing a*, and we represent the collection of all such R-classes by the following type.

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

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

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

The next type is used to represent an 'R'-class with a designated representative.

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

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

This serves as a kind of *introduction rule*. Dually, the next type provides an *elimination rule*.[37]

```
⌜_⌝ : {R : Rel A 𝓦} → A / R → A

⌜ 𝒸 ⌝ = fst ∥ 𝒸 ∥
```

Later we will need the following tools for working with the quotient types defined above.

```
/-subset : {x y : A}{R : Rel A 𝓦} → IsEquivalence R → R x y → [ x ] R ⊆ [ y ] R
/-subset {x}{y} Req Rxy {z} Rxz = (trans Req) y x z (sym Req x y Rxy) Rxz

/-supset : {x y : A}{R : Rel A 𝓦} → IsEquivalence R → R x y → [ y ] R ⊆ [ x ] R
/-supset {x}{y} Req Rxy {z} Ryz = (trans Req) x y z Rxy Ryz

/-≐ : {x y : A}{R : Rel A 𝓦} → IsEquivalence R → R x y → [ x ] R ≐ [ y ] R
/-≐ Req Rxy = /-subset Req Rxy , /-supset Req Rxy
```

3.4 Truncation: continuous propositions, quotient extensionality

This section presents the Relations.Truncation module of the AgdaUALib, slightly abridged.[38] Here we discuss *truncation* and *h-sets* (which we just call *sets*). We first give a brief discussion of standard notions of *truncation* from a viewpoint that seems useful for formalizing mathematics in Agda.[39]

3.4.1 Background and motivation

This brief introduction to *truncation* is intended for novices. Those already familiar with the concept might wish to skip to the next subsection.

In general, we may have multiple inhabitants of a given type, hence (via Curry-Howard) multiple proofs of a given proposition. For instance, suppose we have a type X and an identity relation $\_\equiv_0\_$ on X so that, given two inhabitants of X, say, $a\ b : X$, we can form the type $a \equiv_0 b$. Suppose p and q inhabit the type $a \equiv_0 b$; that is, p and q are proofs of $a \equiv_0 b$, in which case we write $p\ q : a \equiv_0 b$. 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 $\equiv_0$, and whether there is some inhabitant, say, r of this type; i.e., whether there is a proof $r : p \equiv_1 q$ that the proofs of $a \equiv_0$

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

[38] 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`.

[39] Readers wishing to learn more about truncation may wish to consult [9, §34] (url link), [3], or [17, §7.1].

b are the same. If such a proof exists for all $p\ q : a \equiv_0 b$, then the proof of $a \equiv_0 b$ is unique; as a property of the types X and $\equiv_0$, this is sometimes called *uniqueness of identity proofs*.

Now, perhaps we have two proofs, say, $r\ s : p \equiv_1 q$ that the proofs p and q are equivalent. Then of course we wonder whether $r \equiv_2 s$ 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 $p \equiv_k q$. This is called truncation (at level k).

3.4.2 Sets

In homotopy type theory, a type X with an identity relation $\equiv_0$ is called a *set* (or *0-groupoid*) if for every pair $x\ y : X$ there is at most one proof of $x \equiv_0 y$. 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 $x \equiv_0 y$.

This notion is formalized in the Type Topology library using the type is-set which is defined using the is-subsingleton type (§2.4) as follows.

```
is-set : 𝒰 ̇ → 𝒰 ̇
is-set A = (x y : A) → is-subsingleton (x ≡ y)
```

Thus, the pair $(X\ ,\ \equiv_0)$ forms a set iff it satisfies $\forall\ x\ y : X \to$ is-subsingleton $(x \equiv_0 y)$.

