Abstract. We introduce Voevodsky’s univalent foundations and univalent mathematics, and explain how to develop them with the computer system Agda, which is 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 for manipulating mathematical objects. 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.

Compared to most expositions of the subject, we work with explicit universe levels.

We also fully discuss and emphasize that non-constructive classical axioms can be assumed consistently in univalent 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 issues or pull requests by students to fix typos or mistakes and clarify ambiguities are welcome. There is also a pdf version with internal links to sections and Agda definitions, which is automatically generated from the html version. And a pdf version is also available at the arxiv, updated from time to time.

These notes were originally developed for the Midlands Graduate School 2019. They will evolve for a while.

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 \( \mathbb{N} \) 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 $x \equiv y$ 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 $\mathbb{N}$, 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 the notion of monoid isomorphism. Largeness and smallness are taken as relative concepts, with type universes incorporated in the theory to account for the size distinction.

In particular, properties of monoids are automatically invariant under isomorphism, properties of categories are automatically invariant under equivalence, and so on.

Voevodsky’s way to achieve 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 $n=\infty$.
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, half of these notes begin without the univalence axiom.

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 these 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).

We will not assume univalence and truncation globally, so that the students can see clearly when they 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 terminology 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.

In 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:
Regarding the genesis of the subject:

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

Voevodsky says that he was influenced by Makkai’s thinking:

- FOLDS.
- The theory of abstract sets based on first-order logic with dependent types.

An important foundational reference, by Steve Awodey and Michael A. Warren, is

- Homotopy theoretic models of identity types.

Additional expository material:

- An introduction to univalent foundations for mathematicians, a paper at the Bulletin of the AMS by Dan Grayson.
- Voevodsky’s Memorial talk by Dan Grayson.
- Univalent foundations - an introduction by Benedikt Ahrens.
- Introduction to Homotopy Type Theory by Egbert Rijke.
- A course on homotopy (type) theory by Andrej Bauer and Jaka Smrekar.
- 15-819 Homotopy Type Theory by Bob Harper.
- Homotopy type theory: the logic of space by Mike Shulman.
- Logic in univalent type theory by Martin Escardo.

More references as clickable links are given in the course of the notes.

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 omitting 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

1. Front matter
   1. Title, abstract, keywords and about
   2. Introduction
   3. Homotopy type theory
   4. General references
   5. Choice of material
2. MLTT in Agda
   1. A spartan Martin-Löf type theory (MLTT)
   2. What is Agda?
   3. Getting started with Agda
   4. Type universes $\mathcal{U}, \mathcal{V}, \mathcal{W}$
   5. The one-element type $\mathbb{1}$
   6. The empty type $\emptyset$
   7. The type $\mathbb{N}$ of natural numbers
8. The binary sum type constructor \(_+_
9. \Sigma\) types
10. \(\Pi\) types
11. The identity type former \(\text{Id}\), also written \(_\equiv_
12. Basic constructions with the identity type
13. Reasoning with negation
14. Example: formulation of the twin-prime conjecture
15. Remaining Peano axioms and basic arithmetic

3. Univalent Mathematics in Agda
   1. Our univalent type theory
   2.Singletons, subsingletons and sets
      1. Singleton (or contractible) types
      2. Subsingletons (or propositions or truth values)
      3. Sets (or 0-groupoids)
   3. Univalent excluded middle
   4. The types of magmas and monoids
   5. The identity type in univalent mathematics
   6. Identifications that depend on identifications
   7. Equality in \(\Sigma\) types
   8. Voevodsky’s notion of hlevel
      1. Hedberg’s Theorem
      2. A characterization of sets
      3. Subsingletons are sets
      4. The types of hlevel 1 are the subsingletons
      5. The types of hlevel 2 are the sets
      6. The hlevels are upper closed
      7. \(\mathbb{N}\) and \(\mathbb{Z}\) are sets
   9. Retracts
10. Voevodsky’s notion of type equivalence
11. Voevodsky’s univalence axiom
12. Example of a type that is not a set under univalence
13. Exercises
      1. Formulations
      2. Solutions
14. A characterization of univalence
15. Equivalence induction
16. Half adjoint equivalences
17. Function extensionality from univalence
18. Variations of function extensionality and their logical equivalence
19. Universes are map classifiers
20. The univalence axiom is a (sub)singleton type
21. Unique existence in univalent mathematics
22. Universal property of the natural numbers
23. More consequences of function extensionality
24. Propositional extensionality and the powerset
25. Some constructions with types of equivalences
26. Type embeddings
27. The Yoneda Lemma for types
28. What is a function?
29. Partial functions
30. Universe lifting
31. The subtype classifier and other classifiers
32. Magma equivalences
33. Equality of mathematical structures
      1. A structure identity principle for a standard notion of structure
      2. \(\infty\)-Magmas
      3. Adding axioms
      4. Magmas
      5. Pointed types
      6. Combining two mathematical structures
Table of contents

1. Pointed ∞-magmas
2. Monoids
3. Associative ∞-magmas
4. Groups
5. The slice type
6. Subgroups
7. Rings
8. Metric spaces, graphs and ordered structures
9. Topological spaces
10. Selection spaces
11. A contrived example
12. Functor algebras
13. Type-valued preorders
14. Categories

34. Subsingleton truncation
   1. Voevodsky’s approach to subsingleton truncation
   2. An axiomatic approach
   3. Disjunction and existence
   4. Images and surjections
   5. A characterization of equivalences
   6. Exiting truncations
   7. Equality of Noetherian local rings

35. Choice in univalent mathematics
   1. Unique choice
   2. Univalent choice
   3. A second formulation of univalent choice
   4. A third formulation of univalent choice
   5. Univalent choice gives excluded middle
   6. Global choice

36. Propositional resizing, truncation and the powerset
   1. Propositional resizing
   2. Excluded middle gives propositional resizing
   3. The propositional resizing axiom is a subsingleton
   4. Propositional impredicativity
   5. Propositional resizing gives subsingleton truncation
   6. The powerset in the presence of propositional resizing
   7. Topological spaces in the presence of propositional resizing

37. Quotients
38. Summary of consistent axioms for univalent mathematics

4. Appendix
   1. Solutions to some exercises
   2. Additional exercises
   3. Solutions to additional exercises
   4. Operator fixities and precedences
   5. Agda files automatically extracted from these notes
   6. The sources for these notes
   7. License

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
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 \(\emptyset\).
- A one-element type \(\mathbb{1}\).
- A type of \(\mathbb{N}\) natural numbers.
- Type formers + (binary sum), \(\pi\) (product), \(\Sigma\) (sum), \(\text{Id}\) (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:

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

module HoTT-UF-Agda where
```

- The option --without-K disables Streicher’s \(\kappa\) 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.
Type universes

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

1. One use of universes is to define families of types indexed by a type $X$ as functions $X \to \mathcal{U}$.

2. Such a function is sometimes seen as a property of elements of $X$.

3. 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

$$\mathcal{U}_0, \mathcal{U}_1, \mathcal{U}_2, \mathcal{U}_3, \ldots$$

In practice, the first one, two or three universes suffice, but it will be easier to formulate definitions, constructions and theorems in full generality, to avoid making universe choices before knowing how they are going to be applied.

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

$$\mathbb{1} : \mathcal{U}_0^.$$  

where $\mathbb{1}$ 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.

$$\mathcal{U}_0^ : \mathcal{U}_1^,$$

$$\mathcal{U}_1^ : \mathcal{U}_2^,$$

$$\mathcal{U}_2^ : \mathcal{U}_3^,$$

$$\vdots$$

The assumption that $\mathcal{U}_0 \in \mathcal{U}_0$ 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

$$\mathcal{U}^+$$

its successor universe. For example, if $\mathcal{U}$ is $\mathcal{U}_0$ then $\mathcal{U}^+$ is $\mathcal{U}_1$. According to the above discussion, we have

$$\mathcal{U}^+ : \mathcal{U}_1^+.$$  

The least upper bound of two universes $\mathcal{U}$ and $\mathcal{V}$ is written

$$\mathcal{U} \sqcup \mathcal{V}.$$  

For example, if $\mathcal{U}$ is $\mathcal{U}_0$ and $\mathcal{V}$ is $\mathcal{U}_3$, then $\mathcal{U} \sqcup \mathcal{V}$ is $\mathcal{U}_4$.

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.
open import Universes public

The keyword public makes the contents of the file Universes available to importers of our module HoTT-UF-Agda.

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

variable

$\mathcal{U} \uplus \mathcal{V} \uplus \mathcal{W} \uplus \mathcal{T}$ : Universe

In some type theories, the universes are cumulative “on the nose”, in the sense that from $X : \mathcal{U}$ we derive that $X : \mathcal{U} \uplus \mathcal{V}$. We will instead have an embedding $\mathcal{U} \rightarrow \mathcal{U} \uplus \mathcal{V}$ of universes into larger universes.

### The one-element type $\mathbb{1}$

We place it in the first universe, and we name its unique element “$\star$”. We use the data declaration in Agda to introduce it:

```agda
data $\mathbb{1}$ : $\mathcal{U}_0$ where
  $\star$ : $\mathbb{1}$
```

It is important that the point $\star$ lives in the type $\mathbb{1}$ 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 “$\star$”. (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 $\mathbb{1}$ satisfy a given property $A$.

1. The property is a function $A : \mathbb{1} \rightarrow \mathcal{U}$ for some universe $\mathcal{U}$.

2. 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.

3. In MLTT, mathematical statements are types, such as

   $\prod A : \mathbb{1} \rightarrow \mathcal{U}, ~ A \star \rightarrow \prod x : \mathbb{1}, ~ A \ x.$

4. We read this in natural language as “for any given property $A$ of elements of the type $\mathbb{1}$, if $A \star$ holds, then it follows that $A \ x$ holds for all $x : \mathbb{1}$”.

5. In Agda, the above type is written as

   $$(A : \mathbb{1} \rightarrow \mathcal{U}) \rightarrow A \star \rightarrow (x : \mathbb{1}) \rightarrow A \ x.$$

   This is the type of functions with three arguments $A : \mathbb{1} \rightarrow \mathcal{U}$ and $a : A \star$ and $x : \mathbb{1}$, with values in the type $A \ x$.

6. 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 $\mathbb{1}$-induction.

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

```agda
$\mathbb{1}$-induction : (A : $\mathbb{1}$ → $\mathcal{U}$) → A $\star$ → (x : $\mathbb{1}$) → A x
$\mathbb{1}$-induction A a $\star$ = a
```

The universe $\mathcal{U}$ is arbitrary, and Agda knows $\mathcal{U}$ is a universe variable because we said so above.

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

A particular case of $\mathbb{1}$-induction occurs when the family $A$ is constant with value $B$, which can be written variously as

$$A \ x = B$$
or

\[ A = \lambda (x : \mathbb{I}) \to B, \]

or

\[ A = \lambda x \to B \]

if we want Agda to figure out the type of \( x \) by itself, or

\[ A = \lambda _ \to B \]

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

\[ x \mapsto B \] (\( x \) is mapped to \( B \))

so that the above amount to \( A = (x \mapsto B) \).

Given a type \( B \) and a point \( b : B \), we construct the function \( \mathbb{I} \to B \) that maps any given \( x : \mathbb{I} \) to \( b \).

\[ 1\text{-recursion} : (B : \mathcal{U}) \to B \to (\mathbb{I} \to B) \]

\[ 1\text{-recursion} B b x = 1\text{-induction} (\lambda _ \to B) b x \]

The above expression \( B \to (\mathbb{I} \to B) \) can be written as \( B \to \mathbb{I} \to B \), omitting the brackets, as the function-type formation symbol \( \to \) is taken to be right associative.

Not all types have to be seen as mathematical statements (for example the type \( \mathbb{N} \) 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 \( \mathbb{I} \) is the type encoding the truth value \( true \).

The unique function to \( \mathbb{I} \) will be named \( !\mathbb{I} \). 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.

\[ !\mathbb{I}' : (X : \mathcal{U}) \to X \to \mathbb{I} \]
\[ !\mathbb{I}' X x = * \]

\[ !\mathbb{I} : \{X : \mathcal{U}\} \to X \to \mathbb{I} \]
\[ !\mathbb{I} x = * \]

This means that when we write

\[ !\mathbb{I} x \]

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

\[ !\mathbb{I} \{\mathcal{U}\} \{X\} x. \]

This is because \( \mathcal{U} \) 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 \( \mathcal{U} \) too. Occasionally, people define variants of a function with different choices of “implicitness”, as above.

Table of contents  

The empty type \( \emptyset \)

It is defined like \( \mathbb{I} \), except that no elements are listed for it:

\[ \text{data } \emptyset : \mathcal{U}' \text{ where} \]
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 \textit{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.

\[
\begin{align*}
\emptyset\text{-induction} & : (A : \emptyset \rightarrow \text{U}) \rightarrow (x : \emptyset) \rightarrow A \ x \\
\emptyset\text{-induction} & \ A ()
\end{align*}
\]

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 \textit{vacuously true}. The unique function from \emptyset to any type is a particular case of \emptyset\text{-induction}.

\[
\begin{align*}
\emptyset\text{-recursion} & : (A : \text{U}^\prime) \rightarrow \emptyset \rightarrow A \\
\emptyset\text{-recursion} & \ A a = \emptyset\text{-induction} (\lambda _\rightarrow A) a
\end{align*}
\]

We will use the following categorical notation for \emptyset\text{-recursion}:

\[
\begin{align*}
\!\emptyset : (A : \text{U}^\prime) & \rightarrow \emptyset \rightarrow A \\
\!\emptyset = \emptyset\text{-recursion}
\end{align*}
\]

We give the two names \textit{is-empty} and \textit{¬} to the same function now:

\[
\begin{align*}
is\text{-empty} & : \text{U}^\prime \rightarrow \text{U}^\prime \\
is\text{-empty} & \ X = X \rightarrow \emptyset \\
\neg & : \text{U}^\prime \rightarrow \text{U}^\prime \\
\neg & \ X = X \rightarrow \emptyset
\end{align*}
\]

This says that a type is empty precisely when we have a function to the empty type. Assuming \textit{univalence}, once we have defined the identity type former \_≡_, we will be able to prove that \((\text{is-empty} \ X) \equiv (X \equiv \emptyset)\), where \(X \equiv \emptyset\) is the type of bijections, or equivalences, from \(x\) to \(\emptyset\). We will also be able to prove things like \((2 + 2 \equiv 5) \equiv \emptyset\) and \((2 + 2 \equiv 4) \equiv 1\).

This is for \textit{numbers}. If we define \textit{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 \(2 + 2 \equiv 4\) is a type with \(4!\) elements, which is the \textit{number of permutations} of a set with four elements, rather than a truth value \(\emptyset\) or \(1\), as a consequence of the univalence axiom. That is, we will have \((2 + 2 \equiv 4) \simeq (4 + 4 + 4 + 4 + 4 + 4)\), so that the type identity \(2 + 2 \equiv 4\) holds in \textit{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 \(X \equiv Y\) for types \(X\) and \(Y\) (inhabiting the same universe) is allowed.

When we view \(\emptyset\) as \textit{false}, we can read the definition of the \textit{negation} \(\neg X\) as saying that “\(X\) implies \textit{false}”. With univalence we will be able to show that \(\text{(false \rightarrow true) \equiv true}\), which amounts to \((\emptyset \rightarrow 1) \equiv 1\), which in turn says that there is precisely one function \(\emptyset \rightarrow 1\), namely the (vacuous) function.

\textbf{Table of contents} \textup{⇑}

\textbf{The type \(\mathbb{N}\) of natural numbers}

The definition is similar but not quite the same as the one via \textbf{Peano Axioms}.

We stipulate an element \textbf{zero} : \(\mathbb{N}\) and a successor function \textbf{succ} : \(\mathbb{N} \rightarrow \mathbb{N}\), and then define induction. Once we have defined the identity type former \_≡_, we will \textit{prove} the other peano axioms.

\[
\begin{align*}
data \mathbb{N} : \text{U}_0^\prime \ where \\
\textbf{zero} & : \mathbb{N} \\
\textbf{succ} & : \mathbb{N} \rightarrow \mathbb{N}
\end{align*}
\]

In general, declarations with \textbf{data} are inductive definitions. To write the number 5, we have to write

\[
\textbf{succ} (\textbf{succ} (\textbf{succ} (\textbf{succ} (\textbf{succ} \textbf{zero}))))
\]

We can use the following Agda \textit{built-in} to be able to just write 5 as a shorthand:

\[
{-# \ BUILTIN \ NATURAL \ \mathbb{N} \ #-}
\]
Apart from this notational effect, the above declaration doesn’t play any role in the Agda development of these lecture notes.

In the following, the type family $A$ can be seen as playing the role of a property of elements of $\mathbb{N}$, 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.

$$\mathbb{N} \text{-induction} : (A : \mathbb{N} \rightarrow \mathcal{U}) \\
\quad \Rightarrow A \, \emptyset \\
\quad \Rightarrow ((n : \mathbb{N}) \rightarrow A \, n \rightarrow A \, (\text{succ} \, n)) \\
\quad \Rightarrow (n : \mathbb{N}) \rightarrow A \, n$$

$$\text{N-induction } A \, a_0 \, f = h$$

where

- $h : (n : \mathbb{N}) \rightarrow A \, n$
- $h \, \emptyset = a_0$
- $h \, (\text{succ} \, n) = f \, n \, (h \, n)$

The definition of $\text{N-induction}$ is itself by induction. It says that given a point

$$a_0 : A \, \emptyset$$

and a function

$$f : (n : \mathbb{N}) \rightarrow A \, n \rightarrow A \, (\text{succ} \, n),$$

in order to calculate an element of $A \, n$ for a given $n : \mathbb{N}$, we just calculate $h \, n$, that is,

$$f \, n \, (f \, (n-1) \, (f \, (n-2) \, \ldots \, (f \, \emptyset \, a_0) \ldots))$$

So the principle of induction is a construction that given a base case $a_0 : A \, \emptyset$, an induction step $f : (n : \mathbb{N}) \rightarrow A \, n \rightarrow A \, (\text{succ} \, n)$ and a number $n$, says how to get an element of the type $A \, n$ by primitive recursion.

Notice also that $\text{N-induction}$ is the dependently typed version of primitive recursion, where the non-dependently typed version is

$$\text{N-recursion} : (X : \mathcal{U}) \rightarrow X \rightarrow (X \rightarrow X) \rightarrow X$$

$$\text{N-recursion } X = \text{N-induction} (\lambda \_ \rightarrow X)$$

The following special case occurs often (and is related to the fact that $\mathbb{N}$ is the initial algebra of the functor $\mathbb{I} + (-)$):

$$\text{N-iteration} : (X : \mathcal{U}) \rightarrow X \rightarrow (X \rightarrow X) \rightarrow X$$

$$\text{N-iteration } X \times f = \text{N-recursion } X \times (\lambda \_ X \rightarrow f \, X)$$

Agda checks that any recursive definition as above is well founded, with recursive invocations 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 $\text{N-induction}$ functional to define everything else with the natural numbers. But Agda allows us to define functions by structural recursion, like we defined $\text{N-induction}$.

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 $\times$ 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. $\text{Arithmetic._+}_\_$ in the following example.

module Arithmetic where

_+_ _*_ : \mathbb{N} \rightarrow \mathbb{N} \rightarrow \mathbb{N}
\( x + 0 = x \)
\( x + \text{succ} \ y = \text{succ} (x + y) \)
\( x \times 0 = 0 \)
\( x \times \text{succ} \ y = x + x \times y \)

infixl 20 _+_  
infixl 21 _\times_

The above “fixity” declarations 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 \texttt{N-induction} on the second argument \( y \), via \texttt{N-iteration}:

\begin{verbatim}
module Arithmetic' where
  _+_ x_ : N \to N \to N
  x + y = h y
  where
  h : N \to N
  h = N-iteration N x succ

  _\times_ x_ : N \to N \to N
  x \times y = h y
  where
  h : N \to N
  h = N-iteration N 0 (x +_)

  infixl 20 _+_  
infixl 21 _\times_
\end{verbatim}

Here the expression “\( x +_ \)” stands for the function \( N \to N \) 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.

\textbf{Exercise.} Write it using \texttt{N-induction}, recursion or iteration, as appropriate.

\begin{verbatim}
module ℕ-order where
  _≤_ _≥_ : ℕ \to ℕ \to Ω₀
  0 ≤ y = ⊤
  succ x ≤ 0 = ⊥
  succ x ≤ succ y = x ≤ y

  x ≥ y = y ≤ x

  infixl 10 _≤_  
infixl 10 _≥_
\end{verbatim}

\textbf{Exercise.} After learning \( Σ \) and \_≡_ explained below, prove that

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

Later, after learning \texttt{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 (and this is called \texttt{propositional extensionality}).

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 \textit{disjoint} sum of two types \(X\) and \(Y\). The elements of the type \(X + Y\) are stipulated to be of the forms

\[
\text{inl } x \quad \text{and} \quad \text{inr } y
\]

with \(x : X\) and \(y : Y\). If \(X : \mathcal{U}\) and \(Y : \mathcal{V}\), we stipulate that \(X + Y : \mathcal{U} \sqcup \mathcal{V}\), where \(\mathcal{U} \sqcup \mathcal{V}\) is the \textit{least upper bound} of the two universes \(\mathcal{U}\) and \(\mathcal{V}\). In Agda we can define this as follows.

\[
data \_+_\{\mathcal{U} \mathcal{V}\} \{X : \mathcal{U} \rhd\} \{Y : \mathcal{V} \rhd\} : \mathcal{U} \sqcup \mathcal{V} \rhd where\inl : X \rightarrow X + Y\inr : Y \rightarrow X + Y
\]

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

\[
\begin{align*}
\text{+_induction} : \langle X : \mathcal{U} \rhd \rangle \langle Y : \mathcal{V} \rhd \rangle \langle A : X + Y \rightarrow \mathcal{W} \rhd \rangle
&\quad \rightarrow \langle (x : X) \rightarrow A (\text{inl } x) \rangle \\
&\quad \rightarrow \langle (y : Y) \rightarrow A (\text{inr } y) \rangle \\
&\quad \rightarrow \langle z : X + Y \rightarrow A z \rangle
\end{align*}
\]

\[
\text{+_induction} A f g (\text{inl } x) = f x
\]

\[
\text{+_induction} A f g (\text{inr } y) = g y
\]

\[
\begin{align*}
\text{+_recursion} : \langle X : \mathcal{U} \rhd \rangle \langle Y : \mathcal{V} \rhd \rangle \langle A : \mathcal{W} \rhd \rangle \rightarrow \langle X \rightarrow A \rangle \rightarrow \langle Y \rightarrow A \rangle \rightarrow X + Y \rightarrow A
\end{align*}
\]

\[
\begin{align*}
\text{+_recursion} \{\mathcal{U}\} \{\mathcal{V}\} \{\mathcal{W}\} \{X\} \{Y\} \{A\} = \text{+_induction} (\lambda \_ \rightarrow A)
\end{align*}
\]

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 \textit{truncation} \(\| A + B \|\).

But also \(_+_

is used to construct mathematical objects. For example, we can define a two-point type:

\[
\ast : \mathcal{U}_0 \rhd
\ast = \mathbb{1} + \mathbb{1}
\]

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

\[
\begin{align*}
pattern \_0 = \text{inl} * \\
pattern \_1 = \text{inr} *
\end{align*}
\]

We can define induction on \(\ast\) directly by pattern matching:

\[
\begin{align*}
\ast\text{-induction} : \langle A : \ast \rightarrow \mathcal{U} \rhd \rangle \rightarrow \langle A _0 : A \rangle \rightarrow \langle A _1 : A \rangle \rightarrow \langle (n : \ast) \rightarrow A n \rangle
\end{align*}
\]

\[
\begin{align*}
\ast\text{-induction} A a_0 a_1 = A a_0
\end{align*}
\]

Or we can prove it by induction on \(_+_

and \(\mathbb{1}\):

\[
\begin{align*}
\ast\text{-induction}' : \langle A : \ast \rightarrow \mathcal{U} \rhd \rangle \rightarrow \langle A _0 : A \rangle \rightarrow \langle A _1 : A \rangle \rightarrow \langle (n : \ast) \rightarrow A n \rangle
\end{align*}
\]

\[
\begin{align*}
\ast\text{-induction}' A a_0 a_1 = \text{+_induction} A \\
(\mathbb{1}\text{-induction} (\lambda (x : \mathbb{1}) \rightarrow A (\text{inl } x)) a_0)
\end{align*}
\]

\[
(\mathbb{1}\text{-induction} (\lambda (y : \mathbb{1}) \rightarrow A (\text{inr } y)) a_1)
\]
Given universes $\mathcal{U}$ and $\mathcal{V}$, a type

$$x : \mathcal{U}$$

and a type family

$$y : x \to \mathcal{V},$$

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

$$\mathcal{U} \sqcup \mathcal{V}$$

of the universes $\mathcal{U}$ and $\mathcal{V}$. We will write this sum

$$\Sigma Y,$$

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 of sufficiently powerful inference algorithms in the implementation of Agda. In such cases we can write

$$\Sigma \lambda(x : X) \to Y(x),$$

because $Y = \lambda(x : X) \to Y(x)$ by a so-called $\eta$-rule. However, we will define syntax to be able to write this in more familiar form as

$$\Sigma x : X, Y(x).$$

In MLTT we can write this in other ways, for example with the indexing $x : X$ written as a subscript of $\Sigma$ 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:

$$\Sigma z : \mathbb{N}, x + z \equiv y,$$

where $Y(z) = (x + z \equiv y)$ in this case, and where we haven’t defined the identity type former $\equiv$ yet.

We can construct the $\Sigma$ type former as follows in Agda:

```agda
record \Sigma {\mathcal{U} \mathcal{V}} {{x : \mathcal{U}} \to {y : x \to \mathcal{V}}} : \mathcal{U} \sqcup \mathcal{V} \to 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 $\Sigma Y$ has two fields, giving the types for them.

Remark. Beginners may safely ignore this remark: Normally people will call these two fields $x$ and $y$ something like $\text{pr}_1$, and $\text{pr}_2$, or $\text{fst}$ and $\text{snd}$, for first and second projection, rather than $x$ and $y$, and then do open $\Sigma$ 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.

$$\text{pr}_1 : \{x : \mathcal{U}\} \to \{y : x \to \mathcal{V}\} \to \Sigma Y \to x$$

$$\text{pr}_1(x, y) = x$$

$$\text{pr}_2 : \{x : \mathcal{U}\} \to \{y : x \to \mathcal{V}\} \to (z : \Sigma Y) \to Y(\text{pr}_1 z)$$

$$\text{pr}_2(x, y) = y$$
We now introduce syntax to be able to write $\Sigma x : X$, $y$ instead of $\Sigma \lambda(x : X) \to y$. For this purpose, we first define a version of $\Sigma$ making the index type explicit.

\[
\text{syntax } -\Sigma X Y = \Sigma x : X, y 
\]

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.

Notice also that “$\vdash$” in the above syntax definition is not the same as “$:\vdash$”, even though they may look the same. For the above notation $\Sigma x \vdash A$, the symbol “$\vdash$” has to be typed “\:\vdash” in the emacs Agda mode.

To prove that $A \vdash z$ holds for all $z : \Sigma Y$, 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 $\Sigma$ induction or $\Sigma$ elimination, or uncurry, after Haskell Curry.

\[
\Sigma\text{-induction} \quad \begin{cases} X : \mathcal{U} \vdash Y : X \to \mathcal{V} \\ A : \Sigma Y \to \mathcal{W} \end{cases} \quad \rightarrow \quad ((x : X) (y : Y x) \to A (x , y)) \\
\rightarrow \quad ((x , y) : \Sigma Y) \to A (x , y) 
\]

\[
\Sigma\text{-induction} \quad g (x , y) = g \times y 
\]

This function has an inverse:

\[
\text{curry} \quad \begin{cases} X : \mathcal{U} \vdash Y : X \to \mathcal{V} \\ A : \Sigma Y \to \mathcal{W} \end{cases} \quad \rightarrow \quad (((x , y) : \Sigma Y) \to A (x , y)) \\
\rightarrow \quad ((x : X) (y : Y x) \to A (x , y)) 
\]

\[
\text{curry} \quad f \times y = f (x , y) 
\]

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

\[
_\times : \mathcal{U} \vdash \mathcal{V} \to \mathcal{U} \sqcup \mathcal{V} \\
X \times Y = \Sigma x : X, Y 
\]

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

We have the following uses of $\Sigma$.

1. The binary cartesian product represents conjunction “and”. If the types $A$ and $B$ stand for mathematical statements, then the mathematical statement

   \[
   A \text{ and } B
   \]

   is codified as

   \[
   A \times B,
   \]

   because to establish “$A$ 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.

2. The more general type

   \[
   \Sigma x : X, A x,
   \]

   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 establish this, we have to provide a specific element $x : X$ and a proof $a : A \ x$, together in a pair $(x, a)$.

3. Later we will discuss unspecified existence

$$\exists \ x : X , \ A \ x,$$

which will be obtained by a sort of quotient of $\Sigma \ x : X , \ A \ x$, written

$$\| \Sigma \ x : X , \ A \ x \|,$$

that identifies all the elements of the type $\Sigma \ x : X , \ A \ x$ in a single equivalence class, called its subsingleton (or truth value or propositional) truncation.

4. Another reading of

$$\Sigma \ x : X , \ A \ x$$

is as

the type of $x : X$ with $A \ x$,

similar to set-theoretical notation

$$\{ \ x \in X \mid A \ x \}.$$  

But we have to be careful because if there is more than one element in the type $A \ x$, then $x$ will occur more than once in this type. More precisely, for $a_0, a_1 : A \ x$ we have inhabitants $(x, a_0)$ and $(x, a_1)$ of the type $\Sigma \ x : X , \ A \ x$.

In such situations, if we don’t want that, we have to 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

$$\Sigma \ x : X , \ A \ x \|.$$  

An example is the image of a function $f : X \to Y$, which will be defined to be

$$\Sigma \ y : Y , \ \| \Sigma \ x : X , \ f \ x \equiv y \|.$$  

This is the type of $y : Y$ for which there is an unspecified $x : X$ with $f \ x \equiv y$.

(For constructively minded readers, we emphasize that this doesn’t erase the witness $x : X$.)

Table of contents

**Π types**

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

$$\Pi : \{X : U \} \ (A : X \to \mathcal{V}) \to U \sqcup \mathcal{V}.$$  

$$\Pi \ {U \} \ {\mathcal{V}} \ {X} \ A = (x : X) \to A \ x.$$  

$$-\Pi : \{U \mathcal{V} : \text{Universe}\} \ (X : U \} \ (Y : X \to \mathcal{V}) \to U \sqcup \mathcal{V}.$$  

$$-\Pi X Y = \Pi Y.$$  

**syntax**  

$$-\Pi A (\lambda \ x \to b) = \Pi x : A , \ b.$$  

Notice that the function type $X \to Y$ is the particular case of the $\Pi$ 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:

$$id : \{X : U \} \to X \to X$$  

$$id \ x = x.$$
Usually the type of function composition \( \circ \) is given as simply

\[
(Y \rightarrow Z) \rightarrow (X \rightarrow Y) \rightarrow (X \rightarrow Z).
\]

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

\[
\text{\_} \circ \text{\_} : \{X : \mathcal{U} \} \{Y : \mathcal{V} \} \{Z : Y \rightarrow \mathcal{W} \} \\
\rightarrow (\{y : Y \} \rightarrow Z y) \\
\rightarrow (\{x : X \} \rightarrow Z (f x))
\]

\[
g \circ f = \lambda x \rightarrow g (f x)
\]

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 \( f : X \rightarrow Y \) 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:

\[
\text{domain} : \{X : \mathcal{U} \} \{Y : \mathcal{V} \} \{X\} \{Y\} f = X
\]

\[
\text{codomain} : \{X : \mathcal{U} \} \{Y : \mathcal{V} \} \{X\} \{Y\} f = Y
\]

\[
\text{type-of} : \{X : \mathcal{U} \} \rightarrow X \rightarrow \mathcal{U}
\]

\[
\text{type-of} \{\mathcal{U}\} \{\mathcal{X}\} x = X
\]

### The identity type former \textbf{Id}, also written \textbf{\_≡\_}

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:

\[
data \text{Id} \{\mathcal{U}\} \{X : \mathcal{U} \} : X \rightarrow X \rightarrow \mathcal{U} \\
\text{where} \\
\text{refl} : (x : X) \rightarrow \text{Id} X x x
\]

Intuitively, the above definition would say that the only element of the type \( \text{Id} X x x \) is something called \text{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 \textbf{data} or induction: In the previous cases, we defined \textit{types}, namely \( \emptyset \), \( \mathbb{I} \), \( \mathbb{N} \), or a \textit{type depending on type parameters}, namely \( _+_{\_} \), with \( \mathcal{U} \) and \( \mathcal{V} \) fixed,

\[
_+_{\_} : \mathcal{U} \rightarrow \mathcal{V} \rightarrow \mathcal{U} \sqcup \mathcal{V}
\]

But here we are defining a \textit{type family} indexed by the \textit{elements} of a given type, rather than a new type from old types. Given a type \( X \) in a universe \( \mathcal{U} \), we define a \textit{function}

\[
\text{Id} X : X \rightarrow X \rightarrow \mathcal{U}
\]

by some mysterious sort of induction. It is this that prevents us from being able to prove that the only element of the type \( \text{Id} X x x \) would be \text{refl} \( x \), or that the type \( \text{Id} X x y \) would have at most one element no matter what \( y : X \) is.

There is however, one interesting, and crucial, thing we \textbf{can prove}, namely that for any fixed element \( x : X \), the type

\[
\Sigma y : Y , \text{Id} X x y
\]

is always a \textit{singleton}.
We will use the following alternative notation for the identity type former \( \text{Id} \), 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 defining equation of the type former \( \_ \_ \_ \)): 

\[
\_\_\_ : \{ X : U \} \to X \to X \to U
\]

\[
x \equiv y = \text{Id} \_ x y
\]

Another intuition for this type family \( \_\_\_ : X \to X \to U \) is that it gives the least reflexive relation on the type \( X \), as suggested by Martin-Löf’s induction principle \( J \) discussed below.

Whereas we can make the intuition that \( x \equiv x \) 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 \( \text{refl} \ x \) is the only element of \( x \equiv x \) 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. This is in spirit similar to the non-provability of the parallel postulate in Euclidean geometry, which also considers models, which in turn are interesting in their own right.

However, for the elements of some types, such as the type \( \mathbb{N} \) of natural numbers, it is possible to prove that any identity type \( x \equiv y \) has at most one element. Such types are called sets in univalent mathematics.

If instead of the axiom \( K \) we adopt Voevodsky’s univalence axiom, we get specific examples of objects \( x \) and \( y \) such that the type \( x \equiv y \) has multiple elements, within the type theory. It follows that the identity type \( x \equiv y \) 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 under-specification 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 \( \infty \)-groupoids. In fact, the univalence axiom will say, in particular, that for some types \( X \) and elements \( x \ y : X \), the identity type \( x \equiv y \) does have more than one element.

A possible way to understand the element \( \text{refl} \ x \) of the type \( x \equiv x \) is as the “generic identification” between the point \( x \) and itself, but which is by no means necessarily the only identification in univalent foundations. It is generic in the sense that to explain what happens with all identifications \( p : x \equiv y \) between any two points \( x \) and \( y \) of a type \( X \), it suffices to explain what happens with the identification \( \text{refl} \ x : x \equiv x \) 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 \( \equiv \)-induction, but we prefer to honour MLTT tradition):

\[
\_ : (X : U) \to (A : (X \to Y) \to X \equiv Y \to U)
\]

\[
\to ((X : X) \to A \ x \ x \ (\text{refl} \ x))
\]

\[
\to (x : Y) \ x \ (p : x \equiv y) \to A \ x \ x \ p
\]

\[
\_ X A f \ x \ x \ (\text{refl} \ x) = f \ x
\]

This is related to the Yoneda Lemma in category theory, for readers familiar with the 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, \( \_ \) 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

- \( \text{Id} \),
- \( \text{refl} \),
- \( \_ \),
- the above equation for \( \_ \).

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

Exercise. Define

\[
\_ : \{ X : U \} \to X \to X \to U
\]

\[
\to B \ x \ (\text{refl} \ x)
\]

\[
\to (y : X) \ (p : x \equiv y) \to B \ y \ p
\]
\[ \mathbb{H} \times B \times b \times (\text{refl } x) = b \]

Then we can define \( \mathbb{J} \) from \( \mathbb{H} \) as follows:

\[
\begin{align*}
\mathbb{J}' & : (X : \mathcal{U}) \to (A : X \to \mathcal{V}) \to \text{transport } A \times x \times (\text{refl } x) \to A \times y \times p \\
& \quad \to \mathbb{J}' X A f x = \mathbb{H} x (A x) (f x) \\
\text{js-agreement } & : (X : \mathcal{U}) \to (A : X \to \mathcal{V}) \to \text{transport } A \times x \times (\text{refl } x) \to A \times y \times p \\
& \quad \to \text{js-agreement } X A f x (\text{refl } x) = \text{refl } (f x)
\end{align*}
\]

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

**Notational remark.** The symbols “=” and “≡” are swapped with respect to the HoTT book convention for definitional/judgemental equality and type 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.*

\[
\begin{align*}
\text{transport} & : \{X : \mathcal{U}\} \to (A : X \to \mathcal{V}) \to \{x y : X\} \\
& \quad \to x \equiv y \to A x \to A y \\
\text{transport} A (\text{refl } x) & = \text{id } (A x)
\end{align*}
\]

We can equivalently define transport using \( \mathbb{J} \) as follows:

\[
\begin{align*}
\text{transport}\mathbb{J} & : \{X : \mathcal{U}\} \to (A : X \to \mathcal{V}) \to \{x y : X\} \\
& \quad \to x \equiv y \to A x \to A y \\
\text{transport}\mathbb{J} \{U\} \{V\} = \mathbb{J} X (\lambda x \equiv y \to A x \to A y) (\lambda x \to \text{id } (A x)) x y
\end{align*}
\]

In the same way \( \mathbb{N} \)-recursion can be seen as the non-dependent special case of \( \mathbb{N} \)-induction, the following transport function can be seen as the non-dependent special case of the \( \equiv \)-induction principle \( \mathbb{H} \) with some of the arguments permuted and made implicit:

\[
\begin{align*}
\text{nondep-}\mathbb{H} & : \{X : \mathcal{U}\} \to (x : X) (A : X \to \mathcal{V}) \\
& \quad \to A x \to (y : X) \to x \equiv y \to A y \\
\text{nondep-}\mathbb{H} x A & = \mathbb{H} x (\lambda y \equiv y \to A y) \\
\text{transport}\mathbb{H} & : \{X : \mathcal{U}\} \to (A : X \to \mathcal{V}) \to \text{transport}\mathbb{H} A \times x \to A y \\
\text{transport}\mathbb{H} A \{x\} \{y\} p a & = \text{nondep-}\mathbb{H} A x a p
\end{align*}
\]

All the above transports coincide:

\[
\begin{align*}
\text{transports-agreement} & : \{X : \mathcal{U}\} \to (A : X \to \mathcal{V}) \to \text{transport}\mathbb{H} A \times x \equiv y \\
& \quad \to (\text{transport}\mathbb{H} A p \equiv A p) \\
& \quad \times (\text{transport}\mathbb{J} A p \equiv A p)
\end{align*}
\]
The following is for use when we want to recover implicit arguments without mentioning them.

\[
\text{lhs} : \{X : U \} \{x : X \} \to x \equiv y \to X
\]

\[
\text{rhs} : \{U \} \{X \} \{y \} \to p = x
\]

**Composition of identifications.** Given two identifications \( p : x \equiv y \) and \( q : y \equiv z \), we can compose them to get an identification \( p \cdot q : x \equiv z \). 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 \( \equiv \)-induction.

\[
\equiv : \{X : U \} \{x : X \} \to x \equiv y \equiv y \equiv z \to x \equiv z
\]

\[
p \cdot q = \text{transport} (\text{lhs} \ p \equiv) \ q \ p
\]

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

**Exercise.** Can you define an alternative version that uses \( p \) to transport. Do the two versions give equal results?

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

\[
\begin{align*}
x & \equiv (p) \\
y & \equiv (q) \\
z &
\end{align*}
\]

as a synonym of the expression \( p \cdot q \) with some of the implicit arguments of \( \equiv \cdot \) made explicit. We have one ternary \text{mixfix} operator \( \equiv(\_\_\_ \_) \) and one unary \text{postfix} operator \( \_\_ \).

\[
\equiv(\_\_\_ \_) : \{X : U \} \{x : X \} \{y : X \} \to x \equiv y \equiv y \equiv z \to x \equiv z
\]

\[
x \equiv (p \ q) = p \cdot q
\]

\[
\equiv : \{X : U \} \{x : X \} \to x \equiv x
\]

\[
x \equiv = \text{refl} \ x
\]

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

\[
\equiv^{-1} : \{X : U \} \to \{x : X \} \to x \equiv y \equiv y \equiv x
\]

\[
p^{-1} = \text{transport} (\equiv \text{lhs} \ p) \ p (\text{refl} (\text{lhs} \ p))
\]

We can define an alternative of identification composition with this:

\[
\equiv^{-1} : \{X : U \} \{x : X \} \to x \equiv y \equiv y \equiv z \to x \equiv z
\]

\[
p \cdot^{-1} q = \text{transport} (\equiv \text{rhs} \ q) (p^{-1}) \ q
\]

This agrees with the previous one:

\[
\cdot \text{agreement} : \{X : U \} \{x : X \} \{y : X \} \to p : x \equiv y \ (q : y \equiv z)
\]

\[
\to p \cdot^{-1} q = p \cdot q
\]

\[
\cdot \text{agreement} (\text{refl} \ x) (\text{refl} \ x) = \text{refl} (\text{refl} \ x)
\]

But \( \text{refl} \ y \) is a definitional neutral element for one of them on the right and for the other one on the left,

\[
\bullet \ p \cdot \text{refl} \ y = p,
\]

\[
\bullet \ \text{refl} \ y \cdot^{-1} q = q,
\]

which can be checked as follows

\[
\text{rdnel} : \{X : U \} \{x : X \} \{y : X \} \to p : x \equiv y
\]

\[
\to p \cdot \text{refl} \ y \equiv p
\]
rdnel \ p = \text{refl} \ p

\text{rdner} : \{X : \mathcal{U} \} \ \{y \ z : X\} \ (q : y \equiv z)
\quad \rightarrow \ \text{refl} \ y \ , \ ' \ \ q \equiv q

\text{rdner} \ q = \text{refl} \ q

Exercise. The identification \text{refl} \ y is neutral on both sides of each of the two operations \_ \_ \_ and \_ \ ' \_ \_, although not definitionally. This has to be proved by induction on identifications, as in \_ \ - \_ -agreement.

Application of a function to an identification. Given an identification \ p : x \equiv x' \ we get an identification \ \text{ap} f \ p = f x \equiv f x' \ for any \ f : X \rightarrow Y:

\begin{align*}
\text{ap} : \{X : \mathcal{U} \} \ \{Y : \mathcal{V} \} \ (f : X \rightarrow Y) \ \{x \ x' : X\} \rightarrow x \equiv x' \rightarrow f x \equiv f x'
\end{align*}

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 \text{transport}.

Notice that we have so far used the recursion principle \text{transport} only. To reason about \text{transport}, \_ \_ \_ and \text{ap}, we will need to use the full induction principle \mathcal{J} (or equivalently pattern matching on \text{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 : \mathcal{U} \} \ \{A : X \rightarrow \mathcal{V} \} \rightarrow \prod A \rightarrow A \rightarrow \mathcal{U} \cup \mathcal{V}

\_ \_ \_ f \ g = \forall x \rightarrow f x \equiv g x

The symbol \forall is a built-in notation for \prod. We could equivalently write the \text{definiens} as

\( (x : \_ \_ \_ ) \rightarrow f x \equiv g x \),

or, with our \text{domain} notation

\( (x : \text{domain} f) \rightarrow f x \equiv g x \).

Table of contents ⇩

Reasoning with negation

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

\_ \_ : \mathcal{U} \rightarrow \mathcal{U}

\_ \_ A = \neg(\neg A)

\_ \_ \_ A = \neg(\_ \_ A)

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

\text{dni} : (A : \mathcal{U}) \rightarrow A \rightarrow \_ \_ A

\text{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 \rightarrow \emptyset) \rightarrow \emptyset to get a point of \ A. For truth values \ A, we can assume this as an axiom if we wish, because it is \textit{equivalent to the principle excluded middle}. For arbitrary types \ A, this would be a form of \textit{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. But the \textit{axiom of choice} is consistent with univalent type theory, as stated in the \textit{introduction}.

