Introduction to Univalent Foundations of Mathematics with Agda

4th March 2019, version of 29 March 2019, 21:15.

Martín Hötzel Escardó, School of Computer Science, University of Birmingham, UK.

Table of contents ⇓

Abstract. We introduce Voevodsky's univalent foundations and univalent mathematics, and explain how to develop them with the computer system Agda, which based on Martin-Löf type theory.

Agda allows us to write mathematical definitions, constructions, theorems and proofs, for example in number theory, analysis, group theory, topology, category theory or programming language theory, checking them for logical and mathematical correctness.

Agda is a constructive mathematical system by default, which amounts to saying that it can also be considered as a programming language. But we can assume the axiom of choice or the principle of excluded middle for pieces of mathematics that require them, at the cost of losing the implicit programming-language character of the system. For a fully constructive development of univalent mathematics in Agda, we would need to use its new cubical flavour, and we hope these notes provide a base for researchers interested in learning Cubical Type Theory and Cubical Agda as the next step.

Keywords. Univalent mathematics. Univalent foundations. Univalent type theory. Univalence axiom. ∞-Groupoid. Homotopy type. Type theory. Homotopy type theory. Intensional Martin-Löf type theory. Dependent type theory. Identity type. Type universe. Constructive mathematics. Agda. Cubical type theory. Cubical Agda. Computer-verified mathematics.

About this document. This is a set of so-called literate Agda files, with the formal, verified, mathematical development within *code* environments, and the usual mathematical discussion outside them. Most of this file is not Agda code, and is in markdown format, and the html web page is generated automatically from it using Agda and other tools. Github pull requests by students to fix typos or mistakes and clarify ambiguities are welcome. There is also a pdf version automatically generated from the html version. These notes were originally developed for the Midlands Graduate School 2019. They will evolve for a while.

Table of contents ⇓

Introduction

A univalent type theory is the underlying formal system for a foundation of univalent mathematics as conceived by Voevodsky.

In the same way as there isn't just one set theory (we have e.g. ZFC and NBG among others), there isn't just one univalent type theory (we have e.g. the underlying type theory used in UniMath, HoTT-book type theory, and cubical type theory, among others, and more are expected to come in the foreseeable future before the foundations of univalent mathematics stabilize).

The salient differences between univalent mathematics and traditional, set-based mathematics may be shocking at first sight:

1. The kinds of objects we take as basic.

- Certain things called types, or higher-groupoids, rather than sets, are the primitive objects.
- Sets, also called 0-groupoids, are particular kinds of types.
- So we have more general objects as a starting point.
- E.g. the type `ℕ` of natural numbers is a set, and this is a theorem, not a definition.
- E.g. the type of monoids is not a set, but instead a `1`-groupoid, automatically.
- E.g. the type of categories is a `2`-groupoid, again automatically.

2. The treatment of logic.

- Mathematical statements are interpreted as types rather than truth values.
- Truth values are particular kinds of types, called `-1`-groupoids, with at most one element.
- Logical operations are particular cases of mathematical operations on types.
- The mathematics comes first, with logic as a derived concept.
- E.g. when we say "and", we are taking the cartesian product of two types, which may or may not be truth values.

3. The treatment of equality.

- The value of an equality `x ≡ y` is a type, called the *identity type*, which is not necessarily a truth value.
- It collects the ways in which the mathematical objects `x` and `y` are identified.
- E.g. it is a truth value for elements of `ℕ`, as there is at most one way for two natural numbers to be equal.
- E.g. for the type of monoids, it is a set, amounting to the type of monoid isomorphisms, automatically.
- E.g. for the type of categories, it is a `1`-groupoid, amounting to the type of equivalences of categories, again automatically.

The important word in the above description of univalent foundations is *automatic*. For example, we don't *define* equality of monoids to be isomorphism. Instead, we define the collection of monoids as the large type of small types that are sets, equipped with a binary multiplication operation and a unit satisfying associativity of multiplication and neutrality of the unit in the usual way, and then we *prove* that the native notion of equality that comes with univalent type theory (inherited from Martin-Löf type theory) happens to coincide with monoid isomorphism. Largeness and smallness are taken as relative concepts, with type *universes* incorporated in the theory to account for this distinction.

Voevodsky's way to achive this is to start with a Martin-Löf type theory (MLTT), including identity types and type universes, and postulate a single axiom, named *univalence*. This axiom stipulates a canonical bijection between *type equivalences* (in a suitable sense defined by Voevodsky in type theory) and type identifications (in the original sense of Martin-Löf's identity type). Voevodsky's notion of type equivalence, formulated in MLTT, is a refinement of the notion of isomorphism, which works uniformly for all higher groupoids, i.e. types.

In particular, Voevodsky didn't design a new type theory, but instead gave an axiom for an existing type theory (or any of a family of possible type theories, to be more precise).

The main *technical* contributions in type theory by Voevodsky are:

4. The definition of type levels in MLTT, classifying them as n-groupoids including the possibility n=∞.
5. The (simple and elegant) definition of type equivalence that works uniformly for all type levels in MLTT.
6. The formulation of the univalence axiom in MLTT.

Univalent mathematics begins within MLTT with (4) and (5) before we postulate univalence. In fact, as the reader will see, we will do a fair amount of univalent mathematics before we formulate or assume the univalence axiom.

All of (4)-(6) crucially rely on Martin-Löf's identity type. Initially, Voevodsky thought that a new concept would be needed in the type theory to achieve (4)-(6) and hence (1) and (3) above. But he eventually discovered that Martin-Löf's identity type is precisely what he needed.

It may be considered somewhat miraculous that the addition of the univalence axiom alone to MLTT can achieve (1) and (3). Martin-Löf type theory was designed to achieve (2), and, regarding (1), types were imagined/conceived as sets (and even named *sets* in some of the original expositions by Martin-Löf), and, regarding (3), the identity type was imagined/conceived as having at most one element, even if MLTT cannot prove or disprove this statement, as was eventually shown by Hofmann and Streicher with their groupoid model of types in the early 1990's.

Another important aspect of univalent mathematics is the presence of explicit mechanisms for distinguishing

7. property (e.g. an unspecified thing exists),
8. data or structure (e.g. a designated thing exists or is given),

which are common place in current mathematical practice (e.g. cartesian closedness of a category is a property in some sense (up to isomorphism), whereas monoidal closedness is given structure).

In summary, univalent mathematics is characterized by (1)-(8) and not by the univalence axiom alone. In fact, 3/4 of theses notes begin *without* the univalence axiom (as measured by the number of lines in the source code for these lecture notes until we formulate the univalence axiom and start to use it).

Lastly, univalent type theories don't assume the axiom of choice or the principle of excluded middle, and so in some sense they are constructive by default. But we emphasize that these two axioms are consistent and hence can be safely used as assumptions. However, virtually all theorems of univalent mathematics, e.g. in UniMath, have been proved without assuming them, with natural mathematical arguments. The formulations of theses principles in univalent mathematics differ from their traditional formulations in MLTT, and hence we sometimes refer to them as the *univalent* principle of excluded middle and the *univalent* axiom of choice.

In these notes we will explore the above ideas, using Agda to write MLTT definitions, constructions, theorems and proofs, with univalence as an explicit assumption each time it is needed. We will have a further assumption, the existence of certain subsingleton (or propositional, or truth-value) truncations in order to be able to deal with the distinction between property and data, and in particular with the distinction between designated and unspecified existence (for example to be able to define the notions of image of a function and of surjective function). However, we will not assume them globally, so that the students can see clearly when univalence or truncation are or are not needed. In fact, the foundational definitions, constructions, theorems and proofs of univalent mathematics don't require univalence or propositional truncation, and so can be developed in a version of the original Martin-Löf type theories, and this is what happens in these notes, and what Voevodsky did in his brilliant original development in the computer system Coq. Our use of Agda, rather than Coq, is a personal matter of taste only, and the students are encouraged to learn Coq, too.

Table of contents ⇓

Homotopy type theory

Univalent type theory is often called *homotopy type theory*. Here we are following Voevodsky, who coined the phrases *univalent foundations* and *univalent mathematics*. We regard the terminology *homotopy type theory* as probably more appropriate for referring to the *synthetic* development of homotopy theory within univalent mathematics, for which we refer the reader to the HoTT book.

However, the terminlogy *homotopy type theory* is also used as a synonym for univalent type theory, not only because univalent type theory has a model in homotopy types (as defined in homotopy theory), but also because, without considering models, types do behave like homotopy types, automatically. We will not discuss how to do homotopy theory using univalent type theory in these notes. We refer the reader to the HoTT book as a starting point.

A common compromise is to refer to the subject as HoTT/UF.

Table of contents ⇓

General references

- Papers by Martin-Löf.
- Homotopy type theory website references.
- HoTT book.
- `ncatlab` references.

It particular, it is recommended to read the concluding notes for each chapter in the HoTT book for discussion of original sources. Moreover, the whole HoTT book is a recommended complementary reading for this course.

And, after the reader has gained enough experience:

- Voevodsky's original foundations of univalent mathematics in Coq.
- UniMath project in Coq.
- Coq HoTT library.
- Agda HoTT library.

Regarding the computer language Agda, we recommend the following as starting points:

- Agda wiki.
- Agda reference manual.
- Agda further references.
- Cubical Agda blog post.
- Cubical Agda documentation.

Regarding the genesis of the subject:

- A very short note on homotopy λ-calculus.
- Notes on homotopy λ-calculus.

A paper at the Bulletin of the AMS by Dan Grayson:

- An introduction to univalent foundations for mathematicians.

We have based these lecture notes on the slides of our talk *logic in univalent type theory*.

We also have an Agda development of univalent foundations which is applied to work on injective types, compact (or searchable) types, compact ordinals and more.

Table of contents ⇓

Choice of material

This is intended as an introductory graduate course. We include what we regard as the essence of univalent foundations and univalent mathematics, but we are certainly omiting important material that is needed to do univalent mathematics in practice, and the readers who wish to practice univalent mathematics should consult the above references.

Table of contents

Table of contents ⇑

MLTT in Agda

What is Agda?

There are two views:

1. Agda is a dependently-typed functional programming language.

2. Agda is a language for defining mathematical notions (e.g. group or topological space), formulating constructions to be performed (e.g. a type of real numbers, a group structure on the integers, a topology on the reals), formulating theorems (e.g. a certain construction is indeed a group structure, there are infinitely many primes), and proving theorems (e.g. the infinitude of the collection of primes with Euclid's argument).

This doesn't mean that Agda has two sets of features, one for (1) and the other for (2). The same set of features account simultaneously for (1) and (2). Programs are mathematical constructions that happen not to use non-constructive principles such as excluded middle.

In these notes we study a minimal univalent type theory and do mathematics with it using a minimal subset of the computer language Agda as a vehicle.

Agda allows one to construct proofs interactively, but we will not discuss how to do this in these notes. Agda is not an automatic theorem prover. We have to come up with our own proofs, which Agda checks for correctness. We do get some form of interactive help to input our proofs and render them as formal objects.

Table of contents ⇑

A spartan Martin-Löf type theory (MLTT)

Before embarking into a full definition of our Martin-Löf type theory in Agda, we summarize the particular Martin-Löf type theory that we will consider, by naming the concepts that we will include. We will have:

- An empty type `𝟘`.

- A one-element type `𝟙`.

- A type of `ℕ` natural numbers.

- Type formers `+` (binary sum), `Π` (product), `Σ` (sum), `Id` (identity type).

- Universes (types of types), ranged over by $\mathcal{U}, \mathcal{V}, \mathcal{W}$.

This is enough to do number theory, analysis, group theory, topology, category theory and more.

spartan /ˈspɑːt(ə)n/ adjective:

```
showing or characterized by austerity or a lack of comfort or
luxury.
```

We will also be rather spartan with the subset of Agda that we choose to discuss. Many things we do here can be written in more concise ways using more advanced features. Here we introduce a minimal subset of Agda where everything in our spartan MLTT can be expressed.

Table of contents ⇑

Getting started with Agda

We don't use any Agda library. For pedagogical purposes, we start from scratch, and here are our first two lines of code:

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

module HoTT-UF-Agda where
```

- The option `--without-K` disables Streicher's `K` axiom, which we don't want for univalent mathematics.
- The option `--exact-split` makes Agda to only accept definitions with the equality sign "`=`" that behave like so-called *judgmental* or *definitional* equalities.
- The option `--safe` disables features that may make Agda inconsistent, such as `--type-in-type`, postulates and more.
- Every Agda file is a module. These lecture notes are a set of Agda files, which are converted to html by Agda after it successfully checks the mathematical development for correctness.

Table of contents ⇑

Universes

A universe $\mathcal{U}$ is a type of types.

- One use of universes is to define families of types indexed by a type `X` as functions `X → 𝓤`.
- Such a function is sometimes seen as a property of elements of `X`.
- Another use of universes, as we shall see, is to define types of mathematical structures, such as monoids, groups, topological spaces, categories etc.

Sometimes we need more than one universe. For example, the type of groups in a universe lives in a bigger universe, and given a category in one universe, its presheaf category also lives in a larger universe.

We will work with a tower of type universes

```
𝓤₀, 𝓤₁, 𝓤₂, 𝓤₃, ...
```

These are actually universe names (also called levels, not to be confused with hlevels). We reference the universes themselves by a deliberately almost-invisible superscript dot. For example, we will have

```
𝟙 : 𝓤₀ ̇
```

where `𝟙` is the one-point type to be defined shortly. We will sometimes omit this superscript in our discussions, but are forced to write it in Agda code. We have that the universe $\mathcal{U}_0$ is a type in the universe $\mathcal{U}_1$, which is a type in the universe $\mathcal{U}_2$ and so on.

```
𝓤₀ ̇    : 𝓤₁ ̇
```

```
𝓤₁  : 𝓤₂

𝓤₂  : 𝓤₃

 ⋮
```

The assumption that $\mathcal{U}_0 : \mathcal{U}_0$ or that any universe is in itself or a smaller universe gives rise to a contradiction, similar to Russell's Paradox.

Given a universe $\mathcal{U}$, we denote by

```
𝓤 ⁺
```

its successor universe. For example, if $\mathcal{U}$ is $\mathcal{U}_0$ then $\mathcal{U}^+$ is $\mathcal{U}_1$. The least upper bound of two universes $\mathcal{U}$ and $\mathcal{V}$ is written

```
𝓤 ⊔ 𝓥.
```

For example, if $\mathcal{U}$ is $\mathcal{U}_0$ and $\mathcal{V}$ is $\mathcal{U}_1$, then $\mathcal{U} \sqcup \mathcal{V}$ is $\mathcal{U}_1$.

We now bring our notation for universes by importing our Agda file `Universes`. The Agda keyword `open` asks to make all definitions in the file `Universe` visible in our file here. The Agda code in these notes has syntax highlighting and html links, so that we can navigate to other modules, such as `Universe` or to definitions in this file.

```
open import Universes
```

We will refer to universes by letters $\mathcal{U}, \mathcal{V}, \mathcal{W}, \mathcal{T}$:

```
variable
 𝓤 𝓥 𝓦 𝓣 : Universe
```

Table of contents ⇑

The one-element type 𝟙

We place it in the first universe, and we name its unique element "⋆". We use the `data` declaration in Agda to introduce it:

```
data 𝟙 : 𝓤₀ ̇  where
 ⋆ : 𝟙
```

It is important that the point ⋆ lives in the type 𝟙 and in no other type. There isn't dual citizenship in our type theory. When we create a type, we also create freshly new elements for it, in this case "⋆". (However, Agda has a limited form of overloading, which allows us to sometimes use the same name for different things.)

Next we want to give a mechanism to prove that all points of the type 𝟙 satify a given property `A`.

- The property is a function `A : 𝟙 → 𝓤` for some universe $\mathcal{U}$.
- The type `A(x)`, which we will write simply `A x`, doesn't need to be a truth value. It can be any type. We will meet examples shortly.
- Mathematical statements are types, such as

  ```
  Π (A : 𝟙 → 𝓤), A ⋆ → Π (x : 𝟙), A x.
  ```

- We read this in natural language as "for any given property `A` of elements of the type 𝟙, if `A ⋆` holds, then it follows that `A x` holds for all `x : 𝟙`".

- In Agda above Π type is written as

  ```
  (A : 𝟙 → 𝓤 ̇ ) → A ⋆ → (x : 𝟙) → A x.
  ```

  This is the type of functions with three arguments `A : 𝟙 → 𝓤 ̇` and `a : A ⋆` and `x : 𝟙`, with values in the type `A x`.

- A proof of a mathematical statement rendered as a type is a construction of an element of the type. In our example, we have to construct a function, which we will name `𝟙-induction`.

We do this as follows in Agda, where we first declare the type of the function `𝟙-induction` with “`:`” and then define the function by an equation:

```
𝟙-induction : (A : 𝟙 → 𝓤 ̇ )
            → A ⋆ → (x : 𝟙) → A x
𝟙-induction A a ⋆ = a
```

Notice that we supply `A` and `a` as arbitrary arguments, but instead of an arbitrary `x : 𝟙` we have written “`⋆`”. Agda accepts this because it knows from the definition of `𝟙` that “`⋆`” is the only element of the type `𝟙`. This mechanism is called *pattern matching*.

A particular case of `𝟙-induction` occurs when the family `A` is constant with value `B`, which can be written variously as `A = λ (x : 𝟙) → B`, or `A = λ x → B` if we want Agda to figure out the type of `x` by itself, or `A = λ _ → B` if we don't want to name the argument of `A` because it is not used. In usual mathematical practice, such a lambda expression is often written `x ↦ B` (`x` is mapped to `B`) and so `A = (x ↦ B)`. Given a type `B` and a point `b : B`, we construct the function `𝟙 → B` that maps any given `x : 𝟙` to `b`.

```
𝟙-recursion : (B : 𝓤 ̇ ) → B → (𝟙 → B)
𝟙-recursion B b x = 𝟙-induction (λ _ → B) b x
```

Not all types have to be seen as mathematical statements (for example the type `ℕ` of natural numbers defined below). But the above definition has a dual interpretation as a mathematical function, and as the statement “`B` implies (*true* implies `B`)” where `𝟙` is the type encoding the truth value *true*.

The unique function to `𝟙` will be named `!𝟙`. We define two versions to illustrate implicit arguments, which correspond in mathematics to “subscripts that are omitted when the reader can safely infer them”, as for example for the identity function of a set or the identity arrow of an object of a category.

```
!𝟙' : (X : 𝓤 ̇ ) → X → 𝟙
!𝟙' X x = ⋆

!𝟙 : {X : 𝓤 ̇ } → X → 𝟙
!𝟙 x = ⋆
```

This means that when we write `!𝟙 x` we have to recover the (uniquely determined) missing type `X` with `x : X` “from the context”. When Agda can't figure it out, we need to supply it and write `!𝟙 {𝓤} {X} x`. This is because `𝓤` is also an implicit argument (all things declared with the Agda keyword *variable* as above are implicit arguments). There are other, non-positional, ways to indicate `X` without having to indicate `𝓤` too. Occasionally, people define variants of a function with different choices of “implicitness”, as above.

Table of contents ⇑

The empty type 𝟘

It is defined like `𝟙`, except that no elements are listed for it:

```
data 𝟘 : 𝓤₀ ̇  where
```

That's the complete definition. This has a dual interpretation, mathematically as the empty set (we can actually prove that this type is a set, once we know the definition of set), and logically as the truth value *false*. To prove that a property of elements of the empty type holds for all elements of the empty type we have to do nothing.

```
𝟘-induction : (A : 𝟘 → 𝓤 ̇ )
            → (x : 𝟘) → A x
𝟘-induction A ()
```

When we write the pattern `()`, Agda checks if there is any case we missed. If there is none, our definition is accepted. The expression `()` corresponds to the mathematical phrase vacuously true. The unique function from `𝟘` to any type is a particular case of `𝟘-induction`.

```
!𝟘 : (A : 𝓤 ̇ ) → 𝟘 → A
!𝟘 A a = 𝟘-induction (λ _ → A) a
```

We give the two names `is-empty` and `¬` to the same function now:

```
is-empty : 𝓤 ̇ → 𝓤 ̇
is-empty X = X → 𝟘

¬ : 𝓤 ̇ → 𝓤 ̇
¬ X = X → 𝟘
```

This says that a type is empty precisely when we have a function to the empty type. Assuming univalence, once we have defined the equality type former `_≡_`, we will be able to prove that `(is-empty X) ≡ (X ≃ 𝟘)`, where `X ≃ 𝟘` is the type of bijections, or equivalences, from `X` to `𝟘`. We will also be able to prove things like `(2 + 2 ≡ 5) ≡ 𝟘` and `(2 + 2 ≡ 4) ≡ 𝟙`.

This is for *numbers*. If we define *types* `𝟚 = 𝟙 + 𝟙` and `𝟜 = 𝟚 + 𝟚` with two and four elements respectively, where we are anticipating the definition of `_+_` for types, then we will instead have that `𝟚 + 𝟚 ≡ 𝟜` is a type with `4!` elements, which is number of permutations of a set with four elements, rather than a truth value `𝟘` or `𝟙`, as a consequence of the univalence axiom. That is, we will have `(𝟚 + 𝟚 ≡ 𝟜) ≃ (𝟜 + 𝟜 + 𝟜 + 𝟜 + 𝟜 + 𝟜)`, so that the type equality `𝟚 + 𝟚 ≡ 𝟜` holds in many more ways than the numerical equation `2 + 2 ≡ 4`.

The above is possible only because universes are genuine types and hence their elements (that is, types) have identity types themselves, so that writing `X ≡ Y` for types `X` and `Y` (inhabiting the same universe) is allowed.

When we view `𝟘` as *false*, we can read the definition of the *negation* `¬ X` as saying that "`X` implies *false*". With univalence we will be able to show that "(*false* → *false*) ≡ *true*", which amounts to `(𝟘 → 𝟘) ≡ 𝟙`, which in turn says that there is precisely one function `𝟘 → 𝟘`, namely the (vacuous) identity function.

Table of contents ⇑

The type ℕ of natural numbers

The definition is similar but not quite the same as the one via Peano Axioms.

We stipulate an element `zero : ℕ` and a successor function `ℕ → ℕ`, and then define induction. Once we have defined equality `_≡_`, we will *prove* the other peano axioms.

```
data ℕ : 𝓤₀ ̇  where
 zero : ℕ
 succ : ℕ → ℕ
```

In general, declarations with `data` are inductive definitions. To write the number `5`, we have to write

```
     succ (succ (succ (succ (succ zero))))
```

We can use the following Agda *pragma* to be able to just write `5` as a shorthand:

```
{-# BUILTIN NATURAL ℕ #-}
```

Apart from this notational effect, the above pragma doesn't play any role in the Agda development of these lectures notes.

In the following, the type family `A` can be seen as playing the role of a property of elements of `ℕ`, except that it doesn't need to be necessarily subsingleton-valued. When it is, the *type* of the function gives the familiar principle of mathematical induction for natural numbers, whereas, in general, its definition says how to compute with induction.