We will also need the function to-Σ-≡, which is part of Escardó's characterization of *equality in Sigma types*.[40] It is defined as follows.

```
to-Σ-≡ : {A : 𝒰 ̇} {B : A → 𝒲 ̇} {σ τ : Σ B}
  →      Σ p ꞉ ∣ σ ∣ ≡ ∣ τ ∣ , (transport B p ∥ σ ∥) ≡ ∥ τ ∥
         ——————————————————————————————————————
  →      σ ≡ τ

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

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 Overture.Inverses 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, then we can prove that every 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. Assume the context contains the following typing judgments: $\{\mathcal{U}\ \mathcal{W} :$ Universe$\}\{A : \mathcal{U}\ \dot{}\}\{B : \mathcal{W}\ \dot{}\}$.

```
_⟺_ : 𝒰 ̇ → 𝒲 ̇ → 𝒰 ⊔ 𝒲 ̇
A ⟺ B = (A → B) × (B → A)
```

[40] See ([9]), specifically,
https://www.cs.bham.ac.uk/~mhe/HoTT-UF-in-Agda-Lecture-Notes/HoTT-UF-Agda.html#sigmaequality

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

monic-is-embedding|sets f Bset fmon b (u , fu≡b) (v , fv≡b) = γ
  where
  fuv : f u ≡ f v
  fuv = ≡-trans fu≡b (fv≡b ⁻¹)

  uv : u ≡ v
  uv = fmon u v fuv

  arg1 : Σ p ꞉ (u ≡ v) , (transport (λ a → f a ≡ b) p fu≡b) ≡ fv≡b
  arg1 = uv , Bset (f v) b (transport (λ a → f a ≡ b) uv fu≡b) fv≡b

  γ : u , fu≡b ≡ v , fv≡b
  γ = to-Σ-≡ arg1
```

In stating the previous result, we introduce a new convention to which we will try to adhere. If the antecedent of a theorem includes the assumption that one of the types involved is a set, then we add to the name of the theorem the suffix |sets, which 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 A is a *set*. As we just learned, this means there is at most one proof that two inhabitants of A are the same. Analogously, for predicates on A, 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.[41,42]

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

The principle of *proposition extensionality* asserts that logically equivalent propositions are equivalent. That is, if we have $P\ Q : \mathsf{Pred}_1$ and $|\ P\ | \subseteq |\ Q\ |$ and $|\ Q\ | \subseteq |\ P\ |$, then $P \equiv Q$. This is formalized as follows.[43]

```
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 $\doteq$ for predicates as follows: $P \doteq Q = (P \subseteq Q) \times (Q \subseteq P)$.

---

[41] Recall that Pred A 𝓦 is simply the function type $A \to \mathcal{W}\ \dot{}$, so Pred_1 is definitionally equal to $\Sigma\ P : (A \to \mathcal{W}\ \dot{})\ ,\ \forall\ x \to$ is-subsingleton $(P\ x)$.

[42] Agda now has a type called Prop, though we have never tried to use it. It likely provides at least some of the functionality we develop here, however, our preference is to assume only a minimal MLTT foundation and build up the types we need ourselves. For details about Prop, consult the official documentation at `https://agda.readthedocs.io/en/v2.6.1.3/language/prop.html`.

[43] cf. [9], specifically the section on "Prop extensionality and the powerset".

Therefore, if we postulate prop-ext $\mathcal{U}$ $\mathcal{W}$ and $P \doteq Q$, then $P \equiv Q$ obviously follows. Nonetheless, let us record this corollary here, assuming the context includes $\{\mathcal{U}\ \mathcal{W} : \mathsf{Universe}\}\{A : \mathcal{U}\ ̇\}$.

```
prop-ext' : prop-ext 𝓤 𝓦 → {P Q : Pred₁ A 𝓦} → | P | ≐ | Q | → P ≡ Q
prop-ext' pe hyp = pe (fst hyp) (snd hyp)
```

The foregoing easily generalizes to binary relations. If R is a binary relation such that there is at most one way to prove that a given pair of elements is R-related, then we call R a *binary proposition*. As above, we use Type Topology's is-subsingleton type to impose this truncation assumption on a binary relation.[44,45]

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

3.4.5 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. We now define a type called class-extensionality' that will play a crucial role later (e.g., in the formal proof of Birkhoff's HSP theorem).[46]

