The Agda Universal Algebra Library, Part 1: relations, algebras, congruences, and quotients

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 universal algebra, equational logic, and model theory, and as such provides many examples that exhibit the power of inductive and dependent types for representing and reasoning about mathematical structures and equational theories. In this paper we describe some of the most important foundational types and assumptions of the UALib, and we discuss some of the problems that arose when developing it, as well as our solutions to these problems.

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

Keywords and phrases Universal algebra, Equational logic, Model theory, Dependent type theory, Formalization of mathematics, Agda, Proof assistant, Constructive mathematics

Digital Object Identifier 10.4230/LIPIcs..2021.0

Related Version hosted on arXiv
Extended Version: `arxiv.org/pdf/2101.10166`

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

Acknowledgements The author wishes to thank Cliff Bergman, Hyeyoung Shin, and Siva Somayyajula for supporting and contributing to this project, and Martín Escardó for creating the Type Topology library and teaching a course on Univalent Foundations of Mathematics with Agda at the 2019 Midlands Graduate School in Computing Science. Of course, this 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.

Contents

1 Introduction **2**
1.1 Contributions and organization . . . 3

2 Agda Prelude **4**
2.1 Universes and dependent types . . . 4
2.2 Equality . . . 5
2.3 Function Extensionality . . . 7
2.4 Inverses and Inverse Images of Functions . . . 10
2.5 Lifts to Higher Universes . . . 12

3 Relation and Quotient Types **14**
3.1 Discrete Relations . . . 14
3.2 Continuous Relations . . . 17
3.3 Equivalences and Quotients . . . 19
3.4 Truncation, Sets, Propositions . . . 21

4 Algebra Types **25**
4.1 Operations and Signatures . . . 26
4.2 Algebras . . . 27
4.3 Product Algebras . . . 29
4.4 Congruence Relations . . . 31
4.5 Quotient Algebras . . . 32

5 Concluding Remarks **32**

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 (Agda UALib), a software library containing formal statements and proofs of the core definitions and results of universal algebra. The UALib is written in Agda [7], 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.

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

- Capretta [1] (1999) formalized the basics of universal algebra in the Calculus of Inductive Constructions using the Coq proof assistant;
- Spitters and van der Weegen [9] (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 [6] (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; the project (like the UALib) uses Martin-Löf Type Theory and the Agda proof assistant.

Some other projects aimed at formalizing mathematics generally, and algebra in particular, have developed into very extensive libraries that include definitions, theorems, and proofs about algebraic structures, such as groups, rings, modules, etc. However, the goals of these efforts seem to be the formalization of special classical algebraic structures, as opposed to the general theory of (universal) algebras. Moreover, the part of universal algebra and equational logic formalized in the UALib extends beyond the scope of prior efforts and. In particular, the library now includes a proof of Birkhoff's variety theorem. Most other proofs of this theorem that we know of are informal and nonconstructive.[2]

The seminal idea for the Agda UALib 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.

1.1 Contributions and organization

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. We are in the process of writing two follow-up papers covering more advanced aspects of the library. The contents of these sequels are mentioned in the conclusion of the paper.

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

The second part (§3) covers relations 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). Section 3.3 covers standard types for equivalence relations and quotients, and §3.4 discusses 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

---

[2] After the completion of this work, the author learned about a constructive version of Birkhoff's theorem that was proved by Carlström in [2]. The latter is presented in the standard, informal style of mathematical writing in the current literature, and as far as we know it was never implemented formally and type-checked with a proof assistant. Nonetheless, a comparison of the version of the theorem presented in [2] to the Agda proof we give here would be interesting.

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*, including types for representing *products over arbitrary classes of algebraic structures* (§4.3). Finally, we define types for congruence relations and quotient algebras in 4.4–4.5.

To conclude this introduction, we offer the following links to official sources of information about the Agda UALib.

- `ualib.org` (the web site) includes every line of code in the library, rendered as html and accompanied by documentation, and
- `gitlab.com/ualib/ualib.gitlab.io` (the source code) freely available and licensed under the Creative Commons Attribution-ShareAlike 4.0 International License.

These sources include complete proofs of every theorem we mention here, and much more, including the Agda modules that will be covered in the second and third installments in this series of papers on the UALib.