```
ℕ-induction : (A : ℕ → 𝓤 ̇ )
            → A 0
            → ((n : ℕ) → A n → A (succ n))
            → (n : ℕ) → A n
ℕ-induction A a₀ f = h
 where
  h : (n : ℕ) → A n
  h 0        = a₀
  h (succ n) = f n (h n)
```

The definition of `ℕ-induction` is itself by induction. It says that given a point `a₀ : A 0` and a function `f : (n : ℕ) → A n → A (succ n)`, in order to calculate an element of `A n` for a given `n : ℕ`, we just calculate `h n`, that is,

```
     f n (f (n-1) (f (n-2) (... (f 0 a₀)...))).
```

So the principle of induction is a construction that given a *base case* `a₀ : A 0`, an *induction step* `f : (n : ℕ) → A n → A (succ n)` and a number `n`, says how to get an element of the type `A n` by primitive recursion.

Notice also that `ℕ-induction` is the dependently typed version of primitive recursion, where the non-dependently type version is

```
ℕ-recursion : (X : 𝓤 ̇ )
            → X
            → (ℕ → X → X)
            → ℕ → X
ℕ-recursion X = ℕ-induction (λ _ → X)
```

The following special case occurs often (and is related to the fact that `ℕ` is the initial algebra of the functor `𝟙 + (-)`):

```
ℕ-iteration : (X : 𝓤 ̇ )
            → X
            → (X → X)
            → ℕ → X
ℕ-iteration X x f = ℕ-recursion X x (λ _ x → f x)
```

Agda checks that any recursive definition as above is well founded, with recursive invokations with structurally smaller arguments only. If it isn't, the definition is not accepted.

In official Martin-Löf type theories, we have to use the `ℕ-induction` functional to define everything else with the natural numbers. But Agda allows us to define functions by structural recursion, like we defined `ℕ-induction`.

We now define addition and multiplication for the sake of illustration. We first do it in Peano style. We will create a local `module` so that the definitions are not globally visible, as we want to have the symbols `+` and `×` free for type operations of MLTT to be defined soon. The things in the module are indented and are visible outside the module only if we `open` the module or if we write them as e.g. `Arithmetic.+` in the following example.

```
module Arithmetic where

  _+_  _×_  : ℕ → ℕ → ℕ

  x + 0      = x
  x + succ y = succ (x + y)

  x × 0      = 0
  x × succ y = x + x × y

  infixl 0 _+_
  infixl 1 _×_
```

The above `infix` operations allow us to indicate the precedences (multiplication has higher precedence than addition) and their associativity (here we take left-associativity as the convention, so that e.g. `x+y+z` parses as `(x+y)+z`).

Equivalent definitions use `ℕ-induction` on the second argument `y`, via `ℕ-iteration`:

```
module Arithmetic' where

  _+_  _×_  : ℕ → ℕ → ℕ

  x + y = h y
   where
    h : ℕ → ℕ
    h = ℕ-iteration ℕ x succ

  x × y = h y
   where
    h : ℕ → ℕ
    h = ℕ-iteration ℕ 0 (x +_)

  infixl 0 _+_
  infixl 1 _×_
```

Here the expression "`x +_`" stands for the function `ℕ → ℕ` that adds `x` to its argument. So to multiply `x` by `y`, we apply `y` times the function "`x +_`" to `0`.

As another example, we define the less-than-or-equal relation by nested induction, on the first argument and then the second, but we use pattern matching for the sake of readability.

Exercise. Write it using `ℕ-induction`, recursion or iteration, as appropriate.

```
module ℕ-order where

  _≤_ _≥_ : ℕ → ℕ → 𝓤₀ ̇
  0      ≤ y      = 𝟙
  succ x ≤ 0      = 𝟘
  succ x ≤ succ y = x ≤ y

  x ≥ y = y ≤ x
```

Exercise. After you have learned `Σ` and `_≡_` explained below, prove that

> `x ≤ y` if and only if `Σ \(z : ℕ) → x + z ≡ y`.

Later, when you have learned univalence prove that in this case this implies

> `(x ≤ y) ≡ Σ \(z : ℕ) → x + z ≡ y`.

That bi-implication can be turned into equality only holds for types that are subsingletons.

If we are doing applied mathematics and want to actually compute, we can define a type for binary notation for the sake of efficiency, and of course people have done that. Here we are not concerned with efficiency but only with understanding how to codify mathematics in (univalent) type theory and in Agda.

Table of contents ⇑

The binary sum type constructor _+_

We now define the disjoint sum of two types `X` and `Y`. The elements of the type

```
X + Y
```

are stipulated to be of the forms

`inl x` and `inr y`

with `x : X` and `y : Y`. If `X : 𝓤` and `Y : 𝓥`, we stipulate that `X + Y : 𝓤 ⊔ 𝓥` , where

```
𝓤 ⊔ 𝓥
```

is the least upper bound of the two universes 𝓤 and 𝓥. In Agda we can define this as follows.

```
data _+_ {𝓤 𝓥} (X : 𝓤 ̇ ) (Y : 𝓥 ̇ ) : 𝓤 ⊔ 𝓥 ̇  where
 inl : X → X + Y
 inr : Y → X + Y
```

To prove that a property `A` of the sum holds for all `z : X + Y`, it is enough to prove that `A(inl x)` holds for all `x : X` and that `A(inr y)` holds for all `y : Y`. This amounts to definition by cases:

```
+-induction : {X : 𝓤 ̇ } {Y : 𝓥 ̇ } (A : X + Y → 𝓦 ̇ )
            → ((x : X) → A(inl x))
            → ((y : Y) → A(inr y))
            → (z : X + Y) → A z
+-induction A f g (inl x) = f x
+-induction A f g (inr y) = g y
```

When the types `A` and `B` are understood as mathematical statements, the type `A + B` is understood as the statement "`A` or `B`", because to prove "`A` or `B`" we have to prove one of `A` and `B`. When `A` and `B` are simultaneously possible, we have two proofs, but sometimes we want to deliberately ignore which one we get, when we want to get a truth value rather than a possibly more general type, and in this case we use the truncation `∥ A + B ∥`.

But also `_+_` is used to construct mathematical objects. For example, we can define a two-point type:

```
𝟚 : 𝓤₀ ̇
𝟚 = 𝟙 + 𝟙
```

We can name the left and right points as follows, using patterns, so that they can be used in pattern matching:

```
pattern ₀ = inl ⋆
pattern ₁ = inr ⋆
```

We can define induction on 𝟚 directly by pattern matching:

```
𝟚-induction : (A : 𝟚 → 𝓤 ̇ ) → A ₀ → A ₁ → (n : 𝟚) → A n
𝟚-induction A a₀ a₁ ₀ = a₀
𝟚-induction A a₀ a₁ ₁ = a₁
```

Or we can prove it by induction on `_+_` and `𝟙`:

```
𝟚-induction' : (A : 𝟚 → 𝓤 ̇ ) → A ₀ → A ₁ → (n : 𝟚) → A n
𝟚-induction' A a₀ a₁ = +-induction A
```

```
(𝟙-induction (λ (x : 𝟙) → A (inl x)) a₀)
(𝟙-induction (λ (y : 𝟙) → A (inr y)) a₁)
```

Table of contents ⇑

Σ types

Given universes 𝓤 and 𝓥, a type

```
X : 𝓤
```

and a type family

```
Y : X → 𝓥 ,
```

we want to construct its sum, which is a type whose elements are of the form

```
(x , y)
```

with `x : X` and `y : Y x`. This sum type will live in the least upper bound

```
𝓤 ⊔ 𝓥
```

of the universes 𝓤 and 𝓥. We will write this sum

```
Σ Y,
```

with `X`, as well as the universes, implicit. Often Agda, and people, can figure out what the unwritten type `X` is, from the definition of `Y`. But sometimes there may be either lack of enough information, or of enough concentration power by people, or sufficiently powerful inference algorithms in the implementation of Agda. In such cases we can write

```
Σ λ(x : X) → Y x,
```

because `Y = λ (x : X) → Y x` by a so-called η-rule. However, we will often use the synonym `\` of `λ` for `Σ`, as if considering it as part of the `Σ` syntax.

```
Σ \(x : X) → Y x.
```

In MLTT we would write this as `Σ (x : X), Y x` or similar, for example with the indexing `x : X` written as a subscript of `Σ` or under it.

Or it may be that the name `Y` is not defined, and we work with a nameless family defined on the fly, as in the exercise proposed above:

```
Σ \(z : ℕ) → x + z ≡ y,
```

where `Y z = (x + z ≡ y)` in this case, and where we haven't defined the identity type former `_≡_` yet.

We can construct the `Σ` type former as follows in Agda:

```
record Σ {𝓤 𝓥} {X : 𝓤 ̇ } (Y : X → 𝓥 ̇ ) : 𝓤 ⊔ 𝓥 ̇  where
  constructor _,_
  field
   x : X
   y : Y x
```

This says we are defining a binary operator `_,_` to construct the elements of this type as `x , y`. The brackets are not needed, but we will often write them to get the more familiar notation `(x , y)`. The definition says that an element of `Σ Y` has two `fields`, giving the types for them.

Remark. Beginners may safely ignore this remark: Normally people will call these two fields something like `pr₁` and `pr₂`, or `fst` and `snd`, for first and second projection, rather than `x` and `y`, and then do `open Σ public` and have the projections available as functions automatically. But we will deliberately not do that, and instead define the projections ourselves, because this is confusing for beginners, no matter how mathematically or computationally versed they may be, in particular because it will not be immediately clear that the projections have the following types.

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

pr₂ : {X : 𝓤 ̇ } {Y : X → 𝓥 ̇ } → (z : Σ Y) → Y (pr₁ z)
pr₂ (x , y) = y
```

To prove that `A z` holds for all `z : Σ Y`, for a given property `A`, we just prove that we have `A(x , y)` for all `x : X` and `y : Y x`. This is called `Σ` induction or `Σ` elimination, or `uncurry`, after Haskell Curry.

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

This function has an inverse:

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

An important particular case of the sum type is the binary cartesian product, when the type family doesn't depend on the indexing family:

```
_×_ : 𝓤 ̇ → 𝓥 ̇ → 𝓤 ⊔ 𝓥 ̇
X × Y = Σ \(x : X) → Y
```

We have seen by way of examples that the function type symbol `→` represents logical implication, and that a dependent function type `(x : X) → A x` represents a universal quantification.

We have the following uses of `Σ`.

- The binary cartesian product represents conjunction "and". If the types `A` and `B` stand for mathematical statements, then the mathematical statement "`A` and `B`" is codified as `A × B`. This is because to prove "`A` and `B`", we have to provide a pair `(a , b)` of proofs `a : A` and `b : B`.

  So notice that in type theory, proofs are mathematical objects, rather than meta-mathematical entities like in set theory. They are just elements of types.

- The more general type `Σ \(x : X), A x`, if the type `X` stands for a mathematical object and `A` stands for a mathematical statement, represents *designated* existence "there is a designated `x : X` with `A x`". To prove this, we have to provide a specific `x : X` and a proof `a : A x`, together in a pair `(x , a)`.

- Later we will discuss *unspecified* existence `∃ \(x : X) → A x`, which will be obtained by a sort of quotient of `Σ \(x : X), A x`, written `∥ Σ \(x : X), A x ∥` that identifies all the elements of the type `Σ \(x : X), A x` in a single equivalence class, called its propositional or subsingleton truncation.

- Another reading of `Σ \(x : X), A x` is as "the type of `x : X` with `A x`", similar to subset notation `{ x ∈ X | A x }` in set theory. But have to be careful because if there is more than one element in the type `A x`, then `x` is put more than once in this type. In such situations, if we don't want that, we have to be careful and either ensure that the type `A x` has at most one element for every `x : X`, or instead consider the truncated type `∥ A x ∥` and write `Σ \(x : X), ∥ A x ∥`.

Table of contents ⇑

Π types

`Π` types are builtin with a different notation in Agda, as discussed above, but we can introduce the notation `Π` for them, similar to that for `Σ`:

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

Notice that the function type `X → Y` is the particular case of the `Π` type when the family `A` is constant with value `Y`.

We take the opportunity to define the identity function (in two versions with different implicit arguments) and function composition:

```
id : {X : 𝓤 ̇ } → X → X
id x = x

𝑖𝑑 : (X : 𝓤 ̇ ) → X → X
𝑖𝑑 X = id
```

Usually the type of function composition `_∘_` is given as simply

```
    (Y → Z) → (X → Y) → (X → Z).
```

With dependent functions, we can give it a more general type:

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

Notice that this type for the composition function can be read as a mathematical statement: If `Z y` holds for all `y : Y`, then for any given `f : X → Y` we have that `Z (f x)` holds for all `x : X`. And the non-dependent one is just the transitivity of implication.

The following functions are sometimes useful when we are using implicit arguments, in order to recover them explicitly without having to list them as given arguments:

```
domain : {X : 𝓤 ̇ } {Y : 𝓥 ̇ } → (X → Y) → 𝓤 ̇
domain {𝓤} {𝓥} {X} {Y} f = X

codomain : {X : 𝓤 ̇ } {Y : 𝓥 ̇ } → (X → Y) → 𝓥 ̇
codomain {𝓤} {𝓥} {X} {Y} f = Y

type-of : {X : 𝓤 ̇ } → X → 𝓤 ̇
type-of {𝓤} {X} x = X
```

Table of contents ⇑

The identity type former `Id`, also written `_≡_`

We now introduce the central type constructor of MLTT from the point of view of univalent mathematics. In Agda we can define Martin-Löf's identity type as follows:

```
data Id {𝓤} (X : 𝓤 ̇ ) : X → X → 𝓤 ̇  where
 refl : (x : X) → Id X x x
```

Intuitively, the above definition would say that the only element of the type `Id X x x` is something called `refl x` (for reflexivity). But, as we shall see in a moment, this intuition turns out to be incorrect.

Notice a crucial difference with the previous definitions using `data` or induction: In the previous cases, we defined *types*, namely `𝟘`, `𝟙`, `ℕ`, or a *type depending on type parameters*, namely `_+_`, with `𝓤` and `𝓥` fixed,

```
   _+_ : 𝓤 ̇ → 𝓥 ̇ → 𝓤 ⊔ 𝓥 ̇
```

But here we are defining a *type family* indexed by the *elements* of a give type, rather than a new type from old types. Given a type `X` in a universe `𝓤`, we define a *function*

```
   Id X : X → X → 𝓤
```

by some mysterious sort of induction. It is this that prevents us from being able to prove that `refl x` would be the only element of the type `Id X x x`, or that for `Id X x y` would have at most one element no matter what `y : X` is. There is however, one interesting, and crucial, thing we can prove, namely that for any fixed `x : X`, the type

```
   Σ \(y : Y) → Id X x y
```

is always a singleton.

We will use the following alternative notation for the identity type former `Id`, where the symbol "`_`" in the right-hand side of the definition indicates that we ask Agda to infer which type we are talking about (which is `X`, but this name is not available in the scope of the *definiting equation* of `_≡_`):

```
_≡_ : {X : 𝓤 ̇ } → X → X → 𝓤 ̇
x ≡ y = Id _ x y
```

Another intuition for this type family `_≡_ : X → X → 𝓤` is that it gives the least reflexive relation on the type `X`, as indicated by Martin-Löf's induction principle `J` discussed below.

Whereas we can make the intuition that `x ≡ x` has precisely one element good by postulating a certain `K` axiom due to Thomas Streicher, which comes with Agda by default but we have disabled above, we cannot *prove* that `refl x` is the only element of `x ≡ x` for an arbitrary type `X`. This non-provability result was established by Hofmann and Streicher, by giving a model of type theory in which types are interpreted as `1`-groupoids.

However, for the elements of *some* types, such as `ℕ`, it is possible to prove that any identity type `x ≡ y` has at most one element. Such types are called sets in univalent mathematics.

If instead of `K` we adopt Voevodsky's univalence axiom, we get specific examples of objects `x` and `y` such that the type `x ≡ y` has multiple elements, *within* the type theory. It follows that the identity type `x ≡ y` is fairly under-specified in general, in that we can't prove or disprove that it has at most one element.

There are two opposing ways to resolve the ambiguity or underspecification of the identity types: (1) We can consider the `K` axiom, which postulates that all types are sets, or (2) we can consider the univalence axiom, arriving at univalent mathematics, which gives rise to types that are more general than sets, the `n`-groupoids and `∞`-groupoids. In fact, the univalence axiom will say, in particular, that for some types `X` and elements `x y : X`, the identity type `x ≡ y` does have more than one element.

A possible way to understand the point `refl x` of the type `x ≡ x` is as the "generic identification" between `x` and itself, but which is by no means necessarily the *only* identitification in univalent foundations. It is generic in the sense that to explain what happens with all identifications `p : x ≡ y` between any two points `x` and `y` of a type `X`, it suffices to explain what happens with the identification `refl x : x ≡ x` for all points `x : X`. This is what the induction principle for identity given by Martin-Löf says, which he called `J` (we could have called it `≡-induction`, but we prefer to honour MLTT tradition):

```
J : (X : 𝓤 ̇ ) (A : (x y : X) → x ≡ y → 𝓥 ̇ )
  → ((x : X) → A x x (refl x))
  → (x y : X) (p : x ≡ y) → A x y p
J X A f x x (refl x) = f x
```

This is related to the Yoneda Lemma in category theory, if you are familiar with this subject, which says that certain natural transformations are *uniquely determined* by their *action on the identity arrows*, even if they are *defined for all arrows*. Similarly here, `J` is uniquely determined by its action on reflexive identifications, but is *defined for all identifications between any two points*, not just reflexivities.

In summary, Martin-Löf's identity type is given by the data

- `Id`,
- `refl`,
- `J`,
- the above equation for `J`.

However, we will not always use this induction principle, because we can instead work with the instances we need by pattern matching on `refl` (which is just what we did to define the principle itself) and there is a theorem by Jesper Cockx which says that with the Agda option `without-K`, pattern matching on `refl` can define/prove precisely what `J` can.

Exercise. Define

```
H : {X : 𝓤 ̇ } (x : X) (B : (y : X) → x ≡ y → 𝓥 ̇ )
  → B x (refl x)
  → (y : X) (p : x ≡ y) → B y p
H x B b x (refl x) = b
```

Then we can define `J` from `H` as follows:

```
J' : (X : 𝓤 ̇ ) (A : (x y : X) → x ≡ y → 𝓥 ̇ )
   → ((x : X) → A x x (refl x))
   → (x y : X) (p : x ≡ y) → A x y p
J' X A f x = H x (A x) (f x)

Js-agreement : (X : 𝓤 ̇ ) (A : (x y : X) → x ≡ y → 𝓥 ̇ )
               (f : (x : X) → A x x (refl x)) (x y : X) (p : x ≡ y)
             → J X A f x y p ≡ J' X A f x y p
Js-agreement X A f x x (refl x) = refl (f x)
```

Similarly define `H'` from `J` without using pattern matching on `refl` and show that it coincides with `H` (possibly using pattern matching on `refl`). This is harder.

Notational remark. The symbols "`=`" and "`≡`" are swapped with respect to the HoTT book convention for definitional/judgemental equality and typed-valued equality, and there is nothing we can do about that because "`=`" is a reserved Agda symbol for definitional equality. Irrespectively of this, it does make sense to use "`≡`" with a triple bar, if we understand this as indicating that there are multiple ways of identifying two things in general.

With this, we have concluded the rendering of our spartan MLTT in Agda notation. Before embarking on the development of univalent mathematics within our spartan MLTT, we pause to discuss some basic examples of mathematics in Martin-Löf type theory.

Table of contents ⇑

Basic constructions with the identity type

Transport along an identification.

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

We can equivalently define transport using `J` as follows:

```
transportJ : {X : 𝓤 ̇ } (A : X → 𝓥 ̇ ) {x y : X}
           → x ≡ y → A x → A y
transportJ {𝓤} {𝓥} {X} A {x} {y} = J X (λ x y _ → A x → A y) (λ x → id (A x)) x y
```

In the same way `ℕ`-recursion can be seen as the non-dependent special case of `ℕ`-induction, the following transport function can be seen as the non-dependent special case of the `≡`-induction principle `H` with some of the arguments permuted and made implicit:

```
nondep-H : {X : 𝓤 ̇ } (x : X) (A : X → 𝓥 ̇ )
         → A x → (y : X) → x ≡ y → A y
nondep-H {𝓤} {𝓥} {X} x A = H x (λ y _ → A y)

transportH : {X : 𝓤 ̇ } (A : X → 𝓥 ̇ ) {x y : X}
           → x ≡ y → A x → A y
transportH {𝓤} {𝓥} {X} A {x} {y} p a = nondep-H x A a y p
```

All the above transports coincide:

```
transports-agreement : {X : 𝓤 ̇ } (A : X → 𝓥 ̇ ) {x y : X} (p : x ≡ y)
                     → (transportH A p ≡ transport A p)
                     × (transportJ A p ≡ transport A p)
transports-agreement A (refl x) = refl (transport A (refl x)) ,
                                  refl (transport A (refl x))
```

The following is for use when we want to recover implicit arguments without mentioning them.

```
lhs : {X : 𝓤 ̇ } {x y : X} → x ≡ y → X
lhs {𝓤} {X} {x} {y} p = x

rhs : {X : 𝓤 ̇ } {x y : X} → x ≡ y → X
rhs {𝓤} {X} {x} {y} p = y
```

Composition of identifications. Given two identifications `p : x ≡ y` and `q : y ≡ z`, we can compose them to get an identification `p ∙ q : x ≡ z`. This can also be seen as transitivity of equality. Because the type of composition doesn't mention `p` and `q`, we can use the non-dependent version of `≡`-induction.

```
_∙_ : {X : 𝓤 ̇ } {x y z : X} → x ≡ y → y ≡ z → x ≡ z
p ∙ q = transport (lhs p ≡_) q p
```

Here we are considering the family `A t = (x ≡ t)`, and using the identification `q : y ≡ z` to transport `A y` to `A z`, that is `x ≡ y` to `x ≡ z`.

Exercise. define an alternative version that uses `p` to transport. Can you prove that the two versions give equal results?

When writing `p ∙ q`, we lose information on the lhs and the rhs of the identifications `p : x ≡ y` and `q : y ≡ z`, which makes some definitions hard to read. We now introduce notation to be able to write e.g.

```
    x ≡⟨ p ⟩

    y ≡⟨ q ⟩

    z ∎
```

as a synonym of the expression `p · q` with some of the implicit arguments of `_·_` made explicit. We have one ternary mixfix operator `_≡⟨_⟩_` and one unary `postfix` operator `_∎`.

```
_≡⟨_⟩_ : {X : 𝓤 ̇ } (x : X) {y z : X} → x ≡ y → y ≡ z → x ≡ z
x ≡⟨ p ⟩ q = p · q

_∎ : {X : 𝓤 ̇ } (x : X) → x ≡ x
x ∎ = refl x
```

Inversion of identifications. Given an identification, we get an identification in the opposite direction:

```
_⁻¹ : {X : 𝓤 ̇ } → {x y : X} → x ≡ y → y ≡ x
p ⁻¹ = transport (_≡ lhs p) p (refl (lhs p))
```

Application of a function to an identification. Given an identification `p : x ≡ x'` we get an identification `ap f p : f x ≡ f x'` for any `f : X → Y`:

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

Here the symbol "`-`", which is not to be confused with the symbol "`_`", is a variable. We will adopt the convention in these notes of using this variable name "`-`" to make clear which part of an expression we are replacing with `transport`.

Notice that we have so far used the recursion principle `transport` only. To reason about `transport`, `_·_`, `_⁻¹` and `ap`, we will need to use the full induction principle `J` (or equivalently pattern matching on `refl`).

Pointwise equality of functions. We will work with pointwise equality of functions, defined as follows, which, using univalence, will be equivalent to equality of functions.

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

The symbol ∀ is a built-in notation for `Π` . We could equivalently write the definiens as

```
    (x : _) → f x ≡ g x,
```

or, with our `Π` notation,

```
    Π \x → f x ≡ g x
```

Table of contents ⇑

Reasoning with negation

We first introduce notation for double and triple negation to avoid the use of brackets.

```
¬¬ ¬¬¬ : 𝓤 ̇ → 𝓤 ̇
¬¬  A = ¬(¬ A)
¬¬¬ A = ¬(¬¬ A)
```

To prove that `A → ¬¬ A`, that is, `A → ((A → 𝟘) → 𝟘)`, we start with a hypothetical element `a : A` and a hypothetical function `u : A → 𝟘` and the goal is to get an element of `𝟘`. All we need to do is to apply the function `u` to `a`. This gives double-negation introduction:

```
dni : (A : 𝓤 ̇ ) → A → ¬¬ A
dni A a u = u a
```

Mathematically, this says that if we have a point of `A` (we say that `A` is pointed) then `A` is nonempty. There is no general procedure to implement the converse, that is, from a function `(A → 𝟘) → 𝟘` to get a point of `A`. For truth values `A`, we can assume this as an axiom if we wish, because it is equivalent to the principle excluded middle. For arbitrary types `A`, this would be a form of global choice for type theory. However, global choice is inconsistent with univalence [HoTT book, Theorem 3.2.2], because there is no way to choose an element of every non-empty type in a way that is invariant under automorphisms. However, the axiom of choice *is* consistent with univalent type theory, as stated in the introduction.