In the proof of the following, we are given hypothetical functions \ f : A \rightarrow B \ and \ v : B \rightarrow \emptyset, and a hypothetical element \ a : A, and our goal is to get an element of \emptyset. But this is easy, because \ f \ a : B \ and hence \ v (f a) : \emptyset.
We have given a logical name to this function. Mathematically, this says that if we have a function \( A \rightarrow B \) 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:

\[
\text{tno} : (A : \text{}\text{U}^\prime) \rightarrow \rightarrow \rightarrow A \rightarrow \neg A
\]

\[
\text{tno} A = \text{contrapositive} (\text{dni} A)
\]

Hence, using \( \text{dni} \) once again, we get that \( \rightarrow \rightarrow \rightarrow A \) if and only if \( \neg A \). 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

\[
\mathcal{L} : \text{U}' \rightarrow \mathcal{V} \rightarrow \mathcal{U} \cup \mathcal{V}
\]

\[X \mathcal{L} Y = (X \rightarrow Y) \times (Y \rightarrow X)\]

\[\text{lr-implication} : \{X : \text{U}'\} \{Y : \mathcal{V}'\} \rightarrow (X \mathcal{L} Y) \rightarrow (X \rightarrow Y)\]

\[\text{lr-implication} = \text{pr}_1\]

\[\text{rl-implication} : \{X : \text{U}'\} \{Y : \mathcal{V}'\} \rightarrow (X \mathcal{L} Y) \rightarrow (Y \rightarrow X)\]

\[\text{rl-implication} = \text{pr}_2\]

then we can render Brouwer's argument in Agda as follows, where the “established fact” is \( \text{dni} \):

\[
\text{absurdity}^3\text{-is-absurdity} : \{A : \text{U}'\} \rightarrow \rightarrow \rightarrow A \leftrightarrow \neg A
\]

\[
\text{absurdity}^3\text{-is-absurdity} \{\text{U}\} \{A\} = \text{firstly} , \text{secondly}
\]

where

\[
\text{firstly} : \rightarrow \rightarrow \rightarrow A \rightarrow \neg A
\]

\[
\text{firstly} = \text{contrapositive} (\text{dni} A)
\]

\[
\text{secondly} : \neg A \rightarrow \rightarrow \rightarrow A
\]

\[
\text{secondly} = \text{dni} (\neg A)
\]

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.

\[
\mathcal{L} : \{X : \text{U}'\} \rightarrow \mathcal{U} \rightarrow \mathcal{U} \rightarrow \mathcal{U}'
\]

\[X \mathcal{L} Y \equiv \neg (X \equiv Y)\]

In the following proof, we have \( u : x \equiv y \rightarrow \emptyset \) and need to define a function \( y \equiv x \rightarrow \emptyset \). So all we need to do is to compose the function that inverts identifications with \( u \):

\[
\mathcal{L}\text{-sym} : \{X : \text{U}'\} \{X : \mathcal{X} \rightarrow \mathcal{X} \rightarrow \mathcal{Y} \rightarrow \mathcal{Y} \rightarrow \mathcal{X} \rightarrow \mathcal{X} \rightarrow \mathcal{Y} \}
\]

\[
\text{Id-Fun} : \{\text{U}\} \rightarrow \mathcal{U} \rightarrow \mathcal{U} \rightarrow \mathcal{U} \rightarrow \mathcal{U} \rightarrow \mathcal{U} \rightarrow \mathcal{U}
\]

\[
\text{Id-Fun} \{\text{U}\} = \text{transport} (\text{id} \{\text{U}'\})
\]

To show that the type \( l \) is not equal to the type \( \emptyset \), we use that transport id gives \( l \equiv \emptyset \rightarrow \text{id} \equiv \text{id} \) where id is the identity function of the universe \( \text{U} \). More generally, we have the following conversion of type identifications into functions:
Here the identity function is that of the universe $\mathcal{U}$ where the types $X$ and $Y$ live. An equivalent definition is the following, where this time the identity function is that of the type $X$:

\[
\text{Id-Fun}' : \{X \, Y : \mathcal{U} \} \rightarrow \text{id} \cdot X
\]

\[
\text{Id-Funs-agree} : \{X \, Y : \mathcal{U} \} (p : X \equiv Y) \rightarrow \text{Id-Fun} \cdot p \equiv \text{Id-Fun}' \cdot p
\]

\[
\text{Id-Funs-agree} (\text{refl} X) = \text{refl} (\text{id} X)
\]

So if we have a hypothetical identification $p : \mathord{1} \equiv \mathord{0}$, then we get a function $\mathord{1} \rightarrow \mathord{0}$. We apply this function to $\mathord{1} : \mathord{1}$ to conclude the proof.

\[
\text{l-is-not-0} : \mathord{1} \not\equiv \mathord{0}
\]

\[
\text{l-is-not-0} \ p = \text{Id-Fun} \ p \cdot
\]

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

\[
\text{l-is-not-0} : \mathord{1} \not\equiv \mathord{0}
\]

\[
\text{l-is-not-0} \ p = \text{l-is-not-0} \ q
\]

\[
\text{where}
\]

\[
f : \mathord{2} \rightarrow \mathcal{U}_0;
\]

\[
f_0 = \mathord{0};
\]

\[
f_1 = \mathord{1};
\]

\[
q : \mathord{1} \equiv \mathord{0};
\]

\[
q = \text{ap} \ f \ p
\]

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 $\mathord{0}$-\textit{induction} and nowhere else in these notes.

\[
\text{l-is-not-0}[\text{not-an-MLTT-proof}] : \neg(\mathord{1} \equiv \mathord{0})
\]

\[
\text{l-is-not-0}[\text{not-an-MLTT-proof}] ()
\]

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

\[
\text{decidable} : \mathcal{U} \rightarrow \mathcal{U}
\]

\[
\text{decidable} \ A = A + \neg A
\]

\[
\text{has-decidable-equality} : \mathcal{U} \rightarrow \mathcal{U}
\]

\[
\text{has-decidable-equality} \ X = (x \ y : X) \rightarrow \text{decidable} \ (x \equiv y)
\]

\[
\text{2-has-decidable-equality} : \text{has-decidable-equality} \ \mathord{2}
\]

\[
\text{2-has-decidable-equality} \ _0_0 = \text{inl} \ (\text{refl} \ _0)
\]

\[
\text{2-has-decidable-equality} \ _0_1 = \text{inr} \ (\neg\text{-sym} \ \text{l-is-not-0})
\]

\[
\text{2-has-decidable-equality} \ _1_0 = \text{inr} \ \text{l-is-not-0}
\]

\[
\text{2-has-decidable-equality} \ _1_1 = \text{inl} \ (\text{refl} \ _1)
\]

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 $\mathord{0} \equiv \mathord{1} \rightarrow \mathord{0}$ and $\mathord{1} \equiv \mathord{0} \rightarrow \mathord{0}$, which we can take to be $\neg\text{-sym} \text{l-is-not-0}$ and $\text{l-is-not-0}$ respectively.

The following is more interesting. We consider the two possible cases for $n$. The first one assumes a hypothetical function $f : \mathord{0} \equiv \mathord{0} \rightarrow \mathord{0}$, from which we get $f (\text{refl} \ _0) : \mathord{0}$, and then, using $\mathord{0}$, we get an element of any type we like, which we choose to be $\mathord{0}$, and we are done. Of course, we will never be able to use the function $\text{not-zero-is-one}$ with such outrageous arguments. The other case $n = \mathord{1}$ doesn’t need to use the hypothesis $f : \mathord{1} \equiv \mathord{0} \rightarrow \mathord{0}$, because the desired conclusion holds right away, as it is $\mathord{1} \equiv \mathord{1}$, which is proved by $\text{refl} \ _1$. But notice that there is nothing wrong with the hypothesis $f : \mathord{1} \equiv \mathord{0} \rightarrow \mathord{0}$. For example, we can use $\text{not-zero-is-one}$ taking $n$ to be $\mathord{0}$ and $f$ to be $\text{l-is-not-0}$, so that the hypotheses can be fulfilled in the second equation.

\[
\text{not-zero-is-one} : (n : \mathord{2}) \rightarrow n \not\equiv \mathord{0} \rightarrow n \equiv \mathord{1}
\]

\[
\text{not-zero-is-one} \ _0 \ f = \mathord{!} \ (\mathord{0} \equiv \mathord{1}) \ (f \ (\text{refl} \ _0))
\]

\[
\text{not-zero-is-one} \ _1 \ f = \text{refl} \ _1
\]
The following generalizes, both in its statement and its proof (so we could have formulated it first and then used it to deduce):

\[ \text{inl-inr-disjoint-images} : \{X : \mathcal{U}\} \{Y : \mathcal{V}\} \{x : X\} \{y : Y\} \rightarrow \text{inl} x \neq \text{inr} y \]

\[ \text{inl-inr-disjoint-images} \{U\} \{V\} \{X\} \{Y\} p = \bot \neq \top q \]

where

\[ f : X + Y \rightarrow \mathcal{U}_0 \]
\[ f (\text{inl} x) = \bot \]
\[ f (\text{inr} y) = \bot \]

\[ q : \bot \equiv \bot \]
\[ q = \text{ap} f p \]

If \( P \) or \( Q \) holds and \( Q \) fails, then \( P \) holds:

\[ \text{right-fails-gives-left-holds} : \{P : \mathcal{U}\} \{Q : \mathcal{V}\} \rightarrow P + Q \rightarrow \neg Q \rightarrow P \]

\[ \text{right-fails-gives-left-holds} (\text{inl} p) u = p \]
\[ \text{right-fails-gives-left-holds} (\text{inr} q) u = !\bot _{(u q)} \]

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 (_\times_ \rightarrow \text{*} ; _+_ \rightarrow _+_) \open N\text{-order}

\[ \text{is-prime} : \mathbb{N} \rightarrow \mathcal{U}_0 \]
\[ \text{is-prime} n = (n \geq 2) \times ((x y : \mathbb{N}) \rightarrow x \times y \equiv n \rightarrow (x \equiv 1) + (x \equiv n)) \]

\[ \text{twin-prime-conjecture} : \mathcal{U}_0 \]
\[ \text{twin-prime-conjecture} = (n : \mathbb{N}) \times \Sigma p : \mathbb{N} , (p \geq n) \times \text{is-prime} p \times \text{is-prime} (p + 2) \]

Thus, not only can we write down 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

Remaining Peano axioms and basic arithmetic

We first prove the remaining Peano axioms.

\[ \text{positive-not-zero} : (x : \mathbb{N}) \rightarrow \text{succ} x \neq \emptyset \]
\[ \text{positive-not-zero} \times p = \bot \neq \top (g p) \]

where

\[ f : \mathbb{N} \rightarrow \mathcal{U}_0 \]
\[ f \emptyset = \emptyset \]
\[ f (\text{succ} x) = \bot \]

\[ g : \text{succ} x \equiv \emptyset \rightarrow \bot \equiv \emptyset \]
\[ g = \text{ap} f \]

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

\[ \text{pred} : \mathbb{N} \rightarrow \mathbb{N} \]
\[ \text{pred} \emptyset = \emptyset \]
\[ \text{pred} (\text{succ} n) = n \]

\[ \text{succ-lc} : \{x y : \mathbb{N}\} \rightarrow \text{succ} x \equiv \text{succ} y \rightarrow x \equiv y \]
\[ \text{succ-lc} = \text{ap} \text{pred} \]
With this we have proved all the Peano axioms.

Without assuming the principle of excluded middle, we can prove that \( \mathbb{N} \) has decidable equality:

\[
\begin{align*}
\text{N-has-decidable-equality} & : \text{has-decidable-equality} \ \mathbb{N} \\
\text{N-has-decidable-equality} \ 0 \ 0 & = \text{inl} \ (\text{refl} \ 0) \\
\text{N-has-decidable-equality} \ 0 \ (\text{succ} \ y) & = \text{inr} \ (\text{sym} \ \text{positive-not-zero} \ y) \\
\text{N-has-decidable-equality} \ (\text{succ} \ x) \ 0 & = \text{inr} \ \text{positive-not-zero} \ x \\
\text{N-has-decidable-equality} \ (\text{succ} \ x) \ (\text{succ} \ y) & = f \big( \text{N-has-decidable-equality} \ x \ y \big)
\end{align*}
\]

where
\[
\begin{align*}
f : \text{decidable} \ (x \equiv y) & \rightarrow \text{decidable} \ (\text{succ} \ x \equiv \text{succ} \ y) \\
f \ (\text{inl} \ p) & = \text{inl} \ (\text{ap} \ \text{succ} \ p) \\
f \ (\text{inr} \ k) & = \text{inr} \ (\lambda \ (s : \text{succ} \ x \equiv \text{succ} \ y) \rightarrow k \ (\text{succ-1c} \ s))
\end{align*}
\]

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 \( \mathbb{N} \) to use the \text{N-induction} principle instead of pattern matching and recursion, to understand by themselves that this can be done.

We now move to basic arithmetic, and we use a module for that.

\[
\text{module basic-arithmetic-and-order where}
\]

\[
\begin{align*}
\text{open N-order public} \\
\text{open Arithmetic renaming} \ (+_\_ \ \rightarrow \ +_\_) \ \text{hiding} \ (\_\_)
\end{align*}
\]

We can show that addition is associative as follows, by induction on \( z \), where \( \text{IH} \) stands for “induction hypothesis”:

\[
\begin{align*}
+_\text{assoc} & : (x \ y \ z : \mathbb{N}) \rightarrow (x + y) + z \equiv x + (y + z) \\
+_\text{assoc} \ x \ y \ \text{zero} & = (x + y) + 0 \equiv (\text{refl} \ _) \\
+_\text{assoc} \ x \ y \ (\text{succ} \ z) & = (x + y) + \text{succ} \ z \equiv (\text{refl} \ _) \\
\text{succ} \ ((x + y) + z) & \equiv (\text{ap} \ \text{succ} \ \text{IH}) \\
\text{succ} \ (x + (y + z)) & \equiv (\text{refl} \ _) \\
\text{IH} & : (x + y) + z \equiv x + (y + z) \\
\text{IH} & = +_\text{assoc} \ x \ y \ z
\end{align*}
\]

Notice that the proofs \text{refl} \ _ should be read as “by definition” or “by construction”. They are not necessary, because Agda knows the definitions and silently expands them when necessary, but we are writing them here for the sake of clarity. Here is the version with the silent expansion of definitions, for the sake of illustration (the author of these notes can write, but not read it the absence of the above verbose version):

\[
\begin{align*}
+_\text{assoc}' & : (x \ y \ z : \mathbb{N}) \rightarrow (x + y) + z \equiv x + (y + z) \\
+_\text{assoc}' \ x \ y \ \text{zero} & = \text{refl} \ _ \\
+_\text{assoc}' \ x \ y \ (\text{succ} \ z) & = \text{ap} \ \text{succ} \ (+_\text{assoc}' \ x \ y \ z)
\end{align*}
\]

We defined addition by induction on the second argument. Next we show that the base case and induction step of a definition by induction on the first argument hold (but of course not definitionally). We do this by induction on the second argument.

\[
\begin{align*}
+_\text{base-on-first} & : (x : \mathbb{N}) \rightarrow 0 + x \equiv x \\
+_\text{base-on-first} \ 0 & = \text{refl} \ 0 \\
+_\text{base-on-first} \ (\text{succ} \ x) & = 0 + \text{succ} \ x \equiv (\text{refl} \ _) \\
\text{succ} \ (0 + x) & \equiv (\text{ap} \ \text{succ} \ \text{IH}) \\
\text{succ} \ x & \equiv (\text{IH}) \\
\text{IH} & : 0 + x \equiv x \\
\text{IH} & = +_\text{base-on-first} \ x
\end{align*}
\]

\[
\begin{align*}
+_\text{step-on-first} & : (x \ y : \mathbb{N}) \rightarrow \text{succ} \ x + y \equiv \text{succ} \ (x + y) \\
+_\text{step-on-first} \ x \ \text{zero} & = \text{refl} \ (\text{succ} \ x)
\end{align*}
\]
\begin{align*}
\text{+}-\text{step-on-first } x \ (\text{succ } y) &= \text{succ } x + \text{succ } y \ \equiv (\text{refl }_1) \\
\text{succ} \ (\text{succ } x + y) &\equiv (\text{ap succ } \text{IH}) \\
\text{succ} \ (x + \text{succ } y) &\equiv
\end{align*}

where

\begin{align*}
\text{IH} : \text{succ } x + y &\equiv \text{succ} (x + y) \\
\text{IH} &= +\text{-step-on-first } x \ y
\end{align*}

Using this, the commutativity of addition can be proved by induction on the first argument.

\begin{align*}
\text{+}-\text{comm} : (x \ y : \mathbb{N}) &\rightarrow x + y \equiv y + x \\
\text{+}-\text{comm } 0 \ y &\equiv 0 + y \equiv (\text{+}-\text{base-on-first } y) \\
&\quad y \equiv (\text{refl }_1) \\
&\quad y + 0 \equiv
\end{align*}

\begin{align*}
\text{+}-\text{comm } (\text{succ } x) \ y &= \text{succ } x + y \equiv (\text{+}-\text{step-on-first } x \ y) \\
\text{succ}(x + y) &\equiv (\text{ap succ } \text{IH}) \\
\text{succ}(y + x) &\equiv (\text{refl }_1) \\
\text{y} + \text{succ } x &\equiv
\end{align*}

where

\begin{align*}
\text{IH} : x + y &\equiv y + x \\
\text{IH} &= +\text{-comm } x \ y
\end{align*}

We now show that addition is cancellable in its left argument, by induction on the left argument:

\begin{align*}
\text{+}-\text{lc} : (x \ y \ z : \mathbb{N}) &\rightarrow x + y \equiv x + z \rightarrow y \equiv z \\
\text{+}-\text{lc } 0 \ y \ z \ p &= y \equiv (\text{+}-\text{base-on-first } y) \\
&\quad 0 + y \equiv (p) \\
&\quad 0 + z \equiv (\text{+}-\text{base-on-first } z) \\
&\quad z \equiv
\end{align*}

\begin{align*}
\text{+}-\text{lc } (\text{succ } x) \ y \ z \ p &= \text{IH} \\
\text{where} \\
q &= \text{succ } (x + y) \equiv (\text{+}-\text{step-on-first } x \ y) \\
\text{succ } x + y &\equiv (p) \\
\text{succ } x + z &\equiv (\text{+}-\text{step-on-first } x \ z) \\
\text{succ } (x + z) &\equiv
\end{align*}

\begin{align*}
\text{IH} : y \equiv z \\
\text{IH} &= +\text{-lc } x \ y \ z \ (\text{succ-lc } q)
\end{align*}

Now we solve part of an exercise given above, namely that \((x \leq y) \equiv \Sigma z : \mathbb{N}, x + z \equiv y\).

First we name the alternative definition of \(\leq\):

\begin{align*}
\_\leq \_ : \mathbb{N} \rightarrow \mathbb{N} \rightarrow \mathcal{U}_0, \\
x \leq y \equiv \Sigma z : \mathbb{N}, x + z \equiv y
\end{align*}

Next we show that the two relations \(\leq\) and \(\leq\) imply each other.

In both cases, we proceed by induction on both arguments.

\begin{align*}
\text{\leq-gives-\leq} : (x \ y : \mathbb{N}) &\rightarrow x \leq y \rightarrow x \leq y \\
\text{\leq-gives-\leq } 0 \ 0 &\equiv \ 1 = 0 , \ \text{refl } 0 \\
\text{\leq-gives-\leq } 0 \ (\text{succ } y) &\equiv \ 1 = \text{succ } y , \ \text{+}-\text{base-on-first } (\text{succ } y) \\
\text{\leq-gives-\leq } (\text{succ } x) \ 0 &\equiv \ 1 !\ 0 \ (\text{succ } x \leq \text{zero}) \ 1 \\
\text{\leq-gives-\leq } (\text{succ } x) \ (\text{succ } y) &\equiv \ 1 = y \\
\text{where} \\
\text{IH} : x \leq y \\
\text{IH} &= \text{\leq-gives-\leq } x \ y \ 1 \\
\text{z} : \mathbb{N} &\equiv \text{pr}_1, \text{IH} \\
\text{z} &= \text{pr}_1, \text{IH} \\
p : x + z \equiv y &\equiv \text{pr}_2, \text{IH} \\
p &= \text{pr}_2, \text{IH}
\end{align*}
antisymmetric:
We now develop some generally useful material regarding the order \( \leq \) on natural numbers. First, it is reflexive, transitive and antisymmetric:

\[
\begin{align*}
\leq - \text{refl} : & \quad (n : \mathbb{N}) \rightarrow n \leq n \\
\leq - \text{refl} \quad zero & = \star \\
\leq - \text{refl} (\text{succ} n) & = \leq - \text{refl} n \\
\leq - \text{trans} : & \quad (l m n : \mathbb{N}) \rightarrow l \leq m \rightarrow m \leq n \rightarrow l \leq n \\
\leq - \text{trans} \quad zero n p q & = \star \\
\leq - \text{trans} (\text{succ} l) \quad \text{zero} n p q & = \bot (\text{succ} l \leq n) p \\
\leq - \text{trans} (\text{succ} l) (\text{succ} m) \quad \text{zero} p q & = q \\
\leq - \text{trans} (\text{succ} l) (\text{succ} m) (\text{succ} n) p q & = \leq - \text{trans} l m n p q \\
\leq - \text{anti} : & \quad (m n : \mathbb{N}) \rightarrow m \leq n \rightarrow n \leq m \rightarrow m \equiv n \\
\leq - \text{anti} \quad zero p q & = \text{refl} zero \\
\leq - \text{anti} \quad \text{zero} (\text{succ} n) p q & = \bot (\text{zero} \equiv \text{succ} n) q \\
\leq - \text{anti} (\text{succ} m) \quad zero p q & = \bot (\text{succ} m \equiv \text{zero}) p \\
\leq - \text{anti} (\text{succ} m) (\text{succ} n) p q & = \text{ap} \text{ succ} (\leq - \text{anti} m n p q) \\
\leq - \text{succ} : & \quad (n : \mathbb{N}) \rightarrow n \leq \text{succ} n \\
\leq - \text{succ} \quad zero & = \star \\
\leq - \text{succ} (\text{succ} n) & = \leq - \text{succ} n \\
\text{zero-minimal} : & \quad (n : \mathbb{N}) \rightarrow \text{zero} \leq n \\
\text{zero-minimal} n & = \star \\
\text{unique-minimal} : & \quad (n : \mathbb{N}) \rightarrow n \leq \text{zero} \rightarrow n \equiv \text{zero} \\
\text{unique-minimal} \quad \text{zero} p & = \text{refl} zero \\
\text{unique-minimal} (\text{succ} n) p & = \bot (\text{succ} n \equiv \text{zero}) p \\
\leq - \text{split} : & \quad (m n : \mathbb{N}) \rightarrow m \leq \text{succ} n \rightarrow (m \leq n) + (m \equiv \text{succ} n) \\
\leq - \text{split} \quad \text{zero} l & = \text{inl} l \\
\leq - \text{split} (\text{succ} m) \quad \text{zero} l & = \text{inr} (\text{ap} \text{ succ} (\text{unique-minimal} m l)) \\
\leq - \text{split} (\text{succ} m) (\text{succ} n) l & = \text{+-recursion} \text{ inl} (\text{inr} \circ \text{ap} \text{ succ}) (\leq - \text{split} m n l) \\
\_ \_ : & \quad \mathbb{N} \rightarrow \mathbb{N} \rightarrow \mathbb{N} \\
\_ \_ n & = \bot \\
x \lt y & = \text{succ} x \lt y \\
\text{infix} \quad 10 \_ \_
The type of roots of a function:

\[ \text{root} : (\mathbb{N} \to \mathbb{N}) \to \mathcal{U}_0. \]

\[ \text{root} f = \Sigma n : \mathbb{N}, \; f n \equiv \emptyset \]

\[ \text{has-no-root<} : (\mathbb{N} \to \mathbb{N}) \to \mathbb{N} \to \mathcal{U}_0. \]

\[ f \; \text{has-no-root<} \; k = (n : \mathbb{N}) \to n < k \to f n \not\equiv \emptyset \]

\[ \text{is-minimal-root} : (\mathbb{N} \to \mathbb{N}) \to \mathbb{N} \to \mathcal{U}_0. \]

\[ \text{is-minimal-root} \; f \; m = (f \; m \equiv \emptyset) \times (f \; \text{has-no-root<} \; m) \]

\[ \text{at-most-one-minimal-root} : (f : \mathbb{N} \to \mathbb{N}) (m n : \mathbb{N}) \to \text{is-minimal-root} \; f \; m \to \text{is-minimal-root} \; f \; n \to m \equiv n \]

\[ \text{at-most-one-minimal-root} \; f \; m \; n \; (p , \; \varphi) \; (q , \; \psi) = c \; m \; n \; a \; b \]

where

\[ a : \neg (m < n) \]
\[ a \; u = \varphi \; m \; u \; p \]

\[ b : \neg (n < m) \]
\[ b \; v = \varphi \; n \; v \; q \]

\[ c : (m n : \mathbb{N}) \to \neg (m < n) \to \neg (n < m) \to m \equiv n \]
\[ c \; m \; n \; u \; v = s-\text{anti} \; m \; n \; (\text{not<-gives->} \; m \; n \; v) \; (\text{not<-gives->} \; n \; m \; u) \]

The type of minimal roots of a function:

\[ \text{minimal-root} : (\mathbb{N} \to \mathbb{N}) \to \mathcal{U}_0. \]

\[ \text{minimal-root} \; f = \Sigma m : \mathbb{N}, \; \text{is-minimal-root} \; f \; m \]

\[ \text{minimal-root-is-root} : \forall f \to \text{minimal-root} \; f \to \text{root} \; f \]

\[ \text{minimal-root-is-root} \; f \; (m , \; p , \; _) = m , \; p \]

\[ \text{bounded-N-search} : \forall k f \to (\text{minimal-root} \; f) + (f \; \text{has-no-root<} \; k) \]

\[ \text{bounded-N-search} \; \text{zero} f = \text{inr} \; (\lambda n \to !\emptyset \; (f \; n \not\equiv \emptyset)) \]

\[ \text{bounded-N-search} \; (\text{succ} \; k) f = \text{+recursion} \; \varphi \; \gamma \; (\text{bounded-N-search} \; k \; f) \]

where

\[ A : \mathbb{N} \to (\mathbb{N} \to \mathbb{N}) \to \mathcal{U}_0. \]
\[ A \; k \; f = (\text{minimal-root} \; f) + (f \; \text{has-no-root<} \; k) \]

\[ \varphi : \text{minimal-root} \; f \to A \; (\text{succ} \; k) \; f \]
\[ \varphi = \text{inl} \]

\[ y : f \; \text{has-no-root<} \; k \to A \; (\text{succ} \; k) \; f \]
\[ y \; u = \text{+recursion} \; \gamma_0 \; \gamma_1 \; (N\text{-has-decidable-equality} \; (f \; k) \; \emptyset) \]

where

\[ \gamma_0 : f \; k \equiv \emptyset \to A \; (\text{succ} \; k) \; f \]
\[ \gamma_0 \; p = \text{inl} \; (k , \; p , \; u) \]
Given any root, we can find a minimal root.

\[
\text{root-gives-minimal-root} : \forall f \rightarrow \text{root } f \rightarrow \text{minimal-root } f
\]

where
\[
\begin{align*}
g &: \neg(f \text{ has-no-root<}(\text{succ } n)) \\
g \phi &= \phi n (s\text{-refl } n) p \\
y &: \text{minimal-root } f \\
y &= \text{right-fails-gives-left-holds} (\text{bounded-ℕ-search } (\text{succ } n) f) g
\end{align*}
\]

---

## Univalent Mathematics in Agda

### Our univalent type theory

- A spartan MLTT as above.
- Univalence axiom as below.
- Subsingleton (or truth-value or propositional) truncations as below.

But, as discussed above, rather than postulating univalence and truncation, we will use them as explicit assumptions each time they are needed.

We emphasize that there are univalent type theories in which univalence and existence of truncations are theorems, for example cubical type theory, which has a version available in Agda, called [cubical Agda](#).

---

###Singletons, subsingletons and sets

#### Singleton (or contractible) types

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

\[
\begin{align*}
is\text{-center} &: (X : U^\prime) \rightarrow X \rightarrow U^\prime \\
is\text{-center } X c &= (x : X) \rightarrow c \equiv x \\
is\text{-singleton} &: u^\prime \rightarrow u^\prime \\
is\text{-singleton } X &= \Sigma c : X, \text{is-center } X c
\end{align*}
\]

Such an element \(c\) is called a center of contraction of \(X\) in connection with homotopy theory.

\[
\begin{align*}
is\text{-singleton } X &= \Sigma c : X, \text{is-center } X c
\end{align*}
\]

Once we have defined the notion of type equivalence, we will have that a type is a singleton if and only if it is equivalent to \(\mathbb{1}\).

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

\[
\begin{align*}
\text{center} &: (X : U^\prime) \rightarrow \text{is-singleton } X \rightarrow X \\
\text{center } X (c, \phi) &= c \\
\text{centrality} &: (X : U^\prime) (i : \text{is-singleton } X) (x : X) \rightarrow \text{center } X i \equiv x \\
\text{centrality } X (c, \phi) &= \phi
\end{align*}
\]

#### Subsingletons (or propositions or truth values)

A type is a subsingleton if it has at most one element, that is, any two of its elements are equal, or identified.
is-subsingleton : \( \mathcal{U} \to \mathcal{U} \)
is-subsingleton \( X = (x \; y : X) \to x \equiv y \)

\( \emptyset \)-is-subsingleton : is-subsingleton \( \emptyset \)

\( \emptyset \)-is-subsingleton \( x \; y = \bang \, (x \equiv y) \times \)

corestriction:

singletons-are-subsingletons : \((X : \mathcal{U}) \to is-singleton X \to is-subsingleton X\)
singletons-are-subsingletons \( X \, (c \; \varphi) \times y = x \equiv (\varphi \; x) \)

\( c \equiv (\varphi \; y) \)

\( y \)

1-is-subsingleton : is-subsingleton \( \mathbb{1} \)

1-is-subsingleton = singletons-are-subsingletons \( \mathbb{1} \) \( \mathbb{1} \)-is-singleton

pointed-subsingletons-are-singletons : \((X : \mathcal{U}) \to is-subsingleton X \to is-singleton X\)

pointed-subsingletons-are-singletons \( X \times s = (x \, s \, x) \)

singleton-iff-pointed-and-subsingleton : \\{X : \mathcal{U}\} \to is-singleton X \iff (X \times is-subsingleton X)

defined:

given \(\mathcal{U}\) \(\{X\} = (a \; b)\)

where

\( a : is-singleton X \to X \times is-subsingleton X \)

\( a \; s = center \; X \; s \), singletons-are-subsingletons \( X \; s \)

\( b : X \times is-subsingleton X \to is-singleton X \)

\( b \; (x \; t) = pointed-subsingletons-are-singletons \; X \times t \)

The terminology stands for subtype of a singleton and is justified by the fact that a type \( X \) is a subsingleton according to the above definition if and only if the map \( X \to \mathbb{1} \) is an embedding, if and only if \( X \) is embedded into a singleton.

Under univalent excluded middle, the empty type \( \emptyset \) and the singleton type \( \mathbb{1} \) are the only subsingletons, up to equivalence, or up to identity if we further assume univalence.

Subsingletons are also called propositions or truth values:

is-prop is-truth-value : \( \mathcal{U} \to \mathcal{U} \)
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 statements as arbitrary types, rather than just subsingletons. In these notes we prefer the terminology subsingleton, but we occasionally use the terminologies proposition and truth value. When we wish to emphasize this notion of proposition adopted in univalent mathematics, as opposed to “propositions as (arbitrary) types”, we may say univalent proposition.

In some formal systems, for example set theory based on first-order logic, one can tell that something is a proposition by looking at its syntactical form, e.g. “there are infinitely many prime numbers” is a proposition, by construction, in such a theory.

In univalent mathematics, however, propositions are mathematical, rather than meta-mathematical, objects, and the fact that a type turns out to be a proposition requires mathematical proof. Moreover, such a proof may be subtle and not immediate and require significant preparation. An example is the proof that the univalence axiom is a proposition, which relies on the fact that univalence implies function extensionality, which in turn implies that the statement that a map is an equivalence is a proposition. Another non-trivial example, which again relies on univalence or at least function extensionality, is the proof that the statement that a type \( X \) is a proposition is itself a proposition, which can be written as is-prop (is-prop \( X \)).

Singletons and subsingletons are also called -2-groupoids and -1-groupoids respectively.

Sets (or 0-groupoids)

A type is defined to be a set if there is at most one way for any two of its elements to be equal:
At this point, with the definition of these notions, we are entering the realm of univalent mathematics, but not yet needing the univalence axiom.

### Univalent excluded middle

As mentioned above, under excluded middle, the only two subsingletons, up to equivalence, are \(\emptyset\) and \(\mathbb{1}\). In fact, excluded middle in univalent mathematics says precisely that.

\[
EM:\forall\ U \to U^+\\EM\ U = (X : U^+) \to is-subsingleton X \to X + \neg X\\EM'\ U = (X : U^+) \to is-subsingleton X \to is-singleton X + is-empty X
\]

Notice that the above two definitions 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 ways:

\[
EM\text{-gives-}EM' : EM\ U \to EM'\ U\\EM\text{-gives-}EM'\ em\ X\ s = y\ (em\ X\ s)\\\quad where\\\quad y : X + \neg X \to is-singleton X + is-empty X\\\quad y\ (\text{inl}\ x) = \text{inl}\ (\text{pointed-subsingletons-are-singletons} X\ x\ s)\\\quad y\ (\text{inr}\ x) = \text{inr}\ x
\]

\[
EM'\text{-gives-}EM : EM'\ U \to EM\ U\\EM'\text{-gives-}EM'\ em'\ X\ s = y\ (em'\ X\ s)\\\quad where\\\quad y : is-singleton X + is-empty X \to X + \neg X\\\quad y\ (\text{inl}\ i) = \text{inl}\ (\text{center} X\ i)\\\quad y\ (\text{inr}\ x) = \text{inr}\ x
\]

We will not assume or deny excluded middle, which is an independent statement in our spartan univalent type theory - it can’t be proved or disproved, just as the parallel postulate in Euclidean geometry can’t be proved or disproved. We will deliberately keep it undecided, adopting a neutral approach to the constructive vs. non-constructive dichotomy. We will however prove a couple of consequences of excluded middle in discussions of foundational issues such as size and existence of subsingleton truncations. We will also prove that excluded middle is a consequence of the axiom of choice.

It should be emphasized that the potential failure of excluded middle doesn’t say that there may be mysterious subsingletons that fail to be singletons and also fail to be empty. No such things occur in mathematical nature:

\[
\text{no-unicorns} : \neg(\Sigma\ X : U^+, is-subsingleton X \times \neg(is-singleton X) \times \neg(is-empty X))\\\text{no-unicorns} (X, i, f, g) = c\\\quad where\\\quad e : is-empty X\\\quad e\ x = f\ (\text{pointed-subsingletons-are-singletons} X\ x\ i)\\\quad c : \emptyset\\\quad c = g\ e
\]

Given this, what does the potential failure of excluded middle mean? That there is no general way, provided by our spartan univalent type theory, to determine which of the two cases is-singleton X and is-empty X applies for a given subsingleton X. This kind of phenomenon should be familiar even in classical, non-constructive mathematics: although we are entitled to believe that the Goldbach conjecture either holds or fails, we still don’t know which one is the case, despite efforts by the sharpest mathematical minds. A hypothetical element of the type EM would, in particular, be able to solve the Goldbach conjecture. There is nothing wrong or contradictory with assuming the existence of such a magic blackbox (EM stands for “there Exists such a Magic box”). There is only loss of generality and of the implicit algorithmic character of our spartan base type theory, both of which most mathematicians will be perfectly happy to live with.

In these notes we don’t advocate any particular philosophy for or against excluded middle and other non-constructive principles. We confine ourselves to discussing mathematical facts. Axioms that can be assumed consistently but reduce
generality and/or break the implicit computational character of our base type theory are discussed at various parts of these lecture notes, and are summarized at the end.

Exercise. We also have that it is impossible for \( \text{is-singleton } X + \text{is-empty } X \) to fail for a given subsingleton \( X \), which amounts to saying that

\[
\neg \neg (\text{is-singleton } X + \text{is-empty } X)
\]

always holds.

Occasionally one hears people asserting that this says that the double negation of excluded middle holds. But this is incorrect. The above statement, when written in full, is

\[
(X : \mathcal{U}) \to \text{is-subsingleton } X \to \neg \neg (\text{is-singleton } X + \text{is-empty } X).
\]

This is a theorem, which is quite different from the double negation of excluded middle, which is not a theorem and is

\[
\neg \neg ((X : \mathcal{U}) \to \text{is-subsingleton } X \to \text{is-singleton } X + \text{is-empty } X).
\]

Just as excluded middle, this is an independent statement.

Exercise. Continued from the previous exercise. Also for any type \( R \) replacing the empty type, there is a function \((X + (X \to R)) \to R\), so that the kind of phenomenon illustrated in the previous exercise has little to do with the emptiness of the empty type.

Table of contents

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 \( \mathcal{U} \) as follows:

```haskell
module magmas where

Magma : (U : Universe) C Api
Magma U = \sum X : U \text{ is-set } X \times (X \to X \to X)
```

The type \( \text{Magma } \mathcal{U} \) collects all magmas in a universe \( \mathcal{U} \), and lives in the successor universe \( \mathcal{U}^+ \). 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 \((X , i , _\cdot_\cdot)\) with \(X : \mathcal{U}\) and \(i : \text{is-set } X\) and \( _\cdot_ : X \to X \to X\).

Given a magma \( M = (X , i , _\cdot_\cdot)\) we denote by \((M)\) its underlying set \(X\) and by \( \text{magma-operation } M\) its multiplication \(_\cdot_\cdot\):

\[
(\cdot) : \text{Magma } U \to U^+
\]

\[
\cdot : (X , i , _\cdot_\cdot) = X
\]

\[
\text{magma-is-set} : (M : \text{Magma } U) \to \text{is-set } (M)
\text{magma-is-set} (X , i , _\cdot_\cdot) = i
\]

\[
\text{magma-operation} : (M : \text{Magma } U) \to \{M\} \to \{M\} \to \{M\}
\text{magma-operation} (X , i , _\cdot_\cdot) = _\cdot_\cdot
\]

The following syntax declaration allows us to write \( x \cdot (M) y \) as an abbreviation of \( \text{magma-operation } M x y\):

```haskell
syntax magma-operation M x y = x \cdot (M) y
```

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 \(\text{magma-operation}\). The syntax declaration circumvents this.

A function of the underlying sets of two magmas is called a homomorphism when it commutes with the magma operations:
Then a monoid is a set equipped with such module.

We first define the three laws:

- associative
- left-neutral
- right-neutral

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

\[ \text{Id-iso} : \{ M : \text{Magma} \} \to M \equiv N \to \{ M \} \to \{ N \} \]

The isomorphisms can be collected in a type:

\[ \equiv_\equiv : \text{Magma} \equiv \text{Magma} \equiv \equiv \]

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:

\[ \text{magma-Id-iso} : \{ M : \text{Magma} \} \to M \equiv N \to M \equiv N \equiv N \text{Magma} \]

If we omit the sethood condition in the definition of the type of magmas, we get the type of what we could call \( \infty \)-magmas (then the type of magmas could be called \( \Theta \)-Magma).

\[ \infty \text{-Magma} : (U : \text{Universe}) \to U \]

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:

\[ \text{left-neutral} : \{ X : U \} \to X \to (X \times X) \to U \]

Then a monoid is a set equipped with such \( e \) and \( _\cdot_ \) satisfying these three laws:
Remark. People are more likely to use records in Agda rather than iterated Σs 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 \( \infty \)-monoids in any reasonable sense. We would instead get "wild \( \infty \)-monoids" or "incoherent \( \infty \)-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 sethood 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 \( \infty \)-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 associativity identifications.

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 the identifications between points as the arrows.

We have that \( \text{refl} \) provides a neutral element for composition of identifications:

\[
\begin{align*}
\text{refl-left} & : \{X : \mathcal{U} \} \{x : X\} \{y : X\} \{p : x \equiv y\} \to \text{refl } x \cdot p \equiv p \\
\text{refl-left } & \{\mathcal{U}\} \{X\} \{x\} \{\text{refl } x\} = \text{refl } (\text{refl } x) \\
\text{refl-right} & : \{X : \mathcal{U} \} \{x : X\} \{y : X\} \{p : x \equiv y\} \to p \cdot \text{refl } y \equiv p \\
\text{refl-right } & \{\mathcal{U}\} \{X\} \{x\} \{y\} \{p\} = \text{refl } p
\end{align*}
\]

And composition is associative:

\[
\begin{align*}
\cdot \text{assoc} & : \{X : \mathcal{U} \} \{x : X\} \{y : X\} \{z : X\} \{p : x \equiv y\} \{q : y \equiv z\} \{r : z \equiv t\} \\
& \to (p \cdot q) \cdot r \equiv p \cdot (q \cdot r) \\
\cdot \text{assoc } p \ q \ (\text{refl } z) & = \text{refl } (p \cdot q)
\end{align*}
\]

If we wanted to prove the above without pattern matching, this time we would need the dependent version \( \mathcal{J} \) of induction on \( _\equiv_ \).

Exercise. Try to do this with \( \mathcal{J} \) and with \( \mathcal{H} \).

But all arrows, the identifications, are invertible:

\[
\begin{align*}
^1\text{-left} & : \{X : \mathcal{U} \} \{x : X\} \{y : X\} \{p : x \equiv y\} \\
& \to p^{-1} \cdot p \equiv \text{refl } y \\
^1\text{-left } & (\text{refl } x) = \text{refl } (\text{refl } x)
\end{align*}
\]

\[
\begin{align*}
^1\text{-right} & : \{X : \mathcal{U} \} \{x : X\} \{y : X\} \{p : x \equiv y\} \\
& \to p \cdot p^{-1} \equiv \text{refl } x \\
^1\text{-right } & (\text{refl } x) = \text{refl } (\text{refl } x)
\end{align*}
\]
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

\[ p \equiv q : x \equiv y \]

we can consider the identity type \( p \equiv q \), and given

\[ u \equiv v : p \equiv q \]

we can consider the type \( u \equiv v \), 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 \( x \equiv y \) has at most one element. Such types are the sets as defined above.

Voevodsky defined the notion of \( \text{hlevel} \) to measure how long it takes for the process to trivialize.

Here are some more constructions with identifications:

\begin{align*}
\text{-1-involutive} & : \{ X : \mathcal{U} \} \{ x : X \} \{ y : X \} \ ( p : x \equiv y ) \to ( p^{-1} )^{-1} \equiv p \\
\text{-1-involutive} \ ( \text{refl} \ x ) & = \text{refl} \ ( \text{refl} \ x ) \\
\text{ap-refl} : \{ X : \mathcal{U} \} \{ y : Y \} \ ( f : X \to Y ) \ ( x : X ) \\
& \to \text{ap} \ f \ ( \text{refl} \ x ) \equiv \text{refl} \ ( \text{refl} \ f \ x ) \\
\text{ap-refl} \ f \ x & = \text{refl} \ ( \text{refl} \ f \ x ) \\
\text{ap} \ : \{ X : \mathcal{U} \} \{ y : Y \} \ ( f : X \to Y ) \ ( x : X ) \ ( y : Y ) \ ( z : X ) \ ( p : x \equiv y ) \ ( q : y \equiv z ) \\
& \to \text{ap} \ f \ ( p \cdot q ) \equiv \text{ap} \ f \ p \cdot \text{ap} \ f \ q \\
\text{ap} \ f \ p \ ( \text{refl} \ y ) & = \text{refl} \ ( \text{ap} \ f \ p ) \\
\text{ap} \ : \{ X : \mathcal{U} \} \{ y : Y \} \ ( f : X \to Y ) \ ( x : X ) \ ( y : Y ) \ ( z : X ) \\
& \to \text{ap} \ ( g \circ f ) \ p \equiv ( \text{ap} \ g \circ \text{ap} \ f ) \ p \\
\text{ap} \ g \ ( \text{refl} \ x ) & = \text{refl} \ ( \text{ap} \ g \ ( \text{refl} \ f \ x ) ) \\
\text{ap} \ : \{ X : \mathcal{U} \} \{ y : Y \} \ ( f : X \to Y ) \ ( x : X ) \ ( y : Y ) \ ( z : X ) \\
& \to \text{ap} \ ( g \circ f ) \ p \equiv ( \text{ap} \ g \circ \text{ap} \ f ) \ p \\
\text{ap} \ g \ ( \text{refl} \ x ) & = \text{refl} \ ( \text{ap} \ g \ ( \text{refl} \ f \ x ) ) \\
\text{ap} \ g \ ( \text{refl} \ x ) & = \text{refl} \ ( \text{ap} \ g \ ( \text{refl} \ f \ x ) ) \\
\end{align*}

The above functions \( \text{ap-refl} \) and \( \text{ap} \) constitute functoriality in the second argument. We also have functoriality in the first argument, in the following sense:

\begin{align*}
\text{ap-id} : \{ X : \mathcal{U} \} \{ y : Y \} \ ( f : X \to Y ) \ ( x : X ) \ ( p : x \equiv y ) \\
& \to \text{ap} \ \text{id} \ p \equiv p \\
\text{ap-id} \ ( \text{refl} \ x ) & = \text{refl} \ ( \text{refl} \ x ) \\
\text{ap} \ : \{ X : \mathcal{U} \} \{ y : Y \} \ ( f : X \to Y ) \ ( g : Y \to Z ) \ ( x : X ) \ ( p : x \equiv y ) \\
& \to \text{ap} \ ( g \circ f ) \ p \equiv ( \text{ap} \ g \circ \text{ap} \ f ) \ p \\
\text{ap} \ g \ ( \text{refl} \ x ) & = \text{refl} \ ( \text{ap} \ g \ ( \text{refl} \ f \ x ) ) \\
\text{ap} \ g \ ( \text{refl} \ x ) & = \text{refl} \ ( \text{ap} \ g \ ( \text{refl} \ f \ x ) ) \\
\end{align*}

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 the diagramatic order (as is customary in each case).

\begin{align*}
\text{transport} : \{ X : \mathcal{U} \} \{ A : X \to Y \} \ ( x : X ) \ ( y : Y ) \ ( z : X ) \ ( p : x \equiv y ) \ ( q : y \equiv z ) \\
& \to \text{transport} \ A \ p \equiv \text{transport} \ A \ q \circ \text{transport} \ A \ p \\
\end{align*}
transport \cdot A \ p \ (\text{refl} \ y) = \text{refl} \ (\text{transport} \ A \ p)

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

\[
\text{Nat} : \{X : \mathcal{U} \} \to (X \to \mathcal{V}) \to (X \to \mathcal{W}) \to \mathcal{U} \cup \mathcal{V} \cup \mathcal{W}
\]

\[
\text{Nat} \ A \ B = (x : \text{domain} \ A) \to A x \to B x
\]

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

\[
\text{Nats-are-natural} : \{X : \mathcal{U} \} \ (A : X \to \mathcal{V}) \ (B : X \to \mathcal{W}) \ (\tau : \text{Nat} \ A \ B)
\]
\[
\to \{x \ y : X\} \ (p : x \equiv y) \to \tau \ y \circ \text{transport} \ A \ p = \text{transport} \ B \ p \circ \tau \ x
\]

\[
\text{Nats-are-natural} \ A \ B \ \tau \ (\text{refl} \ x) = \text{refl} \ (\tau \ x)
\]

We will use the following constructions a number of times:

\[
\text{Nat} \Sigma : \{X : \mathcal{U} \} \ \{A : X \to \mathcal{V} \} \ \{B : X \to \mathcal{W} \} \to \text{Nat} \ A \ B \to \Sigma \ A \to \Sigma \ B
\]

\[
\text{Nat} \Sigma \ \tau \ (x , a) = (x , \tau \ x \ a)
\]

\[
\text{transport-ap} : \{X : \mathcal{U} \} \ \{Y : \mathcal{V} \} \ (A : Y \to \mathcal{W} \})
\]
\[
\to \{x \ x' : X\} \ (p : x \equiv x') \ (a : A (f \ x)) \to \text{transport} \ (A \cdot f) \ p \ a = \text{transport} \ A (\text{ap} \ f \ p) \ a
\]

\[
\text{transport-ap} \ A \ f \ (\text{refl} \ x) \ a = \text{refl} \ a
\]

Table of contents ⇩

**Identifications that depend on identifications**

If we have an identification \(p : A \equiv B\) of two types \(A\) and \(B\), and elements \(a : A\) and \(b : B\), we cannot ask directly whether \(a \equiv b\), 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

\[
\text{data} \ Color : \mathcal{U}_\infty \ \text{where}
\]

\[
\text{Black White} : \text{Color}
\]

With univalence, we will have that \(\text{Color} \equiv 2\) where \(2\) is the two-point type \(1 + 1\) with elements \(_0\) and \(_1_.\) But there will be two identifications \(p_0\ p_1 : \text{Color} \equiv 2\), one that identifies \text{Black} with \(_0\) and \text{White} with \(_1_.\) and another one that identifies \text{Black} with \(_1_.\) and \text{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 \(\text{Color} \equiv 2\), it doesn’t make sense to ask whether \text{Black} \equiv _0\ holds without specifying one of the possible inhabitants \(p_0\) and \(p_1\).

What we will have is that the functions \text{transport id} \ p_0\ and \text{transport id} \ p_1\ are the two possible bijections \(\text{Color} \to 2\) that identify colors with bits. So, it is not enough to have \(\text{Color} \equiv 2\) to be able to compare a color \(c : \text{Color}\) with a bit \(b : 2\). We need to specify which identification \(p : \text{Color} \equiv 2\) we want to consider for the comparison. The same considerations apply when we consider identifications \(p : 2 \equiv 2\).

So the meaningful comparison in the more general situation is

\[
\text{transport id} \ p \ a \equiv b
\]

for a specific

\[
p : A \equiv B,
\]

where \text{id} is the identity function of the universe where the types \(A\) and \(B\) live, and hence

\[
\text{transport id} : A \equiv B \to (A \to B)
\]

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 $\text{id}$ of the universe where $A$ and $B$ live by an arbitrary type family, which is what we do now.

If we have a type

$$X : \mathcal{U},$$

and a type family

$$A : X \to \mathcal{U},$$

and points

$$x, y : X$$

and an identification

$$p : x \equiv y,$$

then we get the identification

$$\text{ap} \ A \ p : A \ x \equiv A \ y.$$

However, if we have

$$a : A \ x,$$

$$b : A \ y,$$

we again cannot write down the identity type

$$a \equiv b.$$

This is again a non-sensical mathematical statement, 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 $\text{ap} \ A \ p$, 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.