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

2 Agda Prelude

2.1 Universes and dependent types

This section describes certain key components of the Prelude.Preliminaries module of the Agda UALib.[3] For the benefit of readers who are not proficient in Agda or type theory, we briefly describe some of the most important types and features used in the UALib. Of necessity, the descriptions will be terse, but are usually accompanied by pointers to relevant sections of the appendix or the html documentation.

We begin by highlighting some of the key parts of the Prelude.Preliminaries module of the UALib. This module imports everything we need from Martin Escardó's Type Topology library [5], defines some basic types and proves some of their properties. We do not cover the entire Preliminaries module here, but instead call attention to aspects that differ from standard Agda syntax. For more details, see [4, §2].

Agda programs typically begin by setting some options and by importing from existing libraries. Options are specified with the OPTIONS *pragma* and control the way Agda behaves by, for example, specifying which logical foundations should be assumed when the program is type-checked to verify its correctness. Every Agda program in the UALib begins with the following pragma, which has the effects described below.

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

- *without-K* disables Streicher's K axiom; see [10];
- *exact-split* makes Agda accept only definitions that are *judgmental* or *definitional* equalities; see [12];
- *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 [11] and [12].

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

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

2.1.1 Agda's universe hierarchy

The Agda UALib adopts the notation of Martin Escardo's Type Topology library [5]. In particular, *universe levels*[4] are denoted by capitalized script letters from the second half of the alphabet, e.g., $\mathcal{U}$, $\mathcal{V}$, $\mathcal{W}$, etc. Also defined in Type Topology are the operators $\,^{\cdot}$ and $^{+}$. These map a universe level $\mathcal{U}$ to the universe $\mathcal{U}\,^{\cdot} :=$ Set $\mathcal{U}$ and the level $\mathcal{U}\,^{+} :=$ lsuc $\mathcal{U}$, respectively. Thus, $\mathcal{U}\,^{\cdot}$ is simply an alias for the universe Set $\mathcal{U}$, and we have $\mathcal{U}\,^{\cdot} : \mathcal{U}\,^{+}\,^{\cdot}$.

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

2.1.2 Dependent pairs

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

Agda's default syntax for a Sigma type is $\Sigma\ \lambda(x : X) \to Y$, but we prefer the notation $\Sigma\ x : X\ ,\ Y$, which is closer to the standard syntax described in the preceding paragraph. Fortunately, this preferred notation is available in the Type Topology library (see [5, Σ types]).[5]

Convenient notations for the first and second projections out of a product are |_| and ∥_∥, respectively. However, to improve readability or to avoid notation clashes with other modules, we sometimes use more standard alternatives, such as pr_1 and pr_2, or fst and snd, or some combination of these. The definitions are standard so we omit them (see [4] for details).

2.2 Equality

This section describes certain key components of the Prelude.Equality module of the Agda UALib.[6] Perhaps the most important types in type theory are the equality types. The *definitional equality* we use is a standard one and is often referred to as "reflexivity" or "refl". In our case, it is defined in the Identity-Type module of the Type Topology library, but apart from syntax it is equivalent to the identity type used in most other Agda libraries. Here is the definition.

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

Since refl _ is used so often, the following convenient shorthand is also provided in the Type Topology library.

```
pattern refl x = 𝓇ℯ𝒻ℓ {x = x}
```

[4] See https://agda.readthedocs.io/en/v2.6.1.2/language/universe-levels.html.
[5] The symbol : in the expression $\Sigma\ x : X\ ,\ Y$ is not the ordinary colon; rather, it is the symbol obtained by typing `\:4` in agda2-mode.
[6] For more details, see https://ualib.gitlab.io/Prelude.Equality.html.

Thus, whenever we need to complete a proof by simply asserting that x, or the (possibly implicit) thing in question, is definitionally equal to itself, we can invoke refl x, or (in the implicit case) refl _ or even $\mathscr{refl}$. (The *pattern* directive above is what makes last option available.)

Of course, ≡ is an equivalence relation, and the formalization of this fact is trivial. In fact, we don't even need to prove reflexivity, since it is the defining property of ≡.

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

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

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

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

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

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

Many proofs make abundant use of the symmetry of ≡, and the following syntactic sugar can improve the readability of such proofs.[7]

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

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

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

2.2.1 Transport

Alonzo Church characterized equality by declaring two things equal iff no property (predicate) can distinguish them (see [3]). In other terms, x and y are equal iff for all P we have $P\ x \to P\ y$. One direction of this implication is sometimes called *subsitution* or *transport* or *transport along an identity*. It asserts the following: if two objects are equal and one of them satisfies a given predicate, then so does the other. A type representing this notion is defined, along with the (polymorphic) identity function, in the MGS-MLTT module of the Type Topology library.