In the proof of the following, we assume we are given hypothetical functions `f : A → B` and `v : B → 𝟘`, and a hypothetical element `a : A`, and our goal is to get an element of `𝟘`. But this is easy, because `f a : B` and hence `v (f a) : 𝟘`.

```
contrapositive : {A : 𝓤 ̇ } {B : 𝓥 ̇ } → (A → B) → (¬ B → ¬ A)
contrapositive f v a = v (f a)
```

We have given a logical name to this function. Mathematically, this says that if we have a function `A → B` and `B` is empty, then `A` must be empty, too. The proof is by assuming that `A` would have an inhabitant `a`, to get a contradiction, namely that `B` would have an inhabitant, too, showing that there can’t be any such inhabitant `a` of `A` after all. See Bauer for a general discussion.

And from this we get that three negations imply one:

```
tno : (A : 𝓤 ̇ ) → ¬¬¬ A → ¬ A
tno A = contrapositive (dni A)
```

Hence, using `dni` once again, we get that `¬¬¬ A` if and only if `¬ A`. It is entertaining to see how Brouwer formulated and proved this fact in his Cambridge Lectures on Intuitionism:

> Theorem. Absurdity of absurdity of absurdity is equivalent to absurdity.
>
> Proof. *Firstly*, since implication of the assertion *y* by the assertion *x* implies implication of absurdity of *x* by absurdity of *y*, the implication of *absurdity of absurdity* by *truth* (which is an established fact) implies the implication of *absurdity of truth*, that is to say of *absurdity*, by *absurdity of absurdity of absurdity*. *Secondly*, since truth of an assertion implies absurdity of its absurdity, in particular truth of absurdity implies absurdity of absurdity of absurdity.

If we define *logical equivalence* by

```
_⇔_ : 𝓤 ̇ → 𝓥 ̇ → 𝓤 ⊔ 𝓥 ̇
X ⇔ Y = (X → Y) × (Y → X)
```

then we can render Brouwer’s argument in Agda as follows, where the “established fact” is `dni`:

```
absurdity³-is-absurdity : {A : 𝓤 ̇ } → ¬¬¬ A ⇔ ¬ A
absurdity³-is-absurdity {𝓤} {A} = firstly , secondly
 where
  firstly : ¬¬¬ A → ¬ A
  firstly = contrapositive (dni A)
  secondly : ¬ A → ¬¬¬ A
  secondly = dni (¬ A)
```

But of course Brouwer, as is well known, was averse to formalism, and hence wouldn’t approve of such a sacrilege.

We now define a symbol for the negation of equality.

```
_≢_ : {X : 𝓤 ̇ } → X → X → 𝓤 ̇
x ≢ y = ¬(x ≡ y)
```

In the following proof, we have `u : x ≡ y → 𝟘` and need to define a function `y ≡ x → 𝟘`. So all we need to do is to compose the function that inverts identifications with `u`:

```
≢-sym : {X : 𝓤 ̇ } {x y : X} → x ≢ y → y ≢ x
≢-sym {𝓤} {X} {x} {y} u = λ (p : y ≡ x) → u (p ⁻¹)
```

To show that the type `𝟙` is not equal to the type `𝟘`, we use that `transport id` gives `𝟙 ≡ 𝟘 → id 𝟙 → id 𝟘` where `id` is the identity function of the universe $\mathcal{U}_0$. More generally, we have the following conversion of type identifications into functions:

```
Id-to-Fun : {X Y : 𝓤 ̇ } → X ≡ Y → X → Y
Id-to-Fun {𝓤} = transport (id (𝓤 ̇ ))
```

Here the identity function is that of the universe `𝓤` where the types `X` and `Y` live. An equivalent definition is the following, where this time the identity function is that of the type `X`:

```
Id-to-Fun' : {X Y : 𝓤 ̇ } → X ≡ Y → X → Y
Id-to-Fun' (refl X) = id X

Id-to-Funs-agree : {X Y : 𝓤 ̇ } (p : X ≡ Y)
                 → Id-to-Fun p ≡ Id-to-Fun' p
Id-to-Funs-agree (refl X) = refl (id X)
```

So if we have a hypothetical identification `p : 𝟙 ≡ 𝟘`, then we get a function `𝟙 → 𝟘`. We apply this function to `⋆ : 𝟙` to conclude the proof.

```
𝟙-is-not-𝟘 : 𝟙 ≢ 𝟘
𝟙-is-not-𝟘 p = Id-to-Fun p ⋆
```

To show that the elements `₁` and `₀` of the two-point type `𝟚` are not equal, we reduce to the above case. We start with a hypothetical identification `p : ₁ ≡ ₀`.

```
₁-is-not-₀ : ₁ ≢ ₀
₁-is-not-₀ p = 𝟙-is-not-𝟘 q
 where
  f : 𝟚 → 𝓤₀ ̇
  f ₀ = 𝟘
  f ₁ = 𝟙
  q : 𝟙 ≡ 𝟘
  q = ap f p
```

Remark. Agda allows us to use a pattern `()` to get the following quick proof. However, this method of proof doesn't belong to the realm of MLTT. Hence we will use the pattern `()` only in the above definition of `𝟘-induction`.

```
₁-is-not-₀[not-an-MLTT-proof] : ¬(₁ ≡ ₀)
₁-is-not-₀[not-an-MLTT-proof] ()
```

Perhaps the following is sufficiently self-explanatory given the above:

```
𝟚-has-decidable-equality : (m n : 𝟚) → (m ≡ n) + (m ≢ n)
𝟚-has-decidable-equality ₀ ₀ = inl (refl ₀)
𝟚-has-decidable-equality ₀ ₁ = inr (≢-sym ₁-is-not-₀)
𝟚-has-decidable-equality ₁ ₀ = inr ₁-is-not-₀
𝟚-has-decidable-equality ₁ ₁ = inl (refl ₁)
```

So we consider four cases. In the first and the last, we have equal things and so we give an answer in the left-hand side of the sum. In the middle two, we give an answer in the right-hand side, where we need functions `₀ ≡ ₁ → 𝟘` and `₁ ≡ ₀ → 𝟘`, which we can take to be `≢-sym ₁-is-not-₀` and `₁-is-not-₀` respectively.

The following is more interesting. We consider the two possible cases for `n`. The first one assumes a hypothetical function `f : ₀ ≡ ₀ → 𝟘`, from which we get `f (refl ₀) : 𝟘`, and then, using `!𝟘`, we get an element of any type we like, which we choose to be `₀ ≡ ₁`, and we are done. Of course, we will never be able to use the function `not-zero-is-one` with such outrageous arguments. The other case `n = ₁` doesn't need to use the hypothesis `f : ₁ ≡ ₀ → 𝟘`, because the desired conclusion holds right away, as it is `₁ ≡ ₁`, which is proved by `refl ₁`. But notice that there is nothing wrong with the hypothesis `f : ₁ ≡ ₀ → 𝟘`. For example, we can use `not-zero-is-one` with `n = ₀` and `f = ₁-is-not-₀`, so that the hypotheses can be fulfilled in the second equation.

```
not-zero-is-one : (n : 𝟚) → n ≢ ₀ → n ≡ ₁
not-zero-is-one ₀ f = !𝟘 (₀ ≡ ₁) (f (refl ₀))
not-zero-is-one ₁ f = refl ₁
```

The following generalizes `₁-is-not-₀`, both in its statement and its proof (so we could have formulated it first and then used it to deduce `₁-is-not-₀`):

```
inl-inr-disjoint-images : {X : 𝓤 ̇ } {Y : 𝓥 ̇ } {x : X} {y : Y} → inl x ≢ inr y
inl-inr-disjoint-images {𝓤} {𝓥} {X} {Y} p = 𝟙-is-not-𝟘 q
 where
  f : X + Y → 𝓤₀ ̇
  f (inl x) = 𝟙
  f (inr y) = 𝟘
  q : 𝟙 ≡ 𝟘
  q = ap f p
```

Table of contents ⇑

Example: formulation of the twin-prime conjecture

We illustrate the above constructs of MLTT to formulate this conjecture.

```
module twin-primes where

 open Arithmetic renaming (_×_ to _*_ ; _+_ to _∔_)
 open ℕ-order

 is-prime : ℕ → 𝓤₀ ̇
 is-prime n = (n ≥ 2) × ((x y : ℕ) → x * y ≡ n → (x ≡ 1) + (x ≡ n))

 twin-prime-conjecture : 𝓤₀ ̇
 twin-prime-conjecture = (n : ℕ) → Σ \(p : ℕ) → (p ≥ n) × is-prime p × is-prime (p ∔ 2)
```

Thus, can we write down not only definitions, constructions, theorems and proofs, but also conjectures. They are just definitions of types. Likewise, the univalence axiom, to be formulated in due course, is a type.

Table of contents ⇑

Operator fixities and precedences

Without the following list of operator precedences and associativities (left or right), this agda file doesn't parse and is rejected by Agda.

```
infix  4  _~_
infixr 4 _,_
infixr 2 _×_
infixr 1 _+_
infixl 5 _∘_
infix  0 _≡_
infixl 2 _·_
infixr 0 _≡⟨_⟩_
infix  1 _∎
infix  3  _⁻¹
```

Table of contents ⇑

Univalent Mathematics in Agda

Our univalent type theory

- Spartan MLTT as above.
- Univalence axiom as below.

But, as discussed above, rather than postulating univalence we will use it as an explicit assumption each time it is needed.

We emphasize that there are univalent type theories in which univalence is a theorem, for example cubical type theory, which has a version available in Agda, called cubical Agda.

Table of contents ⇑

Subsingletons (or propositions or truth values) and sets

A type is a subsingleton (or a proposition or a truth value) if it has at most one element, that is, any two of its elements are equal, or identified.

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

𝟘-is-subsingleton : is-subsingleton 𝟘
𝟘-is-subsingleton x y = !𝟘 (x ≡ y) x

𝟙-is-subsingleton : is-subsingleton 𝟙
𝟙-is-subsingleton = 𝟙-induction (λ x → ∀ y → x ≡ y) (𝟙-induction (λ y → ⋆ ≡ y) (refl ⋆))

𝟙-is-subsingleton' : is-subsingleton 𝟙
𝟙-is-subsingleton' ⋆ ⋆  = refl ⋆
```

The following are more logic-oriented terminologies for the notion.

```
is-prop is-truth-value : 𝓤 ̇ → 𝓤 ̇
is-prop        = is-subsingleton
is-truth-value = is-subsingleton
```

The terminology `is-subsingleton` is more mathematical and avoids the clash with the slogan propositions as types, which is based on the interpretation of mathematical propositions as arbitrary types, rather than just subsingletons.

A type is defined to be a set if there is at most one way for any two of its elements to be equal:

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

At this point, with the definition of these notions, we are entering the realm of univalent mathematics, but not yet needing the univalence axiom.

Table of contents ⇑

Example: the types of magmas and monoids

A magma is a *set* equipped with a binary operation subject to no laws (Bourbaki). We can define the type of magmas in a universe 𝓤 as follows:

```
Magma : (𝓤 : Universe) → 𝓤 ⁺ ̇
Magma 𝓤 = Σ \(X : 𝓤 ̇ ) → is-set X × (X → X → X)
```

The type `Magma 𝓤` collects all magmas in a universe `𝓤`, and lives in the successor universe `𝓤 ⁺`. Thus, this doesn't define what a magma is as a property. It defines the type of magmas. A magma is an element of this type, that is, a triple `(X , i , _·_)` with `X : 𝓤` and `i : is-set X` and `_·_ : X → X → X`.

Given a magma `M = (X , i , _·_)` we denote by `⟨ M ⟩` its underlying set `X` and by `magma-operation M` its multiplication `_·_`:

```
⟨_⟩ : Magma 𝓤 → 𝓤 ̇
⟨ X , i , _·_ ⟩ = X

magma-is-set : (M : Magma 𝓤) → is-set ⟨ M ⟩
magma-is-set (X , i , _·_) = i

magma-operation : (M : Magma 𝓤) → ⟨ M ⟩ → ⟨ M ⟩ → ⟨ M ⟩
magma-operation (X , i , _·_) = _·_
```

The following syntax declaration allows us to write `x ·⟨ M ⟩ y` as an abbreviation of `magma-operation M x y`:

```
syntax magma-operation M x y = x ·⟨ M ⟩ y
```

For some reason, Agda has this kind of definition backwards: the *definiendum* and the *definiens* are swapped with respect to the normal convention of writing what is defined on the left-hand side of the equality sign. In any case, the point is that this time we need such a mechanism because in order to be able to mention arbitrary `x` and `y`, we first need to know their types, which is `⟨ M ⟩` and hence `M` has to occur before `x` and `y` in the definition of `magma-operation`. The syntax declaration circumvents this.

A function of the underlying sets of two magmas is a called a homomorphism when it commutes with the magma operations:

```
is-magma-hom : (M N : Magma 𝓤) → (⟨ M ⟩ → ⟨ N ⟩) → 𝓤 ̇
is-magma-hom M N f = (x y : ⟨ M ⟩) → f (x ·⟨ M ⟩ y) ≡ f x ·⟨ N ⟩ f y

id-is-magma-hom : (M : Magma 𝓤) → is-magma-hom M M (id ⟨ M ⟩)
id-is-magma-hom M = λ (x y : ⟨ M ⟩) → refl (x ·⟨ M ⟩ y)

is-magma-iso : (M N : Magma 𝓤) → (⟨ M ⟩ → ⟨ N ⟩) → 𝓤 ̇
is-magma-iso M N f = is-magma-hom M N f
                   × Σ \(g : ⟨ N ⟩ → ⟨ M ⟩) → is-magma-hom N M g
                                             × (g ∘ f ∼ id ⟨ M ⟩)
                                             × (f ∘ g ∼ id ⟨ N ⟩)

id-is-magma-iso : (M : Magma 𝓤) → is-magma-iso M M (id ⟨ M ⟩)
id-is-magma-iso M = id-is-magma-hom M ,
                    id ⟨ M ⟩ ,
                    id-is-magma-hom M ,
                    refl ,
                    refl
```

Any identification of magmas gives rise to a magma isomorphism by transport:

```
⌜_⌝ : {M N : Magma 𝓤} → M ≡ N → ⟨ M ⟩ → ⟨ N ⟩
⌜ p ⌝ = transport ⟨_⟩ p

⌜⌝-is-iso : {M N : Magma 𝓤} (p : M ≡ N) → is-magma-iso M N (⌜ p ⌝)
⌜⌝-is-iso (refl M) = id-is-magma-iso M
```

The isomorphisms can be collected in a type:

```
_≅ₘ_ : Magma 𝓤 → Magma 𝓤 → 𝓤 ̇
M ≅ₘ N = Σ \(f : ⟨ M ⟩ → ⟨ N ⟩) → is-magma-iso M N f
```

The following function will be a bijection in the presence of univalence, so that the identifications of magmas are in one-to-one correspondence with the magma isomorphisms:

```
magma-Id-to-iso : {M N : Magma 𝓤} → M ≡ N → M ≅ₘ N
magma-Id-to-iso p = (⌜ p ⌝ , ⌜⌝-is-iso p )
```

If we omit the set-hood condition in the definition of the type of magmas, we get the type of what we could call ∞-magmas (then the type of magmas could be called `0-Magma`).

```
∞-Magma : (𝓤 : Universe) → 𝓤 ⁺ ̇
∞-Magma 𝓤 = Σ \(X : 𝓤 ̇ ) → X → X → X
```

A monoid is a set equipped with an associative binary operation and with a two-sided neutral element, and so we get the type of monoids as follows.

We first define the three laws:

```
left-neutral : {X : 𝓤 ̇ } → X → (X → X → X) → 𝓤 ̇
left-neutral e _·_ = ∀ x → e · x ≡ x

right-neutral : {X : 𝓤 ̇ } → X → (X → X → X) → 𝓤 ̇
right-neutral e _·_ = ∀ x → x · e ≡ x

associative : {X : 𝓤 ̇ } → (X → X → X) → 𝓤 ̇
associative _·_ = ∀ x y z → (x · y) · z ≡ x · (y · z)
```

Then a monoid is a set equipped with such `e` and `_·_` satisfying these three laws:

```
Monoid : (𝓤 : Universe) → 𝓤 ⁺ ̇
Monoid 𝓤 = Σ \(X : 𝓤 ̇ ) → is-set X
                        × Σ \(_·_ : X → X → X)
                        → Σ \(e : X)
                        → left-neutral e _·_
                        × right-neutral e _·_
                        × associative _·_
```

Remark. People are more likely to use records in Agda rather than iterated `Σ`s as above (recall that we defined `Σ` using a record). This is fine, because records amount to iterated `Σ` types (recall that also `_×_` is a `Σ` type, by definition). Here, however, we are being deliberately spartan. Once we have defined our Agda notation for MLTT, we want to stick to it. This is for teaching purposes (of MLTT, encoded in Agda, not of Agda itself in its full glory).

We could drop the `is-set X` condition, but then we wouldn't get ∞-monoids in any reasonable sense. We would instead get "wild ∞-monoids" or "incoherent ∞-monoids". The reason is that in monoids (with sets as carriers) the neutrality and associativity equations can hold in at most one way, by definition of set. But if we drop the set-hood requirement, then the equations can hold in multiple ways. And then one is forced to consider equations between the identifications (all the way up in the ∞-case), which is what "coherence data" means. The wildness or incoherence amounts to the absence of such data.

Similarly to the situation with magmas, identifications of monoids are in bijection with monoid isomorphisms, assuming univalence. For this to be the case, it is absolutely necessary that the carrier of a monoid is a set rather than an arbitrary type, for otherwise the monoid equations can hold in many possible ways, and we would need to consider a notion of monoid isomorphism that in addition to preserving the neutral element and the multiplication, preserves the equations, and the preservations of the equations, and the preservation of the preservations of the equations, *ad infinitum*.

Exercise. Define the type of groups (with sets as carriers).

Exercise. Write down the various types of categories defined in the HoTT book in Agda.

Exercise. Try to define a type of topological spaces.

Table of contents ⇑

The identity type in univalent mathematics

We can view a type `X` as a sort of category with hom-types rather than hom-sets, with composition defined as follows (and written in so-called diagramatic order rather than the usual backwards order like we wrote function composition).

If we wanted to prove the following without pattern matching, this time we would need the dependent version `J` of induction on `_≡_`.

Exercise. Try to do this with `J` and with `H`.

We have that `refl` provides a neutral element for composition of identifications:

```
refl-left : {X : 𝓤 ̇ } {x y : X} {p : x ≡ y} → refl x · p ≡ p
refl-left {𝓤} {X} {x} {x} {refl x} = refl (refl x)

refl-right : {X : 𝓤 ̇ } {x y : X} {p : x ≡ y} → p · refl y ≡ p
refl-right {𝓤} {X} {x} {y} {p} = refl p
```

And composition is associative:

```
·assoc : {X : 𝓤 ̇ } {x y z t : X} (p : x ≡ y) (q : y ≡ z) (r : z ≡ t)
       → (p · q) · r ≡ p · (q · r)
·assoc p q (refl z) = refl (p · q)
```

But all arrows, the identifications, are invertible:

```
⁻¹-left· : {X : 𝓤 ̇ } {x y : X} (p : x ≡ y)
         → p ⁻¹ · p ≡ refl y
⁻¹-left· (refl x) = refl (refl x)

⁻¹-right· : {X : 𝓤 ̇ } {x y : X} (p : x ≡ y)
          → p · p ⁻¹ ≡ refl x
⁻¹-right· (refl x) = refl (refl x)
```

A category in which all arrows are invertible is called a groupoid. The above is the basis for the Hofmann–Streicher groupoid model of type theory.

But we actually get higher groupoids, because given identifications

```
p q : x ≡ y
```

we can consider the identity type `p ≡ q`, and given

```
u v : p ≡ q
```

we can consider the type `u ≡ v`, and so on. See [van den Berg and Garner] and [Lumsdaine].

For some types, such as the natural numbers, we can prove that this process trivializes after the first step, because the type `x ≡ y` has at most one element. Such types are the sets as defined above.

Voevodsky defined the notion of *hlevel* to measure how long it takes for the process to trivialize.

Here are some more constructions with identifications:

```
⁻¹-involutive : {X : 𝓤 ̇ } {x y : X} (p : x ≡ y)
              → (p ⁻¹)⁻¹ ≡ p
⁻¹-involutive (refl x) = refl (refl x)
```

The application operation on identifications is functorial, in the sense that is preserves the neutral element and commutes with composition.:

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

ap-∙ : {X : 𝓤 ̇ } {Y : 𝓥 ̇ } (f : X → Y) {x y z : X} (p : x ≡ y) (q : y ≡ z)
     → ap f (p ∙ q) ≡ ap f p ∙ ap f q
ap-∙ f p (refl y) = refl (ap f p)
```

This is functoriality in the second argument. We also have functoriality in the first argument, in the following sense:

```
ap-id : {X : 𝓤 ̇ } {x y : X} (p : x ≡ y)
      → ap id p ≡ p
ap-id (refl x) = refl (refl x)

ap-∘ : {X : 𝓤 ̇ } {Y : 𝓥 ̇ } {Z : 𝓦 ̇ }
       (f : X → Y) (g : Y → Z) {x y : X} (p : x ≡ y)
     → ap (g ∘ f) p ≡ (ap g ∘ ap f) p
ap-∘ f g (refl x) = refl (refl (g (f x)))
```

Transport is also functorial with respect to identification composition and function composition. By construction, it maps the neutral element to the identity function. The apparent contravariance takes place because we have defined function composition in the usual order, but identification composition in diagramatic order (as is customary in each case).

```
transport∙ : {X : 𝓤 ̇ } (A : X → 𝓥 ̇ ) {x y z : X} (p : x ≡ y) (q : y ≡ z)
           → transport A (p ∙ q) ≡ transport A q ∘ transport A p
transport∙ A p (refl y) = refl (transport A p)
```

Functions of a type into a universe can be considered as generalized presheaves, which we could perhaps call ∞-presheaves. Their morphisms are natural transformations:

```
Nat : {X : 𝓤 ̇ } → (X → 𝓥 ̇ ) → (X → 𝓦 ̇ ) → 𝓤 ⊔ 𝓥 ⊔ 𝓦 ̇
Nat A B = (x : domain A) → A x → B x
```

We don’t need to specify the naturality condition, because it is automatic:

```
Nats-are-natural : {X : 𝓤 ̇ } (A : X → 𝓥 ̇ ) (B : X → 𝓦 ̇ ) (τ : Nat A B)
                 → {x y : X} (p : x ≡ y) → τ y ∘ transport A p ≡ transport B p ∘ τ x
Nats-are-natural A B τ (refl x) = refl (τ x)
```

We will have the opportunity to use the following construction a number of times:

```
NatΣ : {X : 𝓤 ̇ } {A : X → 𝓥 ̇ } {B : X → 𝓦 ̇ } → Nat A B → Σ A → Σ B
NatΣ τ (x , a) = (x , τ x a)

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

Table of contents ⇑

Identifications that depend on identifications

If we have an identification `p : A ≡ B` of two types `A` and `B`, and elements `a : A` and `b : B`, we cannot ask directly whether `a ≡ b`, because although the types are identified by `p`, they are not necessarily the same, in the sense of definitional equality. This is not merely a syntactical restriction of our formal system, but instead a fundamental fact that reflects the philosophy of univalent mathematics. For instance, consider the type

```
data Color : 𝓤₀ ̇  where
 Black White : Color
```

With univalence, we will have that `Color ≡ 𝟚` where `𝟚` is the two-point type `𝟙 + 𝟙` with points `₀` and `₁`. But there will be two identifications `p₀ p₁ : Color ≡ 𝟚`, one that identifies `Black` with `₀` and `White` with `₁`, and another one that identifies `Black` with `₁` and `White` with `₀`. There is no preferred coding of binary colors as bits. And, precisely because of that, even if univalence does give inhabitants of the type `Colour ≡ 𝟚`, it doesn't make sense to ask whether `Black ≡ ₀` holds without specifying one of the possible inhabitants `p₀` and `p₁`.

What we will have is that `transport id p₀` and `transport id p₁` are the two possible bijections `Color → 𝟚` that identify colors with bits. So, it is not enough to have `Color ≡ 𝟚` to be able to compare a color `c : Color` with a bit `b : 𝟚`. We need to specify which identification `p : Color ≡ 𝟚` we want to consider for the comparison. The same considerations apply when we consider identifications `p : 𝟚 ≡ 𝟚`.

So the meaningful comparison in the more general situation is

```
transport id p a ≡ b
```

for a specific

```
p : A ≡ B,
```

where `id` is the identity function of the universe where the types `A` and `B` live, and hence

```
transport id : A ≡ B → (A → B)
```

is the function that transforms identifications into functions, which has already occurred above.

More generally, we want to consider the situation in which we replace the identity function `id` of the universe where `A` and `B` live by an arbitrary type family, which is what we do now.

If we have a type

```
X : 𝓤 ̇ ,
```

and a type family

```
A : X → 𝓥 ̇
```

and points

```
x y : X
```

and an identification

```
p : x ≡ y,
```

then we get the identification

```
ap A p : A x ≡ A y.
```

However, if we have

```
a : A x,

b : A y,
```

we again cannot write down the identity type

~~a ≡ b~~.

This is again a non-sensical mathematical statement in univalent foundations, because the types `A x` and `A y` are not the same, but only identified, and in general there can be many identifications, not just `ap A p`, and so any identification between elements of `A x` and `A y` has to be with respect to a specific identification, as in the above particular case.

So we define a notion of dependent equality between elements `a : A x` and `b : A y`, where the dependency is on an given identification `p : x ≡ y`. We write

```
dId A p a b
```

for the type of "identifications of `a` and `b` dependent on the identification `p : x ≡ y` over the family `A`".

We can define this by

```
dId A (refl x) a b = (a ≡ b).
```

But, because

```
transport A (refl x) a = a,
```

by definition, we may as well define `dId` as follows in Agda:

```
dId : {X : 𝓤 ̇ } (A : X → 𝓥 ̇ ) {x y : X} (p : x ≡ y) → A x → A y → 𝓥 ̇
dId A p a b = transport A p a ≡ b
```

We now define special syntax in Agda to be able to write this in the more symmetrical way

```
a ≡[ p / A ] b.
```

This stands for equality of `a` and `b` dependent on `p` over `A`. Because we have chosen to say *over*, we may as well use the symbol `/` to express this. We define this quaternary mix-fix operator `_≡[_/_]_` with a syntax declaration as follows in Agda.