This time, the meaningful comparison, given $p : x \equiv y$, is

$$\text{transport} \ A \ p \ a \equiv b,$$

For example, this idea applies when comparing the values of a dependent function:

$$\text{apd} : \{X : \mathcal{U}\} \{A : X \to \mathcal{U}\} \{(f : (x : X) \to A \ x) \{x y : X\}\}
\{(p : x \equiv y) \to \text{transport} \ A \ p (f \ x) \equiv f \ y\}$$

$$\text{apd} \ f \ (\text{refl} \ x) = \text{refl} \ (f \ x)$$

### Equality in $\Sigma$ Types

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

$$\text{to-\Sigma-}\equiv : \{X : \mathcal{U}\} \{A : X \to \mathcal{U}\} \{\sigma \ \tau : \Sigma \ A\}
\to (\Sigma \ p : \ \text{pr}_1 \ \sigma \equiv \ \text{pr}_1 \ \tau , \ \text{transport} \ A \ p \ (\ \text{pr}_2 \ \sigma \equiv \ \text{pr}_2 \ \tau))
\to \sigma \equiv \tau$$

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

$$\text{from-\Sigma-}\equiv : \{X : \mathcal{U}\} \{A : X \to \mathcal{U}\} \{\sigma \ \tau : \Sigma \ A\}
\to \sigma \equiv \tau
\to \Sigma \ p : \ \text{pr}_1 \ \sigma \equiv \ \text{pr}_1 \ \tau , \ \text{transport} \ A \ p \ (\ \text{pr}_2 \ \sigma \equiv \ \text{pr}_2 \ \tau)$$
We take the opportunity to establish more equations for transport and to define a dependent version of transport:

\[
\text{transport}_\Sigma (\lambda x. a) = (\text{refl} x, \text{refl} a)
\]

For the sake of readability, the above two definitions can be equivalently written as follows.

\[
\begin{align*}
\text{to}_\Sigma \equiv' & : \{X : \mathcal{U} \} \{A : X \rightarrow \mathcal{V} \} \{x : X\} \{a : A \} \{a' : A\} \rightarrow (x, a) \equiv (x', a') \\
\text{to}_\Sigma \equiv' & : \{X : \mathcal{U} \} \{A : X \rightarrow \mathcal{V} \} \{x : X\} \{a : A \} \{a' : A\} \rightarrow (x, a) \equiv (y, b) \\
\text{from}_\Sigma \equiv & : \{X : \mathcal{U} \} \{A : X \rightarrow \mathcal{V} \} \{x : X\} \{a : A \} \{a' : A\} \rightarrow \text{transport}_\Sigma (x, a) = (\text{refl} x, \text{refl} a)
\end{align*}
\]

The above gives the logical equivalence

\[
(\sigma \equiv \tau) \iff (\Sigma p : \text{pr}_1 \sigma \equiv \text{pr}_1 \tau, \text{transport}_\Sigma A p \ (\text{pr}_2 \sigma) \equiv \text{pr}_2 \tau).
\]

But this is a very weak statement when the left- and right-hand identity type 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 \(=\equiv\) of type equivalence, this characterization will become an equivalence

\[
(\sigma \equiv \tau) \iff (\Sigma p : \text{pr}_1 \sigma \equiv \text{pr}_1 \tau, \text{transport}_\Sigma A p \ (\text{pr}_2 \sigma) \equiv \text{pr}_2 \tau).
\]

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

\[
\mathbb{1} + \mathbb{1} \equiv \mathbb{1} + \mathbb{1},
\]

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 \(\text{from}_\Sigma \equiv\) 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. In other words, being an equivalence is property of \(f\), rather than data for \(f\).

The following special case of \(\text{to}_\Sigma \equiv\) is often useful:

\[
\begin{align*}
\text{to}_\Sigma \equiv' & : \{X : \mathcal{U} \} \{A : X \rightarrow \mathcal{V} \} \{x : X\} \{a : A \} \{a' : A\} \rightarrow a \equiv a' \rightarrow \text{Id} (\Sigma A) (x, a) (x, a') \\
\text{to}_\Sigma \equiv' & : \{X : \mathcal{U} \} \{A : X \rightarrow \mathcal{V} \} \{x : X\} \{a : A \} \rightarrow \text{ap} (\lambda x. (x, -)) (x, -)
\end{align*}
\]

We take the opportunity to establish more equations for transport and to define a dependent version of transport:

\[
\begin{align*}
\text{transport}_\times (\lambda x. a) & = (\text{refl} x, \text{refl} a) \\
\text{transportd} & : \{X : \mathcal{U} \} \{A : X \rightarrow \mathcal{V} \} \{x : X\} \{a : A \} \{a' : A\} \rightarrow \text{transport} (\lambda x. (x, -)) (\lambda x. (x, -)) (a, b) \equiv (\text{transport} A p a, \text{transport} B p b) \\
\text{transportd} A B a b & = \text{id}
\end{align*}
\]

\[
\begin{align*}
\text{transport}_\Sigma & : \{X : \mathcal{U} \} \{A : X \rightarrow \mathcal{V} \} \{x : X\} \{a : A \} \{a' : A\} \rightarrow (\lambda x. (x, -)) (\lambda x. (x, -)) (a, b) \equiv (\text{transport} A p a, \text{transport} B p b) \\
\text{transport}_\Sigma A B & = \text{id}
\end{align*}
\]
Voevodsky’s notion of hlevel

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

The $\text{hlevel}$ relation is defined by induction on $\mathbb{N}$, with the induction step working with the identity types of the elements of the type in question:

\[
\text{is-of-hlevel} : \mathcal{U} \to \mathbb{N} \to \mathcal{U}
\]

\[
\text{is-of-hlevel} \emptyset = \text{is-singleton} X
\]

\[
\text{is-of-hlevel} (\text{succ} n) = \{(x, x') : X \to (x \equiv x') \text{ is-of-hlevel } n\}
\]

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

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:

\[
\text{wconstant} : \{X : \mathcal{U} \} \to \{Y : \mathcal{V} \} \to (X \to Y) \to \mathcal{U} \sqcup \mathcal{V}
\]

\[
\text{wconstant} f = (x, x') : \text{domain } f \to f x \equiv f x'
\]

The prefix “w” officially stands for “weakly”. Perhaps incoherently constant or wildly constant would be better terminologies, with coherence understood in the $\infty$-categorical sense. We prefer to stick to wildly rather than weakly, and luckily both start with the letter “w”.

We first define the type of constant endomaps of a given type:

\[
\text{wconstant-endomap} : \mathcal{U} \to \mathcal{U}
\]

\[
\text{wconstant-endomap} X = \Sigma f : (X \to X), \text{wconstant } f
\]

\[
\text{wcmap} : (X : \mathcal{U}) \to \text{wconstant-endomap } X \to (X \to X)
\]

\[
\text{wcmap} (X, \text{wcmap} X) = f
\]

\[
\text{wcmap-constancy} : (X : \mathcal{U}) (c : \text{wconstant-endomap } X) \to \text{wconstant } (\text{wcmap } X, c)
\]

\[
\text{wcmap-constancy} X (f, w) = w
\]

The point is that a type is a set if and only if its identity types all have designated $\text{wconstant}$ endomaps:

\[
\text{Hedberg} : \{X : \mathcal{U} \} (x : X) \to ((y : X) \to \text{wconstant-endomap } (x \equiv y))
\]

\[
\text{Hedberg} \{\mathcal{U} \} X (x, y, p, q) =
\]

\[
(p x (\text{refl } x)) \equiv (a, y, p) \equiv \text{ap } f x (\text{refl } x)
\]
\[(f \cdot (\text{refl } x))^{-1} \cdot f \ y \ q \equiv (a \ y \ q)^{-1}\]
\[q\]

where
\[f : (y : X) \to x \equiv y \to x \equiv y\]
\[f \ y = \text{wcmap} \ (x \equiv y) \ (c \ y)\]
\[\kappa : (y : X) \ (p \ q : x \equiv y) \to f \ y \ p \equiv f \ y \ q\]
\[\kappa \ y = \text{wcmap-constancy} \ (x \equiv y) \ (c \ y)\]
\[a : (y : X) \ (p : x \equiv y) \to p \equiv (f \cdot (\text{refl } x))^{-1} \cdot f \ y \ p\]
\[a \ x \ (\text{refl } x) = (-^{1} \cdot \text{left} \cdot (f \cdot (\text{refl } x))^{-1}\]

### A characterization of sets

We consider types whose identity types all have designated constant endomaps:

\[\text{wconstant-≡-endomaps} : \mathcal{U} \to \mathcal{U}\]
\[\text{wconstant-≡-endomaps} \ X = (x \ y : X) \to \text{wconstant-endomap} \ (x \equiv y)\]

The following is immediate from the definitions.

\[\text{sets-have-wconstant-≡-endomaps} : (X : \mathcal{U}) \to \text{is-set} \ X \to \text{wconstant-≡-endomaps} \ X\]
\[\text{sets-have-wconstant-≡-endomaps} \ s \ x \ y = (f , \kappa)\]

where
\[f : x \equiv y \to x \equiv y\]
\[f \ p = p\]
\[\kappa : (p \ q : x \equiv y) \to f \ p \equiv f \ q\]
\[\kappa \ p \ q = s \ x \ y \ p \ q\]

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

\[\text{types-with-wconstant-≡-endomaps-are-sets} : (X : \mathcal{U}) \to \text{is-set} \ X \to \text{wconstant-≡-endomaps} \ X\]
\[\text{types-with-wconstant-≡-endomaps-are-sets} \ X \ c \ x = \text{Hedberg} \ x \ (\lambda \ y \to \text{wcmap} \ (x \equiv y) \ (c \ x \ y) , \text{wcmap-constancy} \ (x \equiv y) \ (c \ x \ y))\]

### 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 \(\text{wconstant}\) function:

\[\text{subsingleton-has-wconstant-≡-endomaps} : (X : \mathcal{U}) \to \text{is-subsingleton} \ X \to \text{wconstant-≡-endomaps} \ X\]
\[\text{subsingleton-has-wconstant-≡-endomaps} \ s \ x \ y = (f , \kappa)\]

where
\[f : x \equiv y \to x \equiv y\]
\[f \ p = s \ x \ y\]
\[\kappa : (p \ q : x \equiv y) \to f \ p \equiv f \ q\]
\[\kappa \ p \ q = \text{refl} \ (s \ x \ y)\]

And the corollary is that (sub)singleton types are sets.

\[\text{subsingleton-are-sets} : (X : \mathcal{U}) \to \text{is-subsingleton} \ X \to \text{is-set} \ X\]
\[\text{subsingleton-are-sets} \ X \ s = \text{types-with-wconstant-≡-endomaps-are-sets} \ X \ (\text{subsingleton-has-wconstant-≡-endomaps} \ X \ s)\]
singletons-are-sets : \((X : \mathcal{U}^\prime) \rightarrow is-singleton X \rightarrow is-set X\)
singletons-are-sets \(X = \) subsingletons-are-sets \(X \circ\) singletons-are-subsingletons \(X\)

In particular, the types \(\emptyset\) and \(\mathbb{1}\) are sets.

\(\emptyset\)-is-set : is-set \(\emptyset\)
\(\emptyset\)-is-set = subsingletons-are-sets \(\emptyset \circ\) \(\emptyset\)-is-subsingleton

\(\mathbb{1}\)-is-set : is-set \(\mathbb{1}\)
\(\mathbb{1}\)-is-set = subsingletons-are-sets \(\mathbb{1} \circ\) \(\mathbb{1}\)-is-subsingleton

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:

subsingles-are-of-hlevel-1 : \((X : \mathcal{U}^\prime) \rightarrow is-subsingleton X \rightarrow X is-of-hlevel 1\)

**subsingletons-are-of-hlevel-1** \(X = g\)
where
\(g : ((x y : X) \rightarrow x \equiv y) \rightarrow (x y : X) \rightarrow is-singleton (x \equiv y)\)
\(g t x y = t x y, \quad subsingletons-are-sets X t x y (t x y)\)

**types-of-hlevel-1-are-subsingletons** : \((X : \mathcal{U}^\prime) \rightarrow X is-of-hlevel 1 \rightarrow is-subsingleton X\)

**types-of-hlevel-1-are-subsingletons** \(X = f\)
where
\(f : ((x y : X) \rightarrow is-singleton (x \equiv y)) \rightarrow (x y : X) \rightarrow x \equiv y\)
\(f s x y = center (x \equiv y) (s x y)\)

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 : \((X : \mathcal{U}^\prime) \rightarrow is-set X \rightarrow X is-of-hlevel 2\)
sets-are-of-hlevel-2 \(X = g\)
where
\(g : ((x y : X) \rightarrow is-subsingleton (x \equiv y)) \rightarrow (x y : X) \rightarrow (x \equiv y) is-of-hlevel 1\)
\(g t x y = subsingletons-are-of-hlevel-1 (x \equiv y) (t x y)\)

**types-of-hlevel-2-are-sets** : \((X : \mathcal{U}^\prime) \rightarrow X is-of-hlevel 2 \rightarrow is-set X\)
**types-of-hlevel-2-are-sets** \(X = f\)
where
\(f : ((x y : X) \rightarrow (x \equiv y) is-of-hlevel 1) \rightarrow (x y : X) \rightarrow is-subsingleton (x \equiv y)\)
\(f s x y = types-of-hlevel-1-are-subsingletons (x \equiv y) (s x y)\)

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:
To be consistent with the above terminology, we have to stipulate that all types have hlevel \( \infty \). We don’t need a definition for this 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 \( \infty \).

**Exercise.** Formulate and prove the following. The type \( \mathbb{1} \) has minimal hlevel \( 0 \).

\[
\text{has-minimal-hlevel} : \mathcal{U} \to \mathbb{N} \to \mathcal{U}
\]

\( X \text{ has-minimal-hlevel } 0 = X \text{ is-of-hlevel } 0 \)

\( X \text{ has-minimal-hlevel } (\text{succ } n) = (X \text{ is-of-hlevel } (\text{succ } n)) \times \neg(X \text{ is-of-hlevel } n) \)

The type \( \emptyset \) has minimal hlevel \( 1 \), the type \( \mathbb{N} \) has minimal hlevel \( 2 \). The solution to the fact that \( \mathbb{N} \) has hlevel \( 2 \) is given in the next section. More ambitiously, after univalence is available, show that the type of monoids has minimal hlevel \( 3 \).

Table of contents

**N and 2 are sets**

If a type has decidable equality we can define a \textit{wconstant} function \( x \equiv y \to x \equiv y \) and hence conclude that it is a set. This argument is due to Hedberg.

\[
\text{pointed-types-have-wconstant-endomap} : \{X : \mathcal{U}\} \to X \to \text{wconstant-endomap } X
\]

\( \text{pointed-types-have-wconstant-endomap } X = ((\lambda y \to x), (\lambda y y' \to \text{refl } x)) \)

\[
\text{empty-types-have-wconstant-endomap} : \{X : \mathcal{U}\} \to \text{is-empty } X \to \text{wconstant-endomap } X
\]

\( \text{empty-types-have-wconstant-endomap } e = (id, (\lambda x x' \to \bot (x \equiv x')) (e x)) \)

\[
\text{decidable-has-wconstant-endomap} : \{X : \mathcal{U}\} \to \text{decidable } X \to \text{wconstant-endomap } X
\]

\( \text{decidable-has-wconstant-endomap } (\text{inl } x) = \text{pointed-types-have-wconstant-endomap } X \)

\( \text{decidable-has-wconstant-endomap } (\text{inr } e) = \text{empty-types-have-wconstant-endomap } e \)

\[
\text{hedberg-lemma} : \{X : \mathcal{U}\} \to \text{has-decidable-equality } X \to \text{wconstant-≡-endomaps } X
\]

\( \text{hedberg-lemma } \{U\} \{X\} d x y = \text{decidable-has-wconstant-endomap } (d x y) \)

\[
\text{hedberg} : \{X : \mathcal{U}\} \to \text{has-decidable-equality } X \to \text{is-set } X
\]

\( \text{hedberg } \{U\} \{X\} d = \text{types-with-wconstant-≡-endomaps-are-sets } X (\text{hedberg-lemma } d) \)

\( \mathbb{N}\text{-is-set} : \text{is-set } \mathbb{N} \)

\( \mathbb{N}\text{-is-set} = \text{hedberg } \mathbb{N}\text{-has-decidable-equality} \)

\( \mathbb{Z}\text{-is-set} : \text{is-set } \mathbb{Z} \)

\( \mathbb{Z}\text{-is-set} = \text{hedberg } \mathbb{Z}\text{-has-decidable-equality} \)

Notice that excluded middle implies directly that all sets have decidable equality, so that in its presence a type is a set if and only if it has decidable equality.

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:

$$\text{has-section} : \{X : U\} \{Y : V\} \rightarrow (X \rightarrow Y) \rightarrow U \uplus V$$
$$\text{has-section} \; r \; = \; \Sigma \; s \; : \; (\text{codomain} \; r \rightarrow \text{domain} \; r), \; r \circ s \; \sim \; \text{id}$$

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

We say that $X$ is a retract of $Y$, written $X \triangleleft Y$, if we have a function $Y \rightarrow X$ which has a section:

$$\triangleleft : U \rightarrow V \rightarrow U \uplus V$$
$$X \triangleleft Y = \Sigma \; r \; : \; (Y \rightarrow X), \; \text{has-section} \; r$$

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:

$$\text{retraction} : \{X : U\} \{Y : V\} \rightarrow X \triangleleft Y \rightarrow Y \rightarrow X$$
$$\text{retraction} \; (r \; , \; s \; , \; \eta) \; = \; r$$

$$\text{section} : \{X : U\} \{Y : V\} \rightarrow X \triangleleft Y \rightarrow X \rightarrow Y$$
$$\text{section} \; (r \; , \; s \; , \; \eta) \; = \; s$$

$$\text{retract-equation} : \{X : U\} \{Y : V\} \{\rho : X \triangleleft Y\} \rightarrow \text{retraction} \; \rho \circ \text{section} \; \rho \; \sim \; \text{id} \; X$$

$$\text{retract-equation} \; (r \; , \; s \; , \; \eta) \; = \; \eta$$

$$\text{retraction-has-section} : \{X : U\} \{Y : V\} \{\rho : X \triangleleft Y\} \rightarrow \text{has-section} \; (\text{retraction} \; \rho)$$

$$\text{retraction-has-section} \; (r \; , \; h) \; = \; h$$

We have an identity retraction:

$$\text{id-\triangleleft} : (X : U) \rightarrow X \triangleleft X$$
$$\text{id-\triangleleft} \; X \; = \; \text{id} \; X \; , \; \text{id} \; X \; , \; \text{refl}$$

**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) $\mathbb{N} \triangleleft \mathbb{N}$ with the function $\text{pred} : \mathbb{N} \rightarrow \mathbb{N}$ defined above as the retraction. Try to produce more inhabitants of this type.

We can define the composition of two retraction as follows:

$$\triangleleft \circ \triangleleft : \{X : U\} \{Y : V\} \{Z : W\} \rightarrow X \triangleleft Y \rightarrow Y \triangleleft Z \rightarrow X \triangleleft Z$$

$$(r \; , \; s \; , \; \eta) \; \triangleleft \circ \; (r' \; , \; s' \; , \; \eta') \; = \; (r \circ r' \; , \; s' \circ s \; , \; \eta''')$$

where

$$\eta''' \; = \; \lambda \; x \rightarrow r \; (r' \; (s' \; (s \; x))) \; \; \text{ap} \; r \; (\eta' \; (s \; x)) \; x$$

We also define composition with an implicit argument made explicit:
We have defined \( \Sigma \text{-reindexing-retract} \) and \( \Sigma \text{-retract} \) and \( \text{transport-is-section} \) directly by induction on \( \Sigma \text{-reindexing-retract} \) and \( \Sigma \text{-retract} \).

A pointwise retraction gives a retraction of the total spaces:

\[
\Sigma \text{-retract} \{X : U \} \{A : X \to Y \} \{B : X \to W \} \\
\to \{(x : X) \to A x \triangleleft B x\} \sto \Sigma A \triangleleft \Sigma B
\]

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 \( \Sigma \) types, which are of general use, in particular for dealing with equivalences in the sense of Voevosky in comparison with invertible maps.

We have that \( \text{transport} A (p^{-1}) \) is a two-sided inverse of \( \text{transport} A p \) using the functoriality of \( \text{transport} A \), or directly by induction on \( p \):

\[
\text{transport-is-retraction} : \{X : U \} \{A : X \to Y \} \{X y : X\} \{p : X \equiv y\} \\
\to \text{transport} A p \circ \text{transport} A (p^{-1}) \equiv \text{id} (A y)
\]

Using this, we have the following reindexing retraction of \( \Sigma \) types:

\[
\Sigma \text{-reindexing-retract} : \{X : U \} \{Y : V \} \{A : X \to W \} \{r : Y \to X\} \\
\to \text{has-section} r \\
\to \{(\Sigma x : X, A x) \triangleleft (\Sigma y : Y, A (r y))\}
\]

We have defined the property of a type being a singleton. The singleton type \( \Sigma x : X, x \equiv y \) induced by a point \( x : X \) of a type \( X \) is denoted by \( \text{singleton-type} x \). The terminology is justified by the fact that it is indeed a singleton in the sense
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.

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:

\[ \text{invertible} : \{ X : U \} \rightarrow X \rightarrow U \]

\[ \text{invertible} f = \Sigma g : (\text{codomain } f \rightarrow \text{domain } f) \times (g \circ f \sim \text{id}) \times (f \circ g \sim \text{id}) \]

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 have that the inverse \( g \) is unique when it exists, we cannot in general prove that the identification data \( g \circ f \sim id \) and \( f \circ g \sim id \) 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:

\[
fiber : \{X : \mathcal{U}\} \{Y : \mathcal{V}\} (f : X \to Y) \to Y \cup \mathcal{V}
\]

\[
fiber f y = \Sigma x : \text{domain } f \to f x = y
\]

\[
fiber-point : \{X : \mathcal{U}\} \{Y : \mathcal{V}\} \{f : X \to Y\} \{y : Y\} 
\to fiber f y \to X
\]

\[
fiber-point (x, p) = x
\]

\[
fiber-identification : \{X : \mathcal{U}\} \{Y : \mathcal{V}\} \{f : X \to Y\} \{y : Y\} 
\to (w : fiber f y) \to f (fiber-point w) \equiv y
\]

\[
fiber-identification (x, p) = p
\]

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 \( x : X \) 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”.

\[
is-equiv : \{X : \mathcal{U}\} \{Y : \mathcal{V}\} \to (X \to Y) \to \mathcal{U} \cup \mathcal{V}
\]

\[
is-equiv f = (y : \text{codomain } f) \to is-singleton (fiber f y)
\]

We can read this as saying that for every \( y : Y \) there is a unique \( x : X \) with \( f x = y \), where the uniqueness refers not only to \( x : X \) but also to the identification datum \( p : f x = y \). More precisely, the pair \( (x, p) \) is required to be unique. It is easy to see that equivalences are invertible:

\[
\text{inverse} : \{X : \mathcal{U}\} \{Y : \mathcal{V}\} (f : X \to Y) \to \text{is-equiv } f \to (Y \to X)
\]

\[
\text{inverse } f e y = fiber-point (\text{center } (fiber f y) (e y))
\]

\[
\text{inverses-are-sections} : \{X : \mathcal{U}\} \{Y : \mathcal{V}\} (f : X \to Y) (e : \text{is-equiv } f) 
\to f \circ \text{inverse } f e \sim id
\]

\[
\text{inverses-are-sections } f e y = fiber-identification (\text{center } (fiber f y) (e y))
\]

\[
\text{inverse-centrality} : \{X : \mathcal{U}\} \{Y : \mathcal{V}\} 
\quad (f : X \to Y) (e : \text{is-equiv } f) (y : Y) (t : fiber f y) 
\quad \to (\text{inverse } f e y, \text{inverses-are-sections } f e y) \equiv t
\]

\[
\text{inverse-centrality } f e y = centrality (fiber f y) (e y)
\]
We need to show that the fiber for a map \( f : X \to Y \) is a retract of the type \( \equiv y \) for any given \( y : Y \).

1. We first use the assumption \( \varepsilon \) to show that the type \( f (g \ y) \equiv y \) is a retract of the type \( \equiv y \) for any given \( y : Y \).

   To get the section \( s : f (g \ y) \equiv y \to y \equiv y_0 \), we transport along the identification \( \varepsilon : f (g \ y) \equiv y \) over the family \( \langle - \equiv y_0 \rangle \), which can be abbreviated as \( - \equiv y_0 \).

   To get the retraction \( r \) in the opposite direction, we transport along the inverse of the identification \( \varepsilon \) over the same family.

   We already know that this gives a section-retraction pair by \( \text{transport-is-section} \).

2. Next we have that the type \( \Sigma x : X \, f x \equiv y_0 \) is a retract of the type \( \Sigma y : Y \, f (g \ y) \equiv y_0 \) by \( \Sigma\text{-reindexing-retract} \) using the assumption that \( \eta \) exhibits \( g \) as a section of \( f \), which in turn is a retract of the type \( \Sigma y : Y \, y \equiv y_0 \) by applying \( \Sigma \) to both sides of the retraction \( (f (g \ y) \equiv y_0) \triangleleft (y \equiv y_0) \) of the previous step.

   This amounts to saying that the type \( \text{fiber} \ f \ y_0 \) is a retract of \( \text{singleton-type} \ y_0 \).

3. But then we are done, because singleton types are singletons and retracts of singletons are singletons.
An immediate consequence is that inverses of equivalences are themselves equivalences:

\[
\text{inverses-are-equivs} : \{x : U \} \{y : V \} \{z : W \} \{f : x \to y\} \{g : y \to z\} \rightarrow \text{is-equiv} (g \circ f) \equiv \text{is-equiv} (f \circ g)
\]

We can compose invertible maps:

\[
\circ \text{-invertible} : \{x : U \} \{y : V \} \{z : W \} \{f' : y \to z\} \rightarrow \text{invertible} (f \circ g) \rightarrow \text{invertible} (g \circ f)
\]

where

\[
\begin{align*}
\eta & = \lambda x \to g \ (g' \ (f \ x)) \equiv (\text{ap} \ g \ (g' \ (f \ x))) \\
\varepsilon & = \lambda z \to f' \ (f \ (g' \ z)) \equiv (\text{ap} \ f' \ (f \ (g' \ z)))
\end{align*}
\]

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

\[
\text{id-is-equiv} : \{x : U \} \rightarrow \text{is-equiv} (\text{id} \ x)
\]

Because we have made the above definition abstract, we don’t have access to the given construction when proving things involving \(\circ \text{-invertible}\), such as the contravariance of inversion:
The type of equivalences is defined as follows:

\[
\equiv : \mathcal{U} \times \mathcal{V} \rightarrow \mathcal{U} \cup \mathcal{V}
\]

\[
X = Y = \Sigma f : (X \rightarrow Y), \text{is-equiv } f
\]

Notice that this doesn’t just say that \( X \) and \( Y \) are equivalent: the type \( X = Y \) collects all the ways in which the types \( X \) and \( Y \) are equivalent. For example, the two-point type \( 2 \) is equivalent to itself in two ways, by the identity map, and by the map that interchanges its two points, and hence the type \( 2 = 2 \) has two elements.

Again it is convenient to have special names for its first and second projections:

\[
\begin{align*}
\text{Eq-fun } & \equiv : \{ X : \mathcal{U} \} \{ Y : \mathcal{V} \} \rightarrow X = Y \rightarrow X = Y \\
\text{Eq-fun } & (f, i) = f \\
\text{Eq-fun } & (\_ = \_ ) = \text{Eq-fun}
\end{align*}
\]

\[
\begin{align*}
\text{Eq-fun-is-equiv } & \equiv : \{ X : \mathcal{U} \} \{ Y : \mathcal{V} \} (e : X = Y) \rightarrow \text{is-equiv } (\_ = \_ )
\end{align*}
\]

\[
\begin{align*}
\text{invertibility-gives-}\equiv & : \{ X : \mathcal{U} \} \{ Y : \mathcal{V} \} (f : X \rightarrow Y) \\
& \rightarrow \text{invertible } f \rightarrow X = Y
\end{align*}
\]

\[
\text{invertibility-gives-}\equiv = f \mapsto f, \text{invertibles-are-equivs } f \mapsto f
\]

Examples:

\[
\begin{align*}
\text{Σ-induction-}\equiv & : \{ X : \mathcal{U} \} \{ Y : \mathcal{V} \} \{ A : \Sigma Y \rightarrow \mathcal{W} \} \\
& \rightarrow \text{((} (X : X) (y : Y x) \rightarrow A (x, y)) = ((z : \Sigma Y) \rightarrow A z)}
\end{align*}
\]

\[
\begin{align*}
\text{Σ-induction-}\equiv = \text{invertibility-gives-}\equiv \Sigma \text{-induction } (\text{curry}, \text{refl}, \text{refl})
\end{align*}
\]

\[
\begin{align*}
\text{Σ-flip } & : \{ X : \mathcal{U} \} \{ Y : \mathcal{V} \} \{ A : X \rightarrow Y \rightarrow \mathcal{W} \} \\
& \rightarrow (\Sigma x : X, \Sigma y : Y, A x y) = (\Sigma y : Y, \Sigma x : X, A x y)
\end{align*}
\]

\[
\begin{align*}
\text{Σ-flip } = \text{invertibility-gives-}\equiv (\lambda (x, y, p) \rightarrow (y, x, p)) \\
& \quad ((\lambda (y, x, p) \rightarrow (x, y, p)), \text{refl}, \text{refl})
\end{align*}
\]

\[
\begin{align*}
\text{x-comm } & : \{ X : \mathcal{U} \} \{ Y : \mathcal{V} \} \\
& \rightarrow (X \times Y) = (Y \times X)
\end{align*}
\]

\[
\begin{align*}
\text{x-comm } = \text{invertibility-gives-}\equiv (\lambda (x, y) \rightarrow (y, x)) \\
& \quad ((\lambda (y, x) \rightarrow (x, y)), \text{refl}, \text{refl})
\end{align*}
\]

The identity equivalence and the composition of two equivalences:

\[
\begin{align*}
\text{id-} & \equiv : (X : \mathcal{U}) \rightarrow X = X \\
\text{id-} & = X \mapsto id X, \text{id-is-equiv } X
\end{align*}
\]
Alternatively, we could have used the fact that transport A p is an equivalence.

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

Here is the promised characterization of equality in Σ types:

Σ≡≡ : {X : U'} {A : X → Y'} (s t : Σ A)
→ (s ≡ t) ≡ (Σ p : pr₁ s ≡ pr₁ t , transport A p (pr₂ s) ≡ pr₂ t)
Σ≡≡ = {U} {V} {X} {Y} σ τ = invertibility-gives-≡ from-Σ≡ (to-Σ≡ , η , ε)
where
η : (q : σ ≡ τ) → to-Σ≡ (from-Σ≡ q) ≡ q
η (refl σ) = refl (refl σ)
ε : (w : Σ p : pr₁ σ ≡ pr₁ τ , transport A p (pr₂ σ) ≡ pr₂ τ)
→ from-Σ≡ (to-Σ≡ w) ≡ w
ε (refl p , refl q) = refl (refl p , refl q)

Similarly we have:

to-x≡ : {X : U'} {Y : Y'} {t z : X × Y}
→ (pr₁ z ≡ pr₁ t) × (pr₂ z ≡ pr₂ t) → z ≡ t
to-x≡ (refl x , refl y) = refl (x , y)

from-x≡ : {X : U'} {Y : Y'} {t z : X × Y}
→ z ≡ t → (pr₁ z ≡ pr₁ t) × (pr₂ z ≡ pr₂ t)
from-x≡ {U} {V} {X} {Y} (refl (x , y)) = (refl x , refl y)

x≡≡ : {X : U'} {Y : Y'} {Z t : X × Y}
→ (Z t ≡ t) ≡ (pr₁ Z t ≡ pr₁ t) × (pr₂ Z t ≡ pr₂ t)
x≡≡ = {U} {V} {X} {Y} Z t = invertibility-gives-≡ from-x≡ (to-x≡ , η , ε)
where
η : (p : Z t ≡ t) → to-x≡ (from-x≡ p) ≡ p
η (refl Z) = refl (refl Z)
ε : (q : (pr₁ Z t ≡ pr₁ t) × (pr₂ Z t ≡ pr₂ t)) → from-x≡ (to-x≡ q) ≡ q
ε (refl x , refl y) = refl (refl x , refl y)

The following are often useful:

ap-pr₁-to-x≡ : {X : U'} {Y : Y'} {Z t : X × Y}
→ (p₁ : pr₁ Z t ≡ pr₁ t)
→ (p₂ : pr₂ Z t ≡ pr₂ t)
→ ap (pr₁ (to-x≡ (p₁ , p₂))) ≡ p₁
**Voevodsky’s univalence axiom**

There is a canonical transformation \((X \ Y : \mathcal{U}) \to X \equiv Y \to X \simeq Y\) that sends the identity identification \(\text{refl} \ X : X \equiv X\) to the identity equivalence \(\text{id} \simeq X : X \simeq X\). The univalence axiom, for the universe \(\mathcal{U}\), says that this canonical map is itself an equivalence.

\[
\text{Id-Eq : (X \ Y : \mathcal{U})} \to X \equiv Y \to X \simeq Y
\]

\[
\text{Id-Eq} \ X \ X \ (\text{refl} \ X) = \text{id} \simeq X
\]

\[
\text{is-univalent} : (\mathcal{U} : \text{Universe}) \to \mathcal{U} \vdash \quad \text{is-equiv} (\text{Id-Eq} \ X \ Y)
\]

Thus, the univalence of the universe \(\mathcal{U}\) says that identifications \(X \equiv Y\) of types in \(\mathcal{U}\) are in canonical bijection with equivalences \(X \simeq Y\), if by bijection we mean equivalence, where the canonical bijection is \(\text{Id-Eq}\).
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).

\[
\text{univalence-} ≃ : \text{is-univalent } 𝓤 → (X : 𝓤) → X ≃ Y ≃ (X ≃ Y)
\]

\[
\text{univalence-} ≃ \text{ ua } X Y = \text{Id-Eq } X Y , \text{ ua } X Y
\]

\[
\text{Eq-Id} : \text{is-univalent } 𝓤 → (X : 𝓤) → X ≃ Y \to X ≡ Y
\]

\[
\text{Eq-Id} \text{ ua } X Y = \text{inverse } (\text{Id-Eq } X Y) (\text{ua } X Y)
\]

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

\[
\text{Id-fun} :\{X : 𝓤\} → X ≡ Y \to X \to Y
\]

\[
\text{Id-fun } \{U\} \{X\} \{Y\} p = \text{Id-Eq } X Y p
\]

\[
\text{Id-funs-agree} : \{X : 𝓤\} (p : X ≡ Y) \to \text{Id-fun } p ≡ \text{Id-Fun } p
\]

\[
\text{Id-funs-agree} (\text{refl } X) = \text{refl } (\text{id } X)
\]

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 that are at the heart of univalent mathematics. 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 in MLTT, 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 ⇧**

**Example of a type that is not a set under univalence**

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

\[
\text{swap}_2 : 2 \to 2
\]

\[
\text{swap}_2 0 = 1
\]

\[
\text{swap}_2 1 = 0
\]

\[
\text{swap}_2^\text{-involutive} : (n : 2) \to \text{swap}_2 (\text{swap}_2 n) ≡ n
\]

\[
\text{swap}_2^\text{-involutive} 0 = \text{refl} 0
\]

\[
\text{swap}_2^\text{-involutive} 1 = \text{refl} 1
\]

That is, \(\text{swap}_2\) is its own inverse and hence is an equivalence:

\[
\text{swap}_2^\text{-is-equiv} : \text{is-equiv } \text{swap}_2
\]

\[
\text{swap}_2^\text{-is-equiv} = \text{invertibles-are-equivs}
\]

\[
\text{swap}_2 (\text{swap}_2 , \text{swap}_2^\text{-involutive} , \text{swap}_2^\text{-involutive})
\]

We now use a local module to assume univalence of the first universe in the construction of our example:

\[
\text{module example-of-a-nonset } (\text{ua} : \text{is-univalent } 𝓤_0) \text{ where}
\]

The above gives two distinct equivalences:

\[
e_0 e_1 : 2 ≃ 2
\]

\[
e_0 = \text{id-} ≃ 2
\]

\[
e_1 = \text{swap}_2 , \text{swap}_2^\text{-is-equiv}
\]

\[
e_0^\text{-is-not-e}_1 : e_0 ≢ e_1
\]

\[
e_0^\text{-is-not-e}_1 p = _1^\text{-is-not-}^0 r
\]

where

\[
q : \text{id} ≡ \text{swap}_2
\]

\[
q = \text{ap }^\text{-} p
\]

\[
r : _1 ≡ _0
\]

\[
r = \text{ap } (λ \to \to _1) q
\]
Using univalence, we get two different identifications of the type \( \mathbb{2} \) with itself:

\[
p_0, p_1 : \mathbb{2} \equiv \mathbb{2} \\
p_0 = \text{Eq-Id ua} \ (\mathbb{2} \ 2 \ e_0) \\
p_1 = \text{Eq-Id ua} \ (\mathbb{2} \ 2 \ e_1)
\]

\[
p_0 \text{-is-not-} p_1 : p_0 \not\equiv p_1 \\
p_0 \text{-is-not-} p_1 q = e_0 \text{-is-not-} e_1 r
\]

where

\[
r = e_0 \\
\text{Id-Eq} \ 2 \ 2 \ p_0 \equiv (\text{inverses-are-sections} \ (\text{Id-Eq} \ 2 \ 2) \ (\text{ua} \ 2 \ 2) \ e_0)\^{-1} \\
\text{Id-Eq} \ 2 \ 2 \ p_1 \equiv (\text{inverses-are-sections} \ (\text{Id-Eq} \ 2 \ 2) \ (\text{ua} \ 2 \ 2) \ e_1) \\
e_1
\]

If the universe \( \mathcal{U}_0 \) were a set, then the identifications \( p_0 \) and \( p_1 \) defined above would be equal, and therefore it is not a set.

\[
\mathcal{U}_0 \text{-is-not-a-set} : \neg (\text{is-set} \ (\mathcal{U}_0 \ .))
\]

\[
\mathcal{U}_0 \text{-is-not-a-set} s = p_0 \text{-is-not-} p_1 q
\]

where

\[
q : p_0 \equiv p_1 \\
q = s \ 2 \ 2 \ p_0 \ p_1
\]

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.

Table of contents ⇧

Formulations

Define functions for the following type declarations. As a matter of procedure, we suggest to import this file in a solutions file and add another declaration with the same type and new name e.g. \texttt{sections-are-lc-solution}, because we already have solutions in this file. It is important not to forget to include the option \texttt{--without-K} in the solutions file that imports (the Agda version of) this file.

\[
\text{subsingleton-criterion} : \{ X : \mathcal{U} \ . \} \\
\quad \rightarrow (X \rightarrow \text{is-singleton} \ X) \\
\quad \rightarrow \text{is-subsingleton} \ X
\]

\[
\text{subsingleton-criterion}' : \{ X : \mathcal{U} \ . \} \\
\quad \rightarrow (X \rightarrow \text{is-subsingleton} \ X) \\
\quad \rightarrow \text{is-subsingleton} \ X
\]

\[
\text{retract-of-subsingleton} : \{ X : \mathcal{U} \ . \} \{ Y : \mathcal{V} \ . \} \\
\quad \rightarrow Y \triangleleft X \rightarrow \text{is-subsingleton} \ X \rightarrow \text{is-subsingleton} \ Y
\]

\[
\text{left-cancellable} : \{ X : \mathcal{U} \ . \} \{ Y : \mathcal{V} \ . \} \rightarrow (X \rightarrow Y) \rightarrow \mathcal{U} \sqcup \mathcal{V} \\
\text{left-cancellable} \ f = \{ x \ x' : \text{domain} \ f \} \rightarrow f x \equiv f x' \rightarrow x \equiv x'
\]

\[
\text{lc-maps-reflect-subsingletons} : \{ X : \mathcal{U} \ . \} \{ Y : \mathcal{V} \ . \} (f : X \rightarrow Y) \\
\quad \rightarrow \text{left-cancellable} \ f \\
\quad \rightarrow \text{is-subsingleton} \ Y \\
\quad \rightarrow \text{is-subsingleton} \ X
\]

\[
\text{has-retraction} : \{ X : \mathcal{U} \ . \} \{ Y : \mathcal{V} \ . \} \rightarrow (X \rightarrow Y) \rightarrow \mathcal{U} \sqcup \mathcal{V} \\
\text{has-retraction} \ s = \Sigma r : (\text{codomain} \ s \rightarrow \text{domain} \ s), r \circ s \sim \text{id}
\]
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'}
→ X ≃ Y
→ is-subsingleton Y
→ is-subsingleton X

comp-inverses : {X : U'} {Y : V'} {Z : W'}
(f : X → Y) (g : Y → Z)
(i : is-equiv f) (j : is-equiv g)
(f' : Y → X) (g' : Z → Y)
→ f' ~ inverse f i
→ g' ~ inverse g j
→ f' ⋅ g' ~ inverse (g ⋅ f) (is-equiv j i)

equiv-to-set : {X : U'} {Y : V'}
→ X ≃ Y
→ is-set Y
→ is-set X

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 André 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:

one-inverse : (X : U') (Y : V')
(f : X → Y) (r s : Y → X)
→ (r ⋅ f ~ id)
→ (f ⋅ s ~ id)
→ r ~ s

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

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

invertibles-are-joyal-eqivs : {X : U'} {Y : V'} (f : X → Y)
maps-of-singletons-are-equivs : \{X : U\} \{Y : V\} (f : X \to Y) 
\to \text{is-equiv } f \to \text{is-joyal-equiv } f

equvs-closed-under-\sim : \{X : U\} \{Y : V\} \{f g : X \to Y\} 
\to \text{is-equiv } f 
\to g \sim f 
\to \text{is-equiv } g

equv-to-singleton' : \{X : U\} \{Y : V\} 
\to X \equiv Y \to \text{is-singleton } X \to \text{is-singleton } Y

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

pr_1-lc : \{X : U\} \{A : X \to V\} 
\to ((x : X) \to \text{is-ssubsingleton } (A x)) 
\to \text{left-cancellable } (\lambda (t : \Sigma A) \to \text{pr}_1 t)

subsets-of-sets-are-sets : (X : U) (A : X \to V) 
\to \text{is-set } X 
\to ((x : X) \to \text{is-ssubsingleton } (A x)) 
\to \text{is-set } (\Sigma x : X , A x)

to-subtype-≡ : \{X : W\} \{A : X \to V\} 
\{x y : X\} \{a : A x\} \{b : A y\} 
\to ((x : X) \to \text{is-ssubsingleton } (A x)) 
\to x \equiv y 
\to (x , a) \equiv (y , b)

pr_1-is-equiv : \{X : U\} \{A : X \to V\} 
\to ((x : X) \to \text{is-singleton } (A x)) 
\to \text{is-equiv } (\lambda (t : \Sigma A) \to \text{pr}_1 t)

pr_1-≡ : \{X : U\} \{A : X \to V\} 
\to ((x : X) \to \text{is-singleton } (A x)) 
\to \Sigma A \equiv X

pr_1-≡ i = pr_1 , pr_1-is-equiv i

ΠΣ-distr-≡ : \{X : U\} \{A : X \to V\} \{P : (x : X) \to A x \to W\} 
\to (\Pi x : X , \Sigma a : A x , P x a) 
\equiv (\Sigma f : \Pi A , \Pi x : X , P x (f x))

Σ-assoc : \{X : U\} \{Y : X \to V\} \{Z : \Sigma Y \to W\} 
\to \Sigma Z \equiv (\Sigma x : X , \Sigma y : Y x , Z (x , y))

-¹-≡ : \{X : U\} (x y : X) \to (x \equiv y) \equiv (y \equiv x)

singleton-types-≡ : \{X : U\} (x : X) \to \text{singleton-type}\{x = \text{singleton-type } x\}

singleton-≡ : \{X : U\} \{Y : V\} 
\to \text{is-singleton } X \to \text{is-singleton } Y \to X \equiv Y

maps-of-singletons-are-equivs : \{X : U\} \{Y : V\} (f : X \to Y) 
\to \text{is-singleton } X \to \text{is-singleton } Y \to \text{is-equiv } f
logically-equivalent-subsingletons-are-equivalent : (X : 𝓤) (Y : 𝓥) → is-subsingleton X → is-subsingleton Y → X ≡ Y

singletons-are-equivalent : (X : 𝓤) (Y : 𝓥) → is-singleton X → is-singleton Y → X ≡ Y

NatΣ-fiber-equiv : {X : 𝓤} (A : X → 𝓥) (B : X → 𝓥) (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 → 𝓥} (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-singleton : {X : 𝓤} {Y : 𝓥} → is-singleton X → is-singleton Y → is-singleton (X × Y)

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

×-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 ⇑

Solutions

For the sake of readability, we re-state the formulations of the exercises in the type of sol in a where clause for each exercise.