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

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

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

See [5] for a discussion of transport.[8]

A function is well defined if and only if it maps equivalent elements to a single element and we often use this nature of functions in Agda proofs. If we have a function $f : X \to$ Y, two elements $x\ x' : X$ of the domain, and an identity proof $p : x \equiv x'$, then we obtain a proof of $f\ x \equiv$ f x' by simply applying the ap function like so, ap $f\ p : f\ x \equiv f\ x'$. Escardó defines *ap* in the Type Topology library as follows.

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

Here are some variations of ap that are sometimes useful.

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

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

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

2.2.2 ≡-intro and ≡-elim for nondependent pairs

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

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

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

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

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

2.3 Function Extensionality

This section describes certain key components of the Prelude.Extensionality module of the Agda UALib.[9]

2.3.1 Background and motivation

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

[8] cf. the HoTT-Agda definition of transport at `https://github.com/HoTT/HoTT-Agda/blob/master/core/lib/Base.agda`.

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

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: $f x := x + 2$ and $g x := ((2 * x) - 8)/2 + 6$. Would you say that f and g are equal? Are they the "same" function? What does that even mean?

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

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

What if we had started out this discussion with two functions f and g both of which take a list as input and produce as output a correctly sorted version of that list? Are the functions the same? What if f were defined using the merge sort algorithm, while g used quick sort. I think most of us would now agree that f and g are not equal in this case.

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

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

2.3.2 Definition of function extensionality

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

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

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

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

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

In informal settings, this so-called "pointwise equality of functions" is typically what one means when one asserts that two functions are "equal."[10] 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* ([5]).

[10] If one assumes the *univalence axiom* of Homotopy Type Theory, then the _∼_ relation is equivalent to equality of functions. See the section "Function extensionality from univalence" of Escardó's notes [5].

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).[11] 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} → funext 𝓤 𝓥
```

The next two types define the converse of function extensionality.[12]

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

2.3.4 Alternative extensionality type

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

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

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

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

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

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

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

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

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

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

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

Now we are finally ready to define the type hfunext that gives an alternative means of postulating function extensionality.[13]

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

2.4 Inverses and Inverse Images of Functions

This section describes certain key components of the Prelude.Inverses module of the Agda UALib.[14] We begin by defining an inductive type that represents the semantic concept of *inverse image* of a function.

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

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

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

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

Note that an inhabitant of Image f ∋ b is a dependent pair (a , p), where $a : A$ and $p : b \equiv$ f a is a proof that f maps a to b. Since the proof that b belongs to the image of f is always accompanied by a "witness" $a : A$, we can actually *compute* a *pseudoinverse* of f. For convenience, we define

[13] In previous versions of the UALib we defined the type hfunext (using another name for it) before realizing that an equivalent type was already defined in the Type Topology library.
[14] For more details, see `https://ualib.gitlab.io/Prelude.Inverses.html`.

this inverse function, which we call Inv, and which takes an arbitrary $b : B$ and a witness-proof pair, (a , p) : Image $f \ni b$, and returns a.

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

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

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

2.4.1 Surjective functions

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

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

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

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

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

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

```
EpicInvIsRightInv : funext 𝓦 𝓦 → {A : 𝓤 ̇ } {B : 𝓦 ̇ }
                    (f : A → B) (fepi : Epic f)
                    ----------------
  →                 f ∘ (EpicInv f fepi) ≡ id B

EpicInvIsRightInv fe f fepi = fe (λ x → InvIsInv f (fepi x))
```

2.4.2 Injective functions

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

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

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

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

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

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

2.4.3 Embeddings

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

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

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

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