```
module _ {𝓤 𝓦 : Universe}{A : 𝓤 ̇}{𝐑 : Pred₂ A 𝓦} where

  class-extensionality : prop-ext 𝓤 𝓦 → IsEquivalence | 𝐑 | → {u v : A}
    →                    | 𝐑 | u v → [ u ] | 𝐑 | ≡ [ v ] | 𝐑 |
  class-extensionality pe Reqv {u}{v} Ruv = ap fst PQ where
    P Q : Pred₁ A 𝓦
    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 |
    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))

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

[44] Recall that Rel A 𝓦 is simply the function type $A \to A \to \mathcal{W}\ ̇$, so Pred_2 is definitionally equal to $\Sigma\ R : (A \to A \to \mathcal{W}\ ̇)\ ,\ \forall\ x\ y \to$ is-subsingleton $(R\ x\ y)$.

[45] This is another example of *proof-irrelevance*. Indeed, 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. Note also that we could have used the definition is-subsingleton-valued from § 3.3.1 above to define Pred_2 by $\Sigma\ R : (\mathsf{Rel}\ A\ \mathcal{W})\ ,$ is-subsingleton-valued R, but this seems less transparent than our explicit definition.

[46] Previous proofs of the class-extensionality' theorem required *function extensionality* (§2.3); however, as the proof given here makes clear, this is unnecessary.

```
class-extensionality' : prop-ext 𝓤 𝓦 → (∀ C → is-subsingleton (𝒞 C))
  →                     IsEquivalence | R | → {u v : A}
  →                     | R | u v → ⟦ u ⟧ ≡ ⟦ v ⟧
class-extensionality' pe ssA Reqv Ruv = to-subtype-⟦⟧ ssA (class-extensionality pe Reqv Ruv)
```

We could have presented the last theorem so that the consequent is a Pi type, as follows.

```
class-extensionality'' : prop-ext 𝓤 𝓦 → (∀ C → is-subsingleton (𝒞 C)) → IsEquivalence | R |
  →                      Π u ꞉ A , Π v ꞉ A , (| R | u v → ⟦ u ⟧ ≡ ⟦ v ⟧)
class-extensionality'' pe ssA Reqv u v Ruv = class-extensionality' pe ssA Reqv Ruv
```

3.4.6 Continuous propositions*[33]

In this final subsection of our presentation of relations in type theory, we offer a few interesting new types to complement the types we defined in the module Relations.Continuous. We should point out however that (so far) no other modules of the library depend on the types defined here. Therefore, the reader may safely skip to Section 4 without fear that this will lead to confusion at some point later on.

We defined a type called ConRel in the Relations.Continuous module to represent relations of arbitrary arity. Here we introduce a new type of *truncated continuous relations*, the inhabitants of which we call *continuous propositions*.

```
module continuous-propositions {𝓤 : Universe}{I : 𝓥 ̇} where

  uv : Universe → Universe -- (merely a convenient shorthand)
  uv 𝓦 = 𝓤 ⊔ 𝓥 ⊔ 𝓦 ⁺

  open import Relations.Continuous using (ConRel; DepRel)

  ConProp : 𝓤 ̇ → (𝓦 : Universe) → uv 𝓦 ̇
  ConProp A 𝓦 = Σ P ꞉ (ConRel I A 𝓦) , ∀ a → is-subsingleton (P a)

  con-prop-ext : 𝓤 ̇ → (𝓦 : Universe) → uv 𝓦 ̇
  con-prop-ext A 𝓦 = {P Q : ConProp A 𝓦 } → | P | ⊆ | Q | → | Q | ⊆ | P | → P ≡ Q
```

To see the point of the types just defined, suppose con-prop-ext A $\mathcal{W}$ holds. Then we can prove that logically equivalent continuous propositions of type ConProp A $\mathcal{W}$ are equivalent. In other words, under the stated hypotheses, we obtain the following extensionality lemma for continuous propositions.

```
module _ (A : 𝓤 ̇)(𝓦 : Universe) where

  con-prop-ext' : con-prop-ext A 𝓦 → {P Q : ConProp A 𝓦} → | P | ≐ | Q | → P ≡ Q
  con-prop-ext' 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 → 𝓤 ̇) → (𝓦 : Universe) → uv 𝓦 ̇
  DepProp 𝒜 𝓦 = Σ P ꞉ (DepRel I 𝒜 𝓦) , ∀ a → is-subsingleton (P a)

  dep-prop-ext : (I → 𝓤 ̇) → (𝓦 : Universe) → uv 𝓦 ̇
  dep-prop-ext 𝒜 𝓦 = {P Q : DepProp 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.