```
syntax dId A p a b = a ≡[ p / A ] b
```

We have designed things so that, by construction, we get the following:

```
≡[]-on-refl-is-≡ : {X : 𝓤 ̇ } (A : X → 𝓥 ̇ ) {x : X} (a b : A x)
                 → (a ≡[ refl x / A ] b) ≡ (a ≡ b)
≡[]-on-refl-is-≡ A {x} a b = refl (a ≡ b)
```

Notice the perhaps unfamiliar nested use of equality: the identity type `transport A (refl x) a ≡ b` is equal to the identity type `a ≡ b`. The proof is the reflexivity identification of the type `a ≡ b`. We rewrite the above making the implicit arguments of `refl` explicit so that it becomes apparent that we are using the identity type former of a type that happens to be a universe.

```
≡[]-on-refl-is-≡' : {X : 𝓤 ̇ } (A : X → 𝓥 ̇ ) {x : X} (a b : A x)
                  → (a ≡[ refl x / A ] b) ≡ (a ≡ b)

≡[]-on-refl-is-≡' {𝓤} {𝓥} {X} A {x} a b = refl {𝓥 ⁺} {𝓥 ̇ } (a ≡ b)
```

This says that we are taking the reflexivity proof of the equality type of the universe $\mathcal{V}$, which lives in the next universe $\mathcal{V}^+$, for the element `a ≡ b` (which is a type) of $\mathcal{V}$.

Table of contents ⇑

Equality in Σ types

With the above notion of dependent equality, we can characterize equality in Σ types as follows.

```
to-Σ-≡ : {X : 𝓤 ̇ } {A : X → 𝓥 ̇ } {σ τ : Σ A}
       → (Σ \(p : pr₁ σ ≡ pr₁ τ) → pr₂ σ ≡[ p / A ] pr₂ τ)
       → σ ≡ τ
to-Σ-≡ (refl x , refl a) = refl (x , a)

from-Σ-≡ : {X : 𝓤 ̇ } {A : X → 𝓥 ̇ } {σ τ : Σ A}
         → σ ≡ τ
         → Σ \(p : pr₁ σ ≡ pr₁ τ) → pr₂ σ ≡[ p / A ] pr₂ τ
from-Σ-≡ (refl (x , a)) = (refl x , refl a)
```

The above gives

```
(σ ≡ τ) ⇔ Σ \(p : pr₁ σ ≡ pr₁ τ) → pr₂ σ ≡[ p / A ] pr₂ τ.
```

But this is a very weak statement when the left- and right-hand identity types may have multiple elements, which is precisely the point of univalent mathematics.

What we want is the lhs and the rhs to be isomorphic, or more precisely, equivalent in the sense of Voevodsky.

Once we have defined this notion `_≃_` of type equivalence, this characterization will become an equivalence

```
(σ ≡ τ) ≃ Σ \(p : pr₁ σ ≡ pr₁ τ) → pr₂ σ ≡[ p / A ] pr₂ τ.
```

But even this is not sufficiently precise, because in general there are many equivalences between two types. For example, there are precisely two equivalences

```
𝟙 + 𝟙 ≃ 𝟙 + 𝟙,
```

namely the identity function and the function that flips left and right. What we want to say is that a *specific map* is an equivalence. In our case, it is the function `from-Σ-≡` defined above.

Voevodsky came up with a definition of a type "`f` is an equivalence" which is always a subsingleton: a given function `f` can be an equivalence in at most one way. But we first discuss *hlevels*.

Table of contents ⇑

Voevodsky's notion of hlevel

Voevodsky's hlevels `0,1,2,3,...` are shifted by `2` with respect to the `n`-groupoid numbering convention, and correspond to `-2`-groupoids (singletons), `-1`-groupoids (subsingletons), `0`-groupoids (sets),…

First Voevodsky defined a notion of *contractible type*, which we refer to here as *singleton type*.

```
is-singleton : 𝓤 ̇ → 𝓤 ̇
is-singleton X = Σ \(c : X) → (x : X) → c ≡ x

𝟙-is-singleton : is-singleton 𝟙
𝟙-is-singleton = ⋆ , 𝟙-induction (λ x → ⋆ ≡ x) (refl ⋆)
```

Then the hlevel relation is defined by induction on ℕ, with the induction step working with the identity types of the elements of the type in question:

```
_is-of-hlevel_ : 𝓤 ̇ → ℕ → 𝓤 ̇
X is-of-hlevel 0        = is-singleton X
X is-of-hlevel (succ n) = (x x' : X) → ((x ≡ x') is-of-hlevel n)
```

It is often convenient in practice to have equivalent formulations of the levels 1 (as subsingletons) and 2 (as sets), which we will develop soon.

When working with singleton types, it will be convenient to have distinghished names for the two projections:

```
center : (X : 𝓤 ̇ ) → is-singleton X → X
center X (c , φ) = c

centrality : (X : 𝓤 ̇ ) (i : is-singleton X) (x : X) → center X i ≡ x
centrality X (c , φ) = φ

singletons-are-subsingletons : (X : 𝓤 ̇ ) → is-singleton X → is-subsingleton X
singletons-are-subsingletons X (c , φ) x y = x ≡⟨ (φ x)⁻¹ ⟩
                                             c ≡⟨ φ y ⟩
                                             y ∎

pointed-subsingletons-are-singletons : (X : 𝓤 ̇ ) → X → is-subsingleton X → is-singleton X
pointed-subsingletons-are-singletons X x s = (x , s x)
```

Table of contents ⇑

The univalent principle of excluded middle

Under excluded middle, the only two subsingletons, up to equivalence, are 𝟘 and 𝟙. In fact, excluded middle in univalent mathematics says precisely that.

```
EM EM' : ∀ 𝓤 → 𝓤 ⁺ ̇
EM  𝓤 = (X : 𝓤 ̇ ) → is-subsingleton X → X + ¬ X
EM' 𝓤 = (X : 𝓤 ̇ ) → is-subsingleton X → is-singleton X + is-empty X
```

Notice that the above don't assert excluded middle, but instead say what excluded middle is (like when we said what the twin-prime conjecture is), in two logically equivalent versions:

```
EM-gives-EM' : EM 𝓤 → EM' 𝓤
EM-gives-EM' em X s = γ (em X s)
 where
  γ : X + ¬ X → is-singleton X + is-empty X
  γ (inl x) = inl (pointed-subsingletons-are-singletons X x s)
  γ (inr x) = inr x

EM'-gives-EM : EM' 𝓤 → EM 𝓤
EM'-gives-EM em' X s = γ (em' X s)
 where
  γ : is-singleton X + is-empty X → X + ¬ X
  γ (inl i) = inl (center X i)
  γ (inr x) = inr x
```

We will not assume or deny excluded middle, which is an independent statement (it can't be proved or disproved).

Table of contents ⇑

Hedberg's Theorem

To characterize sets as the types of hlevel 2, we first need to show that subsingletons are sets, and this is not easy. We use an argument due to Hedberg. This argument also shows that Voevodsky's hlevels are upper closed.

We choose to present an alternative formulation of Hedberg's Theorem, but we stress that the method of proof is essentially the same.

We first define a notion of constant map:

```
wconstant : {X : 𝓤 ̇ } {Y : 𝓥 ̇ } → (f : X → Y) → 𝓤 ⊔ 𝓥 ̇
wconstant f = (x x' : domain f) → f x ≡ f x'
```

The prefix "`w`" officially stands for "weakly". Perhaps *incoherently constant* or *wildly constant* would be better terminologies, with *coherence* understood in the ∞-categorical sense. We prefer to stick to *wildly* rather than *weakly*, and luckily both start with the letter "`w`". The following is also probably not very good terminology, but we haven't come up with a better one yet.

```
collapsible : 𝓤 ̇ → 𝓤 ̇
collapsible X = Σ \(f : X → X) → wconstant f

collapser : (X : 𝓤 ̇ ) → collapsible X → X → X
collapser X (f , w) = f

collapser-wconstancy : (X : 𝓤 ̇ ) (c : collapsible X) → wconstant (collapser X c)
collapser-wconstancy X (f , w) = w
```

The point is that a type is a set if and only if its identity types all have `wconstant` endomaps:

```
Hedberg : {X : 𝓤 ̇ } (x : X)
        → ((y : X) → collapsible (x ≡ y))
        → (y : X) → is-subsingleton (x ≡ y)
Hedberg {𝓤} {X} x c y p q =
 p                        ≡⟨ a y p ⟩
 f x (refl x)⁻¹ · f y p   ≡⟨ ap (λ - → (f x (refl x))⁻¹ · -) (κ y p q) ⟩
 f x (refl x)⁻¹ · f y q   ≡⟨ (a y q)⁻¹ ⟩
 q                        ∎
 where
  f : (y : X) → x ≡ y → x ≡ y
  f y = collapser (x ≡ y) (c y)
  κ : (y : X) (p q : x ≡ y) → f y p ≡ f y q
  κ y = collapser-wconstancy (x ≡ y) (c y)
  a : (y : X) (p : x ≡ y) → p ≡ (f x (refl x))⁻¹ · f y p
  a x (refl x) = (⁻¹-left· (f x (refl x)))⁻¹
```

Table of contents ⇑

A characterization of sets

The following is immediate from the definitions:

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

sets-are-≡-collapsible : (X : 𝓤 ̇ ) → is-set X → ≡-collapsible X
sets-are-≡-collapsible X s x y = (f , κ)
 where
  f : x ≡ y → x ≡ y
  f p = p
  κ : (p q : x ≡ y) → f p ≡ f q
  κ p q = s x y p q
```

And the converse is the content of Hedberg's Theorem.

```
≡-collapsibles-are-sets : (X : 𝓤 ̇ ) → ≡-collapsible X → is-set X
≡-collapsibles-are-sets X c x = Hedberg x (λ y → collapser (x ≡ y) (c x y) ,
                                                 collapser-wconstancy (x ≡ y) (c x y))
```

Table of contents ⇑

Subsingletons are sets

In the following definition of the auxiliary function `f`, we ignore the argument `p`, using the fact that `X` is a subsingleton instead, to get a `wconstant` function:

```
subsingletons-are-≡-collapsible : (X : 𝓤 ̇ ) → is-subsingleton X → ≡-collapsible X
subsingletons-are-≡-collapsible X s x y = (f , κ)
 where
  f : x ≡ y → x ≡ y
  f p = s x y
  κ : (p q : x ≡ y) → f p ≡ f q
  κ p q = refl (s x y)
```

And the corollary is that subsingleton types are sets.

```
subsingletons-are-sets : (X : 𝓤 ̇ ) → is-subsingleton X → is-set X
subsingletons-are-sets X s = ≡-collapsibles-are-sets X (subsingletons-are-≡-collapsible X s)
```

Table of contents ⇑

The types of hlevel 1 are the subsingletons

Then with the above we get our desired characterization of the types of hlevel `1` as an immediate consequence:

```
subsingletons-are-of-hlevel-1 : (X : 𝓤 ̇ ) → is-subsingleton X → X is-of-hlevel 1
subsingletons-are-of-hlevel-1 X = g
 where
  g : ((x y : X) → x ≡ y) → (x y : X) → is-singleton (x ≡ y)
  g t x y = t x y , subsingletons-are-sets X t x y (t x y)

types-of-hlevel-1-are-subsingletons : (X : 𝓤 ̇ ) → X is-of-hlevel 1 → is-subsingleton X
types-of-hlevel-1-are-subsingletons X = f
 where
  f : ((x y : X) → is-singleton (x ≡ y)) → (x y : X) → x ≡ y
  f s x y = center (x ≡ y) (s x y)
```

This is an "if and only if" characterization, but, under univalence, it becomes an equality because the types under consideration are subsingletons.

Table of contents ⇑

The types of hlevel 2 are the sets

The same comments as for the previous section apply.

```
sets-are-of-hlevel-2 : (X : 𝓤 ̇ ) → is-set X → X is-of-hlevel 2
sets-are-of-hlevel-2 X = g
 where
  g : ((x y : X) → is-subsingleton (x ≡ y)) → (x y : X) → (x ≡ y) is-of-hlevel 1
  g t x y = subsingletons-are-of-hlevel-1 (x ≡ y) (t x y)

types-of-hlevel-2-are-sets : (X : 𝓤 ̇ ) → X is-of-hlevel 2 → is-set X
types-of-hlevel-2-are-sets X = f
 where
  f : ((x y : X) → (x ≡ y) is-of-hlevel 1) → (x y : X) → is-subsingleton (x ≡ y)
  f s x y = types-of-hlevel-1-are-subsingletons (x ≡ y) (s x y)
```

Table of contents ⇑

The hlevels are upper closed

A singleton is a subsingleton, a subsingleton is a set, … , a type of hlevel `n` is of hlevel `n+1` too, …

Again we can conclude this almost immediately from the above development:

```
hlevel-upper : (X : 𝓤 ̇ ) (n : ℕ) → X is-of-hlevel n → X is-of-hlevel (succ n)
hlevel-upper X zero = γ
```

```
 where
  γ : is-singleton X → (x y : X) → is-singleton (x ≡ y)
  γ h x y = p , subsingletons-are-sets X k x y p
   where
    k : is-subsingleton X
    k = singletons-are-subsingletons X h
    p : x ≡ y
    p = k x y
hlevel-upper X (succ n) = λ h x y → hlevel-upper (x ≡ y) n (h x y)
```

To be consistent with the above terminology, we have to stipulate that all types have hlevel ∞. We don't need a definition for this vacuous notion. But what may happen (and it does with univalence) is that there are types which don't have any finite hlevel. We can say that such types then have minimal hlevel ∞.

Exercise. Formulate and prove the following. The type 𝟙 has minimal hlevel 0. The type 𝟘 has minimal hlevel 1, the type ℕ has minimal hlevel 2. More ambitiously, when you have univalence at your disposal, show that the type of monoids has minimal hlevel 3.

Table of contents ⇑

Example: ℕ is a set

We first prove the remaining Peano axioms.

```
positive-not-zero : (x : ℕ) → succ x ≢ 0
positive-not-zero x p = 𝟙-is-not-𝟘 (g p)
 where
  f : ℕ → 𝓤₀ ̇
  f 0        = 𝟘
  f (succ x) = 𝟙
  g : succ x ≡ 0 → 𝟙 ≡ 𝟘
  g = ap f
```

To show that the successor function is left cancellable, we can use the following predecessor function.

```
pred : ℕ → ℕ
pred 0 = 0
pred (succ n) = n

succ-lc : {x y : ℕ} → succ x ≡ succ y → x ≡ y
succ-lc = ap pred
```

With this we have proved all the Peano axioms.

Without assuming the principle of excluded middle, we can prove that ℕ has decidable equality in the following sense:

```
ℕ-has-decidable-equality : (x y : ℕ) → (x ≡ y) + (x ≢ y)
ℕ-has-decidable-equality 0 0               = inl (refl 0)
ℕ-has-decidable-equality 0 (succ y)        = inr (≢-sym (positive-not-zero y))
ℕ-has-decidable-equality (succ x) 0        = inr (positive-not-zero x)
ℕ-has-decidable-equality (succ x) (succ y) = f (ℕ-has-decidable-equality x y)
 where
  f : (x ≡ y) + x ≢ y → (succ x ≡ succ y) + (succ x ≢ succ y)
  f (inl p) = inl (ap succ p)
  f (inr k) = inr (λ (s : succ x ≡ succ y) → k (succ-lc s))
```

Exercise. Students should do this kind of thing at least once in their academic life: rewrite the above proof of the decidability of equality of ℕ to use the ℕ-induction principle J (or its alternative H) instead of pattern matching and recursion, to understand by themselves that this can be done.

And using the decidability of equality we can define a `wconstant` function `x ≡ y → x ≡ y` and hence conclude that `ℕ` is a set. This argument is due to Hedberg.

```
ℕ-is-set : is-set ℕ
ℕ-is-set = ≡-collapsibles-are-sets ℕ ℕ-≡-collapsible
 where
  ℕ-≡-collapsible : ≡-collapsible ℕ
  ℕ-≡-collapsible x y = f (ℕ-has-decidable-equality x y) ,
                        κ (ℕ-has-decidable-equality x y)
   where
    f : (x ≡ y) + ¬(x ≡ y) → x ≡ y → x ≡ y
    f (inl p) q = p
    f (inr g) q = !𝟘 (x ≡ y) (g q)
    κ : (d : (x ≡ y) + ¬(x ≡ y)) → wconstant (f d)
    κ (inl p) q r = refl p
    κ (inr g) q r = !𝟘 (f (inr g) q ≡ f (inr g) r) (g q)
```

Exercise. Hedberg proved this for any type with decidable equality. Generalize the above to account for this.

Exercise. Prove that the types of magmas, monoids and groups have hlevel `3` (they are `1`-groupoids) but not hlevel `2` (they are not sets). Prove that this is their minimal hlevel. Can you do this with what we have learned so far?

Table of contents ⇑

Retracts

We use retracts as a mathematical technique to transfer properties between types. For instance, retracts of singletons are singletons. Showing that a particular type `X` is a singleton may be rather difficult to do directly by applying the definition of singleton and the definition of the particular type, but it may be easy to show that `X` is a retract of `Y` for a type `Y` that is already known to be a singleton. In these notes, a major application will be to get a simple proof of the known fact that invertible maps are equivalences in the sense of Voevodsky.

A *section* of a function is simply a right inverse, by definition:

```
has-section : {X : 𝓤 ̇ } {Y : 𝓥 ̇ } → (X → Y) → 𝓤 ⊔ 𝓥 ̇
has-section r = Σ \(s : codomain r → domain r) → r ∘ s ∼ id
```

Notice that `has-section r` is the type of all sections `(s , η)` of `r`, which may well be empty. So a point of this type is a designated section `s` of `r`, together with the datum `η`. Unless the domain of `r` is a set, this datum is not property, and we may well have an element `(s , η')` of the type `has-section r` with `η'` distinct from `η` for the same `s`.

We say that *`X` is a retract of `Y`*, written `X ◁ Y`, if we have a function `Y → X` which has a section:

```
_◁_ : 𝓤 ̇ → 𝓥 ̇ → 𝓤 ⊔ 𝓥 ̇
X ◁ Y = Σ \(r : Y → X) → has-section r
```

This type actually collects *all* the ways in which the type `X` can be a retract of the type `Y`, and so is data or structure on `X` and `Y`, rather than a property of them.

A function that has a section is called a retraction. We use this terminology, ambiguously, also for the function that projects out the retraction:

```
retraction : {X : 𝓤 ̇ } {Y : 𝓥 ̇ } → X ◁ Y → Y → X
retraction (r , s , η) = r

section : {X : 𝓤 ̇ } {Y : 𝓥 ̇ } → X ◁ Y → X → Y
section (r , s , η) = s
```

```
retract-equation : {X : 𝓤 ̇ } {Y : 𝓥 ̇ } (ρ : X ◁ Y) → retraction ρ ∘ section ρ ∼ id X
retract-equation (r , s , η) = η
```

We have an identity retraction:

```
◁-refl : (X : 𝓤 ̇ ) → X ◁ X
◁-refl X = id X , id X , refl
```

Exercise. The identity retraction is by no means the only retraction of a type onto itself in general, of course. Prove that we have (that is, produce an element of the type) `ℕ ◁ ℕ` with the function `pred : ℕ → ℕ` defined above as the retraction, to exercise your Agda skills. Can you produce more inhabitants of this type?

We can define the composition of two retractions as follows:

```
_◁∘_ : {X : 𝓤 ̇ } {Y : 𝓥 ̇ } {Z : 𝓦 ̇ } → X ◁ Y → Y ◁ Z → X ◁ Z

(r , s , η) ◁∘ (r' , s' , η') = (r ∘ r' , s' ∘ s , η'')
 where
  η'' = λ x → r (r' (s' (s x))) ≡⟨ ap r (η' (s x)) ⟩
              r (s x)           ≡⟨ η x ⟩
              x                 ∎
```

We also define composition with an implicit argument made explicit:

```
_◁⟨_⟩_ : (X : 𝓤 ̇ ) {Y : 𝓥 ̇ } {Z : 𝓦 ̇ } → X ◁ Y → Y ◁ Z → X ◁ Z
X ◁⟨ ρ ⟩ σ = ρ ◁∘ σ
```

And we introduce the following postfix notation for the identity retraction:

```
_◀ : (X : 𝓤 ̇ ) → X ◁ X
X ◀ = ◁-refl X
```

These last two definitions are for notational convenience. See below for examples of their use.

We conclude this section with some facts about retracts of `Σ` types. The following are technical tools for dealing with equivalences in the sense of Voevosky in comparison with invertible maps.