subsingleton-criterion = sol
where
sol : {X : 𝓤} → (X → is-singleton X) → is-subsingleton X
sol f x = singletons-are-subsingletons (domain f) (f x) x

subsingleton-criterion' = sol
where
sol : {X : 𝓤} → (X → is-subsingleton X) → is-subsingleton X
\[ f \circ x = f \circ x \circ y \]

\begin{align*}
\text{retract-of-subsingleton} &= \text{sol} \\
\text{where} & \quad \text{sol : } \{ X : \mathcal{U} \} \{ Y : \mathcal{V} \} \\
& \quad \rightarrow Y \triangleleft X \rightarrow \text{is-subsingleton } X \rightarrow \text{is-subsingleton } Y \\
& \quad \text{sol } (r, s, \eta) \ i = \text{subsingleton-criterion} \\
& \quad (\lambda x \rightarrow \text{retract-of-singleton } (r, s, \eta)) \\
& \quad (\text{pointed-subsingletons-are-singulartons} \\
& \quad \text{codomain } s) (s x) i)
\end{align*}

\begin{align*}
\text{lc-maps-reflect-subsingletons} &= \text{sol} \\
\text{where} & \quad \text{sol : } \{ X : \mathcal{U} \} \{ Y : \mathcal{V} \} (f : X \rightarrow Y) \\
& \quad \rightarrow \text{left-cancellable } f \rightarrow \text{is-subsingleton } Y \rightarrow \text{is-subsingleton } X \\
& \quad \text{sol } f \ i \ s \ x \ x' = 1 (s (f x) (f x'))
\end{align*}

\begin{align*}
\text{sections-are-lc} &= \text{sol} \\
\text{where} & \quad \text{sol : } \{ X : \mathcal{U} \} \{ A : \mathcal{V} \} (s : X \rightarrow A) \\
& \quad \rightarrow \text{has-retraction } s \rightarrow \text{left-cancellable } s \\
& \quad \text{sol } s \ (r, \varepsilon) \ {\{ x \}} \ {\{ y \}} \ p = x \equiv (\varepsilon x)^{-1} \ \\
& \quad r (s x) \equiv (\text{ap } r \ p) \ \\
& \quad r (s y) \equiv (\varepsilon \ y) \ \\
& \quad y \phantom{\equiv (\varepsilon \ y)}
\end{align*}

\begin{align*}
\text{_equivs-have-retractions} &= \text{sol} \\
\text{where} & \quad \text{sol : } \{ X : \mathcal{U} \} \{ Y : \mathcal{V} \} (f : X \rightarrow Y) \rightarrow \text{is-equiv } f \rightarrow \text{has-retraction } f \\
& \quad \text{sol } f \ e = (\text{inverse } f \ e, \text{inverses-are-retractions } f \ e)
\end{align*}

\begin{align*}
\text{equivs-have-sections} &= \text{sol} \\
\text{where} & \quad \text{sol : } \{ X : \mathcal{U} \} \{ Y : \mathcal{V} \} (f : X \rightarrow Y) \rightarrow \text{is-equiv } f \rightarrow \text{has-section } f \\
& \quad \text{sol } f \ e = (\text{inverse } f \ e, \text{inverses-are-sections } f \ e)
\end{align*}

\begin{align*}
\text{equivs-are-lc} &= \text{sol} \\
\text{where} & \quad \text{sol : } \{ X : \mathcal{U} \} \{ Y : \mathcal{V} \} (f : X \rightarrow Y) \rightarrow \text{is-equiv } f \rightarrow \text{left-cancellable } f \\
& \quad \text{sol } f \ e = \text{sections-are-lc } f \ (\text{equivs-have-retractions } f \ e)
\end{align*}

\begin{align*}
\text{equiv-to-subsingleton} &= \text{sol} \\
\text{where} & \quad \text{sol : } \{ X : \mathcal{U} \} \{ Y : \mathcal{V} \} \rightarrow X \equiv Y \rightarrow \text{is-subsingleton } Y \rightarrow \text{is-subsingleton } X \\
& \quad \text{sol } (f', i) = \text{lc-maps-reflect-subsingletons } f \ (\text{equivs-are-lc } f \ i)
\end{align*}

\begin{align*}
\text{comp-inverses} &= \text{sol} \\
\text{where} & \quad \text{sol : } \{ X : \mathcal{U} \} \{ Y : \mathcal{V} \} \{ Z : \mathcal{W} \} \\
& \quad (f : X \rightarrow Y) (g : Y \rightarrow Z) \\
& \quad (i : \text{is-equiv } f) (j : \text{is-equiv } g) \\
& \quad (f' : Y \rightarrow X) (g' : Z \rightarrow Y) \\
& \quad \rightarrow f' \sim \text{inverse } f \ i \\
& \quad \rightarrow g' \sim \text{inverse } g \ j \\
& \quad \rightarrow f' \circ g' \sim \text{inverse } (g \circ f) (\circ \text{-is-equiv } j \ i) \\
& \quad \text{sol } f \ g \ i \ j \ f' \ g' \ h \ k \ z = \\
& \quad f' (g' z) \equiv (h (g' z)) \\
& \quad \text{inverse } f \ i (g' z) \equiv (\text{ap } (\text{inverse } f \ i) (k z)) \\
& \quad \text{inverse } f \ i (\text{inverse } g \ j \ z) \equiv (\text{inverse-of-} \circ f \ g \ i \ j \ z) \\
& \quad \text{inverse } (g \circ f) (\circ \text{-is-equiv } j \ i) \ z \phantom{\equiv (h (g' z))}
\end{align*}

\begin{align*}
\text{equiv-to-set} &= \text{sol} \\
\text{where}
\end{align*}
equivs-closed-under-invertibles-are-joyal-equivs

joyal-equivs-are-equivs

one-inverse

retractions-closed-under-sections-closed-under

sections-closed-under-~ = sol

where

sol : {X : U} {Y : V} (f g : X → Y)
  has-retraction f → g ~ f → has-retraction g
sol f g (r , rf) h = (r , λ x → r (g x) ≡ ( ap r (h x) )
  r (f x) ≡ ( rf x )
x \∎)

retractions-closed-under-~ = sol

where

sol : {X : U} {Y : V} (f g : X → Y)
  has-section f → g ~ f → has-section g
sol f g (s , fs) h = (s ,
  λ y → g (s y) ≡ ( h (s y) )
  f (s y) ≡ ( fs y )
  y \∎)

one-inverse = sol

where

sol : (X : U) (Y : V) (f : X → Y) (r s : Y → X)
  (r ◦ f ~ id)
  (f ◦ s ~ id)
  r ~ s
sol X Y f r s h y = r y ≡ ( ap r ((k y)⁻¹) )
  r (f (s y)) ≡ ( h (s y) )
  s y \∎)

joyal-equivs-are-invertible = sol

where

sol : {X : U} {Y : V} (f : X → Y)
  is-joyal-equiv f → invertible f
sol f (s , ε) , (r , η)) = (s , sf , ε)
where

sf = λ (x : domain f) → s(f x)
  ≡ ( η (s (f x)))⁻¹ )
  r(f(s(f x))) ≡ ( ap r (ε (f x)) )
  r(f x) ≡ ( η x )
x \∎)

joyal-equivs-are-equivs = sol

where

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

invertibles-are-joyal-equivs = sol

where

sol : {X : U} {Y : V} (f : X → Y)
  invertible f → is-joyal-equiv f
sol f (g , η , ε) = ((g , ε) , (g , η))

equivs-are-joyal-equivs = sol

where

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

equivs-are-invertible = sol

where

sol : {X : U} {Y : V} {f g : X → Y}

sol e = subtypes-of-sets-are-sets \ f e ≡ (equivs-are-lc \ f e)
to-equiv f → g ∼ f ∼ is-equiv g
sol {U} {V} {X} {Y} {f} {g} e h = joyal-equis-are-equis g
  {retractions-closed-under-≡ f g}
  {equivs-have-sections f e h}
  {sections-closed-under-≡ f g}
  {equivs-have-retractions f e h)

equiv-to-singleton' = sol
where
sol : {X : U} {Y : V}
  → X ≃ Y → is-singleton X → is-singleton Y
sol e = retract-of-singleton (≡-gives-→ e)

subtypes-of-sets-are-sets = sol
where
sol : {X : U} {Y : V}
  → left-cancellable m → is-set Y → is-set X
sol {U} {V} {X} m i h = types-with-wconstant≡-endomaps-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 : wconstant≡-endomaps X
  c x x' = f x x', κ x x'
pr₁-lc = sol
where
sol : {X : U} {A : X → V}
  → ((x : X) → is-subsingleton (A x))
  → left-cancellable (λ (t : Σ A) → pr₁ t)
sol i p = to-Σ≡ (p, i _ _ _)

subsets-of-sets-are-sets = sol
where
sol : (X : U) (A : X → V)
  → is-set X
  → ((x : X) → is-subsingleton (A x))
  → is-set (Σ x : X, A x)
sol X A h p = subtypes-of-sets-are-sets pr₁ (pr₁-lc p) h

to-subtype≡ = sol
where
sol : {X : U} {A : X → V}
  → ((x : X) → is-subsingleton (A x))
  → x ≡ y
  → (x, a) ≡ (y, b)
sol {U} {V} {X} {A} {x} {y} {a} {b} s p = to-Σ≡ (p, s y (transport A p a) b)

pr₁-is-equiv = sol
where
sol : {X : U} {A : X → V}
  → ((x : X) → is-singleton (A x))
  → is-equiv (λ (t : Σ A) → pr₁ t)
sol {U} {V} {X} {A} s = invertibles-are-equis 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-subtype≡ (λ x → singletons-are-subsingletons (A x) (s x)) (ε x)
\( \Sigma \text{-distr} \approx = \text{sol} \)
where
\[
\text{sol} : \{ X : \mathcal{U} \} \{ Y : X \to \mathcal{V} \} \{ Z : \Sigma Y \to \mathcal{W} \}
\to \Sigma Z = (\Sigma x : X, \Sigma y : Y x, Z (x, y))
\]
\[
\text{sol} \{ \mathcal{U} \} \{ \mathcal{V} \} \{ \mathcal{W} \} \{ \Sigma \} \{ \mathcal{P} \} = \text{invertibility-gives=} \varphi (\gamma, \eta, \varepsilon)
\]
where
\[
\varphi : (\Pi x : X, \Sigma a : A x, P x a)
\to \Sigma f : \Pi A, \Pi x : X, P x (f x)
\]
\[
\varphi g = ((\lambda x \to \text{pr}_1 (g x)), \lambda x \to \text{pr}_2 (g x))
\]
\[
\gamma : (\Pi f : \Pi A, \Pi x : X, P x (f x))
\to \Pi x : X, \Sigma a : A x, P x a
\]
\[
\gamma (f, \varphi) x = f x, \varphi x
\]
\[
\eta : \gamma \circ \varphi \approx \text{id}
\]
\[
\eta = \text{refl}
\]
\[
\varepsilon : \varphi \circ \gamma \approx \text{id}
\]
\[
\varepsilon = \text{refl}
\]
\( \Sigma \text{-assoc} \approx = \text{sol} \)
where
\[
\text{sol} : \{ X : \mathcal{U} \} \{ Y : X \to \mathcal{V} \} \{ Z : \Sigma Y \to \mathcal{W} \}
\to \Sigma Z = (\Sigma x : X, \Sigma y : Y x, Z (x, y))
\]
\[
\text{sol} \{ \mathcal{U} \} \{ \mathcal{V} \} \{ \mathcal{W} \} \{ \Sigma \} \{ \mathcal{P} \} = \text{invertibility-gives=} f (g, \text{refl}, \text{refl})
\]
where
\[
f : \Sigma Z \to \Sigma x : X, \Sigma y : Y x, Z (x, y)
\]
\[
f ((x, y), z) = (x, (y, z))
\]
\[
g : (\Sigma x : X, \Sigma y : Y x, Z (x, y)) \to \Sigma Z
\]
\[
g (x, (y, z)) = ((x, y), z)
\]
\( ^{-1} \text{-is-equiv} \approx \{ X : \mathcal{U} \} \{ Y : X \} (x : x : X) \to (x \equiv y) \equiv (y \equiv x)
\]
\( ^{-1} \text{-is-equiv} \approx \text{x y} = \text{invertibles-are-equivs} ^{-1}
\)
\( ^{-1} \text{-equiv} \approx = \text{sol} \)
\[
\text{sol} : \{ X : \mathcal{U} \} \{ Y : X \} \to \{ \text{x y} : X \} \to \{ \text{x y} \} \equiv \{ x \} \equiv x
\]
\[
\text{sol} \ x \ y = (-^{-1}, ^{-1} \text{-is-equiv} \ x \ y)
\]
\( \Sigma \text{-cong} \approx = \text{sol} \)
\[
\text{sol} : \{ X : \mathcal{U} \} \{ Y : \mathcal{V} \} \to \Sigma \text{-cong} (\lambda y \to ^{-1} \text{-equiv} \ x \ y)
\]
\( \Sigma \text{-assoc} \approx = \text{sol} \)
\[
\text{sol} : \{ X : \mathcal{U} \} \{ Y : \mathcal{V} \} \to \text{is-singleton} X \to \text{is-singleton} Y \to X \approx Y
\]
\[
\text{sol} \{ \mathcal{U} \} \{ \mathcal{V} \} \{ \mathcal{X} \} \{ \mathcal{Y} \} \{ \mathcal{I} \} \{ \mathcal{J} \} = \text{invertibility-gives=} f (g, \eta, \varepsilon)
\]
where
\[
f : X \to Y
\]
\[
f x = \text{center} Y j
\]
\[
g : Y \to X
\]
\[
g y = \text{center} X i
\]
\[
\eta : (x : X) \to g (f x) \equiv x
\]
\[
\eta = \text{centrality} X i
\]
\[
\varepsilon : (y : Y) \to f (g y) \equiv y
\]
NatΣ-equiv-gives-fiberwise-equiv

where

sol : {X : ℱ } {Y : ℱ } (f : X → Y)
  → is-singleton X → is-singleton Y → is-equiv f

sol {𝒰} {ℱ} {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 = sol

where

sol : (X : ℱ ) (Y : ℱ )
  → is-subsingleton X → is-subsingleton Y → X ↔ Y → X = Y

sol X Y i j (f , g) = invertibility-gives≃ (λ x → i (g (f x)) x ,
                              (λ y → j (f (g y)) y))

singletons-are-equivalent = sol

where

sol : (X : ℱ ) (Y : ℱ )
  → is-singleton X → is-singleton Y → X ≡ Y

sol X Y i j = invertibility-gives≃ (λ _ → center Y j)
                              ((λ _ → center X i ,
                                centrality X i ,
                                centrality Y j)

NatΣ-fiber-equiv = sol

where

sol : {X : ℱ } (A : X → ℱ ) (B : X → ℱ ) (φ : Nat A B)
  (x : X) (b : B x)
  → fiber (φ x) b ≡ fiber (NatΣ φ) (x , b)

sol A B φ x b = invertibility-gives≃ 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 = sol

where

sol : {X : ℱ } (A : X → ℱ ) {B : X → ℱ } (φ : Nat A B)
  → is-equiv (NatΣ φ) → ((x : X) → is-equiv (φ x))

sol {𝒰} {ℱ} {ℱ} {X} {A} {B} φ e x b = γ

where

d : fiber (φ x) b ≡ fiber (NatΣ φ) (x , b)
d = NatΣ-fiber-equiv A B φ x b

s : is-singleton (fiber (NatΣ φ) (x , b))
s = e (x , b)
We begin with two general results, which will be placed in a more general context later.

**equiv-singleton-lemma** : \{X : U \} \{A : X \to V \} (f : X \to Y) \{x \in A \mid f x \equiv f x' \}

**equiv-to-singleton** = \{X : U \} \{A : X \to \text{equiv-singleton} \} \{x \in A \mid x \equiv x' \}

A characterization of univalence

We begin with two general results, which will be placed in a more general context later.
equiv-singleton-lemma \{U\} \{V\} \{X\} \{A\} x f i = y
where
  abstract
  e : (y : X) \to (x \equiv y) \equiv A y
  e y = (f y , i y)
  d : singleton-type' x \equiv \Sigma A
  d = \Sigma-cong e

y : is-singleton (\Sigma A)
y = equiv-to-singleton (\Sigma A)

singleton-equiv-lemma : \{X : U \} \{A : X \to V \} (x : X)
\to is-singleton (\Sigma A)
\to (y : X) \to is-equiv (f y)

singleton-equiv-lemma \{U\} \{V\} \{X\} \{A\} x f i = y
where
  abstract
  g : singleton-type' x \to \Sigma A
  g = Nat\Sigma f
g : is-equiv g
e = maps-of-singletons-are-equivs g (singleton-types'-are-singles X x) i

y : (y : X) \to is-equiv (f y)
y = Nat\Sigma-equiv-gives-fiberwise-equiv f e

With this we can characterize univalence as follows:

univalence-\to : is-univalent U
\to (X : U \) \to is-singleton (\Sigma Y : U \) , Y \equiv Y)
univalence-\to ua X = equiv-singleton-lemma X (Id-Eq X) (ua X)

\Rightarrow univalence : ((X : U \) \to is-singleton (\Sigma Y : U \) , Y \equiv Y))
\to is-univalent U
\Rightarrow univalence i X = singleton-equiv-lemma X (Id-Eq X) (i X)

(\text{Of course, this doesn't say that there is only one type } Y \text{ equivalent to } X, \text{ or only one equivalence from } X \text{ to } Y, \text{ because equality of } \Sigma \text{ types is given by transport in the second component along an identification in the first component.})

We can replace singleton by subsingleton and still have a logical equivalence, and we sometimes need the characterization in this form:

univalence-\to : is-univalent U
\to (X : U \) \to is-subsingleton (\Sigma Y : U \) , Y \equiv Y)
univalence-\to ua X = singletons-are-subsingletons
(\Sigma (X \equiv_\_)) (univalence-\to ua X)

\Rightarrow univalence : ((X : U \) \to is-subsingleton (\Sigma Y : U \) , Y \equiv Y))
\to is-univalent U
\Rightarrow univalence i X = singleton-equiv-lemma X (Id-Eq X) (i X)

Table of contents ✪

Equivalence induction
Under univalence, we get induction principles for type equivalences, corresponding to the induction principles $\mathbb{H}$ and $\mathbb{J}$ for identifications. To prove a property of equivalences, it is enough to prove it for the identity equivalence $id-\approx X$ for all $X$.

Our objective is to get the induction principles $\mathbb{H}-\approx$ and $\mathbb{J}-\approx$ below and their corresponding equations. As above, it is easy to define $\mathbb{J}-\approx$ from $\mathbb{H}-\approx$, and it is harder to define $\mathbb{H}-\approx$ directly, and it is much harder to prove the desired equation for $\mathbb{H}-\approx$ directly. In order to make this easy, we define an auxiliary induction principle $\mathbb{G}-\approx$, from which we trivially derive $\mathbb{H}-\approx$, and whose equation is easy to prove.

The reason the induction principle $\mathbb{G}-\approx$ and its equation are easy to construct and prove is that the type $\Sigma Y : U \mathrel{\mathord{\approx}} Y$ is a singleton by univalence, which considerably simplifies reasoning about transport. For $\mathbb{H}-\approx$ we consider $Y : U$ and $e : X \mathrel{\mathord{\approx}} Y$ separately, whereas for $\mathbb{G}-\approx$ we treat them as a pair $(Y, e)$. The point is that the type of such pairs is a singleton by univalence.

\[
\begin{align*}
\mathbb{G}-\approx & : \text{is-univalent } U \\
& \quad \to (X : U \mathrel{\mathord{\approx}} Y) \ (A : (\Sigma Y : U \mathrel{\mathord{\approx}} Y) \to Y) \\
& \quad \to A \ (X, id\mathrel{\mathord{\approx}} X) \\
\mathbb{G}-\approx & \ {\{U\}} \ ua \ X \ A \ a \ Y \ e = \text{transport} \ A \ p \ a \\
\text{where} & \\
& \ t : \Sigma Y : U \mathrel{\mathord{\approx}} Y, X = Y \\
& \ t = (X, id\mathrel{\mathord{\approx}} X) \\
& \ p : t \equiv (Y, e) \\
& \ p = \text{univalence}\rightarrow \ {\{U\}} \ ua \ X \ t \ (Y, e) \\
\mathbb{G}-\approx -\text{equation} & : (ua : \text{is-univalent } U) \\
& \quad \to (X : U \mathrel{\mathord{\approx}} Y) \ (A : (\Sigma Y : U \mathrel{\mathord{\approx}} Y) \to Y) \ (a : A \ (X, id\mathrel{\mathord{\approx}} X)) \\
& \quad \to \mathbb{G}-\approx \ ua \ X \ A \ a \ X \ (id\mathrel{\mathord{\approx}} X) \equiv a \\
\mathbb{G}-\approx -\text{equation} \ {\{U\}} \ {\{V\}} \ ua \ X \ A \ a \ = \\
\mathbb{G}-\approx & \ ua \ X \ A \ a \ X \ (id\mathrel{\mathord{\approx}} X) \equiv (\text{refl}_a) \\
\text{transport} \ A \ p \ a & \equiv (\text{ap} \ (\lambda t \rightarrow \text{transport} \ A \ - \ a) \ q) \\
\text{transport} \ A \ (\text{refl}_{t}) \ a & \equiv (\text{refl}_a) \\
\end{align*}
\]

\[
\begin{align*}
\text{where} & \\
& \ t : \Sigma Y : U \mathrel{\mathord{\approx}} Y, X = Y \\
& \ t = (X, id\mathrel{\mathord{\approx}} X) \\
& \ p : t \equiv t \\
& \ p = \text{univalence}\rightarrow \ {\{U\}} \ ua \ X \ t \\
& \ q : p \equiv \text{refl}_{t} \\
& \ q = \text{subsingletons-are-sets} \ (\Sigma Y : U \mathrel{\mathord{\approx}} Y, X = Y) \\
& \quad \ (\text{univalence}\rightarrow \ {\{U\}} \ ua \ X) \ t \ t \ p \ (\text{refl}_t) \\
\mathbb{H}-\approx & : \text{is-univalent } U \\
& \quad \to (X : U \mathrel{\mathord{\approx}} Y) \ (A : (Y : U \mathrel{\mathord{\approx}} Y) \to X = Y \to Y) \\
& \quad \to A \ X \ (id\mathrel{\mathord{\approx}} X) \to (Y : U \mathrel{\mathord{\approx}} Y) \ (e : X \mathrel{\mathord{\approx}} Y) \to A \ Y \ e \\
\mathbb{H}-\approx & \ ua \ X \ A \ = \ G-\approx \ ua \ X \ (\Sigma\text{-induction} \ A) \\
\mathbb{H}-\approx -\text{equation} & : (ua : \text{is-univalent } U) \\
& \quad \to (X : U \mathrel{\mathord{\approx}} Y) \ (A : (Y : U \mathrel{\mathord{\approx}} Y) \to X = Y \to Y) \ (a : A \ X \ (id\mathrel{\mathord{\approx}} X)) \\
& \quad \to \mathbb{H}-\approx \ ua \ X \ A \ a \ X \ (id\mathrel{\mathord{\approx}} X) \equiv a \\
\mathbb{H}-\approx -\text{equation} \ ua \ X \ A \ = \ G-\approx -\text{equation} \ ua \ X \ (\Sigma\text{-induction} \ A) \\
\end{align*}
\]

The induction principle $\mathbb{H}-\approx$ keeps $X$ fixed and lets $Y$ vary, while the induction principle $\mathbb{J}-\approx$ lets both vary:

\[
\begin{align*}
\mathbb{J}-\approx & : \text{is-univalent } U \\
& \quad \to (A : (X Y : U \mathrel{\mathord{\approx}} Y) \to X = Y \to Y) \\
& \quad \to ((X : U \mathrel{\mathord{\approx}} Y) \to A \ X \ (id\mathrel{\mathord{\approx}} X)) \\
& \quad \to (X Y : U \mathrel{\mathord{\approx}} Y) \ (a \equiv Y) \to A \ X \ Y \ e \\
\end{align*}
\]
A second set of equivalence induction principles refer to \( \text{is-equiv} \) rather than \( = \) and are proved by reduction to the first version \( \text{H-=} \):

\[
\text{H-equiv} : \text{is-univalent } \mathcal{U} \\
\quad (X : \mathcal{U}^\prime) \Rightarrow (A : (X : \mathcal{U}^\prime) \rightarrow (X : \mathcal{U}^\prime)) \rightarrow (X : \mathcal{U}^\prime) \\
\quad \rightarrow (A X (\text{id-=} X) \rightarrow (Y : \mathcal{U}^\prime) \rightarrow (f : X \rightarrow Y) \rightarrow \text{is-equiv } f \rightarrow A X f)
\]

where

\[
\begin{align*}
\mathcal{B} : (Y : \mathcal{U}^\prime) & \rightarrow X \equiv Y \equiv \mathcal{V}^\prime \\
\mathcal{B} Y (f, i) & = A Y f \\
\mathcal{b} & = a \\
\gamma & = (e : X \equiv Y) \rightarrow \mathcal{B} Y e \\
\gamma & = \text{H-=} \quad \mathcal{u} A \mathcal{B} \mathcal{b} \gamma
\end{align*}
\]

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:

\[
\text{J-equiv} : \text{is-univalent } \mathcal{U} \\
\quad (A : (X Y : \mathcal{U}^\prime) \rightarrow (X : \mathcal{U}^\prime)) \rightarrow (X Y : \mathcal{U}^\prime) \rightarrow (f : X \rightarrow Y) \rightarrow \text{is-equiv } f \rightarrow A X Y f
\]

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

\[
\text{J-invertible} : \text{is-univalent } \mathcal{U} \\
\quad (A : (X Y : \mathcal{U}^\prime) \rightarrow (X : \mathcal{U}^\prime)) \rightarrow (X Y : \mathcal{U}^\prime) \rightarrow (f : X \rightarrow Y) \rightarrow \text{invertible } f \rightarrow A X Y f
\]

For example, using \( \text{H-equiv} \) we see that for any pair of functions

\[
\begin{align*}
F : \mathcal{U}^\prime - \mathcal{U}^\prime, \\
\mathcal{F} : \{X Y : \mathcal{U}^\prime\} & \rightarrow (X : \mathcal{U}^\prime) \rightarrow F X \rightarrow F Y,
\end{align*}
\]

if \( \mathcal{F} \) preserves identities then it automatically preserves composition of equivalences. More generally, it is enough that at least one of the factors is an equivalence:

\[
\text{automatic-equiv-functoriality} : \\
\quad (\mathcal{F} : \mathcal{U}^\prime - \mathcal{U}^\prime) \\
\quad (\mathcal{F} : \{X Y : \mathcal{U}^\prime\} \rightarrow (X : \mathcal{U}^\prime) \rightarrow F X \rightarrow F Y) \\
\quad (\mathcal{F} \circ \text{id} : \{X : \mathcal{U}^\prime\} \rightarrow \mathcal{F} (\text{id-=} X) \equiv \text{id} (F X)) \\
\quad \{X Y Z : \mathcal{U}^\prime\} \\
\quad (f : X \rightarrow Y) \\
\quad (g : Y \rightarrow Z)
\]

\[
\text{is-univalent } \mathcal{U} \rightarrow \text{is-equiv } f + \text{is-equiv } g \rightarrow \mathcal{F} (g \circ f) \equiv \mathcal{F} g \circ \mathcal{F} f
\]

\[
\text{automatic-equiv-functoriality} \{\mathcal{U}\} F \mathcal{F} \circ \text{id} \{X\} \{Y\} \{Z\} f g \mathcal{u} a = \gamma
\]

where
Here is another example (see this for the terminology):

\[ \Sigma\text{-change-of-variable}' : \text{is-univalent } \mathcal{U} \]
\[ \rightarrow \{ X : \mathcal{U} \} \{ Y : \mathcal{U} \} \left( A : X \rightarrow Y \right) \]
\[ \rightarrow \Sigma\text{-equiv } f \]
\[ \rightarrow (\Sigma X : X, A x) \equiv (\Sigma Y : Y, A (\text{inverse } f i y)) \]

\[ \Sigma\text{-change-of-variable}' : \{ Y \} \text{ ua } \{ X \} \text{ A f i = } \mathcal{U}\text{-}\equiv \text{ ua } X B b Y (f, i) \]
\[ \text{ where } \]
\[ B : (Y : \mathcal{U}) \rightarrow X \equiv Y - (U \cup V)' \]
\[ B Y (f, i) = \Sigma A \equiv (\Sigma (A \circ \text{inverse } f i)) \]
\[ b : B X (\text{id-}\equiv X) \]
\[ b = \text{refl } (\Sigma A) \]

An unprimed version of this is given below, after we study half adjoint equivalences.

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

\[ \Sigma\text{-change-of-variable}'' : \text{is-univalent } \mathcal{U} \]
\[ \rightarrow \{ X : \mathcal{U} \} \{ Y : \mathcal{U} \} \left( A : X \rightarrow Y \right) \]
\[ \rightarrow \text{is-equiv } f \]
\[ \rightarrow (\Sigma y : Y, A y) \equiv (\Sigma x : X, A (f x)) \]

\[ \Sigma\text{-change-of-variable}'' \text{ ua } A f i = \Sigma\text{-change-of-variable}' \text{ ua } A \]
\[ (\text{inverse } f i) \]
\[ (\text{inverses-are-equivs } f i) \]

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

As another example we have the following:

\[ \text{transport-map-along=} : \{ X Y Z : \mathcal{U} \} \]
\[ (p : X \equiv Y) \left( g : X \rightarrow Z \right) \]
\[ \rightarrow \text{transport } (\lambda - \rightarrow - \rightarrow Z) \left( \text{p} \circ \text{g} \right) \equiv \text{g} \circ \text{id\text{-fun } (p^{-1})} \]

\[ \text{transport-map-along=} \text{ (refl } X) = \text{refl} \]

\[ \text{transport-map-along= } : (\text{ua : is-univalent } \mathcal{U}) \{ X Y Z : \mathcal{U} \} \]
\[ (e : X \equiv Y) \left( g : X \rightarrow Z \right) \]
\[ \rightarrow \text{transport } (\lambda - \rightarrow - \rightarrow Z) \left( \text{Eq-Id } \text{ua } X Y e \right) \left( \text{g} \circ \text{sym } e^{-1} \right) \equiv \text{g} \circ \text{sym } e^{-1} \]

\[ \text{transport-map-along=} \text{ (ua } \{ X \} \{ Y \} \{ Z \} = \text{j=} \text{ ua } A a X Y \]
\[ \text{ where } \]
\[ A : (X Y : \mathcal{U}) \rightarrow X \equiv Y \rightarrow \mathcal{U} \]
\[ A X Y e = (g : X \rightarrow Z) \rightarrow \text{transport } (\lambda - \rightarrow - \rightarrow Z) \left( \text{Eq-Id } \text{ua } X Y e \right) \left( \text{g} \circ \text{sym } e^{-1} \right) \equiv \text{g} \circ \text{sym } e^{-1} \]
\[ a : (X : \mathcal{U}) \rightarrow A X X (\text{id-}\equiv X) \]
An annoying feature of the use of $\mathcal{J}$ (rather than pattern matching on $\text{refl}$) or $\mathcal{J} \rightsquigarrow$ is that we have to repeat what we want to prove, as in the above examples.

### Half adjoint equivalences

An often useful alternative formulation of the notion of equivalence is that of half adjoint equivalence. If we have a function $f : X \to Y$ with invertibility data

$$g : Y \to X,$$

$$\eta : g \circ f \rightsquigarrow \text{id},$$

$$\varepsilon : f \circ g \rightsquigarrow \text{id},$$

then for any $x : X$ we have that

$$\text{ap } f (\eta x)$$

and

$$\varepsilon (f x)$$

are two identifications of type

$$f (g (f x)) \equiv f x.$$

The half adjoint condition says that these two identifications are themselves identified. The addition of the constraint

$$\tau x : \text{ap } f (\eta x) \equiv \varepsilon (f x)$$

turns invertibility, which is data in general, into property of $f$.

\[
\text{is-hae} : \{X : \mathcal{U} \} \{Y : \mathcal{V} \} \to (X \to Y) \to \mathcal{U} \square \mathcal{V}
\]

\[
\text{is-hae } f = \Sigma g : (\text{codomain } f \to \text{domain } f)
\]

\[
, \Sigma \eta : g \circ f \rightsquigarrow \text{id}
\]

\[
, \Sigma \varepsilon : f \circ g \rightsquigarrow \text{id}
\]

\[
, ((x : \text{domain } f) \to \text{ap } f (\eta x) \equiv \varepsilon (f x))
\]

The following just forgets the constraint $\tau$:

\[
\text{haes-are-invertible} : \{X : \mathcal{U} \} \{Y : \mathcal{V} \} \{f : X \to Y\} \to \text{is-hae } f \to \text{invertible } f
\]

\[
\text{haes-are-invertible } f (g , \eta , \varepsilon , \tau) = g , \eta , \varepsilon
\]

Hence half adjoint equivalences are equivalences, because invertible maps are equivalences. But it is also easy to prove this directly, avoiding the detour via invertible maps. We begin with a construction which will be used a number of times in connection with half adjoint equivalences.

\[
\text{transport-ap-} \equiv : \{X : \mathcal{U} \} \{Y : \mathcal{V} \} \{f : X \to Y\}
\]

\[
\{x x' : X\} (a : x' \equiv x) (b : f x' \equiv f x)
\]
An example of an equivalence of types in different universes is

\[ \text{equivs-are-haes} \]

To recover the constraint for identity maps, under univalence, it is enough to give the constraint for identity maps:

\[ \text{id-is-hae : } (X : \mathcal{U} \rightarrow) \rightarrow \text{is-hae} \]

\[ \text{id-is-hae} X = \text{id} X , \text{refl} , \text{refl} , (\lambda x \rightarrow \text{refl} (\text{refl} x)) \]

\[ \text{ua-equivs-are-haes : is-univalent } \mathcal{U} \]

\[ \rightarrow \{ X , Y : \mathcal{U} \rightarrow \} (f : X \rightarrow Y) \rightarrow \text{is-equiv } f \rightarrow \text{is-hae } f \]

\[ \text{ua-equivs-are-haes } \text{ua } \{ X \} \{ Y \} = \emptyset \text{-equiv } \text{ua } (\lambda X Y f \rightarrow \text{is-hae } f) \text{ id-is-hae } X Y \]

The above can be proved without univalence as follows. This argument also allows us to have \( X \) and \( Y \) in different universes. An example of an equivalence of types in different universes is \( \text{Id-Eq} \), as stated by univalence.

\[ \text{equivs-are-haes : } \{ X : \mathcal{U} \rightarrow \} \{ Y : \mathcal{V} \rightarrow \} (f : X \rightarrow Y) \rightarrow \text{is-equiv } f \rightarrow \text{is-hae } f \]

\[ \text{equivs-are-haes } \{ \mathcal{U} \} \{ \mathcal{V} \} \{ X \} \{ Y \} f i = (g , \eta , \varepsilon , \tau) \]

where

\[ g : Y \rightarrow X \]

\[ g = \text{inverse } f i \]

\[ \eta : g \circ f \sim \text{id} \]

\[ \eta = \text{inverses-are-retractions } f i \]

\[ \varepsilon : f \circ g \sim \text{id} \]

\[ \varepsilon = \text{inverses-are-sections } f i \]

\[ \tau : (x : X) \rightarrow \text{ap } f (\eta x) \equiv \varepsilon (f x) \]

\[ \tau x = y \]

where

\[ \varphi : \text{fiber } f (f x) \]

\[ \varphi = \text{center } (\text{fiber } f (f x)) (i (f x)) \]

\[ \text{by-definition-of-g : } g (f x) \equiv \text{fiber-point } \varphi \]

\[ \text{by-definition-of-g } = \text{refl } \]

\[ p : \varphi \equiv (x , \text{refl } (f x)) \]

\[ p = \text{centrality } (\text{fiber } f (f x)) (i (f x)) (x , \text{refl } (f x)) \]

\[ a : g (f x) \equiv x \]

\[ a = \text{ap } \text{fiber-point } p \]

\[ b : f (g (f x)) \equiv f x \]

\[ b = \text{fiber-identification } \varphi \]
We wrote the above proof of \( \text{equivs-are-haes} \) in a deliberately verbose form to aid understanding. Here is the same proof in a perversely reduced form:

\[
\text{equivs-are-haes}' : \{X : U\} \{Y : V\} (f : X \to Y) \\
\to \text{is-equiv } f \to \text{is-hae } f
\]

\[
\text{equivs-are-haes}' f e = \text{(inverse } f e), \\
\text{inverses-are-retractions } f e, \\
\text{inverses-are-sections } f e, \\
\tau)
\]

where

\[
\tau : \forall x \to ap f (\text{inverses-are-retractions } f e x) \equiv \text{inverses-are-sections } f e (f x)
\]

\[
\tau x = \text{transport-ap } = f (ap pr_1 p) (pr_2 \phi) \sim q
\]

where

\[
\phi : \text{fiber } f (f x)
\]

\[
\phi = pr_1 (e (f x))
\]

\[
p : \phi \equiv (x, \text{refl } (f x))
\]

\[
p = pr_2 (e (f x)) (x, \text{refl } (f x))
\]

\[
q : \text{transport } (\lambda \to f - \equiv f x) (ap pr_1 p) (pr_2 \phi) \equiv \text{refl } (f x)
\]

\[
q = \text{transport-ap } (\lambda \to f - \equiv f x) pr_1 p ((pr_2 \phi))^{-1} \cdot \text{apd } pr_2 p
\]

Notice that we have the following factorization, on the nose, of the construction of invertibility data from the equivalence property:

\[
\text{equiv-invertible-hae-factorization} : \{X : U\} \{Y : V\} (f : X \to Y) \\
\to \text{equivs-are-invertible } f \\
\to \text{haes-are-invertible } f \circ \text{equivs-are-haes } f
\]

\[
\text{equiv-invertible-hae-factorization } f e = \text{refl } (\text{equivs-are-invertible } f e)
\]

Instead of working with the notion of half adjoint equivalence, we can just work with Voevodsky’s notion of equivalence, and use the fact that it satisfies the half adjoint condition:

\[
\text{half-adjoint-condition} : \{X : U\} \{Y : V\} (f : X \to Y) (e : \text{is-equiv } f) (x : X) \\
\to \text{ap } f (\text{inverses-are-retractions } f e x) \equiv \text{inverses-are-sections } f e (f x)
\]

\[
\text{half-adjoint-condition } f e = pr_2 (pr_1 (pr_2 (\text{equivs-are-haes } f e)))
\]

Here is an example, where, compared to \( \Sigma\text{-change-of-variable}' \), 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 \( \Sigma\text{-reindexing-retract} \).

\[
\Sigma\text{-change-of-variable} : \{X : U\} \{Y : V\} (A : Y \to W) (f : X \to Y) \\
\to \text{is-equiv } f \to (\Sigma y : Y, A y) \equiv (\Sigma x : X, A (f x))
\]

\[
\text{where}
\]

\[
g = \text{inverse } f i
\]

\[
\eta = \text{inverses-are-retractions } f i
\]

\[
\epsilon = \text{inverses-are-sections } f i
\]
The idea of the following proof is to improve \( \varepsilon \) to be able to give the required \( \tau \):

\[
\begin{align*}
\tau &= \text{half-adjoint-condition } f \ i \\
\phi : \Sigma A \to \Sigma (A \circ f) \\
\psi (y, a) &= (g \ y, \ \text{transport } A ((\varepsilon \ y)^{-1}) \ a) \\
\psi : \Sigma (A \circ f) \to \Sigma A \\
\psi (x, a) &= (f \ x, \ a) \\
\psi \psi : (z : \Sigma A) \to \psi (\psi \ z) \equiv z \\
\psi \psi (y, a) &= p \\
\end{align*}
\]

where

\[
\begin{align*}
p : (f \ (g \ y), \ \text{transport } A ((\varepsilon \ y)^{-1}) \ a) \equiv (y, \ a) \\
p &= \text{to-\(\Sigma\)-\(\equiv\) (\(\varepsilon \ y\), \ \text{transport-is-retraction} A (\varepsilon \ y) \ a) \\
\end{align*}
\]

\[
\begin{align*}
\phi \psi : (t : \Sigma (A \circ f)) \to \phi (\psi \ t) \equiv t \\
\phi \psi (x, a) &= p \\
\end{align*}
\]

where

\[
\begin{align*}
a' : A (f \ (g \ (f \ x))) \\
a' &= \text{transport } A ((\varepsilon \ (f \ x))^{-1}) \ a \\
q &= \text{transport } (A \circ f) (\eta \ x) \ a' \equiv (\text{transport-ap A f} (\eta \ x) \ a') \\
\text{transport A (ap f (\eta \ x)) a'} &\equiv (\text{ap} (\lambda - \rightarrow \text{transport A - a'}) (\tau \ x)) \\
\text{transport A (\varepsilon \ (f \ x)) a'} &\equiv (\text{transport-is-retraction A (\varepsilon \ (f \ x)) a}) \\
\end{align*}
\]

\[
\begin{align*}
p : (g \ (f \ x), \ \text{transport } A ((\varepsilon \ (f \ x))^{-1}) \ a) \equiv (x, \ a) \\
p &= \text{to-\(\Sigma\)-\(\equiv\) (\(\eta \ x\), \ q) \\
\end{align*}
\]

\[
\begin{align*}
y : \Sigma A \equiv \Sigma (A \circ f) \\
y &= \text{invertibility-gives=\(\equiv\) } \phi (\psi, \ \psi \phi, \ \phi \psi) \\
\end{align*}
\]

For the sake of completeness, we also include the proof from the HoTT book that invertible maps are half adjoint equivalences, which uses a standard argument coming from category theory.

We first need some naturality lemmas:

\[
\begin{align*}
\sim\text{-naturality : } & \{X : \mathcal{U} \sto^1\} \{A : \mathcal{V} \sto^1\} \\
\quad & (f \ g : X \to A) \ (H : f \sim g) \ {x y : X} \ {p : x \equiv y} \\
\quad & \rightarrow H x \cdot \text{ap } g \ p \equiv \text{ap } f \ p \cdot H y \\
\sim\text{-naturality' : } & \{X : \mathcal{U} \sto^1\} \{A : \mathcal{V} \sto^1\} \\
\quad & (f \ g : X \to A) \ (H : f \sim g) \ {x y : X} \ {p : x \equiv y} \\
\quad & \rightarrow H x \cdot \text{ap } g \ p \cdot (H y)^{-1} \equiv \text{ap } f \ p \\
\sim\text{-naturality' : } & f \ g \ H \ {x : \{x\} \ {\text{refl } x} \ = \sim^1\text{-right. } (H x)} \\
\sim\text{id-naturality : } & \{X : \mathcal{U} \sto^1\} \\
\quad & (h : X \to X) \ (\eta : h \sim \text{id}) \ {x : X} \\
\quad & \rightarrow \eta \ (h x) \equiv \text{ap } h \ (\eta x) \\
\sim\text{id-naturality : } & h \ \eta \ {\{x\}} = \\
\quad & \eta \ (h x) \\
\quad & \equiv (\text{refl } x) \\
\quad & \eta \ (h x) \cdot \text{refl } (h x) \equiv (i \ x) \\
\quad & \eta \ (h x) \cdot (\eta x \cdot (\eta x)^{-1}) \equiv (\text{id}) \\
\quad & \eta \ (h x) \cdot \eta x \cdot (\eta x)^{-1} \equiv (\text{iii } x) \\
\quad & \eta \ (h x) \cdot \text{ap } \text{id } (\eta x) \cdot (\eta x)^{-1} \equiv (\text{iv } x) \\
\quad & \text{ap } h \ (\eta x) \\
\end{align*}
\]

where

\[
\begin{align*}
i &= \text{ap } (\eta (h x) \ldots ((\sim^1\text{-right. } (\eta x))^{-1}) \i \\
ii &= (\cdot \text{assoc } (\eta (h x)) (\eta x) (\eta x^{-1}))^{-1} \\
iii &= \text{ap } (\lambda - \rightarrow \eta \ (h x) \ldots \eta x^{-1}) ((\text{ap-id } (\eta x))^{-1}) \\
iv &= \sim\text{-naturality'} \ h \ \eta \ {\{x\}} \ {\{x\}} \\
\end{align*}
\]
invertibles-are-haes : \{X : \mathcal{U}\} \{Y : \mathcal{V}\} \{f : X \to Y\} \\ 
\to \text{invertible } f \to \text{is-haes } f

invertibles-are-haes \ f \ (g, \eta, \varepsilon) = g, \eta, \varepsilon', \tau

where

\varepsilon' = \lambda y \to f \ (g \ y) \equiv (\varepsilon \ (f \ (g \ y)))^{-1}

\begin{align*}
  f \ (g \ (f \ (g \ y))) & \equiv \text{ap } f \ (\eta \ (g \ y)) \\
  f \ (g \ y) & \equiv (\varepsilon \ y)
\end{align*}

\begin{verbatim}
module _ (x : domain f) where

p = \eta \ (g \ (f \ x)) \equiv (~id-naturality \ (g \circ f) \ eta)
    \ap \ (g \circ f) \ (\eta \ x) \equiv (\ap \circ \circ f \ (\eta \ x))
\ap \ g \ (\ap \ (f \ (\eta \ x)))

q = \ap \ f \ (\eta \ (g \ (f \ x))) \cdot \varepsilon \ (f \ x) \equiv (by-p)
    \ap \ f \ (\ap \ g \ (\ap \ f \ (\eta \ x))) \cdot \varepsilon \ (f \ x) \equiv (by-id-ap)
    \ap \ (g \circ f) \ (\ap \ f \ (\eta \ x)) \cdot \varepsilon \ (f \ x) \equiv (by-naturality)
    \varepsilon \ (f \ (g \ (f \ x))) \cdot \ap \ id \ (\ap \ f \ (\eta \ x)) \equiv (by-id-ap)
\varepsilon \ (f \ (g \ (f \ x))) \cdot \ap \ f \ (\eta \ x)
\end{verbatim}

where

\begin{align*}
  \text{by-p} & = \ap \ (\lambda - \to \ap \ f - \cdot \varepsilon \ (f \ x)) \ p \\
  \text{by-id-ap} & = \ap \ (\cdot \ \varepsilon \ (f \ x)) \ ((\ap \circ \circ g \ f \ (\ap \ f \ (\eta \ x))))^{-1}
\end{align*}

\begin{align*}
  \text{by-naturality} & = (~\text{naturality} \ (f \circ g) \ id \ \varepsilon \ (f \ (g \ (f \ x))) \ (f \ x) \ (\ap \ f \ (\eta \ x)))^{-1} \\
  \text{by-id-ap} & = \ap \ (\varepsilon \ (f \ (g \ (f \ x)))) \ (\cdot \ \ap \ id \ (\ap \ f \ (\eta \ x)))
\end{align*}

\begin{verbatim}
\tau = \ap \ f \ (\eta \ x) \equiv \text{refl-left}^{-1}
    \text{refl} \ (f \ (g \ (f \ x))) \cdot \ap \ f \ (\eta \ x) \equiv (by-naturality)
    \varepsilon \ (f \ (g \ (f \ x))) \cdot \varepsilon \ (f \ (g \ (f \ x))) \cdot \ap \ f \ (\eta \ x) \equiv (by-assoc)
    \varepsilon \ (f \ (g \ (f \ x))) \cdot \varepsilon \ (f \ (g \ (f \ x))) \cdot \ap \ f \ (\eta \ x) \equiv (by-q)
    \varepsilon \ (f \ (g \ (f \ x))) \cdot \ap \ f \ (\eta \ (g \ (f \ x))) \cdot \varepsilon \ (f \ x) \equiv (refl)
\varepsilon' \ (f \ x)
\end{verbatim}

where

\begin{align*}
  \text{by-naturality} & = \ap \ (_{-} \ \ap \ f \ (\eta \ x)) \ ((^{-1}.\text{left}) \ (\varepsilon \ (f \ (g \ (f \ x))))^{-1}) \\
  \text{by-assoc} & = \cdot \text{assoc} \ ((\varepsilon \ (f \ (g \ (f \ x))))^{-1}) \ (\varepsilon \ (f \ (g \ (f \ x)))) \ (\ap \ f \ (\eta \ x)) \\
  \text{by-q} & = \ap \ ((\varepsilon \ (f \ (g \ (f \ x))))^{-1} \ (_{-}) \ (q^{-1})
\end{align*}

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 : \forall \mathcal{U} \ \mathcal{V} \to (\mathcal{U} \ \mathcal{U} \ \mathcal{V})

funext \mathcal{U} \ \mathcal{V} = \{X : \mathcal{U}\} \{Y : \mathcal{V}\} \{f \ g : X \to Y\} \to \forall \ g \ f \equiv \ g

There will be two seemingly stronger statements, namely the generalization to dependent functions, and the requirement that the canonical map \ f = g \to f \sim g \text{ is an equivalence.}

Exercise. Assuming funext, prove that if a function \ f : X \to Y \text{ is an equivalence then so is the precomposition map } \cdot \circ \ f : (Y \to Z) \to (X \to 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 \text{ is an identity function, for which precomposition with } \ f \text{ is itself an identity function (of a function type), and hence an equivalence:}

\begin{verbatim}
precomp-is-equiv : \text{is-univalent } \mathcal{U}
  \to (X \ Y : \mathcal{U}^{-1}) \ (f : X \to Y)
  \to \text{is-equiv } f
  \to (Z : \mathcal{U}^{-1}) \to \text{is-equiv} \ (\lambda \ g : Y \to Z \to \ g \circ f)
\end{verbatim}

precomp-is-equiv \ \{\mathcal{U}\} \ ua = j-equiv ua
\[(\lambda X \ Y \ (f \ : X \to Y) \to (Z : \mathcal{U}) \to \text{is-equiv} \ (\lambda \ g \to g \circ f))\]
\[(\lambda X \ Z \to \text{id-is-equiv} \ (X \to Z))\]

With this we can prove the desired result as follows.

\[\text{univalence-gives-funext} : \text{is-univalent} \ \mathcal{U} \to \text{funext} \ \mathcal{V} \ \mathcal{U}\]
\[\text{univalence-gives-funext} \ \{\mathcal{U}\} \ \{\mathcal{V}\} \ \{\mathcal{X}\} \ \{\mathcal{Y}\} \ \{f_0\} \ \{f_1\} = \gamma\]

where
\[
\Delta : \mathcal{U}'
\Delta = \sum_{y_0 : Y} \, \sum_{y_1 : Y} \, y_0 = y_1
\delta : Y \to \Delta
\delta \ y = (y, y, \text{refl} \ y)
\]
\[
p_0 \ p_1 : \Delta \to Y
p_0 (y_0, y_1, p) = y_0
p_1 (y_0, y_1, p) = y_1
\]
\[
\delta\text{-is-equiv} : \text{is-equiv} \ \delta
\delta\text{-is-equiv} = \text{invertibles-are-equis} \ \delta (p_0, \eta, \epsilon)
\]
where
\[
\eta : (y : Y) \to p_0 (\delta \ y) \equiv y
\eta \ y = \text{refl} \ y
\epsilon : (d : \Delta) \to \delta (p_0 \ d) \equiv d
\epsilon (y, y, \text{refl} \ y) = \text{refl} (y, y, \text{refl} \ y)
\]
\[
\phi : (\Delta \to Y) \to (Y \to Y)
\phi \ p = \pi \circ \delta
\]
\[
\phi\text{-is-equiv} : \text{is-equiv} \ \phi
\phi\text{-is-equiv} = \text{precomp-is-equiv} \ \text{ua} \ Y \ \Delta \ \delta \ \delta\text{-is-equiv} \ Y
\]
\[
p = \phi \ p_0 \equiv \phi \ p_1
p = \text{refl} (\text{id} \ Y)
\]
\[
q = \pi_0 \equiv \pi_1
q = \text{equivs-are-lc} \ \phi \ \phi\text{-is-equiv} \ p
\]\n\[
y : f_0 \sim f_1 \to f_0 \equiv f_1
y \ h = \text{ap} (\lambda \ x \to \pi (f_0 \ x, f_1 \ x, h \ x)) \ q
\]

This definition of \(y\) is probably too concise. Here are all the steps performed silently by Agda, by expanding judgmental equalities, indicated with \text{refl} here:

\[
y' : f_0 \sim f_1 \to f_0 \equiv f_1
y' \ h = f_0 \equiv f_1
\]
\[
\text{(refl_)}
\]
\[
\text{(refl_)}
\]
\[
\text{(refl_)}
\]
\[
\text{(refl_)}
\]
\[
\text{(refl_)}
\]
\[
\text{(refl_)}
\]
\[
\text{(refl_)}
\]
\[
\text{(refl_)}
\]

So notice that this relies on the so-called \(\eta\)-rule for judgmental equality of functions, namely
\[
f = \lambda \ x \to f \ x.
\]
Without it, we would only get that
\[
f_0 \sim f_1 \to (\lambda \ x \to f_0 \ x) \equiv (\lambda \ x \to f_1 \ x)
\]

instead.