```
embedding-is-monic : {𝓧 𝓨 : Universe} {X : 𝓧 ̇ }{Y : 𝓨 ̇ }
                     (f : X → Y) → is-embedding f → Monic f

embedding-is-monic f femb x x' fxfx' = ap pr₁ ((femb (f x)) fa fb)
  where
  fa : fiber f (f x)
  fa = x , refl

  fb : fiber f (f x)
  fb = x' , (fxfx' ⁻¹)
```

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

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

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

2.5 Lifts to Higher Universes

This section describes certain key components of the Prelude.Lifts module of the Agda UALib.[15]

2.5.1 The noncumulative universe hierarchy

The hierarchy of universe levels in Agda looks like this:

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

$\mathcal{U}_0 : \mathcal{U}_1\ ^{\cdot}\ ,\quad \mathcal{U}_1 : \mathcal{U}_2\ ^{\cdot}\ ,\quad \mathcal{U}_2 : \mathcal{U}_3\ ^{\cdot}\ , \ldots$

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

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

2.5.2 Lifting and lowering

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

```
Algebras.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 Algebras.lagda, but was expecting to find the universe $\mathcal{O} \sqcup \mathcal{V} \sqcup \mathcal{U}^{+}$ instead.

To make these situations easier to deal with, we have developed some domain specific tools for the lifting and lowering of universe levels inhabited by algebra types. (Later we do the same for other domain specific types like homomorphisms, subalgebras, products, etc). This must be done carefully to avoid making the type theory inconsistent. In particular, we cannot lower the level of a type unless it was previously lifted to a (higher than necessary) universe level.

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

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

Using the Lift type, we define two functions that lift the domain (respectively, codomain) type of a function to a higher universe.

```
lift-dom : {X : 𝓧 ̇}{Y : 𝓨 ̇} → (X → Y) → (Lift{𝓦}{𝓧} X → Y)
lift-dom f = λ x → (f (lower x))

lift-cod : {X : 𝓧 ̇}{Y : 𝓨 ̇} → (X → Y) → (X → Lift{𝓦}{𝓨} Y)
lift-cod f = λ x → lift (f x)
```

Naturally, we can combine these and define a function that lifts both the domain and codomain type.

```
lift-fun : {X : 𝓧 ̇}{Y : 𝓨 ̇} → (X → Y) → (Lift{𝓦}{𝓧} X → Lift{𝓩}{𝓨} Y)
lift-fun f = λ x → lift (f (lower x))
```

For example, lift-dom takes a function f defined on the domain $X : \mathcal{X}\ ^{\cdot}$ and returns a function defined on the domain Lift{$\mathcal{W}$}{$\mathcal{X}$} $X : \mathcal{X} \sqcup \mathcal{W}\ ^{\cdot}$.

The point of a ramified hierarchy of universes is to avoid Russell's paradox, and this would be subverted if we lower the universe of a type that hasn't already be lifted. Fortunately, however, we can prove that lift and lower compose to the identity. Later, there will be some situations that require these facts, so we formalize them and their (trivial) proofs.

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

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

3 Relation and Quotient Types

This section presents some of the most important relation and quotient types that are defined in the Relations module of the Agda UALib.[16]

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

3.1 Discrete Relations

This section describes certain key components of the Relations.Discrete module of the Agda UALib.[17]

3.1.1 Unary relations

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

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

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

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

[16] For more details see `https://ualib.gitlab.io/Relations.html`.
[17] For more details, see `https://ualib.gitlab.io/Relations.Discrete.html`.

3.1.2 Membership and inclusion relations

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

$$\_\in\_ : \{A : \mathfrak{X}\ ^{\cdot}\ \} \to A \to \mathsf{Pred}\ A\ \mathfrak{Y} \to \mathfrak{Y}\ ^{\cdot}$$
$$x \in P = P\ x$$

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

$$\_\subseteq\_ : \{A : \mathfrak{X}\ ^{\cdot}\ \} \to \mathsf{Pred}\ A\ \mathfrak{Y} \to \mathsf{Pred}\ A\ \mathfrak{Z} \to \mathfrak{X} \sqcup \mathfrak{Y} \sqcup \mathfrak{Z}\ ^{\cdot}$$
$$P \subseteq Q = \forall\ \{x\} \to x \in P \to x \in Q$$

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 we typically means in informal mathematics for two subsets to be (extensionally) equal, which is that they contain the same elements. In the UALib we denote this type by $\doteq$ and define it as follows.[18]