A pointwise retraction gives a retraction of the total spaces:

```
Σ-retract : (X : 𝓤 ̇ ) (A : X → 𝓥 ̇ ) (B : X → 𝓦 ̇ )
          → ((x : X) → (A x) ◁ (B x)) → Σ A ◁ Σ B
Σ-retract X A B ρ = NatΣ r , NatΣ s , η'
 where
  r : (x : X) → B x → A x
  r x = retraction (ρ x)
  s : (x : X) → A x → B x
  s x = section (ρ x)
  η : (x : X) (a : A x) → r x (s x a) ≡ a
  η x = retract-equation (ρ x)
  η' : (σ : Σ A) → NatΣ r (NatΣ s σ) ≡ σ
  η' (x , a) = x , r x (s x a) ≡⟨ ap (λ - → x , -) (η x a) ⟩
               x , a          ∎
```

We have that `transport A (p ⁻¹)` is a two-sided inverse of `transport A p` using the functoriality of `transport A`, or directly by induction on `p`:

```
transport-is-retraction : {X : 𝓤 ̇ } (A : X → 𝓥 ̇ ) {x y : X} (p : x ≡ y)
                        → transport A p ∘ transport A (p ⁻¹) ∼ id (A y)
transport-is-retraction A (refl x) = refl

transport-is-section    : {X : 𝓤 ̇ } (A : X → 𝓥 ̇ ) {x y : X} (p : x ≡ y)
                        → transport A (p ⁻¹) ∘ transport A p ∼ id (A x)
transport-is-section A (refl x) = refl
```

Using this, we can reindex retracts of `Σ` types as follows:

```
Σ-reindex-retraction : {X : 𝓤 ̇ } {Y : 𝓥 ̇ } {A : X → 𝓦 ̇ } (r : Y → X)
                     → has-section r
                     → (Σ \(x : X) → A x) ◁ (Σ \(y : Y) → A (r y))
Σ-reindex-retraction {𝓤} {𝓥} {𝓦} {X} {Y} {A} r (s , η) = γ , φ , γφ
 where
  γ : Σ (A ∘ r) → Σ A
  γ (y , a) = (r y , a)
  φ : Σ A → Σ (A ∘ r)
  φ (x , a) = (s x , transport A ((η x)⁻¹) a)
  γφ : (σ : Σ A) → γ (φ σ) ≡ σ
  γφ (x , a) = to-Σ-≡ (η x , transport-is-retraction A (η x) a)
```

We have defined the property of a type being a singleton. The singleton type `Σ \(y : X) → x ≡ y` induced by a point `x : X` of a type `X` is denoted by `singleton-type x`. The terminology is justified by the fact that it is indeed a singleton in the sense defined above.

```
singleton-type : {X : 𝓤 ̇ } → X → 𝓤 ̇
singleton-type x = Σ \y → y ≡ x

singleton-type-center : {X : 𝓤 ̇ } (x : X) → singleton-type x
singleton-type-center x = (x , refl x)

singleton-type-centered : {X : 𝓤 ̇ } (x y : X) (p : y ≡ x)
                        → singleton-type-center x ≡ (y , p)
singleton-type-centered x x (refl x) = refl (singleton-type-center x)

singleton-types-are-singletons : (X : 𝓤 ̇ ) (x : X)
                               → is-singleton (singleton-type x)
singleton-types-are-singletons X x = singleton-type-center x , φ
 where
  φ : (σ : singleton-type x) → singleton-type-center x ≡ σ
  φ (y , p) = singleton-type-centered x y p
```

The following gives a technique for showing that some types are singletons:

```
retract-of-singleton : {X : 𝓤 ̇ } {Y : 𝓥 ̇ }
                     → Y ◁ X → is-singleton X → is-singleton Y
retract-of-singleton (r , s , η) (c , φ) = r c , γ
 where
  γ : (y : codomain r) → r c ≡ y
  γ y = r c     ≡⟨ ap r (φ (s y)) ⟩
        r (s y) ≡⟨ η y ⟩
        y       ∎
```

Table of contents ⇑

Voevodsky's notion of type equivalence

The main notions of univalent mathematics conceived by Voevodsky, with formulations in MLTT, are those of singleton type (or contractible type), hlevel (including the notions of subsingleton and set), and of type equivalence, which we define now. For that purpose, we need to define the notion of fiber of a function first.

But we begin with a discussion of the notion of *invertible function*, whose only difference with the notion of equivalence is that it is data rather than property:

```
invertible : {X : 𝓤 ̇ } {Y : 𝓥 ̇ } → (X → Y) → 𝓤 ⊔ 𝓥 ̇
invertible f = Σ \g → (g ∘ f ∼ id) × (f ∘ g ∼ id)
```

The situation is that we will have a logical equivalence between "data establishing invertibility of a given function" and "the property of the function being an equivalence". Mathematically, what happens is that the type "`f` is an equivalence" is a retract of the type "`f` is invertible". This retraction property is not easy to show, and there are many approaches. We discuss an approach we came up with while developing these lecture notes, which is intended to be relatively simple and direct, but the reader should consult other approaches, such as that of the HoTT book, which has a well-established categorical pedigree.

The problem with the notion of invertibility of `f` is that, while we can prove that the inverse `g` is unique when it exists, we cannot in general prove that the identification data `g ∘ f ∼ id` and `f ∘ g ∼ id` are also unique, and, indeed, they are not in general.

The following is Voevodsky's proposed formulation of the notion of equivalence in MLTT, which relies on the concept of `fiber`:

```
fiber : {X : 𝓤 ̇ } {Y : 𝓥 ̇ } (f : X → Y) → Y → 𝓤 ⊔ 𝓥 ̇
fiber f y = Σ \(x : domain f) → f x ≡ y

fiber-point : {X : 𝓤 ̇ } {Y : 𝓥 ̇ } {f : X → Y} {y : Y}
            → fiber f y → X
fiber-point (x , p) = x

fiber-identification : {X : 𝓤 ̇ } {Y : 𝓥 ̇ } {f : X → Y} {y : Y}
                     → (w : fiber f y) → f (fiber-point w) ≡ y
fiber-identification (x , p) = p
```

So the type `fiber f y` collects the points `x : X` which are mapped to a point identified with `y`, including the identification datum. Voevodsky's insight is that a general notion of equivalence can be formulated in MLTT by requiring the fibers to be singletons. It is important here that not only the `y : Y` with `f x ≡ y` is unique, but also that the identification datum `p : f x ≡ y` is unique. This is similar to, or even a generalization of the categorical notion of "uniqueness up to a unique isomorphism".

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

It is easy to see that equivalences are invertible:

```
inverse : {X : 𝓤 ̇ } {Y : 𝓥 ̇ } (f : X → Y) → is-equiv f → (Y → X)
inverse f e y = fiber-point (center (fiber f y) (e y))

inverse-is-section : {X : 𝓤 ̇ } {Y : 𝓥 ̇ } (f : X → Y) (e : is-equiv f)
                   → (y : Y) → f (inverse f e y) ≡ y
inverse-is-section f e y = fiber-identification (center (fiber f y) (e y))

inverse-centrality : {X : 𝓤 ̇ } {Y : 𝓥 ̇ }
                     (f : X → Y) (e : is-equiv f) (y : Y) (t : fiber f y)
                   → (inverse f e y , inverse-is-section f e y) ≡ t
inverse-centrality f e y = centrality (fiber f y) (e y)

inverse-is-retraction : {X : 𝓤 ̇ } {Y : 𝓥 ̇ } (f : X → Y) (e : is-equiv f)
                      → (x : X) → inverse f e (f x) ≡ x
inverse-is-retraction f e x = ap fiber-point p
 where
  p : inverse f e (f x) , inverse-is-section f e (f x) ≡ x , refl (f x)
  p = inverse-centrality f e (f x) (x , (refl (f x)))

equivs-are-invertible : {X : 𝓤 ̇ } {Y : 𝓥 ̇ } (f : X → Y)
                      → is-equiv f → invertible f
equivs-are-invertible f e = inverse f e ,
                            inverse-is-retraction f e ,
                            inverse-is-section f e
```

The non-trivial direction is the following, for which we use the retraction techniques explained above:

```
invertibles-are-equivs : {X : 𝓤 ̇ } {Y : 𝓥 ̇ } (f : X → Y)
                       → invertible f → is-equiv f
```

```
invertibles-are-equivs {𝓤} {𝓥} {X} {Y} f (g , η , ε) y₀ = γ
 where
  a : (y : Y) → (f (g y) ≡ y₀) ◁ (y ≡ y₀)
  a y = r , s , rs
   where
    r : y ≡ y₀ → f (g y) ≡ y₀
    r p = f (g y) ≡⟨ ε y ⟩
          y       ≡⟨ p ⟩
          y₀      ∎
    s : f (g y) ≡ y₀ → y ≡ y₀
    s q = y       ≡⟨ (ε y)⁻¹ ⟩
          f (g y) ≡⟨ q ⟩
          y₀      ∎
    rs : (q : f (g y) ≡ y₀) → r (s q) ≡ q
    rs q = ε y ∙ ((ε y)⁻¹ ∙ q) ≡⟨ (∙assoc (ε y) ((ε y)⁻¹) q)⁻¹ ⟩
           (ε y ∙ (ε y)⁻¹) ∙ q ≡⟨ ap (_∙ q) (⁻¹-right∙ (ε y)) ⟩
           refl (f (g y)) ∙ q  ≡⟨ refl-left ⟩
           q                   ∎
  b : fiber f y₀ ◁ singleton-type y₀
  b = (Σ \(x : X) → f x ≡ y₀)     ◁⟨ Σ-reindex-retraction g (f , η) ⟩
      (Σ \(y : Y) → f (g y) ≡ y₀) ◁⟨ Σ-retract Y (λ y → f (g y) ≡ y₀) (λ y → y ≡ y₀) a ⟩
      (Σ \(y : Y) → y ≡ y₀)       ◀
  γ : is-singleton (fiber f y₀)
  γ = retract-of-singleton b (singleton-types-are-singletons Y y₀)

inverse-is-equiv : {X : 𝓤 ̇ } {Y : 𝓥 ̇ } (f : X → Y) (e : is-equiv f)
                 → is-equiv (inverse f e)
inverse-is-equiv f e = invertibles-are-equivs
                         (inverse f e)
                         (f , inverse-is-section f e , inverse-is-retraction f e)
```

Notice that inversion is involutive on the nose:

```
inversion-involutive : {X : 𝓤 ̇ } {Y : 𝓥 ̇ } (f : X → Y) (e : is-equiv f)
                     → inverse (inverse f e) (inverse-is-equiv f e) ≡ f
inversion-involutive f e = refl f
```

To see that the above procedures do exhibit the type "`f` is an equivalence" as a retract of the type "`f` is invertible", it suffices to show that it is a subsingleton.

The identity function is invertible:

```
id-invertible : (X : 𝓤 ̇ ) → invertible (id X)
id-invertible X = id X , refl , refl
```

We can compose invertible maps:

```
∘-invertible : {X : 𝓤 ̇ } {Y : 𝓥 ̇ } {Z : 𝓦 ̇ } {f : X → Y} {f' : Y → Z}
             → invertible f' → invertible f → invertible (f' ∘ f)
∘-invertible {𝓤} {𝓥} {𝓦} {X} {Y} {Z} {f} {f'} (g' , gf' , fg') (g , gf , fg) =
  g ∘ g' , η , ε
 where
  η : (x : X) → g (g' (f' (f x))) ≡ x
  η x = g (g' (f' (f x))) ≡⟨ ap g (gf' (f x)) ⟩
        g (f x)           ≡⟨ gf x ⟩
        x                 ∎
  ε : (z : Z) → f' (f (g (g' z))) ≡ z
  ε z = f' (f (g (g' z))) ≡⟨ ap f' (fg (g' z)) ⟩
        f' (g' z)         ≡⟨ fg' z ⟩
        z                 ∎
```

There is an identity equivalence, and we get composition of equivalences by reduction to invertible maps:

```
id-is-equiv : (X : 𝓤 ̇ ) → is-equiv (id X)
id-is-equiv = singleton-types-are-singletons
```

An `abstract` definition is not expanded during type checking. One possible use of this is efficiency. In our case, it saves 30s in the checking of the module `FunExt`.

```
∘-is-equiv : {X : 𝓤 ̇ } {Y : 𝓥 ̇ } {Z : 𝓦 ̇ } {f : X → Y} {g : Y → Z}
           → is-equiv g → is-equiv f → is-equiv (g ∘ f)
∘-is-equiv {𝓤} {𝓥} {𝓦} {X} {Y} {Z} {f} {g} i j = γ
 where
  abstract
   γ : is-equiv (g ∘ f)
   γ = invertibles-are-equivs (g ∘ f)
         (∘-invertible (equivs-are-invertible g i)
         (equivs-are-invertible f j))
```

The type of equivalences is defined as follows:

```
_≃_ : 𝓤 ̇ → 𝓥 ̇ → 𝓤 ⊔ 𝓥 ̇
X ≃ Y = Σ \(f : X → Y) → is-equiv f

Eq-to-fun : {X : 𝓤 ̇ } {Y : 𝓥 ̇ } → X ≃ Y → X → Y
Eq-to-fun (f , i) = f

Eq-to-fun-is-equiv : {X : 𝓤 ̇ } {Y : 𝓥 ̇ } (e : X ≃ Y) → is-equiv (Eq-to-fun e)
Eq-to-fun-is-equiv (f , i) = i
```

Identity and composition of equivalences:

```
≃-refl : (X : 𝓤 ̇ ) → X ≃ X
≃-refl X = id X , id-is-equiv X

_●_ : {X : 𝓤 ̇ } {Y : 𝓥 ̇ } {Z : 𝓦 ̇ } → X ≃ Y → Y ≃ Z → X ≃ Z
_●_ {𝓤} {𝓥} {𝓦} {X} {Y} {Z} (f , d) (f' , e) = f' ∘ f , ∘-is-equiv e d

≃-sym : {X : 𝓤 ̇ } {Y : 𝓥 ̇ } → X ≃ Y → Y ≃ X
≃-sym (f , e) = inverse f e , inverse-is-equiv f e
```

We can use the following for equational reasoning with equivalences:

```
_≃⟨_⟩_ : (X : 𝓤 ̇ ) {Y : 𝓥 ̇ } {Z : 𝓦 ̇ } → X ≃ Y → Y ≃ Z → X ≃ Z
_ ≃⟨ d ⟩ e = d ● e

_■ : (X : 𝓤 ̇ ) → X ≃ X
_■ = ≃-refl
```

We conclude this section with some examples. The function `transport A p` is an equivalence.

```
transport-is-equiv : {X : 𝓤 ̇ } (A : X → 𝓥 ̇ ) {x y : X} (p : x ≡ y)
                   → is-equiv (transport A p)
transport-is-equiv A (refl x) = id-is-equiv (A x)
```

Alternatively, we could have used the fact that `transport A (p ⁻¹)` is an inverse of `transport A p`.

Characterization of equality in `Σ` types:

```
Σ-≡-equiv : {X : 𝓤 ̇ } {A : X → 𝓥 ̇ } (σ τ : Σ A)
          → (σ ≡ τ) ≃ (Σ \(p : pr₁ σ ≡ pr₁ τ) → pr₂ σ ≡[ p / A ] pr₂ τ)
Σ-≡-equiv  {𝓤} {𝓥} {X} {A}  σ τ = from-Σ-≡ ,
                                  invertibles-are-equivs from-Σ-≡ (to-Σ-≡ , ε , η)
 where
  η : (w : Σ \(p : pr₁ σ ≡ pr₁ τ) → transport A p (pr₂ σ) ≡ pr₂ τ) → from-Σ-≡ (to-Σ-≡ w) ≡ w
  η (refl p , refl q) = refl (refl p , refl q)
  ε : (q : σ ≡ τ) → to-Σ-≡ (from-Σ-≡ q) ≡ q
  ε (refl σ) = refl (refl σ)
```

Table of contents ⇑

Voevodsky's univalence axiom

There is a canonical transformation `(X Y : 𝓤 ̇ ) → X ≡ Y → X ≃ Y` that sends the identity identification `refl X : X ≡ X` to the identity equivalence `≃-refl X : X ≃ X` by induction on identifications. The univalence axiom, for the universe `𝓤`, says that this canonical map is itself an equivalence.

```
Id-to-Eq : (X Y : 𝓤 ̇ ) → X ≡ Y → X ≃ Y
Id-to-Eq X X (refl X) = ≃-refl X

is-univalent : (𝓤 : Universe) → 𝓤 ⁺ ̇
is-univalent 𝓤 = (X Y : 𝓤 ̇ ) → is-equiv (Id-to-Eq X Y)
```

Thus, the univalence of the universe `𝓤` says that identifications `X ≡ Y` are in canonical bijection with equivalences `X ≃ Y`, if by bijection we mean equivalence, where the canonical bijection is `Id-to-Eq`.

We emphasize that this doesn't posit that univalence holds. It says what univalence is (like the type that says what the twin-prime conjecture is).

```
Eq-to-Id : is-univalent 𝓤 → (X Y : 𝓤 ̇ ) → X ≃ Y → X ≡ Y
Eq-to-Id ua X Y = inverse (Id-to-Eq X Y) (ua X Y)
```

Here is a third way to convert a type identification into a function:

```
Id-to-fun : {X Y : 𝓤 ̇ } → X ≡ Y → X → Y
Id-to-fun {𝓤} {X} {Y} p = Eq-to-fun (Id-to-Eq X Y p)

Id-to-funs-agree : {X Y : 𝓤 ̇ } (p : X ≡ Y)
                 → Id-to-fun p ≡ Id-to-Fun p
Id-to-funs-agree (refl X) = refl (id X)
```

What characterizes univalent mathematics is not the univalence axiom. We have defined and studied the main concepts of univalent mathematics in a pure, spartan MLTT. It is the concepts of hlevel, including singleton, subsingleton and set, and the notion of equivalence. Univalence *is* a fundamental ingredient, but first we need the correct notion of equivalence to be able to formulate it.

Remark. If we formulate univalence with invertible maps instead of equivalences, we get a statement that is provably false, and this is one of the reasons why Voevodsky's notion of equivalence is important. This is Exercise 4.6 of the HoTT book. There is a solution in Coq by Mike Shulman.

Table of contents ⇑

Equivalence induction

Under univalence, we get an induction principle for type equivalences, corresponding to the induction principles `H` and `J` for identifications.

To prove a property of equivalences, it is enough to prove it for the identity equivalence `≃-refl X` for all `X`:

```
H-≃ : is-univalent 𝓤
    → (X : 𝓤 ̇ ) (A : (Y : 𝓤 ̇ ) → X ≃ Y → 𝓥 ̇ )
    → A X (≃-refl X) → (Y : 𝓤 ̇ ) (e : X ≃ Y) → A Y e
H-≃ {𝓤} {𝓥} ua X A a Y e = γ
 where
  B : (Y : 𝓤 ̇ ) → X ≡ Y → 𝓥 ̇
  B Y p = A Y (Id-to-Eq X Y p)
  b : B X (refl X)
  b = a
  f : (Y : 𝓤 ̇ ) (p : X ≡ Y) → B Y p
  f = H X B b
  c : A Y (Id-to-Eq X Y (Eq-to-Id ua X Y e))
  c = f Y (Eq-to-Id ua X Y e)
  p : Id-to-Eq X Y (Eq-to-Id ua X Y e) ≡ e
  p = inverse-is-section (Id-to-Eq X Y) (ua X Y) e
  γ : A Y e
  γ = transport (A Y) p c
```

With this we have that if a type satisfies a property then so does any equivalent type:

```
transport-≃ : is-univalent 𝓤
            → (A : 𝓤 ̇ → 𝓥 ̇ ) {X Y : 𝓤 ̇ }
            → X ≃ Y → A X → A Y
transport-≃ ua A {X} {Y} e a = H-≃ ua X (λ Y _ → A Y) a Y e
```

The induction principle `H-≃` keeps `X` fixed and lets `Y` vary, while the induction principle `J-≃` let both vary:

```
J-≃ : is-univalent 𝓤
    → (A : (X Y : 𝓤 ̇ ) → X ≃ Y → 𝓥 ̇ )
    → ((X : 𝓤 ̇) → A X X (≃-refl X))
    → (X Y : 𝓤 ̇ ) (e : X ≃ Y) → A X Y e
J-≃ ua A φ X = H-≃ ua X (A X) (φ X)
```

A second set of equivalence induction principles refer to `is-equiv` rather than ≃ and are proved by reduction to the first version `H-≃`:

```
H-equiv : is-univalent 𝓤
        → (X : 𝓤 ̇ ) (A : (Y : 𝓤 ̇ ) → (X → Y) → 𝓥 ̇ )
        → A X (id X) → (Y : 𝓤 ̇ ) (f : X → Y) → is-equiv f → A Y f
H-equiv {𝓤} {𝓥} ua X A a Y f i = γ (f , i) i
 where
  B : (Y : 𝓤 ̇ ) → X ≃ Y → 𝓤 ⊔ 𝓥 ̇
  B Y (f , i) = is-equiv f → A Y f
  b : B X (≃-refl X)
  b = λ (_ : is-equiv (id X)) → a
  γ : (e : X ≃ Y) → B Y e
  γ = H-≃ ua X B b Y
```

The above and the following say that to prove that a property of functions holds for all equivalences, it is enough to prove it for all identity functions:

```
J-equiv : is-univalent 𝓤
        → (A : (X Y : 𝓤 ̇ ) → (X → Y) → 𝓥 ̇ )
        → ((X : 𝓤 ̇ ) → A X X (id X))
        → (X Y : 𝓤 ̇ ) (f : X → Y) → is-equiv f → A X Y f
J-equiv ua A φ X = H-equiv ua X (A X) (φ X)
```

And the follows is an immediate consequence of the fact that invertible maps are equivalences:

```
J-invertible : is-univalent 𝓤
             → (A : (X Y : 𝓤 ̇ ) → (X → Y) → 𝓥 ̇ )
             → ((X : 𝓤 ̇ ) → A X X (id X))
             → (X Y : 𝓤 ̇ ) (f : X → Y) → invertible f → A X Y f
J-invertible ua A φ X Y f i = J-equiv ua A φ X Y f (invertibles-are-equivs f i)
```

Here is an example:

```
Σ-change-of-variables' : is-univalent 𝓤
                       → {X : 𝓤 ̇ } {Y : 𝓤 ̇ } (A : X → 𝓥 ̇ ) (f : X → Y)
                       → (i : is-equiv f)
                       → (Σ \(x : X) → A x) ≡ (Σ \(y : Y) → A (inverse f i y))
Σ-change-of-variables' {𝓤} {𝓥} ua {X} {Y} A f i = H-≃ ua X B b Y (f , i)
 where
   B : (Y : 𝓤 ̇ ) → X ≃ Y →  (𝓤 ⊔ 𝓥)⁺ ̇
   B Y (f , i) = (Σ A) ≡ (Σ (A ∘ inverse f i))
   b : B X (≃-refl X)
   b = refl (Σ A)
```

The above version using the inverse of `f` can be proved directly by induction, but the following version is perhaps more natural

```
Σ-change-of-variables : is-univalent 𝓤
                      → {X : 𝓤 ̇} {Y : 𝓤 ̇ } (A : Y → 𝓥 ̇ ) (f : X → Y)
                      → (i : is-equiv f)
                      → (Σ \(y : Y) → A y) ≡ (Σ \(x : X) → A (f x))
Σ-change-of-variables ua A f i = Σ-change-of-variables' ua A
                                    (inverse f i)
                                    (inverse-is-equiv f i)
```

This particular proof works only because inversion is involutive on the nose.