Table of contents

Variations of function extensionality and their logical equivalence
We need some lemmas to get \( \text{dfunext} \) from \( \text{vvfunext} \):

\[
\text{postcomp-invertible} : \{ X : \mathcal{U} \} \{ Y : \mathcal{V} \} \{ A : \mathcal{W} \} \\
\quad \rightarrow \text{funext} \mathcal{W} \mathcal{U} \\
\quad \rightarrow \text{funext} \mathcal{W} \mathcal{V} \\
\quad \rightarrow (f : X \rightarrow Y) \\
\quad \rightarrow \text{invertible} f \\
\quad \rightarrow \text{invertible} (λ (h : A \rightarrow X) \rightarrow f \circ h)
\]

where

\[
\begin{align*}
\gamma & : (g : \mathcal{W} \mathcal{U}) \rightarrow \text{is-equiv} \ (g, \eta, \epsilon) \\
\gamma & = f, \ c
\end{align*}
\]

We need some lemmas to get \( \text{hfunext} \) from \( \text{vvfunext} \):

\[
\text{postcomp-is-equiv} : \{ X : \mathcal{U} \} \{ Y : \mathcal{V} \} \{ A : \mathcal{W} \} \\
\quad \rightarrow \text{funext} \mathcal{W} \mathcal{U} \rightarrow \text{funext} \mathcal{W} \mathcal{V} \\
\quad \rightarrow (f : X \rightarrow Y) \\
\quad \rightarrow \text{is-equiv} f \\
\quad \rightarrow \text{is-equiv} (λ (h : A \rightarrow X) \rightarrow f \circ h)
\]

\[
\begin{align*}
\gamma & : (g : \mathcal{W} \mathcal{U}) \rightarrow \text{is-equiv} \ (g, \eta, \epsilon) \\
\gamma & = (g', \eta', \epsilon')
\end{align*}
\]
We have the following corollaries. We first formulate the types of some functions:

funext-gives-vvfunext : vvfunext \ U \ V \to hfunext \ U \ V
funext-gives-hfunext vfe \ {X} \ {Y} \ f = \gamma
where
  a : (x : X) \to is-singleton (Σ y : Y \times, f \times y)
a x = singleton-types'-are-singletons (Y \times) (f \times)
  c : is-singleton (Π x : X, Σ y : Y \times, f \times y)
c = vfe a
  ρ : (Σ g : \Pi Y, f \rightarrows g) \triangleleft (Π x : X, Σ y : Y \times, f \times y)
  ρ = \equiv-gives-\Pi Σ-distr-≃
  d : is-singleton (Σ g : \Pi Y, f \rightarrows g)
d = retract-of-singleton ρ c
  e : (Σ g : \Pi Y, f \equiv g) \to (Σ g : \Pi Y, f \rightarrows g)
e = NatΣ (happily f)
i = is-equiv e
  i = maps-of-singletons-are-equis e (singleton-types'-are-singletons (Π Y) f) d
  γ : (g : \Pi Y) \to is-equiv (happily f)
  γ = NatΣ-equiv-gives-fiberwise-equiv (happily 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 \sqcup \V) \to funext \ U \to vvfunext \ U \ V
funext-gives-vvfunext \ U \ V \ f e \ f e' \ {X} \ {Y} \ \eta = γ
where
  f : Σ A \to X
  f = pr₁
  f-is-equiv : is-equiv f
  f-is-equiv = pr₁-is-equiv \ η
  g : (X \to Σ A) \to (X \to X)
g h = f \circ h
  e : is-equiv g
e = postcomp-is-equiv fe fe' f f-is-equiv
  i : is-singleton (Σ h : (X \to Σ A), f \circ h \equiv \text{id} \times)
i = e (\text{id} \times)
r : (Σ h : (X \to Σ A), f \circ h \equiv \text{id} \times) \to Π A
r (h, p) x = transport A (happily f \circ h) (\text{id} \times) p x (pr₂ (h x))
s : Π A \to (Σ h : (X \to Σ A), f \circ h \equiv \text{id} \times)
s \varphi = (\lambda x \to X, \varphi x), refl (\text{id} \times)
  η : \forall \varphi \to r (s \varphi) \equiv \varphi
  η \varphi = refl (r (s \varphi))
  γ : is-singleton (Π A)
  γ = retract-of-singleton (r, s, η) i

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

abstract
  funext-gives-hfunext : funext \ U (\U \sqcup \V) \to funext \ U \to hfunext \ U \ V
dfunext-gives-hfunext : dfunext \ U \ V \to hfunext \ U \ V
funext-gives-dfunext : funext \ U (\U \sqcup \V) \to funext \ U \to dfunext \ U \ V
univalence-gives-dfunext' : is-univalent \ U \to is-univalent (\U \sqcup \V) \to dfunext \ U \ V
univalence-gives-hfunext' : is-univalent \ U \to is-univalent (\U \sqcup \V) \to 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')

dfunext-gives-hfunext fe = vvfunext-gives-hfunext (dfunext-gives-vvfunext 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 ua = univalence-gives-hfunext' ua ua

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

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

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

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

Table of contents ⇪

Universes are map classifiers

Under univalence, a universe \( \mathcal{U} \) becomes a map classifier, in the sense that maps from a type \( X \) in \( \mathcal{U} \) into a type \( Y \) in \( \mathcal{U} \) are in canonical bijection with functions \( Y \to \mathcal{U} \). Using the following slice notation, this amounts to a bijection between \( \mathcal{U}/Y \) and \( Y \to \mathcal{U} \):

\[
_/_ : (\mathcal{U} : \text{Universe}) \to \mathcal{V}' \to \mathcal{U} ' \cup \mathcal{V}'
\]

\[
\mathcal{U}/Y = \Sigma X : \mathcal{U}', (X \to Y)
\]

We need the following lemma, which has other uses:

\[
\text{total-fiber-is-domain} : \{X : \mathcal{U}'\} \{Y : \mathcal{V}'\} \{f : X \to Y\}
\to \Sigma (\text{fiber } f) = X
\]

\[
\text{total-fiber-is-domain} \{\mathcal{U}\} \{\mathcal{V}\} \{X\} \{Y\} f = \text{invertibility-gives-=} g (h , \eta , \epsilon)
\]

where

\[
g : (\Sigma y : Y , \Sigma x : X , f x \equiv y) \to X
\]

\[
h : X \to \Sigma y : Y , \Sigma x : X , f x \equiv y
\]

\[
h x = (f x , x , \text{refl } (f x))
\]

\[
\eta : \forall t \to h (g t) \equiv t
\]

\[
\eta (_ , x , \text{refl } _) = \text{refl } (f x , x , \text{refl } _)
\]

\[
\epsilon : (x : X) \to g (h x) \equiv x
\]

\[
\epsilon = \text{refl}
\]

The function \( \chi \) gives the characteristic function of a map into \( Y \):

\[
\chi : (Y : \mathcal{U}') \to \mathcal{U} / Y \to (Y \to \mathcal{U}')
\]

\[
\chi Y (X , f) = \text{fiber } f
\]

We say that a universe is a map classifier if the above function is an equivalence for every \( Y \) in the universe:
is-map-classifier : (U : Universe) → U ++
is-map-classifier U = (Y : U') → is-equiv (χ Y)

Any Y → U is the characteristic function of some map into Y by taking its total space and the first projection:

ι : (Y : U') → (Y → U') → U / Y
ι Y A = Σ A , pr₁

χη : is-univalent U
→ (Y : U') (σ : U / Y) → ι Y (χ Y σ) ≡ σ

χη ua Y (X , f) = r
where
e : Σ (fiber f) ≃ X
e = total-fiber-is-domain f

p : Σ (fiber f) ≃ X
p = Eq-Id ua (Σ (fiber f)) X e

observation : " = -sym e " ≃ (λ x → f x , x , refl (f x))
observation = refl _

q = transport (λ - → - → Y) p pr₁ ≃ ( transport-map-along-≃ ua pr₁ )
pr₁ ∘ " = -sym e " ≃ ( refl _ )
f

r : (Σ (fiber f)) pr₁ ≃ (X , f)
r = to-Σ-≡ (p , q)

χε : is-univalent U → dfunext U (U')
→ (Y : U') (A : Y → U') → χ Y (ι Y A) ≡ A

χε ua fe Y A = fe Y
where
f : ∀ y → fiber pr₁ y → A y
f y ((y , a) , refl p) = a

g : ∀ y → A y → fiber pr₁ y
g y a = (y , a) , refl y

η : ∀ y σ → g y (f y σ) ≡ σ
η y ((y , a) , refl p) = refl ((y , a) , refl p)

ε : ∀ y a → f y (g y a) ≡ a
ε y a = refl a

y : ∀ y → fiber pr₁ y ≡ A y
y y = Eq-Id ua _ _ (invertibility-gives-≃ (f y) (g y , η y , ε y))

universes-are-map-classifiers : is-univalent U → dfunext U (U')
→ is-map-classifier U

universes-are-map-classifiers ua fe Y = invertibles-are-equivs (χ Y)
(ι Y , χη ua Y , χε ua fe Y)

Therefore we have the following canonical equivalence:

map-classification : is-univalent U → dfunext U (U')
→ (Y : U') → U / Y = (Y → U')

map-classification ua fe Y = χ Y , universes-are-map-classifiers ua fe Y

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.

\[ \Pi \text{-is-subsingleton} \quad \text{dfunext} \quad \mathcal{U} \to \{ X \mid X \subseteq (\mathcal{U} \to \mathcal{V}) \} \]

\[ \Pi \text{-is-subsingleton} \quad \text{fe i f g} = \text{fe} \quad (\lambda x \to i x (f x) (g x)) \]

\[ \text{being-singleton-is-subsingleton} \quad \text{dfunext} \quad \mathcal{U} \to \{ X \mid X \subseteq (\mathcal{U} \to \mathcal{V}) \} \]

\[ \text{being-singleton-is-subsingleton} \quad \text{fe} \quad (x, \phi) (y, y) = p \quad \text{where} \]

\[ i : \Pi \text{-is-subsingleton X} \]

\[ i = \text{singletons-are-subsingletons X} (y, y) \]

\[ s : \text{is-set X} \]

\[ s = \text{subsingletons-are-sets} X i \]

\[ a : (z : X) \to \Pi \text{-is-subsingleton} ((t : X) \to z = t) \]

\[ a z = \Pi \text{-is-subsingleton} \text{ fe} (s z) \]

\[ b : x = y \]

\[ b = \phi y \]

\[ p : (x, \phi) = (y, y) \]

\[ p = \text{to-subtype=} a b \]

\[ \text{being-equiv-is-subsingleton} \quad \text{dfunext} \quad \mathcal{U} \to \text{dfunext} \quad \mathcal{U} \to \mathcal{U} \to \mathcal{U} \to \mathcal{V} \to \mathcal{V} \]

\[ \text{being-equiv-is-subsingleton} \quad \text{fe fe' f} = \Pi \text{-is-subsingleton} \text{ fe} \]

\[ (\lambda x \to \text{being-singleton-is-subsingleton} \text{ fe'} f) \]

In passing, we fulfill a promise made above:

\[ \text{subsingletons-are-retracts-of-logically-equivalent-types} \quad \{ X \mid X \subseteq (\mathcal{U} \to \mathcal{V}) \} \]

\[ \to \text{is-subsingleton X} \]

\[ \to (X \Leftarrow Y) \]

\[ \to X \triangleleft Y \]

\[ \text{subsingletons-are-retracts-of-logically-equivalent-types} \quad i (f, g) = g, f, \eta \quad \text{where} \]

\[ \eta : g \circ f \sim \text{id} \]

\[ \eta x = i (g (f x)) x \]

\[ \text{equivalence-property-is-retract-of-invertibility-data} \quad \text{dfunext} \quad \mathcal{U} \to \text{dfunext} \quad \mathcal{U} \to \mathcal{U} \to \mathcal{V} \to \mathcal{V} \]

\[ \to \text{is-equiv f} \]

\[ \text{equivalence-property-is-retract-of-invertibility-data} \quad \text{fe fe' f} = \quad \text{subsingletons-are-retracts-of-logically-equivalent-types} \]

\[ \text{being-equiv-is-subsingleton} \text{ fe fe' f} \quad \text{equivs-are-invertible f , invertibles-are-equivs f} \]

We now return to our main concern in this section.

\[ \text{univalence-is-subsingleton} \quad \text{is-univalent} \quad (\mathcal{U} \to \mathcal{V}) \]

\[ \to \text{is-subsingleton} \quad \text{is-univalent} \quad \mathcal{U} \]

\[ \text{univalence-is-subsingleton} \quad \text{ua} = \text{subsingleton-criterion'} \gamma \quad \text{where} \]

\[ \gamma : \text{is-univalent} \quad \mathcal{U} \to \text{is-subsingleton} \quad \text{is-univalent} \quad \mathcal{U} \]

\[ \gamma \text{ ua} = i \quad \text{where} \]

\[ \text{dfe}_{\gamma} : \text{dfunext} \quad \mathcal{U} \quad (\mathcal{U} \to \mathcal{V}) \]
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 $\omega$ many universes, because global univalence would have to live in the first universe after them. Agda does have such a universe $\mathcal{U}_\omega$, 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:

**Univalence**

Univalence : $\mathcal{U}_\omega$

Univalence = $\forall \mathcal{U} \rightarrow is-univalent \mathcal{U}$

univalence-is-a-singleton : Univalence $\rightarrow$ is-a-singleton (is-univalent $\mathcal{U}$)

univalence-is-a-singleton $\{\mathcal{U}\} \gamma = is-pointed-subsingletons-are-singletons

(is-univalent $\mathcal{U}$)

(is-univalent $\mathcal{U}$)

That the type Univalence would be a subsingleton can’t even be formulated in the absence of a successor $\mathcal{U}_\omega$ of $\mathcal{U}_\omega$, and Agda doesn’t have such a successor universe (but there isn’t any fundamental reason why it couldn’t have it).

In the absence of a universe $\mathcal{U}_\omega$ in our MLTT, we can simply have an axiom schema, consisting of $\omega$-many axioms, stating that each universe is univalent. Then we can prove in our MLTT that the univalence property for each universe is a (sub)singleton, with $\omega$-many proofs (or just one schematic proof with a free variable for a universe $\mathcal{U}_n$).

It follows immediately from the above that global univalence gives global function extensionality.

**Global-dfunext** : $\mathcal{U}_\omega$

Global-dfunext = $\forall \{\mathcal{U} \mathcal{V}\} \rightarrow dfunext \mathcal{U} \mathcal{V}$

univalence-gives-global-dfunext : Univalence $\rightarrow$ global-dfunext

univalence-gives-global-dfunext $\mathcal{U}$ $\{\mathcal{V}\} = univalence-gives-dfunext' (ua \mathcal{U}) (ua (\mathcal{U} \uplus \mathcal{V}))$

**Global-hfunext** : $\mathcal{U}_\omega$

Global-hfunext = $\forall \{\mathcal{U} \mathcal{V}\} \rightarrow hfunext \mathcal{U} \mathcal{V}$

univalence-gives-global-hfunext : Univalence $\rightarrow$ global-hfunext

univalence-gives-global-hfunext $\mathcal{U}$ $\{\mathcal{V}\} = univalence-gives-hfunext' (ua \mathcal{U}) (ua (\mathcal{U} \uplus \mathcal{V}))$

**vvfunext** $\mathcal{U} \mathcal{V}$ and **hfunext** $\mathcal{U} \mathcal{V}$ are (sub)singletons

We need a version of $n$-is-subsingleton for dependent functions with implicit arguments.

$n$-is-subsingleton' : dfunext $\mathcal{U} \mathcal{V}$ $\rightarrow \{X : \mathcal{U} \} \{A : X \rightarrow \mathcal{V} \} \rightarrow is-is-subsingleton (A X)

\rightarrow is-is-subsingleton (\{X : X \rightarrow A X\})

$n$-is-subsingleton' $\{X\} \{A\} i = \gamma$

where

$\rho : (\{X : X \rightarrow A X\} \triangleleft n A)\triangleleft n A$

$\rho = (\lambda f \{x\} \rightarrow f x), (\lambda g X \rightarrow g \{x\}) \triangleleft refl$

$\gamma : is-is-subsingleton (\{X : X \rightarrow A X\}$

$\gamma = retract-of-subsingleton \rho (n-is-subsingleton \{x\} \{A\} i)$
To show that $\text{vvfunext } U \ V$ and $\text{hfunext } U \ V$ are subsingletons, we need assumptions of function extensionality for higher universes:

$$\text{vv-and-hfunext-are-subsingletons-lemma} : \text{dfunext} (U \ V) \rightarrow \text{dfunext} (U \ U \ V) \rightarrow \text{is-subsingleton} (\text{vvfunext } U \ V) \times \text{is-subsingleton} (\text{hfunext } U \ V)$$

where

$$\varphi : \text{is-subsingleton} (\text{vvfunext } U \ V) \times \text{is-subsingleton} (\text{hfunext } U \ V)$$

$$\varphi = \Pi-is-subsingleton' (\lambda \text{ dfe} \rightarrow \Pi-is-subsingleton' (\lambda \text{ dfe'} \rightarrow \text{being-singleton-is-subsingleton } dfe'))$$

$$\gamma : \text{is-subsingleton} (\text{hfunext } U \ V)$$

$$\gamma = \Pi-is-subsingleton' (\lambda \text{ dfe} \rightarrow \Pi-is-subsingleton' (\lambda \text{ dfe''} \rightarrow \text{being-equiv-is-subsingleton } dfe'' (\lambda f \rightarrow \text{being-equiv-is-subsingleton } dfe'' (\lambda g \rightarrow \text{being-equiv-is-subsingleton } dfe'' (\text{happly } f g))))))$$

Hence they are singletons assuming global univalence (or just global function extensionality, of any kind):

$$\text{vv-and-hfunext-are-singletons} : \text{Univalence} \rightarrow \text{is-singleton} (\text{vvfunext } U \ V) \times \text{is-singleton} (\text{hfunext } U \ V)$$

where

$$f : \text{is-subsingleton} (\text{vvfunext } U \ V) \times \text{is-subsingleton} (\text{hfunext } U \ V) \rightarrow \text{is-singleton} (\text{vvfunext } U \ V) \times \text{is-singleton} (\text{hfunext } U \ V)$$

$$f (i, j) = \text{pointed-subsingletons-are-singletons} (\text{vvfunext } U \ V)$$

$$\text{pointed-subsingletons-are-singletons} (\text{vvfunext } U \ V)$$

However, $\text{funext } U \ U$ and $\text{dfunext } U \ U$ are not subsingletons (see the HoTT book).

### Table of contents

#### Unique existence in univalent mathematics

Unique existence of $x : X$ with $A x$ in univalent mathematics, written

$$\exists! x : X, A x$$

or simply

$$\exists! A,$$

requires that not only the $x : X$ but also the $a : A x$ to be unique. More precisely, we require that there is a unique pair $(x, a) : \Sigma A$.

This is particularly important in the formulation of universal properties involving types that are not necessarily sets, where it generalizes the categorical notion of uniqueness up to unique isomorphism.
The natural numbers have the **following universal property**. What is noteworthy here is that the type \( Y \) need not be a set, so that the two equations can hold in multiple ways, but nevertheless we have unique existence in the sense of the previous section. Moreover, univalence is not needed. Function extensionality suffices.

\[
\text{\textbf{N-is-nno}} : \text{hfunext } \mathcal{U} : \mathcal{U} \\
\rightarrow (Y : \mathcal{U} \to \mathcal{U}) (y_0 : Y) (g : Y \to Y) \\
\rightarrow \exists ! \ h : (N \to Y), \ (h \circ y_0) \times (h \circ \text{succ} \circ g \circ h)
\]

\[
\text{\textbf{N-is-nno}} \ \mathcal{U} \ \text{hfe } Y \ y_0 \ g = Y
\]

where

We apply the same retraction techniques we used in order to prove that invertible maps are equivalences. We first show that, for any \( h : \mathbb{N} \to Y \), the type
The retraction property doesn't need induction on natural numbers:

\[(h \circ \text{succ} \sim g \circ h) \times (h \circ \text{succ} \sim g \circ h)\]

is a retract of the type

\[h \sim \text{N-iteration } Y y_0 g\]

and hence, by function extensionality, we also have a retraction if we replace pointwise equality \(\sim\) by equality \(\equiv\). Thus the type

\[\Sigma h : \mathbb{N} \to Y, (h \Theta \equiv y_0) \times (h \circ \text{succ} \equiv g \circ h)\]

is a retract of the singleton type

\[\Sigma h : \mathbb{N} \to Y, h \equiv \text{N-iteration } Y y_0 g,\]

and therefore is itself a singleton, as required.

Now we do this in Agda. We need both versions \texttt{hfunext} and \texttt{dfunext} of function extensionality, where we have taken the first as an assumption:

\[
\begin{align*}
\texttt{fe} & : \texttt{dfunext } \mathcal{U} \ \mathcal{U} \\
\texttt{fe} & = \texttt{hfunext-gives-dfunext } hfe
\end{align*}
\]

We first establish the following retraction (which is automatically an equivalence, but we don’t need this fact).

\[
\begin{align*}
\texttt{lemma}_0 & : (h : \mathbb{N} \to Y) \to ((h \Theta \equiv y_0) \times (h \circ \text{succ} \equiv g \circ h)) \succeq (h \sim \text{N-iteration } Y y_0 g) \\
\texttt{lemma}_0 & h = r , s , \eta \\
\text{where} \\
& s : (h \Theta \equiv y_0) \times (h \circ \text{succ} \equiv g \circ h) \to h \sim \text{N-iteration } Y y_0 g \\
& s (p , K) = p \\
& s (p , K) (\text{succ } n) = h (\text{succ } n) \equiv (K n) \\
& g (h n) \equiv (\text{ap } g (s (p , K) n)) \\
& g (Y , (h 0 \equiv y_0)) \equiv (\text{refl }_0) \\
& \text{N-iteration } Y y_0 g (\text{succ } n) \equiv (\text{refl }_n)
\end{align*}
\]

The above section \(s\) is defined by induction on natural numbers, but the following retraction \(r\) is defined directly. Above and below, the identifications \(\text{refl }_\_\_\) are included for the sake of clarity. This adds some extra steps to the proof of the retraction property (some of which could be silent, but we choose to make explicit by further uses of \(\text{refl }_\_\_\)).

\[
\begin{align*}
\texttt{r} & : \text{domain } s \to \text{codomain } s \\
\texttt{r} & = \texttt{n} \to h (\text{succ } n) \equiv (H (\text{succ } n)) \\
& \text{N-iteration } Y y_0 g (\text{succ } n) \equiv (\text{refl }_\_\_) \\
& g (\text{N-iteration } Y y_0 g n) \equiv (\text{ap } g ((H n)^{-1})) \\
& g (h n) \equiv (\text{refl }_n)
\end{align*}
\]

The retraction property doesn’t need induction on natural numbers:

\[
\begin{align*}
\texttt{\eta} & : (z : (h \Theta \equiv y_0) \times (h \circ \text{succ} \equiv g \circ h)) \to (s z) \equiv z \\
\texttt{\eta} & (p , K) = q \\
\text{where} \\
& v = \lambda n \to \\
& s (p , K) (\text{succ } n) \cdot (\text{refl }_\_\_ \cdot \text{ap } g ((s (p , K) n)^{-1})) \equiv (\text{refl }_\_\_) \\
& K n \cdot \text{ap } g (s (p , K) n) \cdot (\text{refl }_\_\_ \cdot \text{ap } g ((s (p , K) n)^{-1})) \equiv (ii n) \\
& K n \cdot \text{ap } g (s (p , K) n) \cdot \text{ap } g ((s (p , K) n)^{-1}) \equiv (i n) \\
& K n \cdot (\text{ap } g (s (p , K) n) \cdot \text{ap } g ((s (p , K) n)^{-1})) \equiv (i i n) \\
& K n \cdot (\text{ap } g (s (p , K) n) \cdot \text{ap } g ((s (p , K) n)^{-1})) \equiv (i i i n) \\
& K n \cdot \text{refl }_\_ \equiv (\text{refl }_\_\_) \\
& K n \equiv (\text{refl }_\_\_) \\
\text{where} \\
& i = \lambda n \to \text{ap } (K n \cdot \text{ap } g (s (p , K) n)^{-1}) \\
& (\text{refl-left }_\_\_ \_\_ \_\_) \cdot \text{ap } g ((s (p , K) n)^{-1}) \\
& ii = \lambda n \to \text{assoc } (K n) (\text{ap } g (s (p , K) n)) (\text{ap } g ((s (p , K) n)^{-1})) \\
& i i i = \lambda n \to \text{ap } (\lambda \to K n \cdot \text{ap } g (s (p , K) n) \cdot \text{ap } g ((s (p , K) n)^{-1})) (\text{ap }^{-1} g (s (p , K) n)^{-1}) \\
& i v = \lambda n \to \text{ap } (K n \cdot \text{ap } g (s (p , K) n)) (\text{ap }^{-1} \cdot \text{ap } g (s (p , K) n)) \equiv (\text{refl }_\_\_\_\_)
\[ q = r (s (p, K)) \]
\[ p, (\lambda n \to s (p, K) (\text{succ } n) \cdot (\text{refl } \cdot \text{ap } g ((s (p, K) n)^{-1}))) \]
\[ \text{where} \]
\[ \text{vi} = \text{ap } (p, _) (f \nu) \]

**lemma\textsubscript{i}** : \( \lambda h \to (h \equiv y_0) \times (h \circ \text{succ} \equiv g \circ h) \langle i \ h \rangle \)
\[ (h \equiv y_0) \times (h \circ \text{succ} \sim g \circ h) \langle i \ h \rangle \]
\[ (h \sim \text{N-iteration } \gamma y_0 g) \]
where
\[ i = \lambda h \to \Sigma \text{-retract } \langle \lambda _\cdot \to \equiv \text{gives}-\rangle (\text{happily } (h \circ \text{succ} (g \circ h)) \text{, hfe } _) \]
\[ ii = \lambda h \to \equiv \text{gives}-\rangle (\text{happily } h (\text{N-iteration } \gamma y_0 g) \text{, hfe } _) \]

This concludes the proof of \( \text{N-is-nno} \). We say that \( \mathbb{N} \) is a **natural numbers object**, or, more precisely, the triple \( (\mathbb{N} , \emptyset , \text{succ}) \) is a natural numbers object.

Here is an important example, which given any \( n : \mathbb{N} \) constructs a type with \( n \) elements (and decidable equality):

**module** finite-types \( \text{hfe} : \text{hfunext } \nu \) \( \text{where} \)
\[ \text{fin} : \exists ! \text{Fin } (\mathbb{N} \to \nu) \]
\[ \times (\text{Fin } \emptyset \equiv \emptyset) \]
\[ \text{fin} = \text{N-is-nno } \text{hfe } (\nu') \emptyset (_+ \text{)} \]
\[ \text{Fin} : \mathbb{N} \to \nu' \]
\[ \text{Fin} = \text{pr}_1 (\text{center } \_ \text{fin}) \]

We could use the other projections to derive the following two equations, but they hold definitionally:

**Fin-equation\textsubscript{0}** : \( \text{Fin } \emptyset \equiv \emptyset \)
\[ \text{Fin-equation\textsubscript{0}} = \text{refl } _\_ \]

**Fin-equation-succ** : \( \text{Fin } \circ \text{succ} \equiv \lambda n \to \text{Fin } n + \text{I} \)
\[ \text{Fin-equation-succ} = \text{refl } _\_ \]

We also have

**Fin-equation-succ'** : \( (n : \mathbb{N}) \to \text{Fin } (\text{succ } n) \equiv \text{Fin } n + \text{I} \)
\[ \text{Fin-equation-succ'} n = \text{refl } _\_ \]

and the examples

**Fin-equation\textsubscript{1}** : \( \text{Fin } 1 \equiv \emptyset + \text{I} \)
\[ \text{Fin-equation\textsubscript{1}} = \text{refl } _\_ \]

**Fin-equation\textsubscript{2}** : \( \text{Fin } 2 \equiv (\emptyset + \text{I}) + \text{I} \)
\[ \text{Fin-equation\textsubscript{2}} = \text{refl } _\_ \]

**Fin-equation\textsubscript{3}** : \( \text{Fin } 3 \equiv ((\emptyset + \text{I}) + \text{I}) + \text{I} \)
\[ \text{Fin-equation\textsubscript{3}} = \text{refl } _\_ \]

**Exercise.** Assume univalence. The equation

\[ \text{Fin } \circ \text{succ} \equiv \lambda n \to \text{Fin } n + \text{I} \]
holds in multiple ways, because \( \text{Fin} n \) has \( n! \) automorphisms. Construct an involutive fiberwise equivalence

\[
mirror : (n : \mathbb N) \to \text{Fin} n \to \text{Fin} n
\]
different from the identity and hence an identification \( \text{Fin} \equiv \text{Fin} \) different from \( \text{refl} \text{Fin} \). Consider

\[
\exists! \text{Fin}' : (\text{Fin}' : \mathcal U') \to (\text{Fin}' \circ \text{succ} : \lambda n \to \mathbb I + \text{Fin}' n)
\]
and show that \( \text{Fin}' \circ \text{succ} : \lambda n \to \text{Fin}' n + \mathbb I \) so that \( \text{Fin}' \) satisfies the defining equations of \( \text{Fin} \), although not judgmentally, and hence \( \text{Fin}' \equiv \text{Fin} \) by the universal property of \( \text{Fin} \). But there are many equalities \( \text{Fin}' \equiv \text{Fin} \). Which one do we get by the universal property? Show that the type \( \text{Fin} n \) has decidable equality and hence is a set.

Table of contents

More consequences of function extensionality

\[
\text{being-subsingleton-is-subsingleton} : \text{dfunext} \mathcal U \mathcal U
\to \{X : \mathcal U'\}
\to \text{is-subsingleton} (\text{is-subsingleton} X)
\]
where

\[
\begin{align*}
\text{l} : & \text{is-set} X \\
\text{l} = & \text{subsingleton-is-sets} X i \\
a : & (x y : X) \to i x y \equiv j x y \\
a x y = & l x y (i x y) (j x y) \\
b : & (x : X) \to i x \equiv j x \\
b x = & \text{fe} (a x) \\
c : & i \equiv j \\
c = & \text{fe} b
\end{align*}
\]

\[
\text{being-center-is-subsingleton} : \text{dfunext} \mathcal U \mathcal U
\to \{X : \mathcal U'\} (c : X)
\to \text{is-subsingleton} (\text{is-center} X c)
\]
where

\[
\begin{align*}
i : & \text{is-singleton} X \\
i = & c , \varphi \\
\phi : & (x : X) \to \text{is-subsingleton} (c \equiv x) \\
\phi x = & \text{singleton-is-sets} X i c x \\
k : & \varphi \equiv y \\
k = & \text{fe} (\lambda x \to \phi x (\varphi x) (y x))
\end{align*}
\]
Here the version \( \text{hfunext} \) of function extensionality is what is needed:

\[
\Pi\text{-is-set} : \text{hfunext} \mathcal U \mathcal V
\to \{X : \mathcal U'\} \{A : X \to \mathcal V'\}
\to ((x : X) \to \text{is-set} (A x)) \to \text{is-set} (\Pi A)
\]
where

\[
\begin{align*}
\text{a} : & \text{is-subsingleton} (f \sim g) \\
\text{a} p q = & y \\
\text{h} : & \forall x \to p x \equiv q x \\
\text{h} x = & s x (f x) (g x) (p x) (q x) \\
y : & p \equiv q \\
y = & \text{hfunext-gives-dfunext} \text{~hfe} \text{~h} \\
e : & (f \equiv g) \Rightarrow (f \sim g) \\
e = & (\text{happly} f g , \text{~hfe} f g) \\
b : & \text{is-subsingleton} (f \equiv g)
\end{align*}
\]
Exercise. The hlevels are closed under \( \Sigma \) and, using \( \text{hfunext} \), also under \( \Pi \). Univalence is not needed, but makes the proof easier. (Without univalence, we need to show that the hlevels are closed under equivalence first.)
An application of Π-cong is hfunext₂≃:

\[ hfunext₂≃ : \text{hofunext} \mathcal{U} \mathcal{V} \to \text{hofunext} \mathcal{U} \mathcal{W} \]
\[ \to \{ X : \mathcal{U} \} \{ Y : \mathcal{V} \} \{ Z : \mathcal{W} \} \{ f : X \to Y \}
\]
\[ \to (f \equiv g) \Rightarrow (f \sim g) \]
\[ hfunext₂≃ hfe \ f \ g = (\text{happily f g , hfe f g}) \]

\[ hfunext₂≃ : \text{hofunext} \mathcal{U} \ (\mathcal{V} \sqcup \mathcal{W}) \to \text{hofunext} \mathcal{V} \mathcal{W} \]
\[ \to \{ X : \mathcal{U} \} \{ Y : \mathcal{V} \} \{ A : (x : X) \to Y \ x \to \mathcal{W} \} \]
\[ \to (f \equiv g) \Rightarrow (\forall x y \ f x y \equiv g x y) \]
\[ hfunext₂≃ \text{fe fe'} \{ X \} \{ f \} \{ g \} = \]
\[ (f \equiv g) = (i) \]
\[ (\forall x f x \equiv g x) = (ii) \]
\[ (\forall x y f x y \equiv g x y) \]

\[ \text{where} \]
\[ i = \text{hofunext₂≃} \text{fe f g} \]
\[ ii = \Pi-cong \]
\[ (\text{hofunext-gives-dfunext fe}) \]
\[ (\text{hofunext-gives-dfunext fe}) \]
\[ (\lambda x \to \text{hofunext≃ fe'} (f x) (g x)) \]

\[ \text{precomp-invertible : dfunext} \mathcal{V} \mathcal{W} \to \text{dfunext} \mathcal{U} \mathcal{W} \]
\[ \to \{ X : \mathcal{U} \} \{ Y : \mathcal{V} \} \{ Z : \mathcal{W} \} \{ f : X \to Y \}
\]
\[ \to \text{invertible f} \]
\[ \to \text{invertible (λ (h : Y → Z) → h ∘ f)} \]

\[ \text{precomp-invertible f fe fe'} \{ X \} \{ f \} \{ g, η, ε \} = (g', η', ε') \]

\[ \text{where} \]
\[ f' : (Y → Z) → (X → Z) \]
\[ f' h = h \circ f \]
This amounts to saying that we can also change variables in Π:

\[ g' : (X \to Z) \to (Y \to Z) \]
\[ g' \cdot k = k \circ g \]

\[ \eta' : (h : Y \to Z) \to g' (f' h) \equiv h \]
\[ \eta' h = \text{fe} \ (\lambda y \to \text{ap} \ h \ (\varepsilon y)) \]

\[ \varepsilon' : (k : X \to Z) \to f' (g' k) \equiv k \]
\[ \varepsilon' k = \text{fe}' \ (\lambda x \to \text{ap} \ k \ (\eta x)) \]

We have already proved the following assuming univalence, in order to derive function extensionality from univalence. Now we prove it assuming function extensionality instead.

\[
\text{precomp-is-equiv}' : \text{dfunext} \ V W \to \text{dfunext} \ U W \\
\quad \to \{ X : U \} \{ Y : V \} \{ Z : W \} \ (f : X \to Y) \\
\quad \text{is-equiv} f \\
\quad \text{is-equiv} \ (\lambda h : Y \to Z) \to h \circ f
\]

\[
\text{precomp-is-equiv}' \ \text{fe} \ \text{fe}' \ (\{ X \} \{ Y \} \{ Z \} \ f \ i = \text{inversibles-are-equivs} \ (_\circ f) \\
\quad \text{precomp-invertible} \ \text{fe} \ \text{fe}' \ f \\
\quad \text{equivs-are-invertible} \ f \ i)
\]

More generally, using the half-adjoint condition for equivalences, we have the following dependent version of precomposition, which is also an equivalence assuming function extensionality.

\[
\text{dprecomp} : \{ X : U \} \{ Y : V \} \{ A : Y \to W \} \ (f : X \to Y) \\
\quad \text{precomp} A \ f = _\circ f
\]

\[
\text{dprecomp-is-equiv} : \text{dfunext} \ U W \to \text{dfunext} \ V W \\
\quad \to \{ X : U \} \{ Y : V \} \{ A : Y \to W \} \ (f : X \to Y) \\
\quad \text{is-equiv} f \\
\quad \text{is-equiv} \ (\lambda h : Y \to Z) \to h \circ f
\]

\[
\text{dprecomp-is-equiv} \ \text{fe} \ \text{fe}' \ (\{ X \} \{ Y \} \{ A : Y \to W \}) \ f \ i = \text{inversibles-are-equivs} \ (\varphi, \psi, \psi)
\]

where
\[
\begin{align*}
g &= \text{inverse} f \ i \\
\eta &= \text{inverses-are-retractions} f \ i \\
\varepsilon &= \text{inverses-are-sections} f \ i \\
\tau : (x : X) \to \text{ap} \ f \ (\eta x) &\equiv \varepsilon (f x) \\
\tau &= \text{half-adjoint-condition} f \ i \\
\varphi : \Pi A \to \Pi (A \circ f) \\
\varphi &= \text{dprecomp} A \ f \\
\psi : \Pi (A \circ f) \to \Pi A \\
\psi \ k \ y &= \text{transport} A \ (\varepsilon y) \ (k \ (g \ y)) \\
\varphi \psi : (k : \Pi (A \circ f)) \ (x : X) \to \text{transport} A \ (\varepsilon (f x)) \ (k \ (g \ (f x))) &\equiv k x \\
\varphi \psi \ k \ x &= \text{transport} A \ (\varepsilon f x) \ (k \ (g \ (f x))) \ (a) \\
&\text{transport} A \ (\text{ap} f \ (\eta x)) (k \ (g \ (f x))) \ (b) \\
&\text{transport} A \ (\varphi \psi \ (\eta x)) (k \ (g \ (f x))) \ (c) \\
&k x
\end{align*}
\]

where
\[
\begin{align*}
a &= \text{ap} \ (\lambda x \to \text{transport} A \ (k \ (g \ (f x)))) \ ((\tau x)^{-1}) \\
b &= \ (\text{transport-ap} \ A \ f \ (\eta x)) (k \ (g \ (f x)))^{-1} \\
c &= \text{apd} k \ (\eta x) \\
\varphi \psi : \varphi \circ \psi \sim \text{id} \\
\varphi \psi \ k &= \text{fe} \ (\varphi \psi \ k) \\
\varphi \psi : \varphi \circ \psi \sim \text{id} \\
\varphi \psi : \psi \circ \varphi \sim \text{id} \\
\psi h &= \text{fe}' \ (\varphi \psi h)
\end{align*}
\]

This amounts to saying that we can also change variables in Π:
Recall that a function is a *Joyal equivalence* if it has a section and it has a retraction. We now show that this notion is a subsingleton. For that purpose, we first show that if a function has a retraction then it has at most one section, and that if it has a section then it has at most one retraction.

\[\text{at-most-one-section} : \text{dfunext } \mathcal{U} \, \mathcal{V} \to \text{dfunext } \mathcal{V} \, \mathcal{V}\]
\[\quad \to \{ X : \mathcal{U} \} \{ Y : \mathcal{V} \} (f : X \to Y) \]
\[\quad \to \text{has-section } f \]
\[\quad \to \text{is-subsingleton } (\text{has-section } f)\]

\[\text{at-most-one-section} \{ \mathcal{V} \} \{ \mathcal{U} \} \text{ fe } \text{ hfe } \{ X \} \{ Y \} f (g, gf) (h, fh) = d\]

where
\[\text{fe'} : \text{dfunext } \mathcal{V} \, \mathcal{V}\]
\[\text{fe'} = \text{hfunext-gives-dfunext } \text{hfe}\]

\[a : \text{invertible } f\]
\[a = \text{joyal-equivs-are-invertible } f (\langle h, fh \rangle, \langle g, gf \rangle)\]

\[b : \text{is-singleton } (\text{fiber } (\lambda h \to f \circ h) \text{ id})\]
\[b = \text{invertibles-are-equivs } (\lambda h \to f \circ h) \text{ (postcomp-invertible } \text{fe } \text{fe'} f a) \text{ id}\]

\[r : \text{fiber } (\lambda h \to f \circ h) \text{ id} \to \text{has-section } f\]
\[r (h, p) = (h, \text{ happily } (f \circ h) \text{ id } p)\]

\[s : \text{has-section } f \to \text{fiber } (\lambda h \to f \circ h) \text{ id}\]
\[s (h, \eta) = (h, \text{fe'} \eta)\]

\[\text{rs} : (\sigma : \text{has-section } f) \to r (s \sigma) \equiv \sigma\]
\[\text{rs} (h, \eta) = \text{to-Σ-≡'} q\]

where
\[q : \text{happly } (f \circ h) \text{ id } (\text{inverse } \text{happly } (f \circ h) \text{ id}) (\text{hfe } (f \circ h) \text{ id}) \eta) \equiv \eta\]
\[q = \text{inverses-are-sections } (\text{happly } (f \circ h) \text{ id}) (\text{hfe } (f \circ h) \text{ id}) \eta\]

\[c : \text{is-singleton } (\text{has-section } f)\]
\[c = \text{retract-of-singleton } (r, s, \text{rs}) b\]

\[d : (\sigma : \text{has-section } f) \to h, \text{fh} \equiv \sigma\]
\[d = \text{singletons-are-subsingletons } (\text{has-section } f) c (h, fh)\]

\[\text{at-most-one-retraction} : \text{hfunext } \mathcal{U} \, \mathcal{U} \to \text{dfunext } \mathcal{V} \, \mathcal{U}\]
\[\to \{ X : \mathcal{U} \} \{ Y : \mathcal{V} \} (f : X \to Y) \]
\[\to \text{has-retraction } f \]
\[\to \text{is-subsingleton } (\text{has-retraction } f)\]

\[\text{at-most-one-retraction} \{ \mathcal{U} \} \{ \mathcal{V} \} \text{ fe } \text{fe'} \{ X \} \{ Y \} f (g, gf) (h, hf) = d\]

where
\[\text{fe} : \text{dfunext } \mathcal{U} \, \mathcal{U}\]
\[\text{fe} = \text{hfunext-gives-dfunext } \text{hfe}\]

\[a : \text{invertible } f\]
\[a = \text{joyal-equivs-are-invertible } f (\langle g, gf \rangle, \langle h, hf \rangle)\]

\[b : \text{is-singleton } (\text{fiber } (\lambda h \to h \circ f) \text{ id})\]
\[b = \text{invertibles-are-equivs } (\lambda h \to h \circ f) \text{ (precomp-invertible } \text{fe } \text{fe'} f a) \text{ id}\]

\[r : \text{fiber } (\lambda h \to h \circ f) \text{ id} \to \text{has-retraction } f\]
\[r (h, p) = (h, \text{ happily } (h \circ f) \text{ id } p)\]

\[s : \text{has-retraction } f \to \text{fiber } (\lambda h \to h \circ f) \text{ id}\]
\[s (h, \eta) = (h, \text{fe} \eta)\]

\[\text{rs} : (\sigma : \text{has-retraction } f) \to r (s \sigma) \equiv \sigma\]
\[\text{rs} (h, \eta) = \text{to-Σ-≡'} q\]

where
\[q : \text{happly } (h \circ f) \text{ id } (\text{inverse } \text{happly } (h \circ f) \text{ id}) (\text{hfe } (h \circ f) \text{ id}) \eta) \equiv \eta\]
\[q = \text{inverses-are-sections } (\text{happly } (h \circ f) \text{ id}) (\text{hfe } (h \circ f) \text{ id}) \eta\]
Another consequence of function extensionality is that emptiness is a subsingleton:

\[ \text{emptiness-is-subsingleton : dfunext } \, \forall \, \text{fiber } f \rightarrow \text{is-subsingleton} \, \text{(is-empty } X) \]

The fact that a function with a retraction has at most one section can also be used to prove that the notion of half adjoint equivalence is property. This is because the type is-hae \( f \) is equivalent to the type

\[ \Sigma \, (g \, , \, \varepsilon) : \text{has-section } f , \, \forall \, x \rightarrow (g \,(f \, x) , \, \varepsilon \,(f \, x)) \equiv (x , \, \text{refl } (f \, x)) , \]

where the equality is in the type fiber \( f \,(x) \).

\[ \text{being-hae-is-subsingleton : dfunext } \, \forall \, \text{fiber } f \rightarrow \text{is-subsingleton} \, \text{(is-hae } f) \]

\[ \text{being-hae-is-subsingleton fe}_0 \, \text{fe}_1 \, \text{fe}_2 \, f = \text{at-most-one-section } \text{fe}_2 \, \text{fe}_1 \, f , \]

\[ \text{at-most-one-retraction } \text{fe}_0 \, \text{fe}_2 \, f \]

\[ \text{being-joyal-equiv-is-subsingleton : hfunext } \, \forall \, \text{fiber } f \rightarrow \text{is-subsingleton} \, \text{(is-joyal-equiv } f) \]

\[ \text{being-joyal-equiv-is-subsingleton fe}_0 \, \text{fe}_1 \, \text{fe}_2 \, f = \text{x-is-subsingleton'} \]

\[ \text{(at-most-one-section } \text{fe}_2 \, \text{fe}_1 \, f , \]

\[ \text{at-most-one-retraction } \text{fe}_0 \, \text{fe}_2 \, f \]

Another consequence of function extensionality is that emptiness is a subsingleton:

\[ \text{emptiness-is-subsingleton : dfunext } \, \forall \, \text{fiber } f \rightarrow \text{is-subsingleton} \, \text{(is-empty } X) \]
emptiness-is-subsingleton fe X f g = fe (λ x → !∅ (f x ≡ g x) (f x))

If P is a subsingleton, then so is P + ¬ P. More generally:

+is-subsingleton : {P : U} {Q : U} → is-subsingleton P → is-subsingleton Q → (P → Q → ⊥) → is-subsingleton (P + Q)

Propositional extensionality and the powerset

We have been using the mathematical terminology “subsingleton”, but tradition in the formulation of the next notion demands the logical terminology “proposition”. Propositional extensionality says that any two logically equivalent propositions are equal:

propext : ∀ U → U →
propext U = {P Q : U} → is-prop P → is-prop Q → (P → Q) → (Q → P) → P ≡ Q

global-propext : lΩ

This is directly implied by univalence:

univalence-gives-propext : is-univalent U → propext U

univalence-gives-propext ua {P} {Q} i j f g = Eq-Id ua P Q γ

where

γ : P ≡ Q
γ = logically-equivalent-subsingletons-are-equivalent P Q i j (f , g)

Under the additional hypothesis of function extensionality, the converse of the above holds. We need a lemma for that.