```
module _ (𝒜 : I → 𝒰 ̇)(𝒲 : Universe) where

  dep-prop-ext' : dep-prop-ext 𝒜 𝒲 → {P Q : DepProp 𝒜 𝒲} → ∣ P ∣ ≐ ∣ Q ∣ → P ≡ Q
  dep-prop-ext' 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*4.4.1.

4.1 Signatures: types for operations & signatures

This section presents the Algebras.Signatures module of the AgdaUALib, slightly abridged.[47]

4.1.1 Operation type

We begin by defining the type of *operations*, as follows.

```
Op : 𝒱 ̇ → 𝒰 ̇ → 𝒰 ⊔ 𝒱 ̇
Op I A = (I → A) → A
```

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. For example, the *I-ary projection operations* on A can be codified as inhabitants of the type Op I A in the following way.

```
π : {I : 𝒱 ̇} {A : 𝒰 ̇ } → I → Op I A
π i x = x i
```

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 §3.2, in the UALib the symbol $\mathcal{O}$ always denotes the universe of *operation symbol* types, while $\mathcal{V}$ is always the universe of *arity* types.

In the Overture module we defined special syntax for the first and second projections—namely, ∣_∣ and ∥_∥, respectively. Consequently, if S : Signature $\mathcal{O}$ $\mathcal{V}$ is a signature, then ∣ S ∣

[47] 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.

denotes the set of operation symbols, and $\| S \|$ denotes the arity function. If $f : | S |$ is an operation symbol in the signature S, then $\| S \| f$ is the arity of f.

4.1.2.1 Example

Here is how we could define the signature for *monoids* as an inhabitant of the type Signature $\mathcal{O}$ $\mathcal{V}$.

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

monoid-sig : Signature 𝓞 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).[48]

4.2 Algebras: types for algebras, operation interpretation & compatibility

This section presents the Algebras.Algebras module of the AgdaUALib, slightly abridged.[49]

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 ̇ ,                    -- the domain
              Π f ꞉ | S | , Op (| S |) f A  -- the basic operations
```

To be precise, we might call an inhabitant of this type a "∞-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 of sets in §3.4.2.) We could then take this opportunity to define the type of "0-algebras" (algebras whose domains are sets), which may be closer to what most of us think of when doing informal universal algebra. However, we have found that we only need to assume the domains of our algebras are sets in a few places in the UALib, so it seems preferable to work with general (∞-)algebras throughout and then assume *uniqueness of identity proofs* explicitly and only where needed.

4.2.2 Algebras as record types

Some people find record types more natural and convenient than Sigma types, and might prefer the following representation of algebras.

```
record algebra (𝓤 : Universe) (S : Signature 𝓞 𝓥) : (𝓞 ⊔ 𝓥 ⊔ 𝓤) ⁺ ̇ where
  constructor mkalg
  field
```

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

[49] 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`.

```
univ : 𝓤 ̇
op : (f : ∣ S ∣) → ((∥ S ∥ f) → univ) → univ
```

If for some reason we want to use both representations of algebras and move back and forth between them, this is easily accomplished with the following functions.

```
algebra→Algebra : algebra 𝓤 S → Algebra 𝓤 S
algebra→Algebra A = (univ A , op A)

Algebra→algebra : Algebra 𝓤 S → algebra 𝓤 S
Algebra→algebra A = mkalg ∣ A ∣ ∥ A ∥
```

In developing the more advanced modules of the AgdaUALib, we have used the Sigma type representation of algebras exclusively, though we occasionally use record types to represent other basic objects (e.g., congruences, subalgebras).

4.2.3 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
```

Thus, 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 same arity, then $(f \mathbin{\hat{}} \mathbf{A})\ a$ denotes the operation f interpreted in $\mathbf{A}$ and evaluated at a.