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

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

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

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

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

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

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

3.1.4 Binary Relations

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

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

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

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

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

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

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

or as a relation from A to A,

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

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

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

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

3.1.5 The implication relation

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

```
_on_ : {𝓧 𝓨 𝓩 : Universe}{A : 𝓧 ̇}{B : 𝓨 ̇}{C : 𝓩 ̇}
 →     (B → B → C) → (A → B) → (A → A → C)

R on g = λ x y → R (g x) (g y)


_⇒_ : {𝓦 𝓧 𝓨 𝓩 : Universe}{A : 𝓦 ̇ } {B : 𝓧 ̇ }
 →    REL A B 𝓨 → REL A B 𝓩 → 𝓦 ⊔ 𝓧 ⊔ 𝓨 ⊔ 𝓩 ̇

P ⇒ Q = ∀ {i j} → P i j → Q i j
```

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

```
_=[_]⇒_ : {𝓦 𝓧 𝓨 𝓩 : Universe}{A : 𝓦 ̇ } {B : 𝓧 ̇ }
 →         Rel A 𝓨 → (A → B) → Rel B 𝓩 → 𝓦 ⊔ 𝓨 ⊔ 𝓩 ̇

P =[ g ]⇒ Q = P ⇒ (Q on g)
```

3.1.6 Compatibility of functions and binary relations

Before discussing general and dependent relations, we pause to define some types that are useful for asserting and proving facts about *compatibility* of functions with binary relations. The first definition simply lifts a binary relation on A to a binary relation on tuples of type $I \to A$.[19]

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

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

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

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

```
compatible-fun' : (f : (I → A) → A)(R : Rel A 𝓦) → 𝓥 ⊔ 𝓤 ⊔ 𝓦 ̇
compatible-fun' f R = ∀ a a' → (lift-rel R) a a' → R (f a) (f a')
```

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

3.2 Continuous Relations

This section describes certain key components of the Relations.Continuous module of the Agda UALib.[20] In set theory, an n-ary relation on a set A is simply a subset of the n-fold product $A \times A \times \cdots \times A$. As such, we could model these as predicates over the type $A \times \cdots \times A$, or as relations of type $A \to A \to \cdots \to A \to \mathcal{W}\,̇$ (for some universe $\mathcal{W}$). To implement such a relation in type theory, we would need to know the arity in advance, and then somehow form an n-fold arrow $\to$.

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

Below we will define GenRel to be the type $(I \to A) \to \mathcal{W}\,̇$ and we will call GenRel the type of *general relations*. This generalizes the unary and binary relations we saw earlier in the sense that general relations can have arbitrarily large arities. However, relations of type GenRel are still not completely general because they are defined over a single type. Just as Rel A $\mathcal{W}$ was the "single-sorted" special case of the "multisorted" binary relation type REL A B $\mathcal{W}$, so too will GenRel I A $\mathcal{W}$ be the single-sorted version of a completely general type of relations. The latter will represent relations that not only have arbitrary arities, but also are defined over arbitrary families of types.

To be more concrete, given an arbitrary family $A : I \to \mathcal{U}\,̇$ of types, we may have a relation from $A\ i$ to $A\ j$ to $A\ k$ to …, *ad infinitum*, where the collection represented by the "indexing" type I might not even be enumerable.[21]

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

[20] For more details, see `https://ualib.gitlab.io/Relations.Continuous.html`.

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

We will refer to such relations as *dependent relations* because in order to define a type to represent them, we absolutely need depedent types. The DepRel type that we define below is the manifestation of a completely general notion of relation.

3.2.1 General relations

We now define the type *GenRel* that represents predicates (or relations) of arbitrary arity over a single type A. We call this the type of **general relations**.[22]

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

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

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

  lift-gen-rel : GenRel I A 𝓦 → (I → J → A) → 𝓥 ⊔ 𝓦 ̇
  lift-gen-rel R 𝕒 = ∀ (j : J) → R λ i → (𝕒 i) j

  gen-compatible-fun : (I → (J → A) → A) → GenRel I A 𝓦 → 𝓥 ⊔ 𝓤 ⊔ 𝓦 ̇
  gen-compatible-fun 𝕗 R = ∀ 𝕒 → (lift-gen-rel R) 𝕒 → R λ i → (𝕗 i) (𝕒 i)
```

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

3.2.2 Dependent relations

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

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

We call DepRel the type of *dependent relations*.

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

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

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

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

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

3.3 Equivalences and Quotients

This section describes certain key components of the Relations.Quotients module of the Agda UALib.[23]

3.3.1 Properties of binary relations

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