Id-from-subsingleton : propext U → dfunext U U → is-subsingleton P → (X : U) → is-subsingleton (P ≡ X)

Id-from-subsingleton {U} pe fe P i = Hedberg P (λ X → h X , k X)

where

module _ (X : U) where
  f : P ≡ X → is-subsingleton X × (P ⊥ X)
  f p = transport is-subsingleton p i , Id-fun p , (Id-fun (p⁻¹))
\( g : \text{is-subsingleton} \; X \times (P \leftrightarrow X) \rightarrow P \equiv X \)
\( g \; (1, \varphi, \psi) = pe \; i \; \varphi \; \psi \)

\( h : P \equiv X \rightarrow P \equiv X \)
\( h = g \circ f \)

\( j : \text{is-subsingleton} \; (is\text{-}subsingleton \; X \times (P \leftrightarrow X)) \)
\( j = x\text{-}is\text{-}subsingleton' \)
\((\lambda \; (_:\; P \leftrightarrow X) \rightarrow \text{being\text{-}subsingleton\text{-}is\text{-}subsingleton} \; fe), \)
\((\lambda \; (l : \text{is\text{-}subsingleton} \; X) \rightarrow x\text{-}is\text{-}subsingleton \)
\((\Pi\text{-}is\text{-}subsingleton \; fe \; (\lambda \; p \rightarrow l)), \)
\((\Pi\text{-}is\text{-}subsingleton \; fe \; (\lambda \; x \rightarrow l)))\)

\( k : w\text{constant} \; h \)
\( k \; p \; q = \text{ap} \; g \; (j \; (f \; p) \; (f \; q)) \)

\( \text{subsingleton\text{-}univalence} : \text{propeext} \; \mathcal{U} \rightarrow \text{dfunext} \; \mathcal{U} \; \mathcal{U} \)
\( \quad \rightarrow (P : \mathcal{U}^\prime) \rightarrow \text{is\text{-}subsingleton} \; P \)
\( \quad \rightarrow (X : \mathcal{U}^\prime) \rightarrow \text{is\text{-}equiv} \; (\text{Id\text{-}Eq} \; P \; X) \)

\( \text{subsingleton\text{-}univalence} \; \text{pe} \; \text{fe} \; P \; i \; X = \gamma \)
where
\( l : P \equiv X \rightarrow \text{is\text{-}subsingleton} \; X \)
\( l \; e \; = \text{equiv\text{-}to\text{-}subsingleton} \; (\rightarrow\text{-}sym \; e) \; i \)

\( \text{eqtoid} : P \equiv X \rightarrow P \equiv X \)
\( \text{eqtoid} \; e \; = \text{pe} \; i \; (\text{equiv\text{-}to\text{-}subsingleton} \; (\rightarrow\text{-}sym \; e) \; i) \)
\( \quad r \; e \; r \; \rightarrow\text{-}sym \; e \; r \)

\( m : \text{is\text{-}subsingleton} \; (P \equiv X) \)
\( m \; (f, k) \; (f', k') \; = \text{to\text{-}subtype}\equiv \)
\( \quad (\text{being\text{-}equiv\text{-}is\text{-}subsingleton} \; fe \; fe), \)
\( \quad (fe \; (\lambda \; x \rightarrow j \; (f \; x) \; (f' \; x))) \)
where
\( j : \text{is\text{-}subsingleton} \; X \)
\( j \; = \text{equiv\text{-}to\text{-}subsingleton} \; (\rightarrow\text{-}sym \; (f, k)) \; i \)

\( \varepsilon : (e : P \equiv X) \rightarrow \text{Id\text{-}Eq} \; P \; X \; (\text{eqtoid} \; e) \; \equiv \; e \)
\( \varepsilon \; e \; = \; m \; (\text{Id\text{-}Eq} \; P \; X \; (\text{eqtoid} \; e)) \; e \)

\( \eta : (q : P \equiv X) \rightarrow \text{eqtoid} \; (\text{Id\text{-}Eq} \; P \; X \; q) \; \equiv \; q \)
\( \eta \; q \; = \text{Id\text{-}from\text{-}singleton} \; \text{pe} \; \text{fe} \; P \; i \; X \; (\text{eqtoid} \; (\text{Id\text{-}Eq} \; P \; X \; q)) \; q \)

\( \gamma : \text{is\text{-}equiv} \; (\text{Id\text{-}Eq} \; P \; X) \)
\( \gamma \; = \text{invertibles\text{-}are\text{-}equivs} \; (\text{Id\text{-}Eq} \; P \; X) \; (\text{eqtoid}, \eta, \varepsilon) \)

\( \text{subsingleton\text{-}univalence}\equiv : \text{propeext} \; \mathcal{U} \rightarrow \text{dfunext} \; \mathcal{U} \; \mathcal{U} \)
\( \quad \rightarrow (X \; P : \mathcal{U}^\prime) \rightarrow \text{is\text{-}subsingleton} \; P \rightarrow (P \equiv X) \; \equiv \; (P \equiv X) \)

\( \text{subsingleton\text{-}univalence}\equiv \; \text{pe} \; \text{fe} \; X \; P \; i = \text{Id\text{-}Eq} \; P \; X \),
\( \text{subsingleton\text{-}univalence} \; \text{pe} \; \text{fe} \; P \; i \; X \)

We also need a version of propositional extensionality for the type \( \pi \; \mathcal{U} \) of subsingletons in a given universe \( \mathcal{U} \), which lives in the next universe:

\( \varnothing : (\mathcal{U} : \text{Universe}) \rightarrow \mathcal{U}^\prime \)
\( \varnothing \; \mathcal{U} \; = \; \Sigma \; P : \mathcal{U}^\prime, \text{is\text{-}subsingleton} \; P \)

\( \_\text{holds} : \varnothing \; \mathcal{U} \rightarrow \mathcal{U}^\prime \)
\( \_\text{holds} \; (P, i) \; = \; P \)

\( \text{holds\text{-}is\text{-}subsingleton} : (p : \varnothing \; \mathcal{U}) \rightarrow \text{is\text{-}subsingleton} \; (p \; \text{holds}) \)
\( \text{holds\text{-}is\text{-}subsingleton} \; (P, i) \; = \; i \)

\( \varnothing\text{-}ext : \text{dfunext} \; \mathcal{U} \; \mathcal{U} \rightarrow \text{propeext} \; \mathcal{U} \rightarrow \{p \; q : \varnothing \; \mathcal{U}\} \)
\( \rightarrow (p \; \text{holds} \rightarrow q \; \text{holds}) \rightarrow (q \; \text{holds} \rightarrow p \; \text{holds}) \rightarrow p \equiv q \)
Membership and containment for elements of the powerset are defined as follows:

\[ \text{Hence powersets, even of types that are not sets, are always sets.} \]

\[ \text{With this and Hedberg, we get that } \Omega \text{-is-a-set.} \]

\[ \Omega \text{-is-a-set : dfunext } \mathcal{U} \to \text{propext } \mathcal{U} \to \text{-is-set } (\Omega \mathcal{U}) \]

\[ \Omega \text{-is-a-set } \{\mathcal{U}\} \text{ fe pe } = \text{types-with-wconstant=\-endomaps-are-sets } (\Omega \mathcal{U}) \] \[ \text{where} \]

\[ A : (p q : \Omega \mathcal{U}) \to \mathcal{U} \]

\[ A \ p \ q = (p \text{ holds } \to q \text{ holds}) \times (q \text{ holds } \to p \text{ holds}) \]

\[ i : (p q : \Omega \mathcal{U}) \to \text{is-subsingleton } (A \ p \ q) \]

\[ i \ p \ q = \Sigma \text{-is-subsingleton \)

\[ (\lambda \_ \to \text{holds-is-subsingleton } q)) \]

\[ (\lambda \_ \to \text{n-is-subsingleton } fe \ (\lambda \_ \to \text{holds-is-subsingleton } p)) \]

\[ g : (p q : \Omega \mathcal{U}) \to p \equiv q \to A \ p \ q \]

\[ g \ p \ q \ e = (u \ , v) \]

\[ \text{where} \]

\[ a : p \text{ holds } \equiv q \text{ holds} \]

\[ a = \text{ap } \_\text{holds } e \]

\[ u : p \text{ holds } \to q \text{ holds} \]

\[ u = \text{Id-fun } a \]

\[ v : q \text{ holds } \to p \text{ holds} \]

\[ v = \text{Id-fun } (a ^{-1}) \]

\[ h : (p q : \Omega \mathcal{U}) \to A \ p \ q \to p \equiv q \]

\[ h \ p \ q \ (u \ , v) = \text{Ω-ext } fe \ p e \ u \ v \]

\[ f : (p q : \Omega \mathcal{U}) \to p \equiv q \to p \equiv q \]

\[ f \ p \ q \ e = h \ p \ q \ (g \ p \ q \ e) \]

\[ k : (p q : \Omega \mathcal{U}) \ (d e : p \equiv q) \to f \ p \ q \ d \equiv f \ p \ q \ e \]

\[ k \ p \ q \ d \ e = \text{ap } (h \ p \ q) \ (i \ p \ q \ (g \ p \ q \ d) \ (g \ p \ q \ e)) \]

\[ c : (p q : \Omega \mathcal{U}) \to \Sigma f : (p \equiv q \to p \equiv q), \text{wconstant } f \]

\[ c \ p \ q = (f \ p \ q \ , k \ p \ q) \]

\[ \text{The above considers } X : \mathcal{U} \text{ and } \Omega \mathcal{V}. \text{When the two universes } \mathcal{U} \text{ and } \mathcal{V} \text{ are the same, we adopt the usual notation } P X \text{ for the powerset } X \to \Omega \mathcal{U} \text{ of } X. \]

\[ P : \mathcal{U} \to \mathcal{U} \] \[ P \ {\mathcal{U}} \ X = X \to \Omega \mathcal{U} \]

\[ \text{powersets-are-sets'} : \text{Univalence} \]

\[ \to \{X : \mathcal{U} \} \to \text{-is-set } (P X) \]

\[ \text{powersets-are-sets'} \ {\mathcal{U}} \ ua = \text{powersets-are-sets} \]

\[ \text{(univalence-gives-hfunext'} (ua \ {\mathcal{U}}) \ (ua (\mathcal{U} \'))) \]

\[ \text{(univalence-gives-dfunext } (ua \ {\mathcal{U}}) \) \]

\[ \text{(univalence-gives-propext } (ua \ {\mathcal{U}}) \) \]

Notice also that both \( \Omega \) and the powerset live in the next universe. With \text{propositional resizing}, we get equivalent copies in the same universe.

Membership and containment for elements of the powerset are defined as follows:

\[ \in : \{X : \mathcal{U} \} \to X \to P X \to \mathcal{U} \]

\[ x \in A = A \times \text{holds} \]
We first prove some properties of equivalence symmetrization and composition:

Some constructions with types of equivalences

Table of contents

For set-level mathematics, function extensionality and propositional extensionality are often the only consequences of univalence for sets (see the HoTT book or this).

Although $\mathcal{P} X$ is a set even if $X$ is not, the total space $\Sigma x : X , A x$ holds of a member $A : \mathcal{P} X$ of the powerset need not be a set. For instance, if $A x$ holds $\equiv \mathbb{1}$ for all $x : X$, then the total space is equivalent to $X$, which may not be a set.

Although $\mathcal{P} X$ is a set even if $X$ is not, the total space $\Sigma x : X , A x$ holds of a member $A : \mathcal{P} X$ of the powerset need not be a set. For instance, if $A x$ holds $\equiv \mathbb{1}$ for all $x : X$, then the total space is equivalent to $X$, which may not be a set.

Propositional and functional extensionality give the usual extensionality condition for the powerset:

And hence so does univalence:

For set-level mathematics, function extensionality and propositional extensionality are often the only consequences of univalence that are needed. A noteworthy exception is the theorem that the type of ordinals in a universe is an ordinal in the next universe, which requires univalence for sets (see the HoTT book or this).

Table of contents

Some constructions with types of equivalences

We first prove some properties of equivalence symmetrization and composition:
id⇒ left fe fe' α = to-subtype⇒ (being-equiv-is-subsingleton fe fe') (refl _)

≡-sym-left-inverse : dfunext U V
  → \{X : U\} \{Y : V\} (α : X ≃ Y)
  → ≡-sym α • α ≡ id⇒ Y

≡-sym-left-inverse fe (f , e) = to-subtype⇒ (being-equiv-is-subsingleton fe fe) p
where
  p : f • inverse f e ≡ id
  p = fe (inverses-are-sections f e)

≡-sym-right-inverse : dfunext U U
  → \{X : U\} \{Y : U\} (α : X ≃ Y)
  → α • ≡-sym α ≡ id⇒ X

≡-sym-right-inverse fe (f , e) = to-subtype⇒ (being-equiv-is-subsingleton fe fe) p
where
  p : inverse f e • f ≡ id
  p = fe (inverses-are-retractions f e)

We then transfer the above to equivalence types:

≡-Sym : dfunext U (U U) → dfunext (U U) (U U)
  → \{X : U\} \{Y : U\} (α : X ≃ Y)
  → (α ≃ Y) ≡ (Y ≃ X)

≡-Sym fe₀ fe₁ fe₂ = ≡-sym , ≡-sym-is-equiv fe₀ fe₁ fe₂

≡-Comp : dfunext W (U U) → dfunext (U U) (U U)
  → dfunext U V → dfunext \{X : U\} \{Y : V\} (Z : W \ U)
  → X = Y → (Y = Z) ≡ (X = Z)

≡-Comp fe₀ fe₁ fe₂ fe₃ fe₄ fe₅ Z α = invertibility-gives⇒ (α • _)
  ((≡-sym α • _), p , q)
where
  p = λ β . (≡-sym α • (α • β)) ≡ (≡-sym α • β) (≡-sym left-inverse fe₂ α)
    id⇒ _ • β ≡ (id⇒ left fe₀ fe₁ _)

 q = λ y . (≡-sym α • y) ≡ (≡-sym α • y) (≡-sym right-inverse fe₃ α)
    id⇒ _ • y ≡ (id⇒ left fe₃ fe₄ _)

Using this we get the following self-congruence property of equivalences:

Eq-Eq-cong' : dfunext U (U U) → dfunext (U U) (U U)
  → dfunext U V → dfunext V V (V U)
  → dfunext W (U V) → dfunext V V (V U)
  → dfunext W (V T) → dfunext T T (V U)
  → \{X : U\} \{Y : U\} \{A : W\} \{B : T\}
  → X = A → Y = B → (X = Y) ≡ (A = B)

Eq-Eq-cong' fe₀ fe₁ fe₂ fe₃ fe₄ fe₅ fe₆ fe₇ fe₈ fe₉ fe₁₀ fe₁₁ \{X\} \{Y\} \{A\} \{B\} α β =

 X = Y ≅ (≡-Comp fe₀ fe₁ fe₂ fe₃ fe₄ fe₅ fe₆ fe₇ fe₈ fe₉ fe₁₀ fe₁₁ \{X\} \{Y\} \{A\} \{B\} α β =
 A = Y ≅ (≡-Sym fe₀ fe₁ fe₂ fe₃ fe₄ fe₅ fe₆ fe₇ fe₈ fe₉ fe₁₀ fe₁₁ fe₁₂)
 Y = A ≅ (≡-Comp fe₀ fe₁ fe₂ fe₃ fe₄ fe₅ fe₆ fe₉ fe₁₀ fe₁₁ A (≡-sym β))
 B = A ≅ (≡-Sym fe₀ fe₁ fe₂ fe₃ fe₄ fe₅ fe₆ fe₇ fe₈ fe₉ fe₁₀ fe₁₁ fe₁₂)
 A = B
More generally, with the arguments swapped, the projection $\text{pr}_2 : A \times X \to X$ is an embedding:

$$\text{pr}_2 : (A : U) \to (X : U) \to (A \times X \to X) \to \text{is-subsingleton} (\text{fiber} \ f y) \to \text{is-subsingleton} \ A$$

This says that for every $y : Y$ there is at most one $x : X$ with $f x \equiv y$, or, more precisely, there is at most one pair $(x, p)$ with $x : X$ and $p : f x \equiv y$.

For example, if $A$ is a subsingleton, then the second projection $A \times X \to X$ is an embedding:

$$\text{pr}_2 : A \times X \to X \to \text{is-subsingleton} \ A \to \text{is-subsingleton} \ (A \times X \to X)$$

where

$$p : ((a, x), \text{refl} x) \equiv ((b, x), \text{refl} x)$$

Exercise. Show that the converse of $pr_2$-embedding holds.

More generally, with the arguments swapped, the projection $\Sigma A \to X$ is an embedding if $A \times X$ is a subsingleton for every $x : X$:

$$\text{pr}_1 : \Sigma A \to X \to (A \times X \to X) \to \text{is-subsingleton} (\text{fiber} \ f y) \to \text{is-subsingleton} (\Sigma A \to X)$$

Exercise. Show that the converse of $pr_1$-is-embedding holds.

Equivalences are embeddings:

$$(f : X \to Y) \to \text{is-equiv} f \to \text{is-embedding} f$$

$$\text{equivs-are-embeddings} : \{X : U\} \to \{Y : U\} \to \{f : X \to Y\} \to \text{is-equiv} \ f \to \text{is-embedding} \ f$$

id-is-embedding : $\{X : U\} \to \text{is-embedding} (\text{id} X)$

id-is-embedding $\{i\} \{X\} = \text{equivs-are-embeddings} \ \text{id} (\text{id-is-equiv} X)$
An equivalent formulation of being an embedding is that the map

\[
\text{ap } f \{x\} \{x'\} : x \equiv x' \to f x \equiv f x'
\]

is an equivalence for all \( x, x' : X \).
Later we will see that a necessary and sufficient condition for an embedding to be an equivalence is that it is a surjection.

If an embedding has a section, then it is an equivalence.

Conversely, left cancellable maps into sets are embeddings.

Hence embeddings of arbitrary types are left cancellable, but the converse fails in general.

Conversely, left cancellable maps into sets are always embeddings.

If an embedding has a section, then it is an equivalence.

Later we will see that a necessary and sufficient condition for an embedding to be an equivalence is that it is a surjection.
If a type $Y$ is embedded into $Z$, then the function type $X \rightarrow Y$ is embedded into $X \rightarrow Z$. More generally, if $A \times x$ is embedded into $B \times x$ for every $x : X$, then the dependent function type $\Pi \ A$ is embedded into $\Pi \ B$.

$$\forall \ A_0 \ B_0 \Nat \ A \rightarrow \Nat \ B$$

(Notice that $Nat_\Pi$ is a dependently typed version of the combinator $S$ from combinatory logic. Its logical interpretation, here, is that if $A \times x$ implies $B \times x$ for all $x : X$, and $A \times x$ holds for all $x : X$, then $B \times x$ holds for all $x : X$ too.)

Nat-is-embedding : hfunext $U \ U$  
  \rightarrow hfunext $U \ U$  
  \rightarrow \{ X : U \} \{ A : X \rightarrow Y \} \{ B : X \rightarrow W \} \rightarrow Nat \ A \rightarrow \Pi \ * \rightarrow \Pi \ B$

Nat-is-embedding $\gamma$ $\rightarrow \{ X : U \} \{ A : X \rightarrow Y \} \{ B : X \rightarrow W \} \rightarrow \{ (\tau \times x) \rightarrow \text{is-embedding} \ (\tau \times x) \}$

Nat-is-embedding $\gamma \rightarrow \text{is-embedding} \ (Nat \ \tau)$

Nat-is-embedding $\gamma \rightarrow \text{is-embedding} \ (Nat \ \tau)$

where

$\gamma : (f \ g : \Pi \ A) \rightarrow (Nat \ \tau \ f \equiv Nat \ \tau \ g) \equiv (f \equiv g)$

Nat-is-embedding $\gamma \rightarrow \{ X : U \} \{ A : X \rightarrow Y \} \{ B : X \rightarrow W \} \rightarrow \text{is-embedding} \ (Nat \ \tau)$

where

$\gamma : (f \ g : \Pi \ A) \rightarrow (Nat \ \tau \ f \equiv Nat \ \tau \ g) \equiv (f \equiv g)$

Nat-is-embedding $\gamma \rightarrow \{ X : U \} \{ A : X \rightarrow Y \} \{ B : X \rightarrow W \} \rightarrow \text{is-embedding} \ (Nat \ \tau)$

Postcomposition with an embedding is itself an embedding (of a function type into another). This amounts to saying that any function $f : X \rightarrow A$ and any embedding $g : Y \rightarrow A$ can be completed to a commutative triangle in at most one way:

triangle-lemma : dfunext $U \ U$ (U \ U) \rightarrow hfunext $U \ U$  
  \rightarrow \{ Y : U \} \{ A : Y \rightarrow X \} \{ (\Sigma \ h : (X \rightarrow Y), g \circ h \sim f) \}$

triangle-lemma $\gamma \rightarrow \{ Y : U \} \{ A : Y \rightarrow X \} \{ (\Sigma \ h : (X \rightarrow Y), g \circ h \sim f) \}$

where

$\gamma : (x : X) \rightarrow (Nat \ \tau \ f \ x \equiv Nat \ \tau \ g \ x) \equiv (f \ x \equiv g \ x)$

$a \equiv \text{is-embedding-converse} \ (\tau \ x) \ (i \ x) \ (f \ x) \ (g \ x)$

$b \equiv \Pi \cong \ (\text{hfunext} \ (\gamma \ \text{gives-dfunext} \ w) \ (\text{hfunext} \ (\gamma \ \text{gives-dfunext} \ w) \ a)$

Postcomp-is-embedding : dfunext $U \ U$ (U \ U) \rightarrow hfunext $U \ U$  
  \rightarrow \{ Y : U \} \{ A : Y \rightarrow X \} \{ (\lambda \ h : (X \rightarrow Y), g \circ h \sim f) \}$

Postcomp-is-embedding $\gamma \rightarrow \{ Y : U \} \{ A : Y \rightarrow X \} \{ (\lambda \ h : (X \rightarrow Y), g \circ h \sim f) \}$

where

$\gamma : (f : X \rightarrow A) \rightarrow (Nat \ \tau \ f \ x \equiv Nat \ \tau \ g \ x) \equiv (f \ x \equiv g \ x)$

$a \equiv \text{equiv-to-subsingleton} \ (\equiv \text{-sym} \ \Pi \text{-distr} \equiv \Rightarrow \text{iii})$

Postcomp-is-embedding $\gamma \rightarrow \{ Y : U \} \{ A : Y \rightarrow X \} \{ (\lambda \ h : (X \rightarrow Y), g \circ h \sim f) \}$

where

$\gamma : (f : X \rightarrow A) \rightarrow (Nat \ \tau \ f \ x \equiv Nat \ \tau \ g \ x) \equiv (f \ x \equiv g \ x)$

$a \equiv \text{equiv-to-subsingleton} \ (\equiv \text{-sym} \ \Pi \text{-distr} \equiv \Rightarrow \text{iii})$

Postcomp-is-embedding $\gamma \rightarrow \{ Y : U \} \{ A : Y \rightarrow X \} \{ (\lambda \ h : (X \rightarrow Y), g \circ h \sim f) \}$

where

$\gamma : (f : X \rightarrow A) \rightarrow (Nat \ \tau \ f \ x \equiv Nat \ \tau \ g \ x) \equiv (f \ x \equiv g \ x)$

$a \equiv \text{equiv-to-subsingleton} \ (\equiv \text{-sym} \ \Pi \text{-distr} \equiv \Rightarrow \text{iii})$

We conclude this section by introducing notation for the type of embeddings.

$\leadsto : U \rightarrow \rightarrow U \rightarrow U \ U$  

$X \leadsto Y \equiv \Sigma \ f : (X \rightarrow Y), \text{is-embedding} \ f$
The following justifies the terminology \textit{subsingleton}:

\textit{Exercise.} (1) Show that the type \textit{is-subsingleton} $X$ is logically equivalent to the type $X \hookrightarrow 1$. (2) Hence assuming function extensionality and propositional extensionality, conclude that \textit{is-subsingleton} $X \equiv (X \hookrightarrow 1)$.

\textit{Exercise.} Show that the map $\text{Fin} : \mathbb{N} \to \mathcal{U}_0$ defined above is left-cancellable but not an embedding.

\textbf{Table of contents} \uparrow

\textbf{The Yoneda Lemma for types}

As we have seen, a type $X$ can be seen as an $\infty$-groupoid and hence as an $\infty$-category, with identifications as the arrows. Likewise a universe $\mathcal{U}$ can be seen as the $\infty$-generalization of the category of sets, with functions as the arrows. Hence a type family

$$A : X \to \mathcal{U}$$

can be seen as an $\infty$-presheaf, because groupoids are self-dual categories.

With this view, the identity type former $\text{Id} X : X \to X \to \mathcal{U}$ plays the role of the \textbf{Yoneda embedding}:

$$\mathcal{Y} : \{X : \mathcal{U} \} \to X \to (X \to \mathcal{U})$$

$$\mathcal{Y} \{\mathcal{U}\} \{X\} = \text{Id} X$$

Sometimes we want to make one of the parameters explicit:

$$Y : (X : \mathcal{U}) \to X \to (X \to \mathcal{U})$$

$$Y \{\mathcal{U}\} X = \mathcal{Y} \{\mathcal{U}\} \{X\}$$

By our definition of $\text{Nat}$, for any $A : X \to \mathcal{U}$ and $x : X$ we have

$$\text{Nat} (\mathcal{Y} x) A = (y : X) \to x \equiv y \to A y,$$

and, by $\text{Nats-are-natural}$, we have that $\text{Nat} (\mathcal{Y} x) A$ is the type of natural transformations from the presheaf $\mathcal{Y} x$ to the presheaf $A$. The starting point of the Yoneda Lemma, in our context, is that every such natural transformation is a transport.

$$\text{transport-lemma} : \{X : \mathcal{U}\} (A : X \to \mathcal{U}) (x : X) \to (\tau : \text{Nat} (\mathcal{Y} x) A) \to (y : X) (p : x \equiv y) \to \tau y p \equiv \text{transport} A p (\tau x (\text{refl} x))$$

$$\text{transport-lemma} A x \tau \equiv \text{transport} A \tau x (\text{refl} x) = \text{refl} (\tau x (\text{refl} x))$$

We denote the point $\tau x (\text{refl} x)$ in the above lemma by $\mathcal{E} A x \tau$ as refer to it as the \textbf{Yoneda element} of the transformation $\tau$.

$$\mathcal{E} : \{X : \mathcal{U}\} (A : X \to \mathcal{U}) (x : X) \to \text{Nat} (\mathcal{Y} x) A \to A x$$

$$\mathcal{E} A x \tau = \tau x (\text{refl} x)$$

The function

$$\mathcal{E} A x : \text{Nat} (\mathcal{Y} x) A \to A x$$

is an equivalence with inverse

$$\mathcal{N} A x : A x \to \text{Nat} (\mathcal{Y} x) A,$$

the transport natural transformation induced by $A$ and $x$:

$$\mathcal{N} : \{X : \mathcal{U}\} (A : X \to \mathcal{U}) (x : X) \to A x \to \text{Nat} (\mathcal{Y} x) A$$

$$\mathcal{N} A x a y p = \text{transport} A p a$$
If the transport transformation is a fiberwise equivalence, then the Yoneda Lemma for types says that natural transformations from $\mathcal{U} x$ to $A$ are in canonical bijection with elements of $A x$:

$$\pi = \text{invertibles-are-equivs} \circ (\pi A x) \circ \text{yoneda-η} \circ \text{fe} \circ \text{fe}' A x$$

By a fiberwise equivalence we mean a natural transformation whose components are all equivalences:

$$\pi \text{-is-equiv} : \text{dfunext} \mathcal{U} \mathcal{V} \to \text{dfunext} \mathcal{U} \mathcal{V}$$

This gives the Yoneda Lemma for types, which says that natural transformations from $\mathcal{U} x$ to $A$ are in canonical bijection with elements of $A x$:

$$\text{Yoneda-Lemma} : \text{dfunext} \mathcal{U} \mathcal{V} \to \text{dfunext} \mathcal{U} \mathcal{V}$$

A universal element of a presheaf $A$ corresponds to an element of the type $\text{is-singleton} \ (\Sigma A)$, which can also be written $\exists! A$.

If the transport transformation is a fiberwise equivalence, then $A$ has a universal element. More generally, we have the following:

$$\text{retract-universal-lemma} : \text{dfunext} \mathcal{U} \mathcal{V} \to \text{dfunext} \mathcal{U} \mathcal{V}$$

$$\text{fiberwise-equiv-universal} : \text{dfunext} \mathcal{U} \mathcal{V} \to \text{dfunext} \mathcal{U} \mathcal{V}$$
A presheaf is called fiberwise equivalence if it is pointwise equivalent to a presheaf of the form $\mathcal{V} X$. This says that if we have a fiberwise retraction, then any natural transformation is an equivalence. And the following says that if we have a fiberwise equivalence, then any natural transformation is an equivalence:

$$universal-fiberwise-equiv : \{U\} \{V\} \{X\} A u x = retract-universal-lemma A x$$

where

$\rho : \forall y A y \triangleleft (x \equiv y)$

$\rho y = \_\text{-gives-}\triangleright ((\tau y), e y)$

Conversely:

$$universal-fiberwise-equiv : \{X : U\} (A : X \to \mathcal{V})$$

$\rightarrow \exists! A$

$\rightarrow (x : X) (\tau : Nat (\mathcal{V} x) A) \rightarrow is-fiberwise-equiv \tau$

$$universal-fiberwise-equiv \{\mathcal{U}\} \{\mathcal{V}\} \{X\} A u x = y$$

where

$g : singleton-type' x \to \Sigma A$

$g = Nat\Sigma \tau$

$e : is-equiv g$

$e = maps-of-singletons-are-eqvs g (singleton-types'-are-singletons X x) u$

$y : is-fiberwise-equiv \tau$

$y = Nat\Sigma e\text{-gives-fiberwise-equiv} \tau e$

In particular, the induced transport transformation $\tau = \mathcal{N} A x a$ is a fiberwise equivalence if and only if there is a unique $x : X$ with $A x$, which we abbreviate as $\exists! A$.

A corollary is the following characterization of function extensionality, similar to the above characterization of univalence, by taking the transformation $\tau = \text{happly} f$, because $\text{hfe} f$ says that $\tau$ is a fiberwise equivalence:

$$\text{hfunext}\rightarrow : \text{hfunext} \mathcal{U} \mathcal{V}$$

$$\rightarrow ((X : U) (A : X \to \mathcal{V}) (f : \Pi A) \rightarrow \exists! g : \Pi A, f \sim g)$$

$$\text{hfunext}\rightarrow : \text{hfunext} X A f = universal-fiberwise-equiv universal-fiberwise-equiv (f \sim) (\text{happly} f) (\text{hfe} f)$$

$$\text{hfunext}\rightarrow : \text{hfunext} f X A$$

$$\rightarrow hfunext \mathcal{U} \mathcal{V}$$

$$\rightarrow \text{hfunext} \varphi \{X\} \{A\} f = universal-fiberwise-equiv universal-fiberwise-equiv (f \sim) (\varphi X A f) (\text{happly} f)$$

We also have the following general corollaries:

$$fiberwise-equiv-criterion : \{X : U\} (A : X \to \mathcal{V})$$

$$(x : X)$$

$$\rightarrow ((y : X) \rightarrow A y \triangleleft (x \equiv y))$$

$$(\tau : Nat (\mathcal{V} x) A) \rightarrow is-fiberwise-equiv \tau$$

$$fiberwise-equiv-criterion A x \rho = retract-universal-lemma A x (universal-fiberwise-equiv universal-fiberwise-equiv (retract-universal-lemma A x) x)$$

This says that if we have a fiberwise retraction, then any natural transformation is an equivalence. And the following says that if we have a fiberwise equivalence, then any natural transformation is a fiberwise equivalence:

$$fiberwise-equiv-criterion' : \{X : U\} (A : X \to \mathcal{V})$$

$$(x : X)$$

$$\rightarrow ((y : X) \rightarrow (x \equiv y) = A y)$$

$$(\tau : Nat (\mathcal{V} x) A) \rightarrow is-fiberwise-equiv \tau$$

$$fiberwise-equiv-criterion' A x e = fiberwise-equiv-criterion A x$$

$$\lambda y \rightarrow \_\text{-gives-}\triangleright (e y)$$

A presheaf is called representable if it is pointwise equivalent to a presheaf of the form $\mathcal{V} x$:

$$\triangleleft : \{X : U\} \rightarrow (X \to \mathcal{V}) \rightarrow (X \to \mathcal{W})$$

$$\rightarrow U \cup \mathcal{V} \cup \mathcal{W}$$

$$A \cong B \equiv \mathcal{V} x \rightarrow A x \cong B x$$

$$is-representable : \{X : U\} \rightarrow (X \to \mathcal{V}) \rightarrow U \cup \mathcal{V}$$
representable-universal : {X : U'} {A : X → Y'} → is-representable A → ∃! A

representable-universal A (x , e) = retract-universal-lemma A x
(λ x → ≃-gives-▷ (e x))

universal-representable : {X : U'} {A : X → Y'} → ∃! A
→ is-representable A

universal-representable {U} {V} {X} {A} ((x , a) , p) = x , φ
where
 e : is-fiberwise-equiv (N A x a)
e = universal-fiberwise-equiv A ((x , a) , p) x (N A x a)

φ : (y : X) → (x ≃ y) ≃ A y
φ y = (N A x a y , e y)

Combining retract-universal-lemma and universal-fiberwise-equiv we get the following:

fiberwise-retractions-are-equivs : {X : U'} {A : X → Y'} (x : X)
→ (τ : Nat (Y X) A)
→ ((y : X) → has-section (τ y))
→ is-fiberwise-equiv τ

fiberwise-retractions-are-equivs {U} {V} {X} A x τ s = γ
where
 ρ : (y : X) → A y <λ (x ≃ y)
ρ y = τ y , s y
 i : ∃! A
 i = retract-universal-lemma A x ρ
 γ : is-fiberwise-equiv τ
 γ = universal-fiberwise-equiv A i x τ

Perhaps the following formulation is more appealing:

fiberwise-≼-gives-≃ : {X : U'} {A : X → Y'} (x : X)
→ ((y : X) → A y <≃ (x ≃ y))
→ ((y : X) → A y = (x ≃ y))

fiberwise-≼-gives-≃ X A x ρ = γ
where
 f : (y : X) → (x ≃ y) → A y
 f y = retraction (ρ y)
e : is-fiberwise-equiv f
e = fiberwise-retractions-are-equivs A x f (λ y → retraction-has-section (ρ y))

γ : (y : X) → A y ≃ (x ≃ y)
γ y = ≃-sym(f y , e y)

We have the following corollary:

embedding-criterion' : {X : U'} {Y : V'} (f : X → Y)
→ ((x x' : X) → (f x ≃ f x') ≃ (x ≃ x'))
→ is-embedding f

embedding-criterion' f ρ = embedding-criterion f
(λ x → fiberwise-≼-gives-≃ (domain f))
(λ y → f x ≃ f y) x (ρ y)

Exercise. It also follows that f is an embedding if and only if the map ap f {x} {x'} has a section.
To prove that \( Y \{ u \} \{ x \} \) is an embedding of \( x \) into \( x \to u \) for any type \( x : u \), we need the following two lemmas, which are interesting in their own right:

\[
\begin{align*}
\text{being-fiberwise-equiv-is-subsingleton} : & \quad \text{global-dfunext} \\
& \quad \rightarrow \{ X : U \} \{ A : X \to V \} \{ B : X \to W \} \\
& \quad \rightarrow (\tau : \text{Nat} \ A \ B) \\
& \quad \rightarrow \text{is-subsingleton} (\text{is-fiberwise-equiv} \ \tau) \\
\text{being-fiberwise-equiv-is-subsingleton} \ \fe \ \tau = \\
\Pi\text{-is-subsingleton} \ \fe \ (\lambda \ y \to \text{being-equiv-is-subsingleton} \ \fe \ \fe \ (\tau \ y)) \\
\text{being-representable-is-subsingleton} : & \quad \text{global-dfunext} \\
& \quad \rightarrow \{ X : U \} \{ A : X \to V \} \\
& \quad \rightarrow \text{is-subsingleton} (\text{is-representable} \ A) \\
\text{being-representable-is-subsingleton} \ \fe \ \{ X \} \ A \ r_0 \ r_1 = \gamma \\
\text{where} \\
u : \exists! \ A \\
u = \text{representable-universal} \ A \ r_0 \\
\ i : (x : X) (\tau : \text{Nat} \ (Y \ x) \ A) \rightarrow \text{is-singleton} (\text{is-fiberwise-equiv} \ \tau) \\
i \ \times \ \tau = \text{pointed-subsingletons-are-singletons} \\
(\text{is-fiberwise-equiv} \ \tau) \\
(\text{universal-fiberwise-equiv} \ A \ u \ \times \ \tau) \\
(\text{being-fiberwise-equiv-is-subsingleton} \ \fe \ \tau) \\
\epsilon : (x : X) \to (Y \ x \equiv A) = A \ x \\
\epsilon \ x = ((y : X) \to Y \ x \ y \equiv A \ y) \\
(\Sigma \ \tau : \text{Nat} \ (Y \ x) \ A, \ \text{is-fiberwise-equiv} \ \tau) \\
\text{Nat} \ (Y \ x) \ A \\
A \ x = \text{Yoneda-Lemma} \ \fe \ \fe \ A \ x \\
\delta : \text{is-representable} \ A \equiv \Sigma \ A \\
\delta = \Sigma\text{-cong} \ \epsilon \\
v : \text{is-singleton} (\text{is-representable} \ A) \\
v = \text{equiv-to-singleton} \ \delta \ \ u \\
y : \ r_0 \equiv r_1 \\
y = \text{singletons-are-subsingletons} (\text{is-representable} \ A) \ \nu \ r_0 \ r_1 \\
\end{align*}
\]

With this it is almost immediate that the Yoneda map \( Y \ x \) is an embedding of \( x \) into \( x \to u \):

\[
\begin{align*}
\text{Y-is-embedding} : & \quad \text{Univalence} \to (X : U) \to \text{is-embedding} \ (Y \ x) \\
\text{Y-is-embedding} \ \{ u \} \ \ua \ X \ A = \gamma \\
\text{where} \\
hfe : \text{global-hfunext} \\
hfe = \text{univalence-gives-global-hfunext} \ \ua \\
dfe : \text{global-dfunext} \\
dfe = \text{univalence-gives-global-dfunext} \ \ua \\
p = \lambda \ x \to (Y \ x \equiv A) \\
p = (\text{Y-is-embedding} \ \{ u \} \ \ua \ X \ A = \gamma) \\
(i : (y : X) \to Y \ x \ y \equiv A \ y) \\
(ii : (y : X) \to Y \ x \ y = A \ y) \\
\text{where} \\
i = \lambda \ x \to (\text{happly} \ (Y \ x) \ A, \ hfe \ (Y \ x) \ A) \\
ii = \lambda \ x \to \Pi\text{-cong} \ dfe \ dfe \\
(\lambda \ y \to \text{univalence-} \ (\ua \ u) \\
(Y \ x \ y) (A \ y)) \\
e : \text{fiber} \ Y \ A \equiv \text{is-representable} \ A \\
e = \Sigma\text{-cong} \ p \\
y : \text{is-subsingleton} (\text{fiber} \ Y \ A) \\
y = \text{equiv-to-subsingleton} \ e \ (\text{being-representable-is-subsingleton} \ \dfe \ A) \\
\end{align*}
\]
What is a function?

In set theory, a function is a relation between two sets that associates to every element of the first set exactly one element of the second set. We say that the relation is functional.

In type theory, on the other hand, the notion of function is taken as primitive. However, we can show that the type of functions is equivalent to the type of functional relations. This relies on univalence.

When the types under consideration are sets, the corresponding relations are truth valued. But for the equivalence between functions and functional relations to hold for arbitrary types, we need to work with type valued relations.

More generally, we have a one-to-one correspondence between dependent functions \( f : (x : X) \to A x \) and dependent type valued functional relations \( R : (x : X) \to A x \to \mathcal{V} \). We take the domain \( X \) and codomain \( A \) as parameters for a submodule:

```haskell
module function-graphs
  (ua : Univalence)
  {\mathcal{U} \mathcal{V} : Universe}
  (X : \mathcal{U}.)
  (A : X \to \mathcal{V}.)
  where

  hfe : global-hfunext
  hfe = univalence-gives-global-hfunext ua

  fe : global-dfunext
  fe = univalence-gives-global-dfunext ua
```

The type of dependent functions:

\[
\text{Function} : \mathcal{U} \sqcup (\mathcal{V}.)
\]

\[
\text{Function} = (x : X) \to A x
\]

That of dependent relations:

\[
\text{Relation} : \mathcal{U} \sqcup (\mathcal{V}.)
\]

\[
\text{Relation} = (x : X) \to A x \to \mathcal{V}.
\]

A relation \( R \) is said to be functional if for every \( x : X \) there is a unique \( a : A x \) with \( R x a \):

\[
is-functional : \text{Relation} \to \mathcal{U} \sqcup (\mathcal{V}.)
\]

\[
is-functional R = (x : X) \to \exists! a : A x , R x a
\]

Although the relation is allowed to take values in arbitrary types, its functionality condition is a truth value:

\[
\text{being-functional-is-subsingleton} : (R : \text{Relation})
\]

\[
\to \text{is-subsingleton} \ (\text{is-functional} \ R)
\]

\[
\text{being-functional-is-subsingleton} \ R = \Pi-is-subsingleton \ fe
\]

\[\]

\[
(\lambda x \to \exists!-is-subsingleton \ (R x) \ fe)
\]

The type of functional relations:

\[
\text{Functional-Relation} : \mathcal{U} \sqcup (\mathcal{V}.)
\]

\[
\text{Functional-Relation} = \Sigma R : \text{Relation} , \text{is-functional} \ R
\]

To a function \( f \) we associate the relation \( R \) defined by \( R x a = (f x \equiv a) \). Notice that \( R \) is truth valued if the type \( A x \) is a set for every \( x : X \), by definition of set.

\[
\rho : \text{Function} \to \text{Relation}
\]

\[
\rho f = \lambda x a \to f x \equiv a
\]

To show that the map \( \rho \) is an embedding we apply the Yoneda embedding and the fact that the map \( \pi \text{Nat} \pi \) transforms fiberwise embeddings into embeddings:

\[
\rho-is-embedding : \text{is-embedding} \ \rho
\]

\[
\rho-is-embedding = \Pi\text{Nat}\text{is-embedding} \ hfe \ hfe
\]

\[\]

\[
(\lambda x \to Y \ (A x))
\]
This relies implicitly on the following remarks:

\[ \tau : (x : X) \to A x \to (A x \to Y) \]

\[ \tau x a b = a \equiv b \]

\[ \text{remark}_0 : \tau \equiv \lambda x \to Y (A x) \]

\[ \text{remark}_0 = \text{refl} \]

\[ \text{remark}_1 : \rho \equiv \text{Nat}\Pi \tau \]

\[ \text{remark}_1 = \text{refl} \]

The relation induced by a function is functional, of course:

\[ \rho\text{-is-functional} : (f : \text{Function}) \to \text{is-functional} (\rho f) \]

\[ \rho\text{-is-functional} f = \sigma \]

where

\[ \sigma : (x : X) \to \exists! a : A x, f x \equiv a \]

\[ \sigma x = \text{singleton-types'}-\text{are-singletons} (A x) (f x) \]

The graph map associates functional relations to functions:

\[ \Gamma : \text{Function} \to \text{Functional-Relation} \]

\[ \Gamma f = \rho f, \rho\text{-is-functional} f \]

The function \( \Gamma \) can be seen as the corestriction of \( \rho \) to its image.

We get a function from a functional relation by unique choice, which is just projection:

\[ \Phi : \text{Functional-Relation} \to \text{Function} \]

\[ \Phi (R, \sigma) = \lambda x \to \text{pr}_1 (\text{center} (\Sigma a : A x, R x a) (\sigma x)) \]

To show that these two constructions are mutually inverse, we again apply the Yoneda machinery, but in a different way.

\[ \Gamma\text{-is-equiv} : \text{is-equiv} \Gamma \]

\[ \Gamma\text{-is-equiv} = \text{invertibles-are-eqivs} \Gamma (\Phi, \eta, \varepsilon) \]

where

\[ \eta : \Phi \circ \Gamma \sim \text{id} \]

\[ \eta = \text{refl} \]

\[ \varepsilon : \Gamma \circ \Phi \sim \text{id} \]

\[ \varepsilon (R, \sigma) = a \]

where

\[ f : \text{Function} \]

\[ f = \Phi (R, \sigma) \]

\[ e : (x : X) \to R x (f x) \]

\[ e x = \text{pr}_2 (\text{center} (\Sigma a : A x, R x a) (\sigma x)) \]

\[ \tau : (x : X) \to \text{Nat} (Y (f x)) (R x) \]

\[ \tau x = \mathcal{N} (R x) (f x) (e x) \]

\[ \tau\text{-is-fiberwise-equiv} : (x : X) \to \text{is-fiberwise-equiv} (\tau x) \]

\[ \tau\text{-is-fiberwise-equiv} x = \text{universal-fiberwise-equiv} (R x) (\sigma x) (f x) (\tau x) \]

\[ d : (x : X) (a : A x) \to (f x \equiv a) = R x a \]

\[ d x a = \tau x a, \tau\text{-is-fiberwise-equiv} x a \]

\[ c : (x : X) (a : A x) \to (f x \equiv a) \equiv R x a \]

\[ c x a = \text{Eq-id} (\text{ua} \mathcal{V}) _-_ (d x a) \]

\[ b : \rho f \equiv R \]

\[ b = \text{fe} (\lambda x \to \text{fe} (c x)) \]

\[ a : (\rho f, \rho\text{-is-functional} f) \equiv (R, \sigma) \]

\[ a = \text{to-subtype-equiv being-functional-is-subsingleton} b \]

Therefore we have a bijection between functions and functional relations:
Partial functions

Based on the previous section, we can define a *partial function* to be a relation \( R \) such that for every \( x : X \) there is *at most one* \( a : A \times x \) with \( R \times a \). We use \( \Pi_\beta \) for the type of dependent partial functions and \( \to \) for the type of partial functions.

\[
\begin{align*}
\Pi_\beta : & \{X : U \to \} \to (X \to V) \to U \uplus (V) \to \\
\Pi_\beta \{U\} \{V\} \{X\} A = & \Sigma R : ((x : X) \to A \times x \to V) \\
& , ((x : X) \to \text{is-subsingleton} (\Sigma a : A \times), R \times a)) \\
\end{align*}
\]

**Exercise.** Define partial function composition, both in non-dependent and dependent versions.

\[
\begin{align*}
is\text{-defined} : & \{X : U \to \} \{A : X \to V \to \} \to \Pi_\beta A \to X \to V \\
is\text{-defined} \ (R \to \sigma) x = & \Sigma a : \_, R \times a \\
\end{align*}
\]

\[
\begin{align*}
\text{being-defined-is-subsingleton} : & \{X : U \to \} \{A : X \to V \to \} \ (f : \Pi_\beta A) \ (x : X) \\
& \to \text{is-subsingleton} (\text{is-defined} f x) \\
\end{align*}
\]

\[
\begin{align*}
\text{being-defined-is-subsingleton} \ (R \to \sigma) x = & \sigma x \\
\end{align*}
\]

Notice that we have to write *is-defined* \( f x \), and we say that \( f \) is defined at \( x \), or that \( x \) is in the domain of definition of \( f \), rather than *is-defined* \( f x \). In fact, before being able to evaluate a partial function \( f \) at an argument \( x \), we need to know that \( f \) is defined at \( x \). However, in informal discussions we will say “\( f x \) is defined” by the usual abuse of notation and terminology.