Table of contents ⇑

Half-adjoint equivalences

An often useful alternative formulation of the notion of equivalence is the following, which adds data `τ x : ap f (η x) ≡ ε (f x)`, where identified elements live in the type `f (g (f x)) ≡ f x`, to turn the notion of invertibility into a subsingleton:

```
is-hae : {X : 𝓤 ̇ } {Y : 𝓥 ̇ } → (X → Y) → 𝓤 ⊔ 𝓥 ̇
is-hae f = Σ \(g : codomain f → domain f)
         → Σ \(η : g ∘ f ∼ id)
         → Σ \(ε : f ∘ g ∼ id)
         → (x : domain f) → ap f (η x) ≡ ε (f x)
```

The following just forgets the additional data `τ`:

```
haes-are-invertible : {X Y : 𝓤 ̇ } (f : X → Y)
                    → is-hae f → invertible f
haes-are-invertible f (g , η , ε , τ) = g , η , ε
```

To recover the data for all invertibles maps, under univalence, it is enough to give the data for identity maps:

```
id-is-hae : (X : 𝓤 ̇ ) → is-hae (id X)
id-is-hae X = id X , refl , refl , (λ x → refl (refl x))

invertibles-are-haes : is-univalent 𝓤
                     → (X Y : 𝓤 ̇ ) (f : X → Y)
                     → invertible f → is-hae f
invertibles-are-haes ua = J-invertible ua (λ X Y f → is-hae f) id-is-hae
```

The above can be proved without univalence, as is done in the HoTT book, with a more complicated argument.

Here is a use of the half-adjoint condition, where we remove univalence from the hypothesis, generalize the universe of the type `Y`, and weaken equality to equivalence in the conclusion. Notice that the proof starts as that of `Σ-reindex-retraction`.

```
Σ-change-of-variables-hae : {X : 𝓤 ̇ } {Y : 𝓥 ̇ } (A : Y → 𝓦 ̇ ) (f : X → Y)
                          → is-hae f → Σ A ≃ Σ (A ∘ f)
Σ-change-of-variables-hae {𝓤} {𝓥} {𝓦} {X} {Y} A f (g , η , ε , τ) =
  φ , invertibles-are-equivs φ (γ , γφ , φγ)
 where
  φ : Σ A → Σ (A ∘ f)
  φ (y , a) = (g y , transport A ((ε y)⁻¹) a)
  γ : Σ (A ∘ f) → Σ A
  γ (x , a) = (f x , a)
  γφ : (z : Σ A) → γ (φ z) ≡ z
  γφ (y , a) = to-Σ-≡ (ε y , transport-is-retraction A (ε y) a)
  φγ : (t : Σ (A ∘ f)) → φ (γ t) ≡ t
  φγ (x , a) = to-Σ-≡ (η x , q)
   where
    b : A (f (g (f x)))
    b = transport A ((ε (f x))⁻¹) a

    q = transport (A ∘ f) (η x)  b ≡⟨ transport-ap A f (η x) b ⟩
        transport A (ap f (η x)) b ≡⟨ ap (λ - → transport A - b) (τ x) ⟩
        transport A (ε (f x))    b ≡⟨ transport-is-retraction A (ε (f x)) a ⟩
        a                          ∎
```

Table of contents ⇑

Example of a type that is not a set under univalence

We have two automorphisms of 𝟚, namely the identity function and the function that swaps ₀ and ₁:

```
swap₂ : 𝟚 → 𝟚
swap₂ ₀ = ₁
swap₂ ₁ = ₀

swap₂-involutive : (n : 𝟚) → swap₂ (swap₂ n) ≡ n
swap₂-involutive ₀ = refl ₀
swap₂-involutive ₁ = refl ₁

swap₂-is-equiv : is-equiv swap₂
swap₂-is-equiv = invertibles-are-equivs swap₂ (swap₂ , swap₂-involutive , swap₂-involutive)
```

Hence we have two distinct equivalences:

```
e₀ e₁ : 𝟚 ≃ 𝟚
e₀ = ≃-refl 𝟚
e₁ = swap₂ , swap₂-is-equiv

e₀-is-not-e₁ : e₀ ≢ e₁
e₀-is-not-e₁ p = ₁-is-not-₀ r
 where
  q : id ≡ swap₂
  q = ap Eq-to-fun p
  r : ₁ ≡ ₀
  r = ap (λ - → - ₁) q
```

We now use an anonymous module to assume univalence in the next few constructions:

```
module _ (ua : is-univalent 𝓤₀) where
```

With this assumption, we get two different identifications of the type 𝟚 with itself:

```
  p₀ p₁ : 𝟚 ≡ 𝟚
  p₀ = Eq-to-Id ua 𝟚 𝟚 e₀
  p₁ = Eq-to-Id ua 𝟚 𝟚 e₁

  p₀-is-not-p₁ : p₀ ≢ p₁
  p₀-is-not-p₁ q = e₀-is-not-e₁ r
   where
    r = e₀              ≡⟨ (inverse-is-section (Id-to-Eq 𝟚 𝟚) (ua 𝟚 𝟚) e₀)⁻¹ ⟩
        Id-to-Eq 𝟚 𝟚 p₀ ≡⟨ ap (Id-to-Eq 𝟚 𝟚) q ⟩
        Id-to-Eq 𝟚 𝟚 p₁ ≡⟨ inverse-is-section (Id-to-Eq 𝟚 𝟚) (ua 𝟚 𝟚) e₁ ⟩
        e₁              ∎
```

If the universe $\mathcal{U}_0$ were a set, then the identifications p₀ and p₁ defined above would be equal, and therefore it is not a set.

```
  𝓤₀-is-not-a-set :  ¬(is-set (𝓤₀ ̇ ))
  𝓤₀-is-not-a-set s = p₀-is-not-p₁ q
   where
    q : p₀ ≡ p₁
    q = s 𝟚 𝟚 p₀ p₁
```

For more examples, see Kraus and Sattler.

Table of contents ⇑

Exercises

Here are some facts whose proofs are left to the reader but that we will need from the next section onwards. Sample solutions are given below.

Define functions for the following type declarations. As a matter of procedure, we suggest to import this file and add another declaration with the same type and new name e.g. `section-are-lc-solution`, because we already have solutions in this file.

We start with the notion of left cancellability.

```
left-cancellable : {X : 𝓤 ̇ } {Y : 𝓥 ̇ } → (X → Y) → 𝓤 ⊔ 𝓥 ̇
left-cancellable f = {x x' : domain f} → f x ≡ f x' → x ≡ x'

lc-maps-reflect-subsingletonness : {X : 𝓤 ̇ } {Y : 𝓥 ̇ } (f : X → Y)
                                 → left-cancellable f
                                 → is-subsingleton Y
                                 → is-subsingleton X

has-retraction : {X : 𝓤 ̇ } {Y : 𝓥 ̇ } → (X → Y) → 𝓤 ⊔ 𝓥 ̇
has-retraction s = Σ \(r : codomain s → domain s) → r ∘ s ∼ id

sections-are-lc : {X : 𝓤 ̇ } {A : 𝓥 ̇ } (s : X → A) → has-retraction s → left-cancellable s

equivs-have-retractions : {X : 𝓤 ̇ } {Y : 𝓥 ̇ } (f : X → Y) → is-equiv f → has-retraction f

equivs-have-sections : {X : 𝓤 ̇ } {Y : 𝓥 ̇ } (f : X → Y) → is-equiv f → has-section f

equivs-are-lc : {X : 𝓤 ̇ } {Y : 𝓥 ̇ } (f : X → Y) → is-equiv f → left-cancellable f

equiv-to-subsingleton : {X : 𝓤 ̇ } {Y : 𝓥 ̇ } (f : X → Y)
                      → is-equiv f
                      → is-subsingleton Y
                      → is-subsingleton X

equiv-to-subsingleton' : {X : 𝓤 ̇ } {Y : 𝓥 ̇ } (f : X → Y)
                       → is-equiv f
                       → is-subsingleton X
                       → is-subsingleton Y

sections-closed-under-∼ : {X : 𝓤 ̇ } {Y : 𝓥 ̇ } (f g : X → Y)
                        → has-retraction f
                        → g ∼ f
                        → has-retraction g

retractions-closed-under-∼ : {X : 𝓤 ̇ } {Y : 𝓥 ̇ } (f g : X → Y)
                           → has-section f
                           → g ∼ f
                           → has-section g
```

An alternative notion of equivalence, equivalent to Voevodsky’s, has been given by Andre Joyal:

```
is-joyal-equiv : {X : 𝓤 ̇ } {Y : 𝓥 ̇ } → (X → Y) → 𝓤 ⊔ 𝓥 ̇
is-joyal-equiv f = has-section f × has-retraction f
```

Provide definitions for the following type declarations:

```
joyal-equivs-are-invertible : {X : 𝓤 ̇ } {Y : 𝓥 ̇ } (f : X → Y)
                            → is-joyal-equiv f → invertible f

joyal-equivs-are-equivs : {X : 𝓤 ̇ } {Y : 𝓥 ̇ } (f : X → Y)
                        → is-joyal-equiv f → is-equiv f

invertibles-are-joyal-equivs : {X : 𝓤 ̇ } {Y : 𝓥 ̇ } (f : X → Y)
                             → invertible f → is-joyal-equiv f

equivs-are-joyal-equivs : {X : 𝓤 ̇ } {Y : 𝓥 ̇ } (f : X → Y)
                        → is-equiv f → is-joyal-equiv f

equivs-closed-under-∼ : {X : 𝓤 ̇ } {Y : 𝓥 ̇ } (f g : X → Y)
                      → is-equiv f
                      → g ∼ f
                      → is-equiv g

equivs-closed-under-∼' : {X : 𝓤 ̇ } {Y : 𝓥 ̇ } (f g : X → Y)
                       → is-equiv f
                       → f ∼ g
                       → is-equiv g

≃-gives-◁ : (X : 𝓤 ̇ ) (Y : 𝓥 ̇ ) → X ≃ Y → X ◁ Y

≃-gives-▷ : (X : 𝓤 ̇ ) (Y : 𝓥 ̇ ) → X ≃ Y → Y ◁ X

equiv-to-singleton : (X : 𝓤 ̇ ) (Y : 𝓥 ̇ )
                   → X ≃ Y → is-singleton Y → is-singleton X

equiv-to-singleton' : (X : 𝓤 ̇ ) (Y : 𝓥 ̇ )
                    → X ≃ Y → is-singleton X → is-singleton Y

subtypes-of-sets-are-sets : {X : 𝓤 ̇ } {Y : 𝓥 ̇ } (m : X → Y)
                          → left-cancellable m → is-set Y → is-set X

pr₁-lc : {X : 𝓤 ̇ } {A : X → 𝓥 ̇ } → ((x : X) → is-subsingleton (A x))
       → left-cancellable  (λ (t : Σ A) → pr₁ t)

subsets-of-sets-are-sets : (X : 𝓤 ̇ ) (A : X → 𝓥 ̇ )
                         → is-set X
                         → ((x : X) → is-subsingleton(A x))
                         → is-set(Σ \(x : X) → A x)

pr₁-equivalence : (X : 𝓤 ̇ ) (A : X → 𝓥 ̇ )
                → ((x : X) → is-singleton (A x))
                → is-equiv (λ (t : Σ A) → pr₁ t)

ΠΣ-distr-≃ : {X : 𝓤 ̇ } {A : X → 𝓥 ̇ } {P : (x : X) → A x → 𝓦 ̇ }
           → (Π \(x : X) → Σ \(a : A x) → P x a) ≃ (Σ \(f : Π A) → Π \(x : X) → P x (f x))

Σ-cong : {X : 𝓤 ̇ } {A : X → 𝓥 ̇ } {B : X → 𝓦 ̇ }
       → ((x : X) → A x ≃ B x) → Σ A ≃ Σ B

Σ-assoc : {X : 𝓤 ̇ } {Y : X → 𝓥 ̇ } {Z : Σ Y → 𝓦 ̇ }
        → Σ Z ≃ (Σ \(x : X) → Σ \(y : Y x) → Z (x , y))

⁻¹-≃ : {X : 𝓤 ̇ } (x y : X) → (x ≡ y) ≃ (y ≡ x)

singleton-type' : {X : 𝓤 ̇ } → X → 𝓤 ̇
singleton-type' x = Σ \y → x ≡ y

singleton-types-≃ : {X : 𝓤 ̇ } (x : X) → singleton-type' x ≃ singleton-type x

singleton-types-are-singletons' : (X : 𝓤 ̇ ) (x : X) → is-singleton (singleton-type' x)
```

```
singletons-equivalent : (X : 𝓤 ̇ ) (Y : 𝓥 ̇ )
                      → is-singleton X → is-singleton Y → X ≃ Y

maps-of-singletons-are-equivs : (X : 𝓤 ̇ ) (Y : 𝓥 ̇ ) (f : X → Y)
                              → is-singleton X → is-singleton Y → is-equiv f

logically-equivalent-subsingletons-are-equivalent : (X : 𝓤 ̇ ) (Y : 𝓥 ̇ )
                                                  → is-subsingleton X
                                                  → is-subsingleton Y
                                                  → X ⇔ Y
                                                  → X ≃ Y

NatΣ-fiber-equiv : {X : 𝓤 ̇ } (A : X → 𝓥 ̇ ) (B : X → 𝓦 ̇ ) (φ : Nat A B)
                 → (x : X) (b : B x) → fiber (φ x) b ≃ fiber (NatΣ φ) (x , b)

NatΣ-equiv-gives-fiberwise-equiv : {X : 𝓤 ̇ } (A : X → 𝓥 ̇ ) (B : X → 𝓦 ̇ ) (φ : Nat A B)
                                 → is-equiv (NatΣ φ) → ((x : X) → is-equiv (φ x))

Σ-is-subsingleton : {X : 𝓤 ̇ } {A : X → 𝓥 ̇ }
                  → is-subsingleton X
                  → ((x : X) → is-subsingleton (A x))
                  → is-subsingleton (Σ A)

×-is-subsingleton : {X : 𝓤 ̇ } {Y : 𝓥 ̇ }
                  → is-subsingleton X
                  → is-subsingleton Y
                  → is-subsingleton (X × Y)

to-×-≡ : {X : 𝓤 ̇ } {Y : 𝓥 ̇ } {z t : X × Y}
       → pr₁ z ≡ pr₁ t
       → pr₂ z ≡ pr₂ t
       → z ≡ t

×-is-subsingleton' : {X : 𝓤 ̇ } {Y : 𝓥 ̇ }
                   → ((Y → is-subsingleton X) × (X → is-subsingleton Y))
                   → is-subsingleton (X × Y)

×-is-subsingleton'-back : {X : 𝓤 ̇ } {Y : 𝓥 ̇ }
                        → is-subsingleton (X × Y)
                        → (Y → is-subsingleton X) × (X → is-subsingleton Y)

ap₂ : {X : 𝓤 ̇ } {Y : 𝓥 ̇ } {Z : 𝓦 ̇ } (f : X → Y → Z) {x x' : X} {y y' : Y}
    → x ≡ x' → y ≡ y' → f x y ≡ f x' y'
```

Table of contents ⇑

Operator fixities and precedences

Without the following list of operator precedences and associativity (left or right), this agda file doesn't parse and is rejected by Agda.

```
infix  0 _◁_
infix  1 _◀
infixr 0 _◁⟨_⟩_
infix  0 _≃_
infixl 2 _●_
infixr 0 _≃⟨_⟩_
infix  1 _■
```

Table of contents ⇑

Solutions

Spoiler alert.

```
lc-maps-reflect-subsingletonness f l s x x' = l (s (f x) (f x'))

sections-are-lc s (r , ε) {x} {y} p = x       ≡⟨ (ε x)⁻¹ ⟩
                                      r (s x) ≡⟨ ap r p ⟩
                                      r (s y) ≡⟨ ε y ⟩
                                      y       ∎

equivs-have-retractions f e = (inverse f e , inverse-is-retraction f e)

equivs-have-sections f e = (inverse f e , inverse-is-section f e)

equivs-are-lc f e = sections-are-lc f (equivs-have-retractions f e)

equiv-to-subsingleton f e = lc-maps-reflect-subsingletonness f (equivs-are-lc f e)

equiv-to-subsingleton' f e = lc-maps-reflect-subsingletonness
                               (inverse f e)
                               (equivs-are-lc (inverse f e) (inverse-is-equiv f e))

sections-closed-under-∼ f g (r , rf) h = (r ,
                                          λ x → r (g x) ≡⟨ ap r (h x) ⟩
                                                r (f x) ≡⟨ rf x ⟩
                                                x       ∎)

retractions-closed-under-∼ f g (s , fs) h = (s ,
                                             λ y → g (s y) ≡⟨ h (s y) ⟩
                                                   f (s y) ≡⟨ fs y ⟩
                                                   y ∎)

joyal-equivs-are-invertible f ((s , fs) , (r , rf)) = (s , sf , fs)
 where
  sf = λ (x : domain f) → s(f x)         ≡⟨ (rf (s (f x)))⁻¹ ⟩
                          r(f(s(f x))) ≡⟨ ap r (fs (f x)) ⟩
                          r(f x)       ≡⟨ rf x ⟩
                          x            ∎

joyal-equivs-are-equivs f j = invertibles-are-equivs f (joyal-equivs-are-invertible f j)

invertibles-are-joyal-equivs f (g , gf , fg) = ((g , fg) , (g , gf))

equivs-are-joyal-equivs f e = invertibles-are-joyal-equivs f (equivs-are-invertible f e)

equivs-closed-under-∼ f g e h =
 joyal-equivs-are-equivs g
  (retractions-closed-under-∼ f g (equivs-have-sections    f e) h ,
   sections-closed-under-∼    f g (equivs-have-retractions f e) h)

equivs-closed-under-∼' f g e h = equivs-closed-under-∼ f g e (λ x → (h x)⁻¹)

≃-gives-◁ X Y (f , e) = (inverse f e , f , inverse-is-retraction f e)

≃-gives-▷ X Y (f , e) = (f , inverse f e , inverse-is-section f e)

equiv-to-singleton X Y e = retract-of-singleton (≃-gives-◁ X Y e)

equiv-to-singleton' X Y e = retract-of-singleton (≃-gives-▷ X Y e)

subtypes-of-sets-are-sets {𝓤} {𝓥} {X} m i h = ≡-collapsibles-are-sets X c
 where
  f : (x x' : X) → x ≡ x' → x ≡ x'
  f x x' r = i (ap m r)
  κ : (x x' : X) (r s : x ≡ x') → f x x' r ≡ f x x' s
  κ x x' r s = ap i (h (m x) (m x') (ap m r) (ap m s))
  c : ≡-collapsible X
  c x x' = f x x' , κ x x'

pr₁-lc i p = to-Σ-≡ (p , i _ _ _)
```

```
subsets-of-sets-are-sets X A h p = subtypes-of-sets-are-sets pr₁ (pr₁-lc p) h

pr₁-equivalence {𝓤} {𝓥} X A s = invertibles-are-equivs pr₁ (g , η , ε)
 where
  g : X → Σ A
  g x = x , pr₁(s x)
  ε : (x : X) → pr₁ (g x) ≡ x
  ε x = refl (pr₁ (g x))
  η : (σ : Σ A) → g (pr₁ σ) ≡ σ
  η (x , a) = to-Σ-≡ (ε x , singletons-are-subsingletons (A x) (s x) _ a)

ΠΣ-distr-≃ {𝓤} {𝓥} {𝓦} {X} {A} {P} = φ , invertibles-are-equivs φ (γ , η , ε)
 where
  φ : (Π \(x : X) → Σ \(a : A x) → P x a) → Σ \(f : Π A) → Π \(x : X) → P x (f x)
  φ g = ((λ x → pr₁ (g x)) , λ x → pr₂ (g x))

  γ : (Σ \(f : Π A) → Π \(x : X) → P x (f x)) → Π \(x : X) → Σ \(a : A x) → P x a
  γ (f , φ) x = f x , φ x
  η : γ ∘ φ ∼ id
  η = refl
  ε : φ ∘ γ ∼ id
  ε = refl

Σ-cong {𝓤} {𝓥} {𝓦} {X} {A} {B} φ =
  (NatΣ f , invertibles-are-equivs (NatΣ f) (NatΣ g , NatΣ-η , NatΣ-ε))
 where
  f : (x : X) → A x → B x
  f x = Eq-to-fun (φ x)
  g : (x : X) → B x → A x
  g x = inverse (f x) (Eq-to-fun-is-equiv (φ x))
  η : (x : X) (a : A x) → g x (f x a) ≡ a
  η x = inverse-is-retraction (f x) (Eq-to-fun-is-equiv (φ x))
  ε : (x : X) (b : B x) → f x (g x b) ≡ b
  ε x = inverse-is-section (f x) (Eq-to-fun-is-equiv (φ x))

  NatΣ-η : (w : Σ A) → NatΣ g (NatΣ f w) ≡ w
  NatΣ-η (x , a) = x , g x (f x a) ≡⟨ ap (λ - → x , -) (η x a) ⟩
                   x , a          ∎

  NatΣ-ε : (t : Σ B) → NatΣ f (NatΣ g t) ≡ t
  NatΣ-ε (x , b) = x , f x (g x b) ≡⟨ ap (λ - → x , -) (ε x b) ⟩
                   x , b          ∎

Σ-assoc {𝓤} {𝓥} {𝓦} {X} {Y} {Z} = f , invertibles-are-equivs f (g , refl , refl)
 where
  f : Σ Z → Σ \x → Σ \y → Z (x , y)
  f ((x , y) , z) = (x , (y , z))
  g : (Σ \x → Σ \y → Z (x , y)) → Σ Z
  g (x , (y , z)) = ((x , y) , z)

⁻¹-≃ x y = (_⁻¹ , invertibles-are-equivs _⁻¹ (_⁻¹ , ⁻¹-involutive , ⁻¹-involutive))

singleton-types-≃ x = Σ-cong (λ y → ⁻¹-≃ x y)

singleton-types-are-singletons' X x = equiv-to-singleton
                                       (singleton-type' x)
                                       (singleton-type x)
                                       (singleton-types-≃ x)
                                       (singleton-types-are-singletons X x)

singletons-equivalent X Y i j = f , invertibles-are-equivs f (g , η , ε)
 where
  f : X → Y
  f x = center Y j
  g : Y → X
  g y = center X i
  η : (x : X) → g (f x) ≡ x
  η = centrality X i
  ε : (y : Y) → f (g y) ≡ y
  ε = centrality Y j

maps-of-singletons-are-equivs X Y f i j = invertibles-are-equivs f (g , η , ε)
 where
  g : Y → X
  g y = center X i
  η : (x : X) → g (f x) ≡ x
  η = centrality X i
  ε : (y : Y) → f (g y) ≡ y
  ε y = singletons-are-subsingletons Y j (f (g y)) y

logically-equivalent-subsingletons-are-equivalent X Y i j (f , g) =
  f , invertibles-are-equivs f (g , (λ x → i (g (f x)) x) , (λ y → j (f (g y)) y))

NatΣ-fiber-equiv A B φ x b = (f , invertibles-are-equivs f (g , ε , η))
 where
  f : fiber (φ x) b → fiber (NatΣ φ) (x , b)
  f (a , refl _) = ((x , a) , refl (x , φ x a))
  g : fiber (NatΣ φ) (x , b) → fiber (φ x) b
  g ((x , a) , refl _) = (a , refl (φ x a))
  ε : (w : fiber (φ x) b) → g (f w) ≡ w
  ε (a , refl _) = refl (a , refl (φ x a))
  η : (t : fiber (NatΣ φ) (x , b)) → f (g t) ≡ t
  η ((x , a) , refl _) = refl ((x , a) , refl (NatΣ φ (x , a)))

NatΣ-equiv-gives-fiberwise-equiv A B φ e x b = γ
 where
  γ : is-singleton (fiber (φ x) b)
  γ = equiv-to-singleton
         (fiber (φ x) b)
         (fiber (NatΣ φ) (x , b))
         (NatΣ-fiber-equiv A B φ x b)
         (e (x , b))

Σ-is-subsingleton i j (x , a) (y , b) = to-Σ-≡ (i x y , j y _ _)

×-is-subsingleton i j = Σ-is-subsingleton i (λ _ → j)

to-×-≡ (refl x) (refl y) = refl (x , y)

×-is-subsingleton' {𝓤} {𝓥} {X} {Y} (i , j) = k
 where
  k : is-subsingleton (X × Y)
  k (x , y) (x' , y') = to-×-≡ (i y x x') (j x y y')

×-is-subsingleton'-back {𝓤} {𝓥} {X} {Y} k = i , j
 where
  i : Y → is-subsingleton X
  i y x x' = ap pr₁ (k (x , y) (x' , y))
  j : X → is-subsingleton Y
  j x y y' = ap pr₂ (k (x , y) (x , y'))

ap₂ f (refl x) (refl y) = refl (f x y)
```

Table of contents ⇑

Function extensionality from univalence

Function extensionality says that any two pointwise equal functions are equal. This is known to be not provable or disprovable in MLTT. It is an independent statement, which we abbreviate as `funext`.

```
funext : ∀ 𝓤 𝓥 → (𝓤 ⊔ 𝓥)⁺ ̇
funext 𝓤 𝓥 = {X : 𝓤 ̇ } {Y : 𝓥 ̇ } {f g : X → Y} → f ∼ g → f ≡ g
```

There will be two stronger statements, namely the generalization to dependent functions, and the requirement that the canonical map `(f ≡ g) → (f ∼ g)` is an equivalence.