4.2.4 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.

```
_↠_ : {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 : Universe}{X : 𝓧 ̇}{S : Signature 𝓞 𝓥}
         {𝒦 : (A : Algebra 𝓤 S) → X ↠ A} where
```

Then fst(𝒦 **A**) will denote the surjective map from X onto $\mid \mathbf{A} \mid$, and snd(𝒦 **A**) will be a proof that the map is surjective.

4.2.5 Lifts of algebras

Recall the discussion in §2.5 of the difficulties one encounters when working with a noncumulative universe hierarchy. There we made a promise to provide some domain-specific lifting and lowering methods. Here we fulfill this promise by supplying a couple of bespoke tools designed

to work with our operation and algebra types.

```
lift-op : ((I → A) → A) → (𝓦 : Universe) → ((I → Lift{𝓦} A) → Lift {𝓦} A)
lift-op f 𝓦 = λ x → lift (f (λ i → lower (x i)))

lift-alg : 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 : 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)
```

Our experience has shown lift-alg to be a perfectly adequate tool for resolving any universe level unification errors that arise when working with the Algebra type in Agda. We will see some examples of its effectiveness later.

4.2.6 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.

```
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.

4.2.7 Compatibility of continuous relations*[33]

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

This section presents the Algebras.Products module of the AgdaUALib, slightly abridged.[50] We assume a fixed signature S : Signature $\mathcal{O}$ $\mathcal{V}$ throughout the module by starting with the line module Algebras.Products {S : Signature $\mathcal{O}$ $\mathcal{V}$} where.

In the UALib the *product of S-algebras* is represented by the following type.[51]

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

⨅ 𝒜 = (Π i ꞉ 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
```

The type just defined is the one we use whenever the product of an indexed collection of algebras (of type Algebra) is required. However, for the sake of completeness, here is how one could define a type representing the product of algebras inhabiting the record type algebra.

```
open algebra
⨅' : (𝒜 : I → algebra 𝓤 S) → algebra (ℑ ⊔ 𝓤) S
⨅' 𝒜 = record { univ = ∀ i → univ (𝒜 i) ;                -- domain
                op = λ f a i → (op (𝒜 i)) f λ x → a x i } -- basic operations
```

Notation. Given a signature S : Signature $\mathcal{O}$ $\mathcal{V}$ the type Algebra $\mathcal{U}$ S has universe $\mathcal{O} \sqcup \mathcal{V} \sqcup \mathcal{U}\ ^{+}$‘. Such types occur so often in the [UALib][] that it is worthwhile to define the following shorthand for their universes.

Before going further, let us agree on another convenient notational convention, which is used in many of the later modules of the UALib. Given a signature S : Signature $\mathcal{O}$ $\mathcal{V}$, the type Algebra $\mathcal{U}$ S has universe level $\mathcal{O} \sqcup \mathcal{V} \sqcup \mathcal{U}\ ^{+}$, and the $\mathcal{O} \sqcup \mathcal{V}$ part remains fixed since $\mathcal{O}$ and $\mathcal{V}$ always denote the universe levels of operation and arity types, respectively. Such levels occur so often in the UALib that we define the following shorthand for it: ov $\mathcal{U}$:= $\mathcal{O} \sqcup \mathcal{V} \sqcup \mathcal{U}\ ^{+}$.

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 level $\mathcal{U}$ and signature S. That is, $\mathscr{K}$: Pred(Algebra $\mathcal{U}$ S) $\mathcal{W}$ for some $\mathcal{W}$. Later we will formally state and prove that the product of all subalgebras of algebras in $\mathscr{K}$ belongs to the class SP($\mathscr{K}$) of subalgebras of products of algebras in $\mathscr{K}$. That is, $\bigsqcap$ S($\mathscr{K}$) $\in$ SP($\mathscr{K}$). This turns out to be a nontrivial exercise.