```
module _ {𝓤 : Universe} where

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

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

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

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

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

3.3.2 Equivalence classes

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

```
module _ {𝓤 𝓡 : Universe} where

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

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

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

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

[23] For more details, see `https://ualib.gitlab.io/Relations.Quotients.html`.

function.

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

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

3.3.3 Equivalence classes

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

```
module _ {𝓤 𝓡 : Universe} where

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

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

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

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

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

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

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

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

If the relation is reflexive, then we have the following elimination rules.[24]

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

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


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

⌜ a ⌝ = ∣ ∥ a ∥ ∣
```

Later we will need the following additional quotient tools.

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

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

 open IsEquivalence{𝓤}{𝓡}

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

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

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

3.4 Truncation, Sets, Propositions

This section describes certain key components of the Relations.Truncation module of the Agda UALib.[25] Here we discuss *truncation* and *h-sets* (which we just call *sets*). We first give a brief discussion of standard notions of trunction, and then we describe a viewpoint which seems useful for formalizing mathematics in Agda. Readers wishing to learn more about truncation and proof-relevant mathematics should consult other sources, such as Section 34 and 35 of Martín Escardó's notes [5], or Guillaume Brunerie, Truncations and truncated higher inductive types, or Section 7.1 of the HoTT book [8].

Remark. Agda now has a built in type called Prop which may provide some or all of what we develop in this module. (See the Prop Section of agda.readthedocs.io.) However, we have never tried to use it, and it seems we can do without it.

3.4.1 Background and motivation[26]

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

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

Now, perhaps we have two proofs, say, $r\ s : p \equiv_{x1} q$ that the proofs p and q are equivalent. Then of course we wonder whether $r \equiv_{x2} s$ has a proof! But at some level we may decide that the potential to distinguish two proofs of an identity in a meaningful way (so-called *proof-relevance*) is not useful or desirable. At that point, say, at level k, we would be naturally inclined to assume that there is at most one proof of any identity of the form $p \equiv_{xk} q$. This is called truncation (at level k) (see, e.g., the Truncation section of Escardó's notes [5]).

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

[25] For more details, see `https://ualib.gitlab.io/Relations.Truncation.html`.
[26] The remarks in this subsection serve to introduce novices to the basic notion of truncation. Readers already familiar with this notion may wish to skip to the next subsection.

This notion is formalized in the Type Topology library using the types is-set which is defined using the is-subsingleton type that we saw earlier (§2.4) as follows.[27]

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

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

We will also make use of the function to-Σ-≡, which is part of Escardó's characterization of equality in Sigma types ([5]). It is defined as follows.

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

to-Σ-≡ (refl x , refl a) = refl (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.

3.4.2 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 Prelude.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 (i.e., is a set), then we can prove that any monic function into that codomain will be an embedding. On the other hand, embeddings are always monic, so we will end up with an equivalence. To prepare for this, we define a type _⟺_ with which to represent such equivalences.

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

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

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

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

  aa' : a ≡ a'
  aa' = fmon a a' faa'

  𝒜 : _ → _ ̇
  𝒜 a = f a ≡ b

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

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

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

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

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

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

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

3.4.3 Propositions

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

```
module _ {𝓤 : Universe} where

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

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

The principle of *proposition extensionality* asserts that logically equivalent propositions are equivalent. That is, if we have $P\ Q$: Pred₁ and $\mid P \mid \subseteq \mid Q \mid$ and $\mid Q \mid \subseteq \mid P \mid$, then $P \equiv Q$. This is formalized as follows (cf. the section "Prop extensionality and the powerset" of [5]).

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

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

```
prop-ext' : (A : 𝓤 ̇)(𝓦 : Universe){P Q : Pred₁ A 𝓦} → prop-ext A 𝓦
  →         | P | ≐ | Q | → P ≡ Q

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

Thus, for truncated predicates P and Q, if PropExt holds, then $P \subseteq Q \times Q \subseteq P \to P \equiv Q$, which is a useful extensionality principle.

3.4.4 Binary propositions

Given a binary relation R, it may be necessary or desirable to assume that there is at most one way to prove that a given pair of elements is R-related. If this is true of R, then we call R a *binary proposition*.[28]

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

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

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

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. (Notice that we don't require *function extensionality* (§2.3) here.)

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

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

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

    α : [ u ] ∣ 𝐑 ∣ ⊆ [ v ] ∣ 𝐑 ∣
    α ua = fst (/-≐ Reqv Ruv) ua

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

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

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


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

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


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

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

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

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

3.4.6 General and dependent propositions

We defined a type called GenRel in the Relations.Continuous module to represent relations of arbitrary arity. So, naturally, we define a type of *truncated general relations*, the inhabitants of which we will call *general propositions*.