Exercise. Assuming `funext`, prove that if `f : X → Y` is an equivalence then so is the function `(-) ∘ f : (Y → Z) → (X → Z)`.

The crucial step in Voevodsky's proof that univalence implies `funext` is to establish the conclusion of the above exercise assuming univalence instead. We prove this by equivalence induction on `f`, which means that we only need to consider the case when `f` is an identity function, for which pre-composition with `f` is itself an identity function (of a function type), and hence an equivalence:

```
pre-comp-is-equiv : (ua : is-univalent 𝓤) (X Y : 𝓤 ̇ ) (f : X → Y)
                  → is-equiv f
                  → (Z : 𝓤 ̇ ) → is-equiv (λ (g : Y → Z) → g ∘ f)
pre-comp-is-equiv {𝓤} ua =
   J-equiv ua
     (λ X Y (f : X → Y) → (Z : 𝓤 ̇ ) → is-equiv (λ g → g ∘ f))
     (λ X Z → id-is-equiv (X → Z))
```

With this we can prove the desired result as follows.

```
univalence-gives-funext : is-univalent 𝓤 → funext 𝓥 𝓤
univalence-gives-funext ua {X} {Y} {f₀} {f₁} = γ
 where
  Δ = Σ \(y₀ : Y) → Σ \(y₁ : Y) → y₀ ≡ y₁

  δ : Y → Δ
  δ y = (y , y , refl y)

  π₀ π₁ : Δ → Y
  π₀ (y₀ , y₁ , p) = y₀
  π₁ (y₀ , y₁ , p) = y₁

  δ-is-equiv : is-equiv δ
  δ-is-equiv = invertibles-are-equivs δ (π₀ , η , ε)
   where
    η : (y : Y) → π₀ (δ y) ≡ y
    η y = refl y
    ε : (d : Δ) → δ (π₀ d) ≡ d
    ε (y , y , refl y) = refl (y , y , refl y)

  φ : (Δ → Y) → (Y → Y)
  φ π = π ∘ δ

  φ-is-equiv : is-equiv φ
  φ-is-equiv = pre-comp-is-equiv ua Y Δ δ δ-is-equiv Y

  p : φ π₀ ≡ φ π₁
  p = refl (id Y)

  q : π₀ ≡ π₁
  q = equivs-are-lc φ φ-is-equiv p

  γ : f₀ ∼ f₁ → f₀ ≡ f₁
  γ h = ap (λ π x → π (f₀ x , f₁ x , h x)) q
```

This definition of `γ` is probably too quick. Here are all the steps performed silently by Agda, by expanding judgmental equalities, indicated with `refl` here:

```
  γ' : f₀ ∼ f₁ → f₀ ≡ f₁
  γ' h = f₀                              ≡⟨ refl _ ⟩
        (λ x → f₀ x)                     ≡⟨ refl _ ⟩
        (λ x → π₀ (f₀ x , f₁ x , h x))  ≡⟨ ap (λ π x → π (f₀ x , f₁ x , h x)) q ⟩
        (λ x → π₁ (f₀ x , f₁ x , h x))  ≡⟨ refl _ ⟩
        (λ x → f₁ x)                     ≡⟨ refl _ ⟩
        f₁                               ∎
```

So notice that this relies on the so-called η-rule for judgmental equality of functions, namely `f = λ x → f x`. Without it, we would only get that

```
      f₀ ∼ f₁ → (λ x → f₀ x) ≡ (λ x → f₁ x)
```

instead.

Table of contents ⇑

Variations of function extensionality and their logical equivalence

Dependent function extensionality:

```
dfunext : ∀ 𝓤 𝓥 → (𝓤 ⊔ 𝓥)⁺ ̇
dfunext 𝓤 𝓥 = {X : 𝓤 ̇ } {A : X → 𝓥 ̇ } {f g : Π A} → f ∼ g → f ≡ g
```

The above says that there is some map `f ~ g → f ≡ g`. The following instead says that the canonical map in the other direction is an equivalence:

```
happly : {X : 𝓤 ̇ } {A : X → 𝓥 ̇ } (f g : Π A) → f ≡ g → f ∼ g
happly f g p x = ap (λ - → - x) p

hfunext : ∀ 𝓤 𝓥 → (𝓤 ⊔ 𝓥)⁺ ̇
hfunext 𝓤 𝓥 = {X : 𝓤 ̇ } {A : X → 𝓥 ̇ } (f g : Π A) → is-equiv (happly f g)

hfunext-gives-dfunext : hfunext 𝓤 𝓥 → dfunext 𝓤 𝓥
hfunext-gives-dfunext hfe {X} {A} {f} {g} = inverse (happly f g) (hfe f g)
```

Voevodsky showed that all notions of function extensionality are logically equivalent to saying that products of singletons are singletons:

```
vvfunext : ∀ 𝓤 𝓥 → (𝓤 ⊔ 𝓥)⁺ ̇
vvfunext 𝓤 𝓥 = {X : 𝓤 ̇ } {A : X → 𝓥 ̇ } → ((x : X) → is-singleton (A x)) → is-singleton (Π A)

dfunext-gives-vvfunext : dfunext 𝓤 𝓥 → vvfunext 𝓤 𝓥
dfunext-gives-vvfunext fe {X} {A} i = f , c
 where
  f : Π A
  f x = center (A x) (i x)
  c : (g : Π A) → f ≡ g
  c g = fe (λ (x : X) → centrality (A x) (i x) (g x))
```

We need some lemmas to get `hfunext` from `vvfunext`:

```
post-comp-is-invertible : {X : 𝓤 ̇ } {Y : 𝓥 ̇ } {A : 𝓦 ̇ }
                        → funext 𝓦 𝓤 → funext 𝓦 𝓥
                        → (f : X → Y) → invertible f → invertible (λ (h : A → X) → f ∘ h)
post-comp-is-invertible {𝓤} {𝓥} {𝓦} {X} {Y} {A} nfe nfe' f (g , η , ε) = (g' , η' , ε')
 where
  f' : (A → X) → (A → Y)
  f' h = f ∘ h
  g' : (A → Y) → (A → X)
  g' k = g ∘ k
  η' : (h : A → X) → g' (f' h) ≡ h
  η' h = nfe (η ∘ h)
  ε' : (k : A → Y) → f' (g' k) ≡ k
  ε' k = nfe' (ε ∘ k)

post-comp-is-equiv : {X : 𝓤 ̇ } {Y : 𝓥 ̇ } {A : 𝓦 ̇ } → funext 𝓦 𝓤 → funext 𝓦 𝓥
                   → (f : X → Y) → is-equiv f → is-equiv (λ (h : A → X) → f ∘ h)
post-comp-is-equiv fe fe' f e =
 invertibles-are-equivs
  (λ h → f ∘ h)
  (post-comp-is-invertible fe fe' f (equivs-are-invertible f e))

vvfunext-gives-hfunext : vvfunext 𝓤 𝓥 → hfunext 𝓤 𝓥
vvfunext-gives-hfunext {𝓤} {𝓥} vfe {X} {Y} f = γ
 where
```

```
  a : (x : X) → is-singleton (Σ \(y : Y x) → f x ≡ y)
  a x = singleton-types-are-singletons' (Y x) (f x)
  c : is-singleton ((x : X) → Σ \(y : Y x) → f x ≡ y)
  c = vfe a
  R : (Σ \(g : Π Y) → f ∼ g) ◁ (Π \(x : X) → Σ \(y : Y x) → f x ≡ y)
  R = ≃-gives-▷ _ _ ΠΣ-distr-≃
  r : (Π \(x : X) → Σ \(y : Y x) → f x ≡ y) → Σ \(g : Π Y) → f ∼ g
  r = λ _ → f , (λ x → refl (f x))
  d : is-singleton (Σ \(g : Π Y) → f ∼ g)
  d = retract-of-singleton R c
  e : (Σ \(g : Π Y) → f ≡ g) → (Σ \(g : Π Y) → f ∼ g)
  e = NatΣ (happly f)
  i : is-equiv e
  i = maps-of-singletons-are-equivs (Σ (λ g → f ≡ g)) (Σ (λ g → f ∼ g)) e
       (singleton-types-are-singletons' (Π Y) f) d
  γ : (g : Π Y) → is-equiv (happly f g)
  γ = NatΣ-equiv-gives-fiberwise-equiv (λ g → f ≡ g) (λ g → f ∼ g) (happly f) i
```

And finally the seemingly rather weak, non-dependent version `funext` implies the seemingly strongest version, which closes the circle of logical equivalences.

```
funext-gives-vvfunext : funext 𝓤 (𝓤 ⊔ 𝓥) → funext 𝓤 𝓤 → vvfunext 𝓤 𝓥
funext-gives-vvfunext {𝓤} {𝓥} fe fe' {X} {A} φ = γ
 where
  f : Σ A → X
  f = pr₁
  f-is-equiv : is-equiv f
  f-is-equiv = pr₁-equivalence X A φ
  g : (X → Σ A) → (X → X)
  g h = f ∘ h
  g-is-equiv : is-equiv g
  g-is-equiv = post-comp-is-equiv fe fe' f f-is-equiv
  i : is-singleton (Σ \(h : X → Σ A) → f ∘ h ≡ id X)
  i = g-is-equiv (id X)
  r : (Σ \(h : X → Σ A) → f ∘ h ≡ id X) → Π A
  r (h , p) x = transport A (happly (f ∘ h) (id X) p x) (pr₂ (h x))
  s : Π A → (Σ \(h : X → Σ A) → f ∘ h ≡ id X)
  s φ = (λ x → x , φ x) , refl (id X)
  rs : ∀ φ → r (s φ) ≡ φ
  rs φ = refl (r (s φ))
  γ : is-singleton (Π A)
  γ = retract-of-singleton (r , s , rs) i
```

We have the following corollaries. We first formulate the types of some functions:

```
funext-gives-hfunext       : funext 𝓤 (𝓤 ⊔ 𝓥) → funext 𝓤 𝓤 → hfunext 𝓤 𝓥
funext-gives-dfunext       : funext 𝓤 (𝓤 ⊔ 𝓥) → funext 𝓤 𝓤 → dfunext 𝓤 𝓥
univalence-gives-dfunext'  : is-univalent 𝓤 → is-univalent (𝓤 ⊔ 𝓥) → dfunext 𝓤 𝓥
univalence-gives-hfunext'  : is-univalent 𝓤 → is-univalent (𝓤 ⊔ 𝓥) → hfunext 𝓤 𝓥
univalence-gives-vvfunext' : is-univalent 𝓤 → is-univalent (𝓤 ⊔ 𝓥) → vvfunext 𝓤 𝓥
univalence-gives-hfunext   : is-univalent 𝓤 → hfunext 𝓤 𝓤
univalence-gives-dfunext   : is-univalent 𝓤 → dfunext 𝓤 𝓤
univalence-gives-vvfunext  : is-univalent 𝓤 → vvfunext 𝓤 𝓤
```

And then we give their definitions (Agda makes sure there are no circularities):

```
funext-gives-hfunext fe fe' = vvfunext-gives-hfunext (funext-gives-vvfunext fe fe')

funext-gives-dfunext fe fe' = hfunext-gives-dfunext (funext-gives-hfunext fe fe')

univalence-gives-dfunext' ua ua' = funext-gives-dfunext
                                    (univalence-gives-funext ua')
                                    (univalence-gives-funext ua)

univalence-gives-hfunext' ua ua' = funext-gives-hfunext
                                     (univalence-gives-funext ua')
                                     (univalence-gives-funext ua)
```

```
univalence-gives-vvfunext' ua ua' = funext-gives-vvfunext
                                     (univalence-gives-funext ua')
                                     (univalence-gives-funext ua)

univalence-gives-hfunext ua = univalence-gives-hfunext' ua ua

univalence-gives-dfunext ua = univalence-gives-dfunext' ua ua

univalence-gives-vvfunext ua = univalence-gives-vvfunext' ua ua
```

Table of contents ⇑

The univalence axiom is a (sub)singleton

If we use a type as an axiom, it should better have at most one element. We prove some generally useful lemmas first.

```
Π-is-subsingleton : dfunext 𝓤 𝓥 → {X : 𝓤 ̇ } {A : X → 𝓥 ̇ }
                  → ((x : X) → is-subsingleton (A x))
                  → is-subsingleton (Π A)
Π-is-subsingleton fe i f g = fe (λ x → i x (f x) (g x))

being-singleton-is-a-subsingleton : dfunext 𝓤 𝓤 → {X : 𝓤 ̇ }
                                  → is-subsingleton (is-singleton X)
being-singleton-is-a-subsingleton fe {X} (x , φ) (y , γ) = p
 where
  i : is-subsingleton X
  i = singletons-are-subsingletons X (y , γ)
  s : is-set X
  s = subsingletons-are-sets X i
  p : (x , φ) ≡ (y , γ)
  p = to-Σ-≡ (φ y , fe (λ (z : X) → s y z _ _))

being-equiv-is-a-subsingleton : dfunext 𝓥 (𝓤 ⊔ 𝓥) → dfunext (𝓤 ⊔ 𝓥) (𝓤 ⊔ 𝓥)
                              → {X : 𝓤 ̇ } {Y : 𝓥 ̇ } (f : X → Y)
                              → is-subsingleton (is-equiv f)
being-equiv-is-a-subsingleton fe fe' f =
 Π-is-subsingleton fe (λ x → being-singleton-is-a-subsingleton fe')

univalence-is-a-subsingleton : is-univalent (𝓤 ⁺) → is-subsingleton (is-univalent 𝓤)
univalence-is-a-subsingleton {𝓤} ua⁺ ua ua' = p
 where
  fe₀  :  funext  𝓤     𝓤
  fe₁  :  funext  𝓤    (𝓤 ⁺)
  fe₂  :  funext (𝓤 ⁺) (𝓤 ⁺)
  dfe₁ : dfunext  𝓤    (𝓤 ⁺)
  dfe₂ : dfunext (𝓤 ⁺) (𝓤 ⁺)

  fe₀  = univalence-gives-funext ua
  fe₁  = univalence-gives-funext ua⁺
  fe₂  = univalence-gives-funext ua⁺
  dfe₁ = funext-gives-dfunext fe₁ fe₀
  dfe₂ = funext-gives-dfunext fe₂ fe₂

  i : is-subsingleton (is-univalent 𝓤)
  i = Π-is-subsingleton dfe₂
       (λ X → Π-is-subsingleton dfe₂
               (λ Y → being-equiv-is-a-subsingleton dfe₁ dfe₂ (Id-to-Eq X Y)))

  p : ua ≡ ua'
  p = i ua ua'
```

So if all universes are univalent then “being univalent” is a subsingleton, and hence a singleton. This hypothesis of global univalence cannot be expressed in our MLTT that only has ω many universes, because global univalence would have to live in the first universe after them. Agda does have such a universe 𝓤ω, and so we can formulate it here. There would be no problem in extending our MLTT to have such a universe if we so wished, in which case we would be able to formulate and prove:

```
global-univalence : 𝓤ω
global-univalence = ∀ 𝓤 → is-univalent 𝓤

univalence-is-a-subsingletonω : global-univalence → is-subsingleton (is-univalent 𝓤)
univalence-is-a-subsingletonω {𝓤} γ = univalence-is-a-subsingleton (γ (𝓤 ⁺))

univalence-is-a-singleton : global-univalence → is-singleton (is-univalent 𝓤)
univalence-is-a-singleton {𝓤} γ = pointed-subsingletons-are-singletons
                                    (is-univalent 𝓤)
                                    (γ 𝓤)
                                    (univalence-is-a-subsingletonω γ)
```

That the type `global-univalence` would be a subsingleton can't even be formulated in the absence of a universe of level `ω+1`, which this time Agda doesn't have.

In the absence of a universe 𝓤ω in our MLTT, we can simply have an axiom schema, consisting of ω-many axioms, stating that each universe is univalent. Then we can prove in our MLTT that the univalence property for each inverse is a (sub)singleton, with ω-many proofs (or just one schematic proof with a free variable for a universe 𝓤ₙ.).

Table of contents ⇑

`hfunext` and `vvfunext` are subsingletons

This is left as an exercise. Like univalence, the proof that these two forms of function extensional extensionality require assumptions of function extensionality at higher universes. So perhaps it is more convenient to just assume global univalence. An inconvenience is that the natural tool to use, Π-is-subsingleton, needs products with explicit arguments, but we made some of the arguments of hfunext and vvfunext implicit. One way around this problem is to define equivalent versions with the arguments explicit, and establish an equivalence between the new version and the original version.

Table of contents ⇑

More applications of function extensionality

```
being-subsingleton-is-a-subsingleton : {X : 𝓤 ̇ } → dfunext 𝓤 𝓤
                                     → is-subsingleton (is-subsingleton X)
being-subsingleton-is-a-subsingleton {𝓤} {X} fe i j = c
 where
  l : is-set X
  l = subsingletons-are-sets X i
  a : (x y : X) → i x y ≡ j x y
  a x y = l x y (i x y) (j x y)
  b : (x : X) → i x ≡ j x
  b x = fe (a x)
  c : i ≡ j
  c = fe b
```

Here is a situation where `hfunext` is what is needed:

```
Π-is-set : hfunext 𝓤 𝓥 → {X : 𝓤 ̇ } {A : X → 𝓥 ̇ }
         → ((x : X) → is-set(A x)) → is-set(Π A)
Π-is-set hfe s f g = b
 where
  a : is-subsingleton (f ∼ g)
  a p q = hfunext-gives-dfunext hfe ((λ x → s x (f x) (g x) (p x) (q x)))
  b : is-subsingleton(f ≡ g)
  b = equiv-to-subsingleton (happly f g) (hfe f g) a
```

```
being-set-is-a-subsingleton : dfunext 𝓤 𝓤 → {X : 𝓤 ̇ } → is-subsingleton (is-set X)
being-set-is-a-subsingleton {𝓤} fe {X} =
 Π-is-subsingleton fe
   (λ x → Π-is-subsingleton fe
           (λ y → being-subsingleton-is-a-subsingleton fe))
```

More generally:

```
hlevel-relation-is-subsingleton : dfunext 𝓤 𝓤
                                → (n : ℕ) (X : 𝓤 ̇ ) → is-subsingleton (X is-of-hlevel n)
hlevel-relation-is-subsingleton {𝓤} fe zero     X = being-singleton-is-a-subsingleton fe
hlevel-relation-is-subsingleton {𝓤} fe (succ n) X =
  Π-is-subsingleton fe
    (λ x → Π-is-subsingleton fe
            (λ x' → hlevel-relation-is-subsingleton {𝓤} fe n (x ≡ x')))
```

Composition of equivalences is associative:

```
●-assoc : dfunext 𝓣 (𝓤 ⊔ 𝓣) → dfunext (𝓤 ⊔ 𝓣) (𝓤 ⊔ 𝓣)
        → {X : 𝓤 ̇ } {Y : 𝓥 ̇ } {Z : 𝓦 ̇ } {T : 𝓣 ̇ }
          (α : X ≃ Y) (β : Y ≃ Z) (γ : Z ≃ T)
        → α ● (β ● γ) ≡ (α ● β) ● γ
●-assoc fe fe' (f , a) (g , b) (h , c) = ap (h ∘ g ∘ f ,_) q
 where
  d e : is-equiv (h ∘ g ∘ f)
  d = ∘-is-equiv (∘-is-equiv c b) a
  e = ∘-is-equiv c (∘-is-equiv b a)

  q : d ≡ e
  q = being-equiv-is-a-subsingleton fe fe' (h ∘ g ∘ f) _ _

≃-sym-involutive : dfunext 𝓥 (𝓤 ⊔ 𝓥) → dfunext (𝓥 ⊔ 𝓤) (𝓥 ⊔ 𝓤) →
                   {X : 𝓤 ̇ } {Y : 𝓥 ̇ } (α : X ≃ Y)
                 → ≃-sym (≃-sym α) ≡ α
≃-sym-involutive fe fe' (f , a) = to-Σ-≡ (inversion-involutive f a ,
                                          being-equiv-is-a-subsingleton fe fe' f _ _)
```

Exercise. The hlevels are closed under `Σ` and, using `hfunext`, also under `Π`. Univalence is not needed, but makes the proof easier. (If you don't use univalence, you will need to show that hlevels are closed under equivalence.)

Table of contents ⇑

Subsingleton truncation

The following is Voevosky's approach to saying that a type is inhabited in such a way that the statement of inhabitation is a subsingleton, using `funext`.

```
is-inhabited : 𝓤 ̇ → 𝓤 ⁺ ̇
is-inhabited {𝓤} X = (P : 𝓤 ̇ ) → is-subsingleton P → (X → P) → P
```

This says that if we have a function from `X` to a subsingleton `P`, then `P` must have a point. So this fails when `X=𝟘`. Considering `P=𝟘`, we conclude that `¬¬ X` if `X` is inhabited, which says that `X` is non-empty. However, in the absence of excluded middle, inhabitation is stronger than non-emptiness.

For simplicity in the formulation of the theorems, we assume global `dfunext`.

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

A type can be pointed in many ways, but inhabited in at most one way:

```
inhabitation-is-a-subsingleton : global-dfunext → (X : 𝓤 ̇ )
                               → is-subsingleton (is-inhabited X)
```

```
inhabitation-is-a-subsingleton {𝓤} fe X =
  Π-is-subsingleton (fe (𝓤 ⁺) 𝓤)
    λ P → Π-is-subsingleton (fe 𝓤 𝓤)
           (λ (s : is-subsingleton P)
                → Π-is-subsingleton (fe 𝓤 𝓤) (λ (f : X → P) → s))

pointed-is-inhabited : {X : 𝓤 ̇ } → X → is-inhabited X
pointed-is-inhabited x = λ P s f → f x

inhabited-recursion : (X P : 𝓤 ̇ ) → is-subsingleton P → (X → P) → is-inhabited X → P
inhabited-recursion X P s f φ = φ P s f
```

Although we don't necessarily have that `¬¬ P → P`, we do have that `is-inhabited P → P` if `P` is a subsingleton:

```
inhabited-gives-pointed-for-subsingletons : (P : 𝓤 ̇ )
                                          → is-subsingleton P → is-inhabited P → P
inhabited-gives-pointed-for-subsingletons P s = inhabited-recursion P P s (id P)

inhabited-functorial : global-dfunext → (X : 𝓤 ⁺ ̇ ) (Y : 𝓤 ̇ )
                     → (X → Y) → is-inhabited X → is-inhabited Y
inhabited-functorial fe X Y f = inhabited-recursion
                                  X
                                  (is-inhabited Y)
                                  (inhabitation-is-a-subsingleton fe Y)
                                  (pointed-is-inhabited ∘ f)
```

This universe assignment for functoriality is fairly restrictive, but is the only possible one.