To begin, we need to define types that represent products over arbitrary (nonindexed) families such as $\mathscr{K}$ or S($\mathscr{K}$). Observe that Π $\mathscr{K}$ is definitely *not* what we want. To see why, recall that Pred(*Algebra* $\mathcal{U}$ S) $\mathcal{W}$‘ is just an alias for the function type Algebra $\mathcal{U}$ S $\to$ $\mathcal{W}$ ˙. We interpret the latter semantically by taking $\mathscr{K}$ **A** to be the assertion that $\mathscr{K}$ **A** belongs to $\mathscr{K}$ **A**, denoted **A** $\in$ $\mathscr{K}$. Therefore, by definition, we have

$$\begin{aligned} \Pi\ \mathscr{K} &= \Pi\ \mathbf{A} : (\text{Algebra}\ \mathcal{U}\ S)\ ,\ \mathscr{K}\,\mathbf{A} \\ &= \Pi\ \mathbf{A} : (\text{Algebra}\ \mathcal{U}\ S)\ ,\ \mathbf{A} \in \mathscr{K}. \end{aligned}$$

Semantically, this is the assertion that every algebra of type Algebra $\mathcal{U}$ S belongs to $\mathscr{K}$, and this bears little resemblance to the product of algebras that we seek.

---

[50] 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.

[51] Alternative equivalent notation for the domain of the product is $\forall\ i \to \mid \mathcal{A}\ i \mid$.

What we need is a type that serves to index the class $\mathscr{K}$, and a function $\mathfrak{A}$ that maps an index to the inhabitant of $\mathscr{K}$ at that index. But $\mathscr{K}$ is a predicate (of type (Algebra $\mathcal{U}$ S) $\to \mathcal{W}$ ˙) and the type Algebra $\mathcal{U}$ S seems rather nebulous in that there is no natural indexing class with which to "enumerate" all inhabitants of Algebra $\mathcal{U}$ S that belong to $\mathscr{K}$.[52]

The solution is to essentially take '$\mathscr{K}$' itself to be the indexing type; at least heuristically that is how one can view the type '$\mathfrak{I}$' that we now define.[53]

```
module class-products {𝓤 : Universe} (𝒦 : Pred (Algebra 𝓤 S)(ov 𝓤)) where

  ℑ : ov 𝓤 ̇
  ℑ = Σ 𝐀 ꞉ (Algebra 𝓤 S) , (𝐀 ∈ 𝒦)
```

Taking the product over the index type $\mathfrak{I}$ requires a function that maps an index $i : \mathfrak{I}$ to the corresponding algebra. Each $i : \mathfrak{I}$ denotes a pair, $(\mathbf{A}\ ,\ p)$, where $\mathbf{A}$ is an algebra and p is a proof that $\mathbf{A}$ belongs to $\mathscr{K}$, so the function mapping such an index to the corresponding algebra is simply the first projection.

```
  𝔄 : ℑ → Algebra 𝓤 S
  𝔄 = λ (i : ℑ) → ∣ i ∣
```

Finally, we represent the product of all members of the class $\mathscr{K}$ by the following type.

```
  class-product : Algebra (ov 𝓤) S
  class-product = ⨅ 𝔄
```

Observe that the application $\mathfrak{A}\ (\mathbf{A}\ ,\ p)$ of $\mathfrak{A}$ to the pair $(\mathbf{A}\ ,\ p)$ (the result of which is simply the algebra $\mathbf{A}$) may be viewed as the *projection* of the product $\bigsqcap \mathfrak{A}$ onto the "$(\mathbf{A}\ ,\ p)$-th component" of the product.

4.4 Congruences: types for congruences & quotient algebras

This section presents the Algebras.Congruences module of the AgdaUALib, slightly abridged.[54] 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 alternative ways, not only in type theory, but also in the informal presentation. Informally, a relation is a congruence if and only if it is both an equivalence relation on the domain of $\mathbf{A}$ and a subalgebra of the square of $\mathbf{A}$. Formally, a compatible equivalence relation can be represented as an inhabitant of a certain Sigma type (which we denote by 'Con') or a certain record type (which we denote by 'Congruence').

```
  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
```

[52] If you haven't already seen this before, do yourself a favor and give it some thought; see if the correct type comes to you organically.