```
GenProp : 𝓥 ̇ → 𝓤 ̇ → (𝓦 : Universe) → 𝓥 ⊔ 𝓤 ⊔ 𝓦 ⁺ ̇
GenProp I A 𝓦 = Σ P ꞉ (GenRel I A 𝓦) , ∀ a → is-subsingleton (P a)

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

The point of this is that if we assume gen-prop-ext I A $\mathscr{W}$ holds for some I, A and $\mathscr{W}$, then we can prove that logically equivalent general propositions of type GenProp I A $\mathscr{W}$ are equivalent.

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

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

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

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


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

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

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

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

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

4 Algebra Types

A standard way to define algebraic structures in type theory is using record types. However, we feel the dependent pair (or Sigma) type (§2.1.2) 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*4.5.

4.1 Operations and Signatures

This section describes certain key components of the Algebras.Signatures module of the Agda UALib.[29]

4.1.1 Operation type

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

```
module _ {𝓤 : Universe} where

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

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

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

4.1.2 Signature type

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

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

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

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

4.1.2.1 Example

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

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

[29] For more details, see `https://ualib.gitlab.io/Algebras.Signatures.html`.

```
  · : 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 λ { e → 𝟘 ; · → 𝟚 } which maps e to the empty type 𝟘 (since e is the nullary identity) and maps · to the two element type 𝟚 (since · is binary).[30]

4.2 Algebras

This section describes certain key components of the Algebras.Algebras module of the Agda UALib.[31]

4.2.1 The Algebra type

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

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

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

We could refer to an inhabitant of this type as a "∞-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 to be a set. (See Prelude.Preliminaries.html#truncation.)

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

4.2.2 Operation interpretation syntax

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

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

f ̂ 𝐀 = λ a → (∥ 𝐀 ∥ f) a
```

So, if f : ∣ S ∣ is an operation symbol in the signature S, and if a : ∥ S ∥ f → ∣ $\mathbf{A}$ ∣ is a tuple of the appropriate arity, then (f ̂ $\mathbf{A}$) a denotes the operation f interpreted in $\mathbf{A}$ and evaluated at a.

[30] The types 𝟘 and 𝟚 are defined in the MGS-MLTT module of the Type Topology library.
[31] For more details, see `https://ualib.gitlab.io/Algebras.Algebras.html`.

4.2.3 Arbitrarily many variable symbols

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

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

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

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

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

4.2.4 Lifts of algebras

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

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

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

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

  open algebra

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

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

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

4.2.5 Compatibility of binary relations

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

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

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

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

```
compatible : (A : Algebra 𝓤 S) → Rel ∣ A ∣ 𝓦 → 𝓞 ⊔ 𝓤 ⊔ 𝓥 ⊔ 𝓦 ̇
compatible A R = ∀ f → compatible-fun (f ̂ A) R
```

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

4.2.6 Compatibility of general relations

Next we define a type that represents *compatibility of a general relation* with all operations of an algebra. We start by defining compatibility of a general relations with a single operation.

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

 gen-compatible-op : ∣ S ∣ → GenRel I ∣ A ∣ 𝓦 → 𝓤 ⊔ 𝓥 ⊔ 𝓦 ̇
 gen-compatible-op f R = gen-compatible-fun (λ _ → (f ̂ A)) R
```

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

```
gen-compatible-op' : ∣ S ∣ → GenRel I ∣ A ∣ 𝓦 → 𝓤 ⊔ 𝓥 ⊔ 𝓦 ̇
gen-compatible-op' f R = ∀ 𝕒 → (lift-gen-rel R) 𝕒 → R (λ i → (f ̂ A) (𝕒 i))
```

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

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

```
gen-compatible : GenRel I ∣ A ∣ 𝓦 → 𝓞 ⊔ 𝓤 ⊔ 𝓥 ⊔ 𝓦 ̇
gen-compatible R = ∀ (f : ∣ S ∣ ) → gen-compatible-op f R
```

4.3 Product Algebras

This section describes certain key components of the Algebras.Products module of the Agda UALib.[32] We begin this module by assuming a signature S : Signature 𝓞 𝓥 which is then present and available throughout the module. Because of this, in contrast to our (highly abridged) descriptions of previous modules, we present the first few lines of the Algebras.Products module in full. They are as follows.