With this notion, we can define the image of a function as follows:

```
image : {X : 𝓤 ̇ } {Y : 𝓥 ̇ } → (X → Y) → (𝓤 ⊔ 𝓥)⁺ ̇
image f = Σ \(y : codomain f) → is-inhabited (Σ \(x : domain f) → f x ≡ y)
```

Exercise. An attempt to define the image of `f` without the inhabitation predicate would be to take it to be `Σ \(y : codomain f) → Σ \(x : domain f) → f x ≡ y`. Show that this type is equivalent to `X`. This is similar to what happens in set theory: the graph of any function is isomorphic to its domain.

We can define the restriction and corestriction of a function to its image as follows:

```
restriction : {X : 𝓤 ̇ } {Y : 𝓥 ̇ } (f : X → Y)
            → image f → Y
restriction f (y , _) = y

corestriction : {X : 𝓤 ̇ } {Y : 𝓥 ̇ } (f : X → Y)
              → X → image f
corestriction f x = f x , pointed-is-inhabited (x , refl (f x))
```

And we can define the notion of surjection as follows:

```
is-surjection : {X : 𝓤 ̇ } {Y : 𝓥 ̇ } → (X → Y) → (𝓤 ⊔ 𝓥)⁺ ̇
is-surjection f = (y : codomain f) → is-inhabited (Σ \(x : domain f) → f x ≡ y)
```

Exercise. The type `(y : codomain f) → Σ \(x : domain f) → f x ≡ y` is equivalent to the type `has-section f`, which is stronger than saying that `f` is a surjection.

There are two problems with this definition of inhabitation:

- Inhabitation has values in the next universe.
- We can eliminate into subsingletons of the same universe only.

In particular, it is not possible to show that the map `X → is-inhabited X` is a surjection, or that `X → Y` gives `is-inhabited X → is-inhabited Y` for `X` and `Y` in arbitrary universes.

There are two proposed ways to solve this:

- Voevodsky works with certain resizing rules for subsingletons. At the time of writing, the (relative) consistency of the system with such rules is an open question.
- The HoTT book works with certain higher inductive types (HIT's), which are known to have models and hence to be (relatively) consistent. This is the same approach adopted by cubical type theory and cubical Agda.

A third possibility is to work with subsingleton truncations axiomatically, which is compatible with the above two proposals. We write this axiom as a record type rather than as an iterated `Σ type` for simplicity, where we use the HoTT-book notation `∥ X ∥` for the inhabitation of `X`, called the propositional, or subsingleton, truncation of `X`:

```
record subsingleton-truncations-exist : 𝓤ω where
 field
  ∥_∥                    : {𝓤 : Universe} → 𝓤 ̇ → 𝓤 ̇
  ∥∥-is-a-subsingleton : {𝓤 : Universe} {X : 𝓤 ̇ } → is-subsingleton ∥ X ∥
  ∣_∣                   : {𝓤 : Universe} {X : 𝓤 ̇ } → X → ∥ X ∥
  ∥∥-rec                 : {𝓤 𝓥 : Universe} {X : 𝓤 ̇ } {P : 𝓥 ̇ }
                         → is-subsingleton P → (X → P) → ∥ X ∥ → P
```

This is the approach we adopt in our personal Agda development.

We now assume that subsingleton truncations exist for the remainder of this file, and we `open` the assumption to make the above fields visible.

```
module basic-truncation-development
          (pt : subsingleton-truncations-exist)
          (fe : global-dfunext)
       where

  open subsingleton-truncations-exist pt public

  ∥∥-functor : {X : 𝓤 ̇ } {Y : 𝓥 ̇ } → (X → Y) → ∥ X ∥ → ∥ Y ∥
  ∥∥-functor f = ∥∥-rec ∥∥-is-a-subsingleton (λ x → ∣ f x ∣)

  ∃ : {X : 𝓤 ̇ } → (X → 𝓥 ̇ ) → 𝓤 ⊔ 𝓥 ̇
  ∃ A = ∥ Σ A ∥

  ∃! : {X : 𝓤 ̇ } → (X → 𝓥 ̇ ) → 𝓤 ⊔ 𝓥 ̇
  ∃! A = is-singleton (Σ A)

  _∨_ : 𝓤 ̇ → 𝓥 ̇ → 𝓤 ⊔ 𝓥 ̇
  A ∨ B = ∥ A + B ∥
```

The subsingleton truncation of a type and its inhabitation are logically equivalent propositions:

```
  ∥∥-agrees-with-inhabitation : (X : 𝓤 ̇ ) → ∥ X ∥ ⇔ is-inhabited X
  ∥∥-agrees-with-inhabitation X = a , b
   where
    a : ∥ X ∥ → is-inhabited X
    a = ∥∥-rec (inhabitation-is-a-subsingleton fe X) pointed-is-inhabited
    b : is-inhabited X → ∥ X ∥
    b = inhabited-recursion X ∥ X ∥ ∥∥-is-a-subsingleton ∣_∣
```

Hence they differ only in size, and when size matters don't get on the way, we can use `is-inhabited` instead of `∥_∥` if we wish.

Exercise. If `X` and `Y` are types obtained by summing `x`- and `y`-many copies of the type `𝟙`, respectively, as in `𝟙 + 𝟙 + ... + 𝟙`, where `x` and `y` are natural numbers, then `∥ X ≡ Y ∥ ≃ (x ≡ y)` and the type `X ≡ X` has `x!` elements.

Table of contents ⇑

The univalent axiom of choice

The axiom of choice says that if for every `x : X` there exists `a : A x` with `R x a`, where `R` is some given relation, then there exists a choice function `f : (x : X) → A x` with `R x (f x)` for all `x : X`. This doesn't hold in general in univalent mathematics, but it does hold in Voevodsky's simplicial model of our univalent type theory, and hence is consistent, provided:

- `X` is a set,
- `A` is a family of sets,
- the relation `R` is subsingleton valued.

```
AC : ∀ 𝓣 (X : 𝓤 ̇ ) (A : X → 𝓥 ̇ )
   → is-set X → ((x : X) → is-set (A x)) → 𝓣 ⁺ ⊔ 𝓤 ⊔ 𝓥 ̇
AC 𝓣 X A i j = (R : (x : X) → A x → 𝓣 ̇ )
               → ((x : X) (a : A x) → is-subsingleton (R x a))

               → ((x : X) → ∃ \(a : A x) → R x a)
               → ∃ \(f : (x : X) → A x) → (x : X) → R x (f x)
```

We define the axiom of choice in the universe `𝓤` to be the above with `𝓣 = 𝓤`, for all possible `X` and `A` (and `R`).

```
Choice : ∀ 𝓤 → 𝓤 ⁺ ̇
Choice 𝓤 = (X : 𝓤 ̇ ) (A : X → 𝓤 ̇ )
           (i : is-set X) (j : (x : X) → is-set (A x))
         → AC 𝓤 X A i j
```

It is important that we have the condition that `A` is a set-indexed family of sets and that the relation `R` is subsingleton valued. For arbitrary higher groupoids, it is not in general possible to perform the choice functorially.

The above is equivalent to another familiar formulation of choice, namely that a set-indexed product of non-empty sets is non-empty, where in a constructive setting we generalize `non-empty` to `inhabited` (but this generalization is immaterial, because choice implies excluded middle, and excluded middle implies that non-emptiness and inhabitation are the same notion).

```
IAC : (X : 𝓤 ̇ ) (Y : X → 𝓥 ̇ )
    → is-set X → ((x : X) → is-set (Y x)) → 𝓤 ⊔ 𝓥 ̇
IAC X Y i j = ((x : X) → ∥ Y x ∥) → ∥ Π Y ∥

IChoice : ∀ 𝓤 → 𝓤 ⁺ ̇
IChoice 𝓤 = (X : 𝓤 ̇ ) (Y : X → 𝓤 ̇ )
            (i : is-set X) (j : (x : X) → is-set (Y x))
          → IAC X Y i j
```

These two forms of choice are logically equivalent (and hence equivalent, as both are subsingletons assuming function extensionality):

```
Choice-gives-IChoice : Choice 𝓤 → IChoice 𝓤
Choice-gives-IChoice {𝓤} ac X Y i j φ = γ
 where
  R : (x : X) → Y x → 𝓤 ̇
  R x y = x ≡ x -- Any singleton type in 𝓤 will do.
  k : (x : X) (y : Y x) → is-subsingleton (R x y)
  k x y = i x x
  h : (x : X) → Y x → Σ \(y : Y x) → R x y
  h x y = (y , refl x)
  g : (x : X) → ∃ \(y : Y x) → R x y
  g x = ∥∥-functor (h x) (φ x)
  c : ∃ \(f : Π Y) → (x : X) → R x (f x)
  c = ac X Y i j R k g
```

```
    γ : ∥ Π Y ∥
    γ = ∥∥-functor pr₁ c

  IChoice-gives-Choice : IChoice 𝓤 → Choice 𝓤
  IChoice-gives-Choice {𝓤} iac X A i j R k ψ = γ
   where
    Y : X → 𝓤 ̇
    Y x = Σ \(a : A x) → R x a
    l : (x : X) → is-set (Y x)
    l x = subsets-of-sets-are-sets (A x) (R x) (j x) (k x)
    a : ∥ Π Y ∥
    a = iac X Y i l ψ
    h : Π Y → Σ \(f : Π A) → (x : X) → R x (f x)
    h g = (λ x → pr₁ (g x)) , (λ x → pr₂ (g x))
    γ : ∃ \(f : Π A) → (x : X) → R x (f x)
    γ = ∥∥-functor h a
```

For more information with Agda code, see this, which in particular has a proof that univalent choice implies univalent excluded middle.

Table of contents ⇑

Structure identity principle

For the moment, see this.

Appendix

Additional exercises

Solutions are available at the end.

Exercise. A sequence of elements of a type `X` is just a function `ℕ → X`. Use Cantor's diagonal argument to show in Agda that the type of sequences of natural numbers is uncountable. Prove a positive version and then derive a negative version from it:

```
positive-cantors-diagonal : (e : ℕ → (ℕ → ℕ)) → Σ \(α : ℕ → ℕ) → (n : ℕ) → α ≢ e n

cantors-diagonal : ¬(Σ \(e : ℕ → (ℕ → ℕ)) → (α : ℕ → ℕ) → Σ \(n : ℕ) → α ≡ e n)
```

Hint. You may wish to prove that the function `succ` has no fixed points, first.

```
𝟚-has-𝟚-automorphisms : dfunext 𝓤₀ 𝓤₀ → (𝟚 ≃ 𝟚) ≃ 𝟚
```

Now we would like to have `(𝟚 ≡ 𝟚) ≡ 𝟚` with univalence, but the problem is that the type `𝟚 ≡ 𝟚` lives in $\mathcal{U}_1$ whereas `𝟚` lives in $\mathcal{U}_0$ and so, having different types, can't be compared for equality.

Universes are not cumulative in Agda, in the sense that from `X : 𝓤` we would get `X : 𝓤⁺` or `X : 𝓤 ⊔ 𝓥`. The usual approach is to consider embeddings of universes into larger universes:

```
data Up {𝓤 : Universe} (𝓥 : Universe) (X : 𝓤 ̇ ) : 𝓤 ⊔ 𝓥 ̇  where
 up : X → Up 𝓥 X
```

This gives an embedding `Up 𝓥 : 𝓤 ̇ → 𝓤 ⊔ 𝓥 ̇` and an embedding `up : X → Up 𝓥 X`. Prove the following:

```
Up-induction : ∀ {𝓤} 𝓥 (X : 𝓤 ̇ ) (A : Up 𝓥 X → 𝓦 ̇ )
             → ((x : X) → A (up x))
             → ((l : Up 𝓥 X) → A l)

Up-recursion : ∀ {𝓤} 𝓥 {X : 𝓤 ̇ } {B : 𝓦 ̇ }
             → (X → B) → Up 𝓥 X → B
```

```
down : {X : 𝓤 ̇ } → Up 𝓥 X → X

down-up : {X : 𝓤 ̇ } (x : X) → down {𝓤} {𝓥} (up x) ≡ x

up-down : {X : 𝓤 ̇ } (l : Up 𝓥 X) → up (down l) ≡ l

Up-≃ : (X : 𝓤 ̇ ) → Up 𝓥 X ≃ X

Up-left-≃ : (X : 𝓤 ̇ ) (Y : 𝓥 ̇ ) → X ≃ Y → Up 𝓦 X ≃ Y

ap-Up-≃ : (X : 𝓤 ̇ ) (Y : 𝓥 ̇ ) → X ≃ Y → Up 𝓦 X ≃ Up 𝓣 Y
```

With this we can show:

```
uptwo : is-univalent 𝓤₀
      → is-univalent 𝓤₁
      → (𝟚 ≡ 𝟚) ≡ Up 𝓤₁ 𝟚
```

We now discuss alternative formulations of the principle of excluded middle.

```
DNE : ∀ 𝓤 → 𝓤 ⁺ ̇
DNE 𝓤 = (P : 𝓤 ̇ ) → is-subsingleton P → ¬¬ P → P

neg-is-subsingleton : dfunext 𝓤 𝓤₀ → (X : 𝓤 ̇ ) → is-subsingleton (¬ X)

emsanity : dfunext 𝓤 𝓤₀ → (P : 𝓤 ̇ ) → is-subsingleton P → is-subsingleton (P + ¬ P)

ne : (X : 𝓤 ̇ ) → ¬¬(X + ¬ X)

DNE-gives-EM : dfunext 𝓤 𝓤₀ → DNE 𝓤 → EM 𝓤

EM-gives-DNE : EM 𝓤 → DNE 𝓤
```

The following says that, under univalence, excluded middle holds if and only if every subsingleton is the negation of some type (maybe you want to formulate and prove this - no solution given).

```
SN : ∀ 𝓤 → 𝓤 ⁺ ̇
SN 𝓤 = (P : 𝓤 ̇ ) → is-subsingleton P → Σ \(X : 𝓤 ̇ ) → P ⇔ ¬ X

SN-gives-DNE : SN 𝓤 → DNE 𝓤

DNE-gives-SN : DNE 𝓤 → SN 𝓤
```

Table of contents ⇑

Solutions to additional exercises

Spoiler alert.

```
succ-no-fixed-point : (n : ℕ) → succ n ≢ n
succ-no-fixed-point 0        = positive-not-zero 0
succ-no-fixed-point (succ n) = γ
 where
  IH : succ n ≢ n
  IH = succ-no-fixed-point n
  γ : succ (succ n) ≢ succ n
  γ p = IH (succ-lc p)

positive-cantors-diagonal e = (α , φ)
 where
  α : ℕ → ℕ
  α n = succ(e n n)
  φ : (n : ℕ) → α ≢ e n
  φ n p = succ-no-fixed-point (e n n) q
   where
```

```
    q = succ (e n n)  ≡⟨ refl (α n) ⟩
        α n           ≡⟨ ap (λ - → - n) p ⟩
        e n n         ∎

cantors-diagonal (e , γ) = c
 where
  α : ℕ → ℕ
  α = pr₁ (positive-cantors-diagonal e)
  φ : (n : ℕ) → α ≢ e n
  φ = pr₂ (positive-cantors-diagonal e)
  b : Σ \(n : ℕ) → α ≡ e n
  b = γ α
  c : 𝟘
  c = φ (pr₁ b) (pr₂ b)

𝟚-has-𝟚-automorphisms fe =
 (f , invertibles-are-equivs f (g , η , ε))
 where
  f : (𝟚 ≃ 𝟚) → 𝟚
  f (h , e) = h ₀
  g : 𝟚 → (𝟚 ≃ 𝟚)
  g ₀ = id , id-is-equiv 𝟚
  g ₁ = swap₂ , swap₂-is-equiv
  η : (e : 𝟚 ≃ 𝟚) → g (f e) ≡ e
  η (h , e) = γ (h ₀) (h ₁) (refl (h ₀)) (refl (h ₁))
   where
    γ : (m n : 𝟚) → h ₀ ≡ m → h ₁ ≡ n → g (h ₀) ≡ (h , e)
    γ ₀ ₀ p q = !𝟘 (g (h ₀) ≡ (h , e))
                   (₁-is-not-₀ (equivs-are-lc h e (h ₁ ≡⟨ q ⟩
                                                   ₀   ≡⟨ p ⁻¹ ⟩
                                                   h ₀ ∎)))
    γ ₀ ₁ p q = to-Σ-≡ (fe (𝟚-induction (λ n → pr₁ (g (h ₀)) n ≡ h n)
                             (pr₁ (g (h ₀)) ₀ ≡⟨ ap (λ - → pr₁ (g -) ₀) p ⟩
                              pr₁ (g ₀) ₀     ≡⟨ refl ₀ ⟩
                              ₀               ≡⟨ p ⁻¹ ⟩
                              h ₀             ∎)
                             (pr₁ (g (h ₀)) ₁ ≡⟨ ap (λ - → pr₁ (g -) ₁) p ⟩
                              pr₁ (g ₀) ₁     ≡⟨ refl ₁ ⟩
                              ₁               ≡⟨ q ⁻¹ ⟩
                              h ₁             ∎)),
                       being-equiv-is-a-subsingleton fe fe _ _ e)
    γ ₁ ₀ p q = to-Σ-≡ (fe (𝟚-induction (λ n → pr₁ (g (h ₀)) n ≡ h n)
                             (pr₁ (g (h ₀)) ₀ ≡⟨ ap (λ - → pr₁ (g -) ₀) p ⟩
                              pr₁ (g ₁) ₀     ≡⟨ refl ₁ ⟩
                              ₁               ≡⟨ p ⁻¹ ⟩
                              h ₀             ∎)
                             (pr₁ (g (h ₀)) ₁ ≡⟨ ap (λ - → pr₁ (g -) ₁) p ⟩
                              pr₁ (g ₁) ₁     ≡⟨ refl ₀ ⟩
                              ₀               ≡⟨ q ⁻¹ ⟩
                              h ₁             ∎)),
                       being-equiv-is-a-subsingleton fe fe _ _ e)
    γ ₁ ₁ p q = !𝟘 (g (h ₀) ≡ (h , e))
                   (₁-is-not-₀ (equivs-are-lc h e (h ₁ ≡⟨ q ⟩
                                                   ₁   ≡⟨ p ⁻¹ ⟩
                                                   h ₀ ∎)))

  ε : (n : 𝟚) → f (g n) ≡ n
  ε ₀ = refl ₀
  ε ₁ = refl ₁

Up-induction 𝓥 X A φ (up x) = φ x

Up-recursion 𝓥 {X} {B} = Up-induction 𝓥 X (λ _ → B)

down = Up-recursion _ id

down-up = refl
```

```
Up-≃ {𝓤} {𝓥} X = down {𝓤} {𝓥} , invertibles-are-equivs down (up , up-down , down-up {𝓤} {𝓥})

up-down {𝓤} {𝓥} {X} = Up-induction 𝓥 X
                       (λ l → up (down l) ≡ l)
                       (λ x → up (down {𝓤} {𝓥} (up x)) ≡⟨ ap up (down-up {𝓤} {𝓥}x) ⟩
                              up x                      ∎)

Up-left-≃ {𝓤} {𝓥} {𝓦} X Y e = Up 𝓦 X ≃⟨ Up-≃ X ⟩
                               X      ≃⟨ e ⟩
                               Y      ■

ap-Up-≃ {𝓤} {𝓥} {𝓦} {𝓣} X Y e = Up 𝓦 X  ≃⟨ Up-left-≃ X Y e ⟩
                                 Y       ≃⟨ ≃-sym (Up-≃ Y) ⟩
                                 Up 𝓣 Y  ■

uptwo ua₀ ua₁ = Eq-to-Id ua₁ (𝟚 ≡ 𝟚) (Up 𝓤₁ 𝟚) e
 where
  e = (𝟚 ≡ 𝟚) ≃⟨ Id-to-Eq 𝟚 𝟚 , ua₀ 𝟚 𝟚 ⟩
      (𝟚 ≃ 𝟚) ≃⟨ 𝟚-has-𝟚-automorphisms (univalence-gives-dfunext ua₀) ⟩
      𝟚       ≃⟨ ≃-sym (Up-≃ 𝟚) ⟩
      Up 𝓤₁ 𝟚 ■

neg-is-subsingleton fe X f g = fe (λ x → !𝟘 (f x ≡ g x) (f x))

emsanity fe P i (inl p) (inl q) = ap inl (i p q)
emsanity fe P i (inl p) (inr n) = !𝟘 (inl p ≡ inr n) (n p)
emsanity fe P i (inr m) (inl q) = !𝟘 (inr m ≡ inl q) (m q)
emsanity fe P i (inr m) (inr n) = ap inr (neg-is-subsingleton fe P m n)

ne X = λ (f : ¬(X + ¬ X)) → f (inr (λ (x : X) → f (inl x)))

DNE-gives-EM fe dne P i = dne (P + ¬ P) (emsanity fe P i) (ne P)

EM-gives-DNE em P i = γ (em P i)
 where
  γ : P + ¬ P → ¬¬ P → P
  γ (inl p) φ = p
  γ (inr n) φ = !𝟘 P (φ n)

SN-gives-DNE {𝓤} sn P i = h
 where
  X : 𝓤 ̇
  X = pr₁ (sn P i)
  f : P → ¬ X
  f = pr₁ (pr₂ (sn P i))
  g : ¬ X → P
  g = pr₂ (pr₂ (sn P i))
  f' : ¬¬ P → ¬(¬¬ X)
  f' = contrapositive (contrapositive f)
  h : ¬¬ P → P
  h = g ∘ tno X ∘ f'
  h' : ¬¬ P → P
  h' φ = g (λ (x : X) → φ (λ (p : P) → f p x))

DNE-gives-SN dne P i = (¬ P) , dni P , dne P i
```

Table of contents ⇑

Introduction to Univalent Foundations of Mathematics with Agda

Table of contents ⇑

Universes

We define our notation for type universes used in these notes, which is different from the standard Agda notation, but closer to the standard notation in HoTT/UF.

Readers unfamiliar with Agda should probably try to understand this only after doing some MLTT in Agda and HoTT/UF in Agda.

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

module Universes where

open import Agda.Primitive public
 renaming (
            Level to Universe -- We speak of universes rather than of levels.
          ; lzero to 𝓤₀       -- Our first universe is called 𝓤₀
          ; lsuc to _⁺        -- The universe after 𝓤 is 𝓤 ⁺
          ; Setω to 𝓤ω        -- There is a universe 𝓤ω strictly above 𝓤₀, 𝓤₁, ⋯ , 𝓤ₙ, ⋯
          )
 using    (_⊔_)               -- Least upper bound of two universes, e.g. 𝓤₀ ⊔ 𝓤₁ is 𝓤₁
```

The elements of `Universe` are universe names. Given a name 𝓤, the universe itself will be written 𝓤 ̇ in these notes, with a deliberately almost invisible superscript dot.

We actually need to define this notation, because traditionally in Agda if one uses `ℓ` for a universe level, then `Set ℓ` is the type of types of level `ℓ`. However, this notation is not good for univalent foundations, because not all types are sets. Also the terminology “level” is not good, because the hlevels in univalent type theory refer to the complexity of equality rather than size.

The following should be the only use of the Agda keyword `Set` in these notes.

```
Type = λ ℓ → Set ℓ

_̇   : (𝓤 : Universe) → Type (𝓤 ⁺)

𝓤 ̇  = Type 𝓤
```

This says that given the universe level 𝓤, we get the type universe 𝓤 ̇, which lives in the next next type universe universe 𝓤 ⁺. So the superscript dot notation is just a (postfix) synonym for (prefix) `Type`, which is just a synonym for `Set`, which means type in Agda.

We name a few of the initial universes:

```
𝓤₁ = 𝓤₀ ⁺
𝓤₂ = 𝓤₁ ⁺
𝓤₃ = 𝓤₂ ⁺
```

The following is sometimes useful:

```
universe-of : {𝓤 : Universe} (X : 𝓤 ̇ ) → Universe
universe-of {𝓤} X = 𝓤
```

Fixities:

```
infix  1 _̇
```

Table of contents ⇑