[53] **Unicode Hints**. Some of our types are denoted with with Gothic ("mathfrak") symbols. To produce them in agda2-mode, type `\Mf` followed by a letter. For example, `\MfI` ⇝ $\mathfrak{I}$.

[54] 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`.

```
⟨_⟩ : Rel | A | 𝓦
Compatible : compatible A ⟨_⟩
IsEquiv : IsEquivalence ⟨_⟩
```

We defined the zero relation **0**-rel in the Relations.Discrete module, and we now demonstrate how to build the trivial congruence out of it. 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 Overture.Equality module, so we simply apply the corresponding proofs.

```
0-IsEquivalence : {A : 𝓤 ̇} → IsEquivalence {A = A} 0
0-IsEquivalence = record {rfl = λ x → refl{x = x}; sym = ≡-symmetric; trans = ≡-transitive}
```

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

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

Finally, we have the ingredients need to construct the zero congruence of any algebra we like. (For example, see the proof of ⟦0⟧ A ╱ θ below.

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

4.4.1 Quotient Algebras

In many areas of abstract mathematics, and especially in universal algebra, the quotient of an algebra **A** with respect to a congruence relation θ of **A** plays a central role. This quotient is typically denoted by **A** ╱ θ and Agda allows us to define and express quotients using this standard notation.[55]

```
_╱_ : (A : Algebra 𝓤 S) → Congruence{𝓤}{𝓦} A → Algebra (𝓤 ⊔ 𝓦 ⁺) 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
```

Example. If we adopt the notation [0] **A** ╱ θ for the zero (or identity) relation on the quotient algebra **A** ╱ θ, then we would define this relation as follows.

```
[0]_╱_ : (A : Algebra 𝓤 S)(θ : Congruence{𝓤}{𝓦} A) → Rel (| A | / ⟨ θ ⟩)(𝓤 ⊔ 𝓦 ⁺)
[0] A ╱ θ = λ x x₁ → x ≡ x₁
```

We obtain from this the zero congruence relation of **A** ╱ θ, which we denote by ⟦0⟧ **A** ╱ θ, by applying the Δ function defined above.

```
⟦0⟧_╱_ : (A : Algebra 𝓤 S)(θ : Congruence{𝓤}{𝓦} A){fe : funext 𝓥 (𝓤 ⊔ (𝓦 ⁺))}
  →      Congruence (A ╱ θ)
(⟦0⟧ A ╱ θ) {fe} = Δ fe
```

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

Finally, the following *elimination rule* is sometimes useful.

```
module _ {𝐀 : Algebra 𝓤 S} where

  ⁄-≡ : (θ : Congruence{𝓤}{𝓦} 𝐀){u v : ∣ 𝐀 ∣} → ⟦ u ⟧{⟨ θ ⟩} ≡ ⟦ v ⟧ → ⟨ θ ⟩ u v
  ⁄-≡ θ refl = IsEquivalence.rfl (IsEquiv θ) _
```

5 Concluding Remarks

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

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

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 Guillaume Brunerie. Truncations and truncated higher inductive types, September 2012. URL: https://homotopytypetheory.org/2012/09/16/truncations-and-truncated-higher-inductive-types/.

4 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.

5 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.

6 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.

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

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

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

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

11 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.

12 nLab authors. constructive mathematics. http://ncatlab.org/nlab/show/constructive%20mathematics, March 2021. Revision 65.

13 nLab authors. predicative mathematics. http://ncatlab.org/nlab/show/predicative%20mathematics, March 2021. Revision 22.

14 nLab authors. propositions as types. http://ncatlab.org/nlab/show/propositions%20as%20types, March 2021. Revision 40.

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

16 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.

17 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.

18 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.

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

20 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.

21 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.

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