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

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

The product of S-algebras is defined as follows.

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

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

       λ f a i → (f ̂ 𝒜 i) λ x → a x i      – basic operations of the product algebra
```

[32] For more details, see `https://ualib.gitlab.io/Algebras.Products.html`.

4.3.1 Products of classes of algebras

An arbitrary class $\mathscr{K}$ of algebras is represented as a predicate over the type Algebra $\mathcal{U}$ S, for some universe $\mathcal{U}$ and signature S. That is, $\mathscr{K}$: Pred (Algebra $\mathcal{U}$ S) _.[33] Later we will formally state and prove that the product of all subalgebras of algebras in such a class belongs to SP($\mathscr{K}$) (subalgebras of products of algebras in $\mathscr{K}$). That is, $\prod$ S($\mathscr{K}$) $\in$ SP($\mathscr{K}$). This turns out to be a nontrivial exercise. In fact, it is not even clear (at least not to this author) how one should express the product of an entire class of algebras as a dependent type. However, if one ponders this for a while, the right type will eventually reveal itself, and will then seem obvious.[34] The solution is the class-product type whose construction is the main goal of this section.

First, we need a type that will serve to index the class, as well as the product of its members.[35]

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

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

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

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

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

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

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

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

  𝔄 𝒦 = λ (i : (ℑ 𝒦)) → ∣ i ∣
```

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

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

  class-product 𝒦 = ⨅ ( 𝔄 𝒦 )
```

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

---

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

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

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

4.4 Congruence Relations

This section describes certain key components of the Algebras.Congruences module of the Agda UALib.[36] A *congruence relation* of an algebra **A** is defined to be an equivalence relation that is compatible with the basic operations of **A**. This concept can be represented in a number of different ways in type theory. For example, we define both a Sigma type Con and a record type Congruence, each of which captures the informal notion of congruence, and each one is useful in certain contexts. (We will see examples later.)

```
Con : {𝓤 : Universe}(𝐀 : Algebra 𝓤 S) → ov 𝓤 ̇
Con {𝓤} 𝐀 = Σ θ ꞉ ( Rel ∣ 𝐀 ∣ 𝓤 ) , IsEquivalence θ × compatible 𝐀 θ

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

open Congruence
```

4.4.1 Example

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

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

```
module _ {𝓤 : Universe} where

  𝟎-IsEquivalence : {A : 𝓤 ̇ } → IsEquivalence{𝓤}{A = A} 𝟎-rel
  𝟎-IsEquivalence = record { rfl = ≡-rfl; sym = ≡-sym; trans = ≡-trans }
```

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

```
module _ {𝓤 : Universe} where

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

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

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

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

[36] For more details, see `https://ualib.gitlab.io/Algebras.Congruences.html`.

4.5 Quotient Algebras

An important construction in universal algebra is the quotient of an algebra $\mathbf{A}$ with respect to a congruence relation θ of $\mathbf{A}$. This quotient is typically denote by $\mathbf{A}$ / θ and Agda allows us to define and express quotients using the standard notation.[37]

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

𝐀 ╱ θ = ( ∣ 𝐀 ∣ / ⟨ θ ⟩ ) , -- the domain of the quotient algebra

        λ f 𝒂 → ⟦ (f ̂ 𝐀) (λ i → ∣ ∥ 𝒂 i ∥ ∣) ⟧ -- the basic operations of the quotient algebra
```

4.5.1 Examples

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

```
module _ {𝓤 𝓡 : Universe} where

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

  Zero╱ θ = λ x x₁ → x ≡ x₁
```

Finally, the following elimination rule is sometimes useful.

```
╱-refl : {𝐀 : Algebra 𝓤 S}(θ : Congruence{𝓤}{𝓡} 𝐀){a a' : ∣ 𝐀 ∣}
 → ⟦ a ⟧{⟨ θ ⟩} ≡ ⟦ a' ⟧ → ⟨ θ ⟩ a a'

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

5 Concluding Remarks

We've reached the end of Part 1 of our three-part series describing the Agda UALib. 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 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`.

2 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`.

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

3 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`.

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

5 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`.

6 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`.

7 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`.

8 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`.

9 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`.

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

11 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`.

12 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`.