We will write the application of a partial function \( f \) to \( x \), under the information \( i \) that \( f x \) is defined, as \( f \ [ x \ , i \ ] \).

\[
\begin{align*}
\_\ [\ , \_] : & \{X : U \to \} \{A : X \to V \to \} \ (f : \Pi_\beta A) \ (x : X) \to \text{is-defined} f x \times A x \\
& (R \to \sigma) \ [ x \ , \ (a \ , r)] = a \\
\end{align*}
\]

**Exercise.** Define *Kleene equality* of two partial functions \( f g : \Pi_\beta A \) by saying that for all \( x : X \), if whenever one of \( f x \) and \( g x \) is defined then so is the other, and whenever they are both defined, then they are equal:

\[
\begin{align*}
\_\equiv_\_ : & \{X : U \to \} \{A : X \to V \to \} \to \Pi_\beta A \to \Pi_\beta A \to U \uplus V \to \\
& f \equiv_\_ g = \forall x \to (\text{is-defined} f x \iff \text{is-defined} g x) \\
& \times ((i : \text{is-defined} f x) \ (j : \text{is-defined} g x) \to f [ x \ , i \ ] \equiv g [ x \ , j \ ])) \\
\end{align*}
\]

Show that the equality of two partial functions, in the sense of the identity type, is equivalent to their Kleene equality. This needs univalence. If all types \( A \times x \) are sets, then functional and propositional extensionality suffice. In the general case, it is easier, or less hard, to approach this problem using the chapter on *equality of mathematical structures*.

**Example.** The famous *\( \mu \)-operator* from recursion theory is a partial function.

\[
\begin{align*}
\text{module} \ & \ \mu\text{-operator} \ (fe : \text{dfunext} \ U_0 \ U_0) \ \text{where} \\
& \ \text{open} \ \text{basic-arithmetic-and-order} \\
\end{align*}
\]

First we need to show that the property of being a minimal root is a truth value and that the type of minimal roots has at most one element. It is this that requires function extensionality.

\[
\begin{align*}
\text{being-minimal-root-is-subsingleton} : & (f : N \to N) \ (m : N) \\
& \to \text{is-subsingleton} (\text{is-minimal-root} f m) \\
\end{align*}
\]

\[
\begin{align*}
\text{being-minimal-root-is-subsingleton} f m = & x\text{-is-subsingleton} \\
& (N\text{-is-set} (f m) \emptyset) \\
& (\Pi\text{-is-subsingleton} fe) \\
& (\lambda _\to N\text{-is-subsingleton} fe \\
\end{align*}
\]
minimal-root-is-singleton : (f : ℕ → ℕ) → is-singleton (minimal-root f)

minimal-root-is-singleton f (m , p , φ) (m' , p' , φ') =
to-subtype≡
(being-minimal-root-is-singleton f)
(at-most-one-minimal-root f m m' (p , φ) (p' , φ'))

We now define μ so that if f has a root then μ f is defined, and, conversely, if μ f is defined then it is the minimal root of f. Most of the work has already been done in the module basic-arithmetic-and-order.

μ : (ℕ → ℕ) → ℕ
μ = is-minimal-root , minimal-root-is-singleton

μ-property₀ : (f : ℕ → ℕ) → (Σ n : ℕ , f n ≡ 0) → is-defined μ f
μ-property₀ = root-gives-minimal-root

μ-property₁ : (f : ℕ → ℕ) (i : is-defined μ f)
→ ((n : ℕ) → n < μ [ f , i ] → f n ≢ 0)
× (λ _ → n-is-singleton f)
(λ _ → Θ-is-singleton)))

Exercise. Define

is-total : {X : U} {A : X → Y} → Π p A → U ⊔ Y
is-total f = ∀ x → is-defined f x

Show that the type Σ f ꞉ Π A , is-total f of total partial functions is equivalent to the type Π A of functions. In particular, the type Σ f ꞉ X ꞉ Y , is-total f is equivalent to the type X → Y.

Exercise. Two other natural renderings of the notion of partial function, for X Y : U, are given by the equivalences

(X → Y) ≃ (X → L Y)
≃ Σ D ꞉ U , (D ⊆ X) × (D → Y)

where

L Y = Σ P ꞉ U , is-singleton P × (P → Y)
≃ (⊤ → Y)

are two equivalent formulations of the type of partial elements of Y. Generalize the universes, and generalize these alternative descriptions of the type of partial functions to dependent partial functions, and prove them.

Table of contents

Universe lifting

Universe lifting Universes are not cumulative on the nose in Agda, in the sense that from

X : U

we would get that

X : U₁

or that

X : U ⊔ V.

Instead we work with embeddings of universes into larger universes.
The following together with its induction principle should be considered as part of the universe handling of our spartan Martin-Löf type theory:

```
record Lift (U : Universe) (V : Universe) (X : U') : U ⊔ V' where
  constructor
  lift
  field
    lower : X

open Lift public
```

The functions `Lift`, `lift` and `lower` have the following types:

- `type-of-Lift : type-of (Lift {U} V) ≡ (U' → U ⊔ V')`
- `type-of-lift : {X : U'} → type-of (lift {U} {V} {X}) ≡ (X → Lift V X)`
- `type-of-lower : {X : U'} → type-of (lower {U} {V} {X}) ≡ (Lift V X → X)`

```
type-of-Lift = refl
type-of-lift = refl
type-of-lower = refl
```

The induction and recursion principles are as follows:

- `Lift-induction : ∀ {U} V {X : U'} (A : Lift V X → W')
  → ((X : X) → A (lift x))
  → (l : Lift V X) → A l`

```
Lift-induction V X A φ (lift x) = φ x
```

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

```
Lift-recursion V {X} {B} = Lift-induction V X (λ _ → B)
```

This gives an equivalence `lift : X → Lift V X` and hence an embedding `Lift V : U' → U ⊔ V`. The following two constructions can be performed with induction, but actually hold on the nose by the so-called `η` rule for records:

- `lower-lift : {X : U'} (x : X) → lower {U} {V} (lift x) ≡ x`

```
lower-lift = refl
```

- `lift-lower : {X : U'} (l : Lift V X) → lift (lower l) ≡ l`

```
lift-lower = refl
```

- `Lift≃ : (X : U') → Lift V X ≃ X`
- `Lift≃ {U} {V} X = invertibility-gives≃ lower
  (lift , lift-lower , lower-lift {U} {V})`

```
≃-Lift : (X : U') → Lift V X
≃-Lift {U} {V} X = invertibility-gives≃ lift
  (lower , lower-lift {U} {V} , lift-lower)
```

With universe lifting, we can generalize equivalence induction as follows, in several steps.

Firstly, function extensionality for a pair of universes gives function extensionality for any pair of lower universes:

```
lower-dfunext : ∀ W T U V → dfunext (U ⊔ W) (V ⊔ T) → dfunext U V
```

where
```
A' : Lift W X → V ⊔ T'
A' y = Lift T (A (lower y))

f' g' : ⨿ A'

f' y = lift (f (lower y))
g' y = lift (g (lower y))
```

h' : f' ~ g'
\[ h' \ y = ap \ lift \ (h \ (lower \ y)) \]

\[ p' : f' \equiv g' \]
\[ p' = fe \ h' \]

\[ p : f \equiv g \]
\[ p = ap \ (\lambda \ f' \ x \to \ lower \ (f' \ (lift \ x))) \ p' \]

Secondly, a function from a universe to a higher universe is an embedding provided it maps any type to an equivalent type and the two universes are univalent:

\[
\text{universe-embedding-criterion} : \text{is-univalent } \mathcal{U} \\
\to \text{is-univalent } (\mathcal{U} \uplus \mathcal{V}) \\
\to (f : \mathcal{U} ' \to \mathcal{U} \uplus \mathcal{V}) \\
\to ((X : \mathcal{U} ') \to f \ X \simeq X) \\
\to \text{is-embedding } f
\]

\[
\text{universe-embedding-criterion} \{\mathcal{U}\} \{\mathcal{V}\} \ ua \ ua' \ f \ e = \text{embedding-criterion } f \ y
\]

where

\[
fe : \text{dfunext } (\mathcal{U} \uplus \mathcal{V}) (\mathcal{U} \uplus \mathcal{V})
\]
\[
fe = \text{univalence-gives-dfunext } ua'
\]

\[
fe_0 : \text{dfunext } \mathcal{U} \mathcal{U}
\]
\[
fe_0 = \text{lower-dfunext } \mathcal{V} \mathcal{V} \mathcal{U} \mathcal{U} \ fe
\]

\[
fe_1 : \text{dfunext } (\mathcal{U} \uplus \mathcal{V})
\]
\[
fe_1 = \text{lower-dfunext } \mathcal{V} \mathcal{V} (\mathcal{U} \uplus \mathcal{V}) \ fe
\]

\[
y : (X X' : \mathcal{U} ') \to (f X \equiv f X') \simeq (X \equiv X')
\]

\[
y X X' = (f X \equiv f X') \simeq (i)
\]
\[
(f X \equiv f X') \simeq (ii)
\]
\[
(X \equiv X') \simeq (iii)
\]

where

\[
i = \text{univalence-=} \ ua' (f X) (f X')
\]
\[
ii = \text{Eq-Eq-cong}' \ fe \ fe \ fe \ fe \ fe_0 \ fe_1 \ fe \ fe_0 \ fe_0 \ fe_0 \ fe_0 \ fe_0 \ e \ X (e X')
\]
\[
iii = \simeq\text{-sym } (\text{univalence-=} \ ua \ X X')
\]

In particular, the function \( \text{Lift} \) is an embedding:

\[
\text{Lift-is-embedding} : \text{is-univalent } \mathcal{U} \to \text{is-univalent } (\mathcal{U} \uplus \mathcal{V})
\]
\[
\to \text{is-embedding } (\text{Lift } \{\mathcal{U}\} \mathcal{V})
\]

\[
\text{Lift-is-embedding } \{\mathcal{U}\} \{\mathcal{V}\} \ ua \ ua' = \text{universe-embedding-criterion } \{\mathcal{U}\} \{\mathcal{V}\} \ ua \ ua'
\]

Thirdly, we have a generalization of \( \text{univalence-=} \) from a single universe to a pair of universes. We work with two symmetrical versions, where the second is derived from the first. We use an anonymous module to assume univalence in the following couple of constructions:

\[
\text{module } _\{\mathcal{U} \mathcal{V} : \text{Universe}\}
\]
\[
(ua : \text{is-univalent } \mathcal{V})
\]
\[
(ua' : \text{is-univalent } (\mathcal{U} \uplus \mathcal{V}))
\]

where

\[
fe : \text{dfunext } (\mathcal{U} \uplus \mathcal{V}) (\mathcal{U} \uplus \mathcal{V})
\]
\[
fe = \text{univalence-gives-dfunext } ua'
\]

\[
fe_0 : \text{dfunext } \mathcal{V} (\mathcal{U} \uplus \mathcal{V})
\]
\[
fe_0 = \text{lower-dfunext } \mathcal{U} \mathcal{V} (\mathcal{U} \uplus \mathcal{V}) \ fe
\]

\[
fe_1 : \text{dfunext } (\mathcal{U} \uplus \mathcal{V})
\]
\[
fe_1 = \text{lower-dfunext } (\mathcal{U} \uplus \mathcal{V}) \mathcal{U} \mathcal{V} (\mathcal{U} \uplus \mathcal{V}) \ fe
\]

\[
fe_2 : \text{dfunext } \mathcal{V} \mathcal{V}
\]
\[
fe_2 = \text{lower-dfunext } \mathcal{U} \mathcal{V} \mathcal{V} \mathcal{U} \mathcal{V} \ fe
\]

\[
fe_3 : \text{dfunext } \mathcal{U} \mathcal{U}
\]
\( \text{fe}_3 = \text{lower-dfunext} \forall \forall \ u \ u \ \text{fe} \)

\begin{verbatim}
univalence-' : (X : U`) → is-subsingleton (Σ Y : V` , X ≃ Y)
univalence-' X = s
where
  abstract
  e : (Y : V`) → (X ≃ Y) = (Lift U Y ≡ Lift V X)
  e Y = (X ≃ Y) ≃ (i)
  (Y ≃ X) ≃ (ii)
  (Lift U Y ≡ Lift V X) ≃ (iii)
  (Lift U Y ≡ Lift V X) □

  where
  i = ≃-Sym fe_0 fe_i fe
  ii = Eq-Eq-cong' fe_i fe fe_i fe fe fe fe
  iii = ≃-sym (univalence-≡ ua' (Lift U Y) (Lift V X))

d : (Σ Y : V` , X ≃ Y) = (Σ Y : V` , Lift U Y ≡ Lift V X)
d = Σ-cong e

j : is-subsingleton (Σ Y : V` , Lift U Y ≡ Lift V X)
j = Lift-is-embedding ua ua' (Lift V X)

s : is-subsingleton (Σ Y : V` , X ≃ Y)
s = equiv-to-subsingleton d j

univalence-'-dual : (Y : U`) → is-subsingleton (Σ X : V` , X ≃ Y)
univalence-'-dual Y = equiv-to-subsingleton e i
where
  e : (Σ X : V` , X ≃ Y) ≃ (Σ X : V` , Y ≃ X)
e = Σ-cong (λ X → ≃-Sym fe_i fe_0 fe)

  i : is-subsingleton (Σ X : V` , Y ≃ X)
i = univalence-' Y

This is the end of the anonymous module. We are interested in these corollaries:

\begin{verbatim}
univalence-" : is-univalent (U ⊔ V)
  → (X : U`) → is-subsingleton (Σ Y : U ⊔ V` , X ≃ Y)
univalence-" ua = univalence-' ua ua

univalence-"-dual : is-univalent (U ⊔ V)
  → (Y : U`) → is-subsingleton (Σ X : U ⊔ V` , X ≃ Y)
univalence-"-dual ua = univalence-"-dual ua ua
\end{verbatim}

The first one is applied to get the following, where \( Y \) lives in a universe above that of \( X \):

\begin{verbatim}
G↓-≡ : is-univalent (U ⊔ V)
  → (X : U`) (A : (Σ Y : U ⊔ V` , X ≃ Y) → W`)
  → A (Lift V X , ≃-Lift X) → (Y : U ⊔ V`) (e : X ≃ Y) → A (Y , e)

G↓-≡ \{U\} \{V\} ua X A a Y e = transport A p a
where
  t : Σ Y : U ⊔ V` , X ≃ Y
t = (Lift V X , ≃-Lift X)

  p : t ≡ (Y , e)
p = univalence-"-" \{U\} \{V\} ua X t (Y , e)
\end{verbatim}

\begin{verbatim}
H↓-≡ : is-univalent (U ⊔ V)
  → (X : U`) (A : (Y : U ⊔ V`) → X ≃ Y → W`)
  → A (Lift V X) (≃-Lift X) → (Y : U ⊔ V`) (e : X ≃ Y) → A Y e

H↓-≡ ua X A = G↓-≡ ua X (Σ-induction A)
\end{verbatim}
Hence all invertible maps going up universe levels are half adjoint equivalences:

\( J \uparrow \)-invertible satisfies the property for all

rather than

\( A \times (\mathord{\text{id}}) \).

And we have a similar development with a similar example:

\[
\begin{align*}
J \uparrow \equiv & : \text{is-univalent } (U \sqcup V) \\
& \quad \to (A : (X : U \to \top) \to (Y : U \sqcup V) \to X \equiv Y \to V \equiv) \\
& \quad \to (X : U \to \top) \to A \times (\text{Lift } U X) (\equiv \text{- } \text{Lift } X) \\
& \quad \to (X : U \to \top) \to (Y : U \sqcup V) \mapsto (e : X \equiv Y) \to A \times Y \equiv \varepsilon
\end{align*}
\]

\( J \uparrow \equiv \) ua \( A \varphi X = H \uparrow \equiv \) ua \( A \times (A X) (\varphi X) \)

\[
\begin{align*}
H \uparrow \equiv & : \text{is-univalent } (U \sqcup V) \\
& \quad \to (X : U \to \top) \to (A : (Y : U \sqcup V) \to (X \equiv Y \to V \equiv)) \\
& \quad \to A \times (\text{Lift } U X) \mapsto (Y : U \sqcup V) \mapsto (f : X \equiv Y) \mapsto \text{is-equiv } f \to A \times Y \equiv \varepsilon
\end{align*}
\]

where

\[
\begin{align*}
B : (Y : U \sqcup V) \to X \equiv Y \to V \equiv
\end{align*}
\]

\[
\begin{align*}
b & \equiv a \\
y & : (e : X \equiv Y) \to B \equiv e \\
y & = H \uparrow \equiv \) ua \( X \times B \equiv y
\end{align*}
\]

\[
\begin{align*}
J \uparrow \equiv & : \text{is-univalent } (U \sqcup V) \\
& \quad \to (A : (X : U \to \top) \to (Y : U \sqcup V) \to (X \equiv Y \to V \equiv)) \\
& \quad \to (X : U \to \top) \to A \times (\text{Lift } U X) \mapsto \text{is-equiv } f \to A \times Y \equiv \varepsilon
\end{align*}
\]

\( J \uparrow \equiv \) ua \( A \varphi X = H \uparrow \equiv \) ua \( A \times (A X) (\varphi X) \)

All invertible functions from a type in a universe \( U \) to a type in a higher universe \( U \sqcup V \) satisfy a given property if (and only if) the functions

\[
\begin{align*}
\text{lift } \{U\} \\{V\} \{X\} : X \to \text{Lift } \{U\} \{V\} \{X\}
\end{align*}
\]

satisfy the property for all \( X : U \) (where we don’t write the implicit arguments for \( \text{lift} \)):

\[
\begin{align*}
J \uparrow \text{-invertible} & : \text{is-univalent } (U \sqcup V) \\
& \quad \to (A : (X : U \to \top) \to (Y : U \sqcup V) \to (X \equiv Y \to V \equiv)) \\
& \quad \to (X : U \to \top) \to A \times (\text{Lift } U X) \mapsto \text{is-equiv } f \to A \times Y \equiv \varepsilon
\end{align*}
\]

\( J \uparrow \text{-invertible} \) ua \( A \varphi X \equiv \) \( J \uparrow \equiv \) ua \( A \equiv X \equiv Y \equiv f \equiv i \equiv \) \( \text{invertibles-are-equis } f \equiv i \)

Here is an example. First, \( \text{lift} \) is a half adjoint equivalence on the nose:

\[
\begin{align*}
\text{lift-is-hae} & : (X : U \to \top) \to \text{is-hae } \{U\} \\{V\} \{X\} \\{\text{Lift } U X\} \\{\text{lift } \{U\} \\{V\}\} \\
\text{lift-is-hae } \{U\} \\{V\} \{X\} & = \text{lower} , \\
& \quad \text{lower-lift } \{U\} \\{V\} , \\
& \quad \text{lift-lower} , \\
& \quad \lambda x \to \text{refl } \\text{refl } (\text{lift } X)
\end{align*}
\]

Hence all invertible maps going up universe levels are half adjoint equivalences:
We have a dual development with the universes going down, where we consider lower in place of lift:

\[ \text{equivs-are-hae} : \text{is-univalent} (\mathcal{U} \uplus \mathcal{V}) \]
\[ \rightarrow \{ X : \mathcal{U} \} \{ Y : \mathcal{U} \uplus \mathcal{V} \} \text{ (f : X \rightarrow Y) } \]
\[ \rightarrow \text{is-equiv f} \rightarrow \text{is-hae f} \]

equivs-are-hae: \{\mathcal{U}\} \{\mathcal{V}\} \text{ ua } \{X\} \{Y\} = J↓-equiv \{\mathcal{U}\} \{\mathcal{V}\} \text{ ua } (\lambda \text{ X Y f } \rightarrow \text{is-hae f})

\quad \text{lift-is-hae X Y}

All invertible functions from a type in a universe \( \mathcal{U} \uplus \mathcal{V} \) to a type in the lower universe \( \mathcal{U} \) satisfy a given property if (and only if) the functions

\[ \text{lower } \{\mathcal{U}\} \{\mathcal{V}\} \{Y\} : \text{Lift } \mathcal{V} \ Y \rightarrow Y \]

satisfy the property for all \( Y : \mathcal{U} \).

\[ \text{J↓-invertible} : \text{is-univalent} (\mathcal{U} \uplus \mathcal{V}) \]
\[ \rightarrow (A : (X : \mathcal{U} \uplus \mathcal{V} \uplus \mathcal{V} \uplus \mathcal{V})) \text{ (Y : } \mathcal{U} \uplus \mathcal{V} \text{) (f : X \rightarrow Y) } \rightarrow \text{is-equiv f} \rightarrow \text{is-hae f} \]
However, the above notion of global property is very restrictive. For example, $\equiv$ global-cumulative is preserved by the embedding $\equiv X Y\ f\rightarrow A X Y f$.

We may hope to have that if $h$ level $n$ holds then so does $\equiv X Y\ f\rightarrow A X Y f$.

The remainder of this section is not used anywhere else.

A crucial example of an equivalence “going down one universe” is $\text{Id}\rightarrow\text{Eq}$ $X Y$. This is because the identity type $X\equiv Y$ lives in the successor universe $\cal U'\cal U'$ if $X$ and $Y$ live in $U$, whereas the equivalence type $X\equiv Y$ lives in the same universe as $X$ and $Y$. Hence we can apply the above function invertibles-are-haes↓ to $\text{Id}\rightarrow\text{Eq}$ $X Y$ to conclude that it is a half adjoint equivalence.

We can be parsimonious with the uses of univalence by instead using invertibles-are-haes↓, which doesn’t require univalence. However, that $\text{Id}\rightarrow\text{Eq}$ is invertible of course requires univalence.

The remainder of this section is not used anywhere else. Using the universe $\cal U\omega$ discussed above, we can consider global properties:

We may hope to have that if $A$ is a global property of types, then, in the presence of univalence, for any $X : \cal U$ and $Y : \cal V$, if $A X$ holds then so does $A Y$. However, because we have a type of universes, or universe levels, we may define e.g. $A \{\cal U\} X = (\cal U \equiv \cal U\omega)$, which violates this hope. To get this conclusion, we need an assumption on $A$. We say that $A$ is cumulative if it is preserved by the embedding $\text{Lift}$ of universes into higher universes.

We can prove the following:

However, the above notion of global property is very restrictive. For example, $\text{is-inhabited}$ defined below is a global property of type $\{\cal U\} X \equiv \cal U\omega \rightarrow \cal U\omega\rightarrow \cal U\omega$. Hence we prove something more general, where in this example we take $\text{F} \equiv \cal U\omega \rightarrow \cal U\omega$.
The following generalizes the slice constructor 
\[
\text{\textunderscore}/
\]\:

\[
\text{\textunderscore}/\text{\textunderscore} : (U : \text{Universe}) \rightarrow (U \to U \rightarrow V) \rightarrow (U \to U \to U \to V)
\]
\[
U /\text{\textunderscore} P ] Y = \Sigma X : U \to U, X \to Y, (f : (X \to Y), \phi : (Y \to P \text{fiber } f y))
\]

\[
\chi \text{-special} : (P : U \to V) \rightarrow (Y : U) \rightarrow U /\text{\textunderscore} P ] Y \rightarrow (Y \to \Sigma P)
\]

\[
\text{is-special-map-classifier} : (U \to V) \rightarrow U \to U \to V
\]

\[
\text{is-special-map-classifier} \{U\} P = (Y : U) \rightarrow \text{is-equiv} (\chi \text{-special } P Y)
\]

If a universe is a map classifier then \(\Sigma P\) is the classifier of maps with \(P\)-fibers, for any \(P : U \rightarrow V\):

\[
\text{mc-gives-sc} : \text{is-map-classifier } U
\]
\[
\rightarrow (P : U \to V) \rightarrow \text{is-special-map-classifier } U P
\]

\[
\text{mc-gives-sc } \{U\} s P Y = y
\]

where

\[
e = (U /\text{\textunderscore} P ] Y) = (\approx \text{-sym } a )
\]
With this we can derive a fact we already know, as follows. First the type of singletons (in a universe) is itself a singleton (in the next universe):

\[
\text{the-singletons-form-a-singleton} : \text{propext } \mathcal{U} \to \text{dfunext } \mathcal{U} \ \text{is-singleton } (S \ \mathcal{U})
\]

\[
\text{the-singletons-form-a-singleton } \{\mathcal{U}\} \text{ pe fe } = \ i , \ \varphi
\]

\[
\text{where}
\]

\[
i : \text{is-singleton } (\text{Lift } \mathcal{U} \ \mathbb{1})
\]

\[
i = \text{equiv-to-singleton } (\text{Lift=} \mathbb{1}) \ \mathbb{1}\text{-is-singleton}
\]

\[
c : S \ \mathcal{U}
\]

\[
c = \text{Lift } \mathcal{U} \ \mathbb{1} , \ i
\]

\[
\varphi : (x : S \ \mathcal{U}) \to c \equiv x
\]
The magma homomorphism and isomorphism conditions are subsingleton types by virtue of the fact that the underlying type of a magma is a set by definition.

```
being-magma-hom-is-subsingleton : \( M \ N : \text{Magma } \mathcal{U} \) \( f : ( M ) \to ( N ) \) 
\mapsto \text{is-subsingleton } (\text{is-magma-hom } M \ N \ f)
```

```
being-magma-hom-is-subsingleton \( M \ N \ f \) =
\( \Pi \text{-is-subsingleton } \text{dfe} \)
\( \lambda x \to \text{Pi-is-subsingleton } \text{dfe} \)
\( \lambda y \to \text{magmas-is-set } \text{dfe} \)
\( \lambda \to ( f x \cdot ( M ) y ) ) ( f x \cdot ( N ) f y ) ) )
```

```
being-magma-iso-is-subsingleton : \( M \ N : \text{Magma } \mathcal{U} \) \( f : ( M ) \to ( N ) \) 
\mapsto \text{is-subsingleton } (\text{is-magma-iso } M \ N \ f)
```

```
being-magma-iso-is-subsingleton \( M \ N \ f \) ( h , g , k , \eta , \varepsilon ) ( h' , g' , k' , \eta' , \varepsilon' ) = y
where
p : h \equiv h' 
p = being-magma-hom-is-subsingleton \( M \ N \ f \) h h'
q : g \equiv g'
q = \text{dfe} ( \lambda y \to g y ) \equiv ( \text{ap } g ( \varepsilon' y ) ) \equiv ( \text{ap } \eta ( g' y ) )
```

**Exercise.** (1) Show that the retractions into \( Y \) are classified by the type \( \Sigma A \ni \mathcal{U} \) \( \to \text{is-singleton } (\Sigma X \ni \mathcal{U} , X \equiv Y ) \)

univalence→-again {\cal U} ua \( Y = \text{equiv-to-singleton } (\text{equiv-classification } ua \ Y) i \)

where
\( i : \text{is-singleton } (Y \to \text{蚓 } \mathcal{U}) \)
\( i = \text{univalence-gives-vvfunext'} (\text{ua } \mathcal{U}) (u a (u \times)) \)
\( (\lambda y \to \text{the-singles form a singleton}
 (\text{univalence-gives-propext } (ua \mathcal{U})))
 (\text{univalence-gives-dfunext } (ua \mathcal{U})))
```

**Magma equivalences**

We now define magma equivalences and show that the type of magma equivalences is identified with the type of magma isomorphisms. In the next section, which proves a *structure identity principles*, we apply this to characterize magma equality and equality of other mathematical structures in terms of equivalences of underlying types.

For simplicity we assume global univalence here.

```
module magma-equivalences (ua : Univalence) where
open magmas

dfe : global-dfunext
dfe = univalence-gives-vvfunext' (ua \mathcal{U}) (ua (u \times))

hfe : global-hfunext
hfe = univalence-gives-global-hfunext ua
```

The magma homomorphism and isomorphism conditions are subsingleton types by virtue of the fact that the underlying type of a magma is a set by definition.

```
being-magma-hom-is-subsingleton : \( M \ N : \text{Magma } \mathcal{U} \) \( f : ( M ) \to ( N ) \) 
\mapsto \text{is-subsingleton } (\text{is-magma-hom } M \ N \ f)
```

```
being-magma-hom-is-subsingleton \( M \ N \ f \) =
\( \Pi \text{-is-subsingleton } \text{dfe} \)
\( \lambda x \to \text{Pi-is-subsingleton } \text{dfe} \)
\( \lambda y \to \text{magmas-is-set } \text{dfe} \)
\( \lambda \to ( f x \cdot ( M ) y ) ) ( f x \cdot ( N ) f y ) ) )
```

```
being-magma-iso-is-subsingleton : \( M \ N : \text{Magma } \mathcal{U} \) \( f : ( M ) \to ( N ) \) 
\mapsto \text{is-subsingleton } (\text{is-magma-iso } M \ N \ f)
```

```
being-magma-iso-is-subsingleton \( M \ N \ f \) ( h , g , k , \eta , \varepsilon ) ( h' , g' , k' , \eta' , \varepsilon' ) = y
where
p : h \equiv h' 
p = being-magma-hom-is-subsingleton \( M \ N \ f \) h h'
q : g \equiv g'
q = \text{dfe} ( \lambda y \to g y ) \equiv ( \text{ap } g ( \varepsilon' y ) ) \equiv ( \text{ap } \eta ( g' y ) )
```

**Exercise.** (2) After we have defined propositional truncation and surjections, show that the surjections into \( Y \) are classified by the type \( \Sigma A \ni \mathcal{U} \) \( \parallel A \) of inhabited types.
A function is a magma isomorphism if and only if it is a magma equivalence.

By a magma equivalence we mean an equivalence which is a magma homomorphism. This notion is again a subsingleton type.

\[
\text{is-magma-equiv} : (M N : \text{Magma } \mathcal{U}) \to (\{ M \} \to \{ N \}) \to \mathcal{U}'
\]

\[
\text{is-magma-equiv} M N f = \text{is-equiv} f \times \text{is-magma-hom} M N f
\]

\[
\text{being-magma-equiv-is-subsingleton} : (M N : \text{Magma } \mathcal{U}) (f : \{ M \} \to \{ N \}) \to \text{is-subsingleton} (\text{is-magma-equiv} M N f)
\]

\[
\text{being-magma-equiv-is-subsingleton} M N f =
\]

\[
\text{x-is-subsingleton}
\]

\[
(\text{being-equiv-is-subsingleton} dfe dfe f)
\]

\[
(\text{being-magma-hom-is-subsingleton} M N f)
\]

A function is a magma isomorphism if and only if it is a magma equivalence.

\[
\text{magma-isos-are-magma-equivs} : (M N : \text{Magma } \mathcal{U}) (f : \{ M \} \to \{ N \}) \\
\to \text{is-magma-iso} M N f \\
\to \text{is-magma-equiv} M N f
\]

\[
\text{magma-isos-are-magma-equivs} M N f (h, g, k, \eta, \epsilon) = i, h
\]

where

\[
i : \text{is-equiv} f
\]

\[
i = \text{invertibles-are-equis} f (g, \eta, \epsilon)
\]

\[
\text{magma-eqvs-are-magma-isos} : (M N : \text{Magma } \mathcal{U}) (f : \{ M \} \to \{ N \}) \\
\to \text{is-magma-equiv} M N f \\
\to \text{is-magma-iso} M N f
\]

\[
\text{magma-eqvs-are-magma-isos} M N f (i, h) = h, g, k, \eta, \epsilon
\]

where

\[
g : \{ N \} \to \{ M \}
\]

\[
g = \text{inverse} f i
\]

\[
\eta : g \circ f \sim \text{id}
\]

\[
\eta = \text{inverses-are-retractions} f i
\]

\[
\epsilon : f \circ g \sim \text{id}
\]

\[
\epsilon = \text{inverses-are-sections} f i
\]

\[
k : (a b : \{ N \}) \to g (a \cdot (N) b) \equiv g a \cdot (M) g b
\]

\[
k a b = g (a \cdot (N) b) \equiv (\text{ap} \text{2} (\lambda a b \to g (a \cdot (N) b)) ((\epsilon \text{a})^{-1})
\]

\[
(g f (g a \cdot (M) g b)) \equiv (\eta (g a \cdot (M) g b))
\]

\[
g a \cdot (M) g b
\]

Because these two notions are subsingleton types, we conclude that they are equivalent.

\[
\text{magma-iso-charac} : (M N : \text{Magma } \mathcal{U}) (f : \{ M \} \to \{ N \}) \\
\to \text{is-magma-iso} M N f = \text{is-magma-equiv} M N f
\]

\[
\text{magma-iso-charac} M N f = \text{logically-equivalent-subsingletons-are-equivalent}
\]

\[
(\text{is-magma-iso} M N f) \\
(\text{is-magma-equiv} M N f) \\
(\text{being-magma-iso-is-subsingleton} M N f)
\]
And hence they are equal by univalence.

\[
magma-iso-charac' : (M N : Magma \mathcal{U}) (f : (M) \to (N)) \\ \rightarrow is-magma-iso M N f \equiv is-magma-equiv M N f
\]

\[
magma-iso-charac' M N f = Eq-Id (ua (universe-of (M))) \\ (is-magma-iso M N f) \\ (is-magma-equiv M N f) \\ (magma-iso-charac M N f)
\]

And by function extensionality the properties of being a magma isomorphism and a magma equivalence are the same:

\[
magma-iso-charac'' : (M N : Magma \mathcal{U}) \\ \rightarrow is-magma-iso M N \equiv is-magma-equiv M N
\]

\[
magma-iso-charac'' M N = dfe (magma-iso-charac' M N)
\]

Hence the type of magma equivalences is equivalent, and therefore equal, to the type of magma isomorphisms.

\[
\equiv_{\sim} : Magma \mathcal{U} \rightarrow Magma \mathcal{U} \rightarrow \mathcal{U}' \\ M \equiv_{\sim} N = \Sigma f : ((M) \to (N)), is-magma-equiv M N f
\]

\[
\equiv_{\sim}-charac : (M N : Magma \mathcal{U}) \\ \rightarrow (M \equiv_{\sim} N) \Rightarrow (M \equiv_{\sim} N)
\]

\[
\equiv_{\sim}-charac M N = \Sigma-cong (magma-iso-charac M N)
\]

\[
\equiv_{\sim}-charac' : (M N : Magma \mathcal{U}) \\ \rightarrow (M \equiv_{\sim} N) \equiv (M \equiv_{\sim} N)
\]

\[
\equiv_{\sim}-charac' M N = ap \Sigma (magma-iso-charac'' M N)
\]

It follows from the results of this and the next section that magma equality amounts to magma isomorphism.

Table of contents

### Equality of mathematical structures

Independently of any choice of foundation, we regard two groups to be the same, for all mathematical purposes, if they are isomorphic. Likewise, we consider two topological spaces to be the same if they are homeomorphic, two metric spaces to be the same if they are isometric, two categories to be the same if they are equivalent, and so on.

With Voevodsky’s univalence axiom, we can prove that these notions of sameness are automatically captured by Martin-Löf’s identity type. In particular, properties of groups are automatically invariant under isomorphism, properties of topological spaces are automatically invariant under homeomorphism, properties of metric spaces are automatically invariant under isometry, properties of categories are automatically invariant under equivalence, and so on, simply because, by design, properties are invariant under the notion of equality given by the identity type. In other foundations, the lack of such automatic invariance creates practical difficulties.

A structure identity principle describes the identity type of types of mathematical structures in terms of equivalences of underlying types, relying on univalence. The first published structure identity principle, for a large class of algebraic structures, is [Coquand and Danielsson]. The HoTT book (section 9.8) has a categorical version, whose formulation is attributed to Peter Aczel.

Here we formulate and prove a variation for types equipped with structure. We consider several versions:

1. One for raw structures subject to no axioms, such as ∞-magmas and pointed types.
2. One that adds axioms to a structure, so as to e.g. get an automatic characterization of magma identifications from a characterization of $\infty$-magma identifications.

3. One that joins two kinds of structure, so as to e.g. get an automatic characterization of identifications of pointed $\infty$-magmas from characterizations of identifications for pointed types and for $\infty$-magmas.

4. In particular, adding axioms to pointed $\infty$-magmas we get monoids with an automatic characterization of their identifications.

5. And then adding an axiom to monoids we get groups, again with an automatic characterization of their identifications.

6. We also show that while two groups are equal precisely when they are isomorphic, two subgroups of a group are equal precisely when they have the same elements, if we define a subgroup to be a subset closed under the group operations.

We also apply these ideas to characterize identifications of metric spaces, topological spaces, graphs, partially ordered sets, categories and more.

Table of contents

A structure identity principle for a standard notion of structure

We consider mathematical structures specified by a function

$$S : \mathcal{U} \to \mathcal{V}$$

and we consider types $X : \mathcal{U}$ equipped with such structure $s : S X$, collected in the type

$$\Sigma X : \mathcal{U}, \ S X,$$

which, as we have seen, can be abbreviated as

$$\Sigma S.$$

For example, for the type of $\infty$-magmas we will take $\mathcal{V} = \mathcal{U}$ and

$$S X = X \to X \to X.$$

Our objective is to describe the identity type $\text{Id} (\Sigma S) A B$, in favourable circumstances, in terms of equivalences of the underlying types $\langle A \rangle$ and $\langle B \rangle$ of $A B : \Sigma S$.

$$\langle _\_ \rangle : \{ S : \mathcal{U} \to \mathcal{V} \} \to \Sigma S \to \mathcal{U}$$

$$\langle X, s \rangle = X$$

structure : \{ S : \mathcal{U} \to \mathcal{V} \} (A : \Sigma S) \to S \{ A \}

structure (X, s) = s

Our favourable circumstances will be given by data

$$\iota : (A B : \Sigma S) \to \langle A \rangle \simeq \langle B \rangle \to \mathcal{V},$$

$$\rho : (A : \Sigma S) \to \iota A A (\text{id-}\simeq \langle A \rangle).$$

The idea is that

- $\iota$ describes favourable equivalences, which will be called homomorphisms, and
- $\rho$ then stipulates that all identity equivalences are homomorphisms.

We require that any two structures on the same type making the identity equivalence a homomorphism must be identified in a canonical way:
The canonical map
\[ s \equiv t \mapsto (X, s) (X, t) (\text{id-}\simeq X) \]
defined by induction on identifications by
\[ \text{refl } s \mapsto \rho (X, s) \]
must be an equivalence for all \( X : \mathcal{U} \) and \( s, t : S X \).

This may sound a bit abstract at this point, but in practical examples of interest it is easy to fulfill these requirements, as we will illustrate soon.

We first define the canonical map:
\[
\text{canonical-map} : \{ S : \mathcal{U} \to \mathcal{V} \} \to \{ A : \Sigma S \to \{ A \} \simeq \{ B \} \to \mathcal{W} \}
\]
\[
(\text{canonical-map} \ s t) = \rho (X, s) \]

We refer to such favourable data as a standard notion of structure and collect them in the type
\[
\text{SNS} S \ \mathcal{W}
\]
as follows:
\[
\text{SNS} : (\mathcal{U} \to \mathcal{V}) \to (\mathcal{W} : \text{Universe}) \to \mathcal{U} \cup \mathcal{V} \cup (\mathcal{W} \to \mathcal{V})
\]
\[
\text{SNS} \{ \mathcal{U} \} \{ \mathcal{V} \} S \ \mathcal{W} = \Sigma t : ((A B : \Sigma S) \to \{ A \} \simeq \{ B \} \to \mathcal{W})
\]
\[
, \Sigma \rho : ((A : \Sigma S) \to \text{is-equiv } \{ A \})(id-\simeq \{ A \})
\]
\[
, \text{is-equiv } (\text{canonical-map} \ s t) \]

We write \( \text{homomorphic} \) for the first projection (we don’t need names for the other two projections):
\[
\text{homomorphic} : \{ S : \mathcal{U} \to \mathcal{V} \} \to \text{SNS} S \ \mathcal{W}
\]
\[
\text{homomorphic} (\text{canonical-map} \ s t) = \rho (X, s)
\]

For example, when \( S \) specifies \( \infty \)-magma structure, we will have that \( \text{homomorphic } \sigma \ A \ B \ (f, i) \) amounts to \( f \) being a magma homomorphism.

We then collect the homomorphic equivalences of \( A \ B : \Sigma S \), assuming that \( S \) is a standard notion of structure witnessed by \( \sigma \), in a type
\[
A \simeq [ \sigma ] B.
\]

Notice that only the first component of \( \sigma \), namely \( \text{homomorphic } \sigma \), is used in the definition. The other two components are used to prove properties of \( A \simeq [ \sigma ] B \).

\[
\text{Id-homEq} : \{ S : \mathcal{U} \to \mathcal{V} \} (\sigma : \text{SNS} S \ \mathcal{W}) \to (A B : \Sigma S) \to (A \equiv B) \to (A \simeq [ \sigma ] B)
\]
\[
\text{Id-homEq} (_, \rho, _) A \ A \ (\text{refl } A) = \text{id}, \text{id-is-equiv } \{ A \}, \rho A
\]
With this we are ready to prove the promised characterization of identity on $\Sigma S$:

$$
\text{homomorphism-lemma} : \{ S : U \to V \} \ (\sigma : \text{SNS} \ S W) \\
(A B : \Sigma S) \ (p : \{ A \} \equiv \{ B \}) \\
\to \ (\text{transport} \ S \ p \ \text{structure} \ A \equiv \text{structure} \ B) \equiv \ \text{homomorphic} \ \sigma \ A \ B \ (\text{Id-Eq} \ (A) \ (B) \ p) \\
\text{homomorphism-lemma} \ (I, \rho, \theta) \ (X, s) \ (X, t) \ (\text{refl} \ X) = \gamma
$$

where

$$
\gamma : (s \equiv t) \equiv I (X, s) (X, t) (\text{id} \equiv X) \\
\gamma = (\text{canonical-map} \ I \ p \ s \ t, \ \theta \ s \ t)
$$

characterization-of-$\equiv$ : $\text{is-univalent} \ U \\
\to \{ S : U \to V \} \ (\sigma : \text{SNS} \ S W) \\
\to \ (A B : \Sigma S) \\
\to \ (A \equiv B) = (A \equiv [\sigma] B)
$$

characterization-of-$\equiv \ ua \{S\} \ \sigma \ A \ B =$

$$
(A \equiv B) \equiv (\ i \ ) \\
(\Sigma p : \{ A \} \equiv \{ B \}, \ \text{transport} \ S \ p \ \text{structure} \ A \equiv \text{structure} \ B) \equiv (\ ii \ ) \\
(\Sigma p : \{ A \} \equiv \{ B \}, \ I \ A \ B \ (\text{Id-Eq} \ (A) \ (B) \ p)) \equiv (\ iii \ ) \\
(\Sigma e : \{ A \} \equiv \{ B \}, \ \iota \ A \ B \ e) \equiv (\ iv \ ) \\
(A \equiv [\sigma] B)
$$

where

$$
i = \text{homomorphic} \ \sigma \\
ii = \Sigma-cong \ \text{homomorphism-lemma} \ \sigma \ A \ B \\
iii = \equiv \text{-sym} \ \Sigma-change-of-variable \ (\iota \ A \ B) \ (\text{Id-Eq} \ (A) \ (B) \ p) \ (\text{ua} \ (A) \ (B)) \\
iv = \Sigma-\text{assoc}
$$

This equivalence is pointwise equal to $\text{Id-homEq}$, and hence $\text{Id-homEq}$ is itself an equivalence:

$$
\text{Id-homEq-is-equiv} : (\text{ua} : \text{is-univalent} \ U) \ \{ S : U \to V \} \ (\sigma : \text{SNS} \ S W) \\
\to \ (A B : \Sigma S) \to \text{is-equiv} \ (\text{Id-homEq} \ \sigma \ A \ B)
$$

Id-homEq-is-equiv \ ua \{S\} \ \sigma \ A \ B = \gamma

where

$$
h : (A B : \Sigma S) \to \text{Id-homEq} \ \sigma \ A \ B \sim \ \text{characterization-of-$\equiv \ ua \ \sigma \ A \ B$} \\
h A A \ (\text{refl} \ A) = \text{refl } _-
$$

$$
\gamma : \text{is-equiv} \ (\text{Id-homEq} \ \sigma \ A \ B) \\
\gamma = \text{equivs-closed-under-~} \\
\ (\text{"-is-equiv} \ \text{characterization-of-$\equiv \ ua \ \sigma \ A \ B$}) \\
\ (h A B)
$$

We conclude this submodule with the following characterization of the canonical map and of when it is an equivalence, applying Yoneda.

module _ {S : U \to V} \\
\ (\iota : (A B : \Sigma S) \to \{ A \} \equiv \{ B \} \to W) \\
\ (p : (A : \Sigma S) \to \iota \ A \ A \ (\text{id} \equiv \{ A \})) \\
\ \{X : U\}

where

$$
\text{canonical-map-charac} : (s t : S X) \ (p : s \equiv t) \\
\to \ \text{canonical-map} \ I \ p \ s \ t \ p \\
\equiv \ \text{transport} \ (\lambda - \to \iota \ (X, s) \ (X, -) (\text{id} \equiv X)) \ p \ (p \ (X, s))
$$

$$
\text{canonical-map-charac} \ s = \ \text{transport-lema} \ (\lambda t \to \iota \ (X, s) \ (X, t) (\text{id} \equiv X)) \ s \\
\ (\text{canonical-map} \ I \ p \ s)
$$
Another criterion is the following: It is enough to have any equivalence for the canonical map to be an equivalence:

\[
\text{canonical-map-equiv-criterion} : ((s : S X) \rightarrow \text{is-equiv } (\text{canonical-map } \triangleright \rho s t)) \\
\rightarrow ((s : S X) \rightarrow \exists t : S X, \triangleright (X, s) (X, t) (\text{id} \simeq X)) \\
\]

where

\[
A = \lambda s t \rightarrow \triangleright (X, s) (X, t) (\text{id} \simeq X) \\
\triangleright = \text{canonical-map } \triangleright \rho
\]

And in fact it is enough to have any retraction for the canonical map to be an equivalence:

\[
\text{canonical-map-equiv-criterion'} : ((s : S X) \rightarrow \triangleright (X, s) (X, t) (\text{id} \simeq X)) \\
\rightarrow (s : S X) \rightarrow \text{is-equiv } (\text{canonical-map } \triangleright \rho s t)
\]

Another criterion is the following: It is enough to have any equivalence for the canonical map to be an equivalence:

\[
\text{canonical-map-equiv-criterion} : ((s : S X) \rightarrow (s \equiv t) \Rightarrow \triangleright (X, s) (X, t) (\text{id} \simeq X)) \\
\rightarrow (s : S X) \rightarrow \text{is-equiv } (\text{canonical-map } \triangleright \rho s t)
\]

And in fact it is enough to have any retraction for the canonical map to be an equivalence:

\[
\text{canonical-map-equiv-criterion'} : ((s : S X) \rightarrow \triangleright (X, s) (X, t) (\text{id} \simeq X)) \\
\rightarrow (s : S X) \rightarrow \text{is-equiv } (\text{canonical-map } \triangleright \rho s t)
\]

This concludes the module sip, and we now consider some examples of uses of this.

---

**∞-Magmas**

We now make precise the example outlined above:

```lean
module ∞-magma {U : Universe} where

open sip

∞-magma-structure : U → U
∞-magma-structure X = X → X → X

∞-Magma : U → ∞-Magma = Σ X : U, ∞-magma-structure X

sns-data : SNS ∞-magma-structure U
sns-data = (⊂, ρ, θ)
where

\[
\Subset : (A : ∞-Magma) \rightarrow (A \equiv {B} \rightarrow U) \\
\Subset (X, _·_) (Y, _·_) (f, _·_) = (\lambda x x' \rightarrow f (x \cdot x')) \equiv (\lambda x x' \rightarrow f x \cdot f x')
\]

\[
\rho : (A : ∞-Magma) \rightarrow \Subset A A (\text{id} \simeq {A})
\]

\[
\rho (X, _·_) = \text{refl } _·_
\]

\[
h : {X : U} \rightarrow {\_·_{\_·_{\_·}} : \text{∞-magma-structure } X} \\
\rightarrow \text{canonical-map } \triangleright \rho _·_{\_·_{\_·}} \simeq {id} (\_·_{\_·_{\_·}} \equiv _·_{\_·_{\_·}})
\]

\[
h (\text{refl } _·_) = \text{refl } (\text{refl } _·_)
\]

\[
\theta : {X : U} \rightarrow {\_·_{\_·_{\_·}} : \text{∞-magma-structure } X} \\
\rightarrow \text{is-equiv } (\text{canonical-map } \triangleright \rho _·_{\_·_{\_·}})
\]

\[
\theta _·_{\_·_{\_·}} = \text{equivs-closed-under-} \leadsto (\text{id} \simeq \text{equiv } (_·_{\_·_{\_·}})) h
\]

\[
\equiv : \text{∞-Magma } \rightarrow \text{∞-Magma } \rightarrow U
\]

(X, _·_) \equiv (Y, _·_) =
```
\[ \Sigma \mathcal{f} : (X \to Y), \text{is-equiv} \mathcal{f} \times ((\lambda x \ x' \to f (x \cdot x')) \equiv (\lambda x \ x' \to f x \ast f x')) \]

The above equivalence is characterized by induction on identifications as the function that maps the reflexive identification to the identity equivalence:

\[
\text{characterization-of-\(\infty\)-Magma-\(\equiv\) : is-univalent } \mathcal{U} \to (A B : \infty\text{-Magma}) \to (A \equiv B) = (A \equiv B)
\]

\[
\text{characterization-of-\(\infty\)-Magma-\(\equiv\) \text{ua} = characterization-of-\(\equiv\) \text{ua sns-data}
\]

Next we want to account for situations in which axioms are considered, for example that the underlying type is a set, or that the monoid structure satisfies the unit and associativity laws. We do this in a submodule, by reduction to the characterization of identifications given in the module sip.

module sip-with-axioms where
open sip

The first construction, given \(S\) as above, and given subsingleton-valued axioms for types equipped with structure specified by \(S\), builds SNS data on \(S'\) defined by

\[ S' X = \Sigma s : S X, \text{axioms } X s \]

from given SNS data on \(S\).

For that purpose we first define a forgetful map \(\Sigma S' \to \Sigma S\) and an underlying-type function \(\Sigma S \to \mathcal{U}\):

\[
[_] : \{S : U \to V\} \{\text{axioms} : (X : U \to V) \to S X \to \mathcal{W}\} \to (\Sigma X : U \to V, \Sigma \ s : S X, \text{axioms } X s) \to \Sigma S
\]

\[
[X, s, _] = (X, s)
\]

\[
\llangle_\llangle : \{S : U \to V\} \{\text{axioms} : (X : U \to V) \to S X \to \mathcal{W}\} \to (\Sigma X : U \to V, \Sigma \ s : S X, \text{axioms } X s) \to \mathcal{U}
\]

\[
\llangle X, _, _, _\rrangle = X
\]

In the following construction:

- For \(\iota'\) and \(\rho'\) we use \(\iota\) and \(\rho\) ignoring the axioms.
- For \(\theta'\) we need more work, but the essence of the construction is the fact that the projection

\[ S' X \to S X \]

that forgets the axioms is an embedding precisely because the axioms are subsingleton-valued.

add-axioms : \{S : U \to V\}
\(\text{axioms} : (X : U \to V) \to S X \to \mathcal{W}\)
\(\to ((X : U \to V) (s : S X) \to \text{is-subsingleton} (axioms X s))\)
As discussed above, we get magmas from $\infty$-magmas by adding an axiom:

$$Magma$$

And this concludes the module. We now consider some examples.

**characterization-of-$≡$-with-axioms**:

- **is-univalent $\mathcal{U}$**
  - $\{S : \mathcal{U} → \mathcal{V} \}$
  - $(σ : SNS S T) (axioms : (X : \mathcal{U}’) → S X → \mathcal{U}’)$
  - $→ ((X : \mathcal{U}’) (s : S X) → is-subsingleton (axioms X s))$
  - $→ (A B : Σ X : \mathcal{U}’, Σ s : S X, axioms X s)
  - $→ (A ≡ B) = ([ A ] = [ σ ] [ B ])$

And this concludes the module sip-with-axioms. We now consider some examples.

**Magmas**

As discussed above, we get magmas from $\infty$-magmas by adding an axiom:

```
module magma {U : Universe} where

open sip-with-axioms

Magma : U’
Magma = Σ X : U’, (X → X → X) × is-set X
```
\( \_ \equiv : \text{Magma} \rightarrow \text{Magma} \rightarrow \mathcal{U} \)  
\[
(X \ , \ _* \ , \ _) \equiv (Y \ , \ _* \ , \ _) = \\
\Sigma \ f : (X \to Y), \text{is-equiv} \ f \times ((\lambda \ x \ x' \to f \ (x \cdot x')) \equiv (\lambda \ x \ x' \to f \ x \cdot f \ x'))
\]

characterization-of-Magma-≡ : is-univalent \( \mathcal{U} \to (A \ B : \text{Magma}) \to (A \equiv B) \equiv (A \equiv B) \)

characterization-of-Magma-≡ ua = 
- characterization-of-≡-with-axioms ua
- \( \Rightarrow \text{magma.sns-data} \)
- \( \ (\lambda \ x \ s \to \text{is-set} \ X) \)
- \( \ (\lambda \ x \ s \to \text{being-set-is-subsingleton} \ (\text{univalence-gives-dfunext} \ ua)) \)

Exercise. The above equivalence is characterized by induction on identifications as the function that maps the reflexive identification to the identity equivalence.

Exercise. Characterize identifications of monoids along the above lines. It is convenient to redefine the type of monoids to an equivalent type in the above format of structure with axioms. The following development solves this exercise.

Pointed types

We now discuss equality of pointed types, where a pointed type is a type equipped with a designated point.

module pointed-type \{\( \mathcal{U} : \text{Universe} \)\} where

open sip

Pointed : \( \mathcal{U} \rightarrow \mathcal{U} \)
Pointed \( X = X \)

sns-data : SNS Pointed \( \mathcal{U} \)

sns-data = (ι , ρ , θ)

where

ι : (A \ B : Σ Pointed) → \{A\} \equiv \{B\} → \mathcal{U} \ 
ι (X \ , \ x_0) (Y \ , \ y_0) (f \ , \ _) = (f \ x_0 \equiv y_0) 

ρ : (A : Σ Pointed) → ι A A (id-≡ \{A\})
ρ (X \ , \ x_0) = refl \ x_0 

θ : \{X : \mathcal{U}\} \ (x_0 \ x_1 : \text{Pointed} X) → \text{is-equiv} \ (\text{canonical-map} \ ι \ ρ \ x_0 \ x_1) 
θ x_0 x_1 = \text{equivs-closed-under-=} \ (\text{id-is-equiv} (x_0 \equiv x_1)) \ h 

where

h : \text{canonical-map} \ ι \ ρ \ x_0 \ x_1 \sim id \ (x_0 \equiv x_1) 

h (refl \ x_0) = refl (refl \ x_0) 

≡_ : Σ Pointed → Σ Pointed → \mathcal{U} \ 
\( X \ , \ x_0 \) ≡ \( Y \ , \ y_0 \) = Σ \( f : (X \rightarrow Y) \), \text{is-equiv} \ f \times (f \ x_0 \equiv y_0) 

characterization-of-pointed-type≡ : is-univalent \( \mathcal{U} \)  
\( \rightarrow (A \ B : \Sigma \text{Pointed}) \rightarrow (A \equiv B) \equiv (A \equiv B) \)

characterization-of-pointed-type≡ ua = characterization-of-≡ ua sns-data

Exercise. The above equivalence is characterized by induction on identifications as the function that maps the reflexive identification to the identity equivalence.

Combining two mathematical structures
We consider two given mathematical structures specified by their characterizations of the equalities of the structures. For example, we build the characterization of identifications of pointed ∞-magmas from the characterizations of the identifications of pointed types and the characterization of the identifications of magmas. Moreover, adding axioms, we get a characterization of identifications of monoids which amounts to the characterization of identifications of pointed ∞-magmas. Further adding an axiom, we get an automatic characterization of group identifications.

module sip-join where

We begin with the following technical lemma, whose proof uses the Yoneda machinery:

\[
\begin{align*}
\text{technical-lemma} & : \{X : \mathcal{U} \} \{A : X \rightarrow Y \rightarrow \mathcal{V} \} \\
& \{Y : \mathcal{V} \} \{B : Y \rightarrow Y \rightarrow \mathcal{T} \} \\
& (f : (x_0, x_1 : X) \rightarrow x_0 \equiv x_1 \rightarrow A x_0 x_1) \\
& (g : (y_0, y_1 : Y) \rightarrow y_0 \equiv y_1 \rightarrow B y_0 y_1) \\
& \rightarrow ((x_0, x_1 : X) \rightarrow \text{is-equiv} \ (f \ x_0 \ x_1)) \\
& \rightarrow ((y_0, y_1 : Y) \rightarrow \text{is-equiv} \ (g \ y_0 \ y_1)) \\
& \rightarrow ((x_0, y_0) \ (x_1, y_1) : X \times Y) \rightarrow \text{is-equiv} \ \left(\lambda \ (p : (x_0, y_0) \equiv (x_1, y_1)) \rightarrow f \ x_0 \ x_1 \ (\text{ap} \ \text{pr}_1 \ p), \ g \ y_0 \ y_1 \ (\text{ap} \ \text{pr}_1 \ p)\right)
\end{align*}
\]

\[
\text{technical-lemma} \ \{U \} \ \{V \} \ \{W \} \ \{T \} \ \{X \} \ \{A \} \ \{Y \} \ \{B \} \ f \ g \ i \ j \ (x_0, y_0) = y
\]

where

\[
\begin{align*}
u : & \exists! \ (x_1 : X, A \ x_0 x_1) \\
u & = \text{fiberwise-equiv-universal} \ (A \ x_0) \ \ x_0 \ (f \ x_0) \ (i \ x_0)
\end{align*}
\]

\[
\begin{align*}
v : & \exists! \ (y_1 : Y, B \ y_0 y_1) \\
v & = \text{fiberwise-equiv-universal} \ (B \ y_0) \ \ y_0 \ (g \ y_0) \ (j \ y_0)
\end{align*}
\]

\[
\begin{align*}
C : & X \times Y \rightarrow \mathcal{V} \sqcup \mathcal{T} \\
C \ (x_1, y_1) & = A x_0 x_1 \times B y_0 y_1
\end{align*}
\]

\[
\begin{align*}
w : & \exists! \ (x_1 : X, A \ x_0 x_1) \\
& \rightarrow \exists! \ (y_1 : Y, B \ y_0 y_1) \\
& \rightarrow \exists! \ (x_1, y_1) : X \times Y, C \ (x_1, y_1)
\end{align*}
\]

\[
\begin{align*}
w ((x_1, a_1), \varphi) ((y_1, b_1), \psi) & = ((x_1, y_1), (a_1, b_1)), \delta
\end{align*}
\]

where

\[
\begin{align*}
p : & \forall x y a b \\
p & \rightarrow (x_1, a_1) \equiv (x, a) \\
p & \rightarrow (y_1, b_1) \equiv (y, b) \\
p & \rightarrow (x_1, y_1), (a_1, b_1) \equiv (x, y), (a, b)
\end{align*}
\]

\[
\begin{align*}
p x_1 y_1 a_1 b_1 (\text{refl} \ (x_1, a_1)) (\text{refl} \ (y_1, b_1)) & = \text{refl} \ ((x_1, y_1), (a_1, b_1))
\end{align*}
\]

\[
\begin{align*}
delta : & ((x, y), (a, b)) : \Sigma \ C \\
delta & = (p x y a b (\varphi (x, a)) (\psi (y, b)))
\end{align*}
\]

\[
\begin{align*}
\tau : & \text{Nat} \ (\mathcal{V} \ (x_0, y_0)) \ C \\
\tau \ (x_1, y_1) & = f \ x_0 \ x_1 \ (\text{ap} \ \text{pr}_1 \ p), \ g \ y_0 \ y_1 \ (\text{ap} \ \text{pr}_2 \ p)
\end{align*}
\]

\[
\begin{align*}
\gamma : & \text{is-fiberwise-equiv} \ \tau \\
\gamma & = \text{universal-fiberwise-equiv} \ \ C \ (w \ u) \ (x_0, y_0) \ \tau
\end{align*}
\]

We consider two given mathematical structures specified by $S_0$ and $S_1$, and work with structures specified by their combination $\lambda \ X \rightarrow S_0 \ X \times S_1 \ X$.

variable

\[
\mathcal{V}_0 \ \mathcal{V}_1 \ \mathcal{W}_0 \ \mathcal{W}_1 : \text{Universe}
\]

open sip

\[
\{ X : \mathcal{U} \rightarrow \mathcal{V}_0 \} \ {S_1 : \mathcal{U} \rightarrow \mathcal{V}_1} \}
\rightarrow (\Sigma X : \mathcal{U}, S_0 X \times S_1 X) \rightarrow \mathcal{U}
\]

\[
\{ X, S_0, S_1 \} = X
\]

\[
[_-] : \ {S_0 : \mathcal{U} \rightarrow \mathcal{V}_0 \} \ {S_1 : \mathcal{U} \rightarrow \mathcal{V}_1} \}
\]
The following is then immediate from the join construction and the general structure identity principle:

\[ [X, s_0, s_1]_0 = (X, s_0) \]

\[ \vdash \{ S_0 : u \to v \} \{ S_1 : u \to v \} \]

\[ \to (\Sigma X : u \times s_1 X) \to S S_0 \]

\[ [X, s_0, s_1]_1 = (X, s_1) \]

The main construction in this submodule is this:

\[ \text{join} : \{ S_0 : u \to v \} \{ S_1 : u \to v \} \]

\[ \to \text{SNS} S_0 \mathcal{W}_0 \]

\[ \to \text{SNS} S_1 \mathcal{W}_1 \]

\[ \to \text{SNS} (\lambda X : S_0 X \times s_1 X) (W_0 \cup W_1) \]

\[ \text{join} \{ U \} \{ V_0 \} \{ V_1 \} \{ W_0 \} \{ W_1 \} \{ S_0 \} \{ S_1 \} (\lambda_0, \rho_0, \theta_0) (\lambda_1, \rho_1, \theta_1) = \lambda, \rho, \theta \]

where

\[ S : u \to v \cup v' \]

\[ S X = S_0 X \times s_1 X \]

\[ \lambda : (A B : \Sigma S) \to \{ A \to B \} \to W_0 \cup W_1 \]

\[ \lambda A B e = \lambda_0 [A]_0 [B]_0 e \times \lambda_1 [A]_1 [B]_1 e \]

\[ \rho : (A : \Sigma S) \to \lambda A A (id-\equiv (A)) \]

\[ \rho A = (\rho_0 [A]_0, \rho_1 [A]_1) \]

\[ \theta : \{ X : u \} (s t : S X) \to \text{is-equiv} \ (\text{canonical-map} \ \lambda \rho s) \]

\[ \theta \{ X \} \{ S_0, s_1 \} (t_0, t_1) = \gamma \]

where

\[ c : (p : S_0, s_1 : t_0, t_1) \to \lambda_0 (X, s_0) (X, t_0) (id-\equiv X) \times \lambda_1 (X, s_1) (X, t_1) (id-\equiv X) \]

\[ c p = (\text{canonical-map} \ \lambda_0 \rho_0 s_0 t_0 (ap \ pr_1, p), \text{canonical-map} \ \lambda_1 \rho_1 s_1 t_1 (ap \ pr_2, p)) \]

\[ i = \text{is-equiv} \ c \]

\[ i = \text{technical-lemma} \]

\[ (\text{canonical-map} \ \lambda_0 \rho_0) \]

\[ (\text{canonical-map} \ \lambda_1 \rho_1) \]

\[ \theta_0 \theta_1 (s_0, s_1) (t_0, t_1) \]

\[ e : \text{canonical-map} \ \lambda \rho (s_0, s_1) (t_0, t_1) \sim c \]

\[ e (\text{refl} (s_0, s_1)) = \text{refl} (\rho_0 (X, s_0), \rho_1 (X, s_1)) \]

\[ \gamma : \text{is-equiv} \ (\text{canonical-map} \ \lambda \rho (s_0, s_1) (t_0, t_1)) \]

\[ \gamma = \text{equivs-closed-under-\sim} \ i e \]

We then can characterize the identity type of structures in the join by the following relation:

\[ \vdash \}_{\ \equiv} [\_, \_] : \{ S_0 : u \to v \} \{ S_1 : u \to v \} \]

\[ \to (\Sigma X : u \times s_1 X) \]

\[ \to \text{SNS} S_0 \mathcal{W}_0 \]

\[ \to \text{SNS} S_1 \mathcal{W}_1 \]

\[ \to (\Sigma X : u \times s_1 X) \]

\[ \to u \cup \mathcal{W}_0 \cup \mathcal{W}_1 \]

\[ A \equiv [\sigma_0, \sigma_1] B = \Sigma f : (\{ A \} \to \{ B \}) \]

\[ \Sigma i : \text{is-equiv} f, \text{homomorphic} \sigma_0 [A]_0 [B]_0 (f, i) \times \text{homomorphic} \sigma_1 [A]_1 [B]_1 (f, i) \]

The following is then immediate from the join construction and the general structure identity principle:

\[ \text{characterization-of-join} = \text{is-univalent} u \]

\[ \to \{ S_0 : u \to v \} \{ S_1 : u \to v \} \]
This concludes the submodule. Some examples of uses of this follow.

Exercise. The above equivalence is characterized by induction on identifications as the function that maps the reflexive identification to the identity equivalence.
Monoid : \( \mathcal{U} \) 
\[ \text{Monoid} = \Sigma X : \mathcal{U}, \Sigma s : \text{monoid-structure} X, \text{monoid-axioms} X s \]

\[ \text{monoid-axioms-subsingleton} : (X : \mathcal{U}) (s : \text{monoid-structure} X) \rightarrow \text{is-subsingleton} (\text{monoid-axioms} X s) \]

\[ \text{monoid-axioms-subsingleton} X (\_\_ , e) = \text{subsingleton-criterion}' \gamma \]

where 
\[ \gamma : \text{monoid-axioms} X (\_\_ , e) \rightarrow \text{is-subsingleton} (\text{monoid-axioms} X (\_\_ , e)) \]
\[ \gamma (i , \_\_) = \times\text{-is-subsingleton} \]
\[ (\times\text{-is-subsingleton} dfe) \]
\[ (\times\text{-is-subsingleton} \lambda x \rightarrow i (e \cdot x) x) \]
\[ (\times\text{-is-subsingleton} \lambda x \rightarrow \pi-is-subsingleton dfe) \]
\[ (\pi-is-subsingleton dfe \lambda x \rightarrow i (x \cdot e) x) \]
\[ (\pi-is-subsingleton dfe \lambda x \rightarrow \pi-is-subsingleton dfe) \]
\[ \lambda y \rightarrow \pi-is-subsingleton dfe) \]
\[ (\lambda z \rightarrow i (((x \cdot y) \cdot z) (x \cdot (y \cdot z)))) \]

\[ \text{sns-data} : \text{SNS} (\lambda X \rightarrow \Sigma s : \text{monoid-structure} X, \text{monoid-axioms} X s) \mathcal{U} \]
\[ \text{sns-data} = \text{add-axioms} \]
\[ \begin{align*}
\text{monoid-axioms} & \text{ monoid-axioms-subsingleton} \\
& (\text{join} \\
& \begin{align*}
\text{magma.sns-data} \\
\text{pointed-type.sns-data}
\end{align*}
\end{align*}
\]

\[ \equiv_\_ : \text{Monoid} \rightarrow \text{Monoid} \rightarrow \mathcal{U} \]

\[ (X , (\_\_ , d) , \_\_ ) \equiv (Y , (\_\_ , e) , \_\_ ) = \]
\[ \Sigma f : (X \rightarrow Y), \text{is-equiv} f \\
\times ((\lambda x x' \rightarrow f (x \cdot x')) \equiv (\lambda x x' \rightarrow f x \cdot f x')) \\
\times (f d \equiv e) \]

\[ \text{characterization-of-monoid-}\equiv : (A B : \text{Monoid}) \rightarrow (A \equiv B) \equiv (A \equiv B) \]
\[ \text{characterization-of-monoid-}\equiv = \text{characterization-of-}\equiv \text{ ua sns-data} \]

Exercise. The above equivalence is characterized by induction on identifications as the function that maps the reflexive identification to the identity equivalence.

Exercise. A bijection that preserves the monoid multiplication automatically preserves the unit. We can say that the unit is property-like structure. This is because an associative magma, or semigroup, has at most one unit.

If we alternatively define monoids as associative magmas (that is, semigroups) with the property that a unit exists, then the structure identity principle automatically shows that identifications of monoids are equivalent to bijections that preserve the multiplication, without referring to the unit. However, functions that preserve the multiplication don’t necessarily preserve the unit (exercise), that is, are not automatically monoid homomorphisms.

\[ \text{monoid-axioms}' : \mathcal{U} \rightarrow \mathcal{U} \]
\[ \text{monoid-axioms} X = X \times X \rightarrow X \]

\[ \text{has-unit} : \{X : \mathcal{U} \} \rightarrow \text{monoid-axioms}' X \rightarrow \mathcal{U} \]
\[ \text{has-unit} \{X\} \_\_\_ = \Sigma e : X, \text{left-neutral} e \_\_\_ \times \text{right-neutral} e \_\_\_ \]

As discussed above, the difference is that now the unit is taken to be property rather than structure:

\[ \text{monoid-axioms}' : (X : \mathcal{U}) \rightarrow \text{monoid-axioms}' X \rightarrow \mathcal{U} \]
\[ \text{monoid-axioms} X \_\_\_ = \text{is-set} X \times \text{has-unit} \_\_\_ \times \text{associative} \_\_]
We can define the type of semigroup isomorphisms of monoids as follows:

\[
\text{Monoid' : } \mathcal{U}^\prime \\
\text{Monoid' } = \Sigma X : \mathcal{U}^\prime, \Sigma s : \text{monoid-structure}' X, \text{monoid-axioms}' X s
\]

The equivalence of the alternative type \(\text{Monoid}'\) with the original type \(\text{Monoid}\) is just tuple reshuffling:

\[
\text{to-Monoid} : \text{Monoid}' \rightarrow \text{Monoid} \\
\text{to-Monoid} (X, \_\_\_\_\_, i, (e, l, r), a) = (X, \_\_\_\_\_, e, (i, l, r), a)
\]

\[
\text{from-Monoid} : \text{Monoid} \rightarrow \text{Monoid}' \\
\text{from-Monoid} (X, (\_\_\_\_\_, e), (i, l, r), a) = (X, \_\_\_\_\_, i, (e, l, r), a)
\]

\[
\text{to-Monoid-is-equiv} : \text{is-equiv to-Monoid} \\
\text{to-Monoid-is-equiv} = \text{invertibles-are-equis} \text{to-Monoid} (\text{from-Monoid}, \text{refl}, \text{refl})
\]

\[
\text{from-Monoid-is-equiv} : \text{is-equiv from-Monoid} \\
\text{from-Monoid-is-equiv} = \text{invertibles-are-equis from-Monoid} (\text{to-Monoid}, \text{refl}, \text{refl})
\]

\[
\text{the-two-types-of-monoids-coincide} : \text{Monoid}' \equiv \text{Monoid} \\
\text{the-two-types-of-monoids-coincide} = \text{to-Monoid}, \text{to-Monoid-is-equiv}
\]

And because the existence of a unit is property, the alternative monoid axioms are also property:

\[
\text{monoid-axioms}'\text{-subsingleton} : (X : \mathcal{U}^\prime) (s : \text{monoid-structure}' X) \\
\rightarrow \text{is-subsingleton} (\text{monoid-axioms}' X s)
\]

\[
\text{monoid-axioms}'\text{-subsingleton} X \_\_\_ = \text{subsingleton-criterion}' Y
\]

where
\[
y : \text{monoid-axioms}' X \_\_\_ \rightarrow \text{is-subsingleton} (\text{monoid-axioms}' X \_\_\_) \\
y (i, \_\_\_, \_\_) =
\]

\[
\times\text{-is-subsingleton} \\
\text{being-set-is-subsingleton dfe}
\]

\[
x\text{-is-subsingleton} \\
\k
\]

\[
(\Pi\text{-is-subsingleton dfe} (\lambda x \rightarrow) \\
\Pi\text{-is-subsingleton dfe} (\lambda y \rightarrow) \\
\Pi\text{-is-subsingleton dfe} (\lambda z \rightarrow ((x \cdot y) \cdot z) (x \cdot (y \cdot z)))))))
\]

where
\[
j : (e : X) \rightarrow \text{is-subsingleton} (\text{left-neutral } e \_\_ \_ \times \text{right-neutral } e \_\_\_)
\]

\[
\text{j e} = \times\text{-is-subsingleton} \\
(\Pi\text{-is-subsingleton dfe} (\lambda x \rightarrow i (e \cdot x) x)) \\
(\Pi\text{-is-subsingleton dfe} (\lambda x \rightarrow i (x \cdot e) x))
\]

\[
k : \text{is-subsingleton} (\text{has-unit } \_\_\_)
\]

\[
k (e, l, r) (e', l', r') = \text{to-subtype=} j p
\]

where
\[
p = e \text{ } \equiv \text{ } (r' e)^{-1} \text{ }
\]
\[
(e \cdot e') \equiv (l e') \\
e'
\]

\[
sns-data' : SNS (\lambda X \rightarrow \Sigma s : \text{monoid-structure}' X, \text{monoid-axioms}' X s) \mathcal{U}
\]

\[
sns-data' = \text{add-axioms} \\
\text{monoid-axioms}' \text{monoid-axioms}'\text{-subsingleton} \\
\text{∞-magma.sns-data}
\]

As promised above, the characterization of equality doesn’t refer to preservation of the unit:

\[
\_\equiv\_ : \text{Monoid'} \rightarrow \text{Monoid'} \rightarrow \mathcal{U} \\
(X, \_\_\_\_\_, \_\_) \equiv' (Y, \_\_\_\_\_, \_\_) =
\]

\[
\Sigma f : (X \rightarrow Y), \text{is-equiv} f \\
\times (((\lambda x x' \rightarrow f (x \cdot x')) \equiv (\lambda x x' \rightarrow f x \cdot f x')))
\]

\[
\text{characterization-of-monoid}'\equiv' : (A B : \text{Monoid'}) \rightarrow (A \equiv B) \equiv (A \equiv' B) \\
\text{characterization-of-monoid}'\equiv' = \text{characterization-of-}'\equiv' \text{ ua sns-data}'
\]

We can define the type of semigroup isomorphisms of monoids as follows:

\[
\_\equiv\_ : \text{Monoid} \rightarrow \text{Monoid} \rightarrow \mathcal{U} \\
(X, (\_\_\_\_\_, \_\_), \_\_) \equiv (Y, (\_\_\_\_\_, \_\_), \_\_) =
\]

\[
\Sigma f : (X \rightarrow Y), \text{is-equiv} f \\
\times (((\lambda x x' \rightarrow f (x \cdot x')) \equiv (\lambda x x' \rightarrow f x \cdot f x')))
\]
The notion of preservation of the associativity data depends not only on the homomorphism but also on the underlying type being a set. In the absence of the requirement that the underlying type is a set, the equivalences in the characterization of equality of associative $\infty$-magmas not only have to be homomorphic with respect to the magma operations but also need to respect the associativity data.

We then get the following alternative characterization of monoid equality as semigroup-isomorphism, solving the above exercise, and a bit more.

### 2nd-characterization-of-monoid-≡

<table>
<thead>
<tr>
<th>From-Monoid A ≡ from-Monoid B</th>
<th>= (i )</th>
</tr>
</thead>
<tbody>
<tr>
<td>From-Monoid A ≡' from-Monoid B</td>
<td>= (ii )</td>
</tr>
<tr>
<td>From-Monoid A ≡ B</td>
<td>= (iii )</td>
</tr>
</tbody>
</table>

**where**

φ : A ≡ B → from-Monoid A ≡ from-Monoid B
φ = ap from-Monoid

from-Monoid-is-embedding : is-embedding from-Monoid
from-Monoid-is-embedding = equivs-are-embeddings from-Monoid is-equiv

φ-is-equiv : is-equiv φ
φ-is-equiv = embedding-gives-ap-is-equiv from-Monoid from-Monoid-is-embedding A B

clearly : (from-Monoid A ≡' from-Monoid B) ≡ (A ≡ B)
clearly = refl _

i = (φ , φ-is-equiv)

ii = characterization-of-monoid'-≡ _ _

iii = Id-Eq _ _ clearly

### Table of contents

**Associative $\infty$-magmas**

In the absence of the requirement that the underlying type is a set, the equivalences in the characterization of equality of associative $\infty$-magmas not only have to be homomorphic with respect to the magma operations but also need to respect the associativity data.

**module** associative-ω-magma
{
  {U : Universe}
  (ua : is-univalent U)
  where

  fe : dfunext U U
  fe = univalence-gives-dfunext ua

  associative : {X : U} → (X → X → X) → U

  associative _ _ = ∀ x y z → (x · y) · z ≡ x · (y · z)

  ω-amagma-structure : U → U
  ω-amagma-structure X = Σ _ _ : (X → X → X), (associative _ _)

  ω-aMagma : U → U
  ω-aMagma = Σ X : U , ω-amagma-structure X

  homomorphic : {X Y : U} → (X → X → X) → (Y → Y → Y) → (X → Y) → U

  homomorphic _ _ _ _ * f = (λ x y → f (x · y)) ≡ (λ x y → f x * f y)

The notion of preservation of the associativity data depends not only on the homomorphism $f$ but also on the homomorphism data $h$ for $f$:

respect-assoc : {X A : U} (_ _ : X → X → X) (_ _ : A → A → A)
  → associative _ _ _ _ associative _ _ _ _
  → (f : X → A) → homomorphic _ _ _ _ _ _ _ _ f → U

respect-assoc _ _ _ _ _ _ _ a β f h = fa ≡ β f

**where**

l = λ x y z → f ((x · y) · z) ≡ (ap (λ → → (x · y) z) h)

f (x · y) * f z ≡ (ap (λ → → x y * f z) h)
The functions \(1\) and \(r\), defined from the binary homomorphism condition \(h\), give the homomorphism condition for the two induced ternary magma operations of each magma.

The following, which holds by construction, will be used implicitly:

\[
\begin{align*}
\text{open sip hiding (homomorphic)} \\
sns-data : \text{SNS } \Rightarrow \text{amagma-structure } \mathcal{U} \\
sns-data = (\iota, p, \theta) \\
\text{where} \\
\iota : (X, A : \Rightarrow \text{amagma}) \to (X) = \{A\} \to \mathcal{U} \\
\iota (X, _\cdot_\cdot_, \alpha) (A, _\cdot_\cdot_, \beta) (f, i) = \Sigma h \colon \text{homomorphic } _\cdot_\cdot_\cdot_\cdot_\cdot_\cdot_\cdot f \\
\text{where} \\
\rho : (X : \Rightarrow \text{amagma}) \to \iota X' X' (id-\approx (X')) \\
\rho (X, _\cdot_\cdot_, \alpha) = h, p \\
\text{where} \\
h : \text{homomorphic } _\cdot_\cdot_\cdot_\cdot_\cdot_\cdot id \\
h = \text{refl } _\cdot_\cdot_\cdot_\cdot_\cdot_\cdot \\
p : (\lambda x y z \to \text{refl } ((x \cdot y) \cdot z) \cdot \text{ap id } (\alpha x y z)) \equiv \alpha \\
p = \text{fe } (\lambda x \to \text{fe } (\lambda y \to \text{fe } (\lambda z \to \text{refl-left } \cdot \text{ap-id } (\alpha x y z))))
\end{align*}
\]

We prove the canonicity condition \(\theta\) with the Yoneda machinery.

\[
\begin{align*}
u : (X : \mathcal{U}' \cdot) \to \forall s \to \exists! t : \Rightarrow \text{amagma-structure } X, \iota (X, s) (X, t) (id-\approx X) \\
u X (\cdot_\cdot_, \alpha) = c, \phi \\
\text{where} \\
c : \Sigma t : \Rightarrow \text{amagma-structure } X, \iota (X, _\cdot_\cdot_, \alpha) (X, t) (id-\approx X) \\
c = (\cdot_\cdot_, \alpha), p (X, _\cdot_\cdot_, \alpha) \\
\phi : (\sigma : \Sigma t : \Rightarrow \text{amagma-structure } X, \iota (X, _\cdot_\cdot_, \alpha) (X, t) (id-\approx X)) \to c \equiv \sigma \\
\phi ((\cdot_\cdot_, \beta), \text{refl } _\cdot_\cdot_\cdot_\cdot_\cdot_\cdot, k) = \gamma \\
\text{where} \\
a : \text{associative } _\cdot_\cdot_\cdot_\cdot_\cdot_\cdot_\cdot \\
a x y z = \text{refl } ((x \cdot y) \cdot z) \cdot \text{ap id } (\alpha x y z) \\
g : \text{singleton-type}' a \to \Sigma t : \Rightarrow \text{amagma-structure } X, \iota (X, _\cdot_\cdot_, \alpha) (X, t) (id-\approx X) \\
g (\beta, k) = (_\cdot_\cdot_, \beta), \text{refl } _\cdot_\cdot_\cdot_\cdot_\cdot_\cdot, k \\
i : \text{is-subsingleton } (\text{singleton-type}' a) \\
i = \text{singletons-are-subsingletons } (\text{singleton-types}'-\text{are-singletons } a) \\
q : \alpha, \text{pr}_2 (p (X, _\cdot_\cdot_, \alpha)) \equiv \beta, k \\
q = i _\cdot_\cdot_\cdot_\cdot_\cdot_\cdot \\
y : c \equiv (_\cdot_\cdot_, \beta), \text{refl } _\cdot_\cdot_\cdot_\cdot_\cdot_\cdot, k \\
y = \text{ap g } q
\end{align*}
\]

\[
\theta : \{X : \mathcal{U}' \cdot\} (s t : \Rightarrow \text{amagma-structure } X) \to \text{is-equiv } \text{(canonical-map } \iota p s t) \]
We add an axiom to monoids to get groups, so that we get a characterization of group equality from that of monoid equality.

The promised characterization of associative ∞-magma equality then follows directly from the general structure of identity principle:

\[ \_\_ = \infty\text{-}aMagma \rightarrow \infty\text{-}aMagma \rightarrow \mathcal{U} \]

\[ (X, \_\_ \cdot \_, a) \cong (Y, \_\_ \cdot \_, \beta) = \Sigma f : (X \rightarrow Y), \text{is-equiv } f \times (\Sigma h : \text{homomorphic } \_\_ \cdot \_\_ \cdot \_\_ \text{ and respect-associative } \_\_ \cdot \_\_ \cdot \_\_ \text{ of } \alpha \beta f h) \]

characterization-of-∞-aMagma-≡ : (A B : ∞-aMagma) → (A ≡ B) ≡ (A ≡ B)
characterization-of-∞-aMagma-≡ = characterization-of-≡ ua sns-data

Table of contents

Groups

We add an axiom to monoids to get groups, so that we get a characterization of group equality from that of monoid equality.

module group {U : Universe} (ua : is-univalent U) where
hfe : hfunext U U
hfe = univalence-gives-hfunext ua
open sip
open sip-with-axioms
open monoid {U} ua hiding (sns-data ; _≡ ; _≡')
group-structure : U' → U'
group-structure X = Σ s X monoid-structure X, monoid-axioms X s
group-axiom : (X : U') → monoid-structure X → U'
group-axiom X (\_\_ \cdot \_, e) = (x : X) → Σ x' : X, (x \cdot x' \equiv e) \times (x' \cdot x \equiv e)

Group : U'
Group = Σ X : U', Σ ((\_\_ \cdot \_, e), a) : group-structure X, group-axiom X (\_\_ \cdot \_, e)

inv-lemma : (X : U') (\_\_ : X → X → X) (e : X)
→ monoid-axioms X (\_\_ \cdot \_, e)
→ (x y z : X)
→ (y \cdot x) ≡ e
→ (x \cdot z) ≡ e
→ y ≡ z

inv-lemma X _\_ e (s, l, r, a) x y z q p =

\[
\begin{align*}
y & \equiv (r \cdot y)^{-1} \\
(y \cdot e) & \equiv (ap (y \cdot \_) (p^{-1})) \\
(y \cdot (x \cdot z)) & \equiv (a \cdot y \cdot x \cdot z)^{-1} \\
(y \cdot x) \cdot z & \equiv (ap (_\_ \cdot z) q) \\
(e \cdot z) & \equiv (l \cdot z) \\
z & \equiv
\end{align*}
\]

group-axiom-is-subsingleton : (X : U')
→ (s : group-structure X)
→ is-subsingleton (group-axiom X (pr1 s))
group-axiom-is-subsingleton X ((\_\_ \cdot \_, e), (s, l, r, a)) = \gamma

where
i : (X : X) → is-subsingleton (Σ x' : X, (x \cdot x' \equiv e) \times (x' \cdot x \equiv e))
i x (y, \_, q) (z, p, \_, \_) = u

where
t : y ≡ z
t = inv-lemma X _\_ e (s, l, r, a) x y z q p
We first name various projections and introduce notation. We now solve this exercise and do a bit more on the way, but in a different way as we did for monoids, for the sake of variation.

**Exercise.** The above equivalence is characterized by induction on identifications as the function that maps the reflexive identification to the identity equivalence.

**Exercise.** As in monoids, the preservation of the unit follows from the preservation of the multiplication, and hence one can remove \( f \ d \equiv e \) from the above definition. Prove that

\[
(A \equiv B) \equiv (A \equiv B)
\]

and hence, by transitivity,

\[
(A \equiv B) \equiv (A \equiv B)
\]

where

\[
_\equiv' : \text{Group} \to \text{Group} \to \mathcal{U}'
\]

\[
(X, (_\cdot, \ d), \ _, \ _) \equiv' (Y, (_\cdot', \ e), \ _, \ _) =
\]

\[
\Sigma \; f : (X \to Y), \ \text{is-equiv} \; f
\]

\[
\times ((λ \ x \ x' \rightarrow f \ (x \cdot x')) \equiv (λ \ x \ x' \rightarrow f \ x \cdot f \ x'))
\]

We now solve this exercise and do a bit more on the way, but in a different way as we did for monoids, for the sake of variation. We first name various projections and introduce notation.
We now solve the exercise.

\[
group-is-set (X, ((\_\_\_), e), i, l, r, a), y) = i
\]

\[
unit-left : (G : Group) (x : G) \\
\rightarrow unit G \cdot (G) x \equiv x
\]

\[
unit-left (X, ((\_\_\_), e), i, l, r, a), y) x = l x
\]

\[
unit-right : (G : Group) (x : G) \\
\rightarrow x \cdot (G) unit G \equiv x
\]

\[
unit-right (X, ((\_\_\_), e), i, l, r, a), y) x = r x
\]

\[
assoc : (G : Group) (x y z : G) \\
\rightarrow (x \cdot (G) y) \cdot (G) z \equiv x \cdot (G) (y \cdot (G) z)
\]

\[
assoc (X, ((\_\_\_), e), i, l, r, a), y) = a
\]

\[
inv : (G : Group) \rightarrow \{ G \} \rightarrow G
\]

\[
inv (X, ((\_\_\_), e), i, l, r, a), y) x = \text{pr}_1 (y x)
\]

\[
inv-left : (G : Group) (x : G) \\
\rightarrow inv G x \cdot (G) x \equiv \text{unit G}
\]

\[
inv-left (X, ((\_\_\_), e), i, l, r, a), y) x = \text{pr}_2 (\text{pr}_2 (y x))
\]

\[
inv-right : (G : Group) (x : G) \\
\rightarrow x \cdot (G) inv G x \equiv \text{unit G}
\]

\[
inv-right (X, ((\_\_\_), e), i, l, r, a), y) x = \text{pr}_1 (\text{pr}_2 (y x))
\]

We now solve the exercise.

\[
\text{preserves-multiplication} : (G H : \text{Group}) \rightarrow ((\{ G \} \rightarrow (H)) \rightarrow U
\]

\[
\text{preserves-multiplication} G H f = (\lambda (x y : \{ G \}) \rightarrow f (x \cdot (G) y)) \\
\equiv (\lambda (x y : \{ G \}) \rightarrow f x \cdot (H) f y)
\]

\[
\text{preserves-unit} : (G H : \text{Group}) \rightarrow ((\{ G \} \rightarrow (H)) \rightarrow U
\]

\[
\text{preserves-unit} G H f = f (\text{unit G}) \equiv \text{unit H}
\]

\[
\text{idempotent-is-unit} : (G : \text{Group}) (x : \{ G \}) \\
\rightarrow x \cdot (G) x \equiv x
\]

\[
\text{idempotent-is-unit} G x p = y
\]

where

\[
x' = inv G x
\]

\[
y = x \equiv ((\text{unit-left } G x)^{-1})
\]

\[
unit G \cdot (G) x \equiv (\text{ap } (\lambda - \rightarrow \cdot (G) x) (\text{inv-left } G x)^{-1})
\]

\[
x' \cdot (G) (x \cdot (G) x) \equiv (\text{ap } (\lambda - \rightarrow x' \cdot (G) -) p)
\]

\[
x' \cdot (G) x \equiv (\text{inv-left } G x)
\]

\[
unit G
\]

\[
\text{unit-preservation-lemma} : (G H : \text{Group}) (f : \{ G \} \rightarrow (H)) \\
\rightarrow \text{preserves-multiplication} G H f
\]

\[
\text{unit-preservation-lemma} \rightarrow \text{preserves-unit} G H f
\]

where

\[
e = f (\text{unit G})
\]
If a map preserves multiplication then it also preserves inverses:

\[
\begin{align*}
\text{inv-Lemma} & : (G : \text{Group}) (x y z : G) \\
& \to (y \cdot (G) x) \equiv \text{unit } G \\
& \to (x \cdot (G) z) \equiv \text{unit } G \\
& \to y \equiv z
\end{align*}
\]

\[
\text{inv-Lemma } G = \text{inv-lemma } (G) (\text{multiplication } G) (\text{unit } G) (\text{monoid-axioms-of } G)
\]

\[
\begin{align*}
\text{one-left-inv} & : (G : \text{Group}) (x x' : G) \\
& \to (x' \cdot (G) x) \equiv \text{unit } G \\
& \to x' \equiv \text{inv } G x
\end{align*}
\]

\[
\text{one-left-inv } G x x' p = \text{inv-Lemma } G x x' (\text{inv } G x) p (\text{inv-right } G x)
\]

\[
\begin{align*}
\text{one-right-inv} & : (G : \text{Group}) (x x' : G) \\
& \to (x \cdot (G) x') \equiv \text{unit } G \\
& \to x' \equiv \text{inv } G x
\end{align*}
\]

\[
\text{one-right-inv } G x x' p = (\text{inv-Lemma } G x (\text{inv } G x) x' (\text{inv-left } G x) p)^{-1}
\]

\[
\begin{align*}
\text{preserves-inv} & : (G H : \text{Group}) \to (G) \to (H) \to \mathcal{U}' \\
& \text{preserves-inv } G H f = (x : G) \to f (\text{inv } G x) \equiv \text{inv } H (f x)
\end{align*}
\]

\[
\begin{align*}
\text{inv-preservation-lemma} & : (G H : \text{Group}) (f : G) \to (H) \\
& \to \text{preserves-multiplication } G H f \\
& \to \text{preserves-inv } G H f
\end{align*}
\]

\[
\text{inv-preservation-lemma } G H f m x = y \\
\text{where} \\
p = f (\text{inv } G x) \cdot H f x \equiv (\text{ap } (\lambda x \to -) (\text{inv } G x) x)^{-1} \) \\
f (\text{inv } G x) \cdot H f x \equiv (\text{ap } f (\text{inv-left } G x) ) \\
f (\text{unit } G) \equiv (\text{unit-preservation-lemma } G H f m ) \\
\text{unit } H
\]

\[
\begin{align*}
y & : f (\text{inv } G x) \equiv \text{inv } H (f x) \\
y & = \text{one-left-inv } H (f x) (f (\text{inv } G x)) p
\end{align*}
\]

The usual notion of group homomorphism is that of multiplication-preserving function. But it is known that a group homomorphism in this sense is the same thing as a function that preserves both the multiplication and the unit.

\[
\begin{align*}
\text{is-homomorphism} & : (G H : \text{Group}) \to (G) \to (H) \to \mathcal{U}' \\
& \text{is-homomorphism } G H f = \text{preserves-multiplication } G H f \\
& \times \text{preserves-unit } G H f
\end{align*}
\]

\[
\begin{align*}
\text{preservation-of-mult-is-subsingleton} & : (G H : \text{Group}) (f : G) \to (H) \\
& \to \text{is-subsingleton } (\text{preserves-multiplication } G H f)
\end{align*}
\]

\[
\text{preservation-of-mult-is-subsingleton } G H f = j
\]

\[
\text{where} \\
j : \text{is-subsingleton } (\text{preserves-multiplication } G H f) \\
j = \mathbb{N}-is-set hfe \\
(\lambda x \to \mathbb{N}-is-set hfe) \\
(\lambda x \to \text{group-is-set } H) \\
(\lambda (x y : G) \to f (x \cdot (G) y)) \\
(\lambda (x y : G) \to f x \cdot (H) f y)
\]

\[
\text{being-homomorphism-is-subsingleton} : (G H : \text{Group}) (f : G) \to (H) \\
\to \text{is-subsingleton } (\text{is-homomorphism } G H f)
\]

\[
\text{being-homomorphism-is-subsingleton } G H f = i
\]

\[
\text{where} \\
i : \text{is-subsingleton } (\text{is-homomorphism } G H f)
\]
\( i = \text{is-subsingleton} \)

\[(\text{preservation-of-mult-is-subsingleton} \ G \ H \ f) \]

\[(\text{group-is-set} \ (f \ (\text{unit} \ G)) \ (\text{unit} \ H)) \]

**notions-of-homomorphism-agree :** \((G \ H : \text{Group}) \to (f : \{G\} \to \{H\}) \to \text{is-homomorphism} \ G \ H \ f \to \text{preserves-multiplication} \ G \ H \ f\)

\( \text{notions-of-homomorphism-agree} \ G \ H \ f = \gamma \)

where

\( \alpha : \text{is-homomorphism} \ G \ H \ f \to \text{preserves-multiplication} \ G \ H \ f \)

\( \alpha = \text{pr}_1 \)

\( \beta : \text{preserves-multiplication} \ G \ H \ f \to \text{is-homomorphism} \ G \ H \ f \)

\( \beta \ m = m \), \text{unit-preservation-lemma} \ G \ H \ f \ m \)

\( \gamma : \text{is-homomorphism} \ G \ H \ f \to \text{preserves-multiplication} \ G \ H \ f \)

\( \gamma = (\text{being-homomorphism-is-subsingleton} \ G \ H \ f) \)

\( (\text{preservation-of-mult-is-subsingleton} \ G \ H \ f) \)

\( (\alpha, \beta) \)

\( \equiv\text{-agreement} : (G \ H : \text{Group}) \to (G \equiv H) \equiv (G \equiv' H) \)

\( \equiv\text{-agreement} \ G \ H = \Sigma\text{-cong} (\lambda \to \Sigma\text{-cong} (\lambda \to \text{notions-of-homomorphism-agree} \ G \ H \ f)) \)

This equivalence is that which forgets the preservation of the unit:

\( \text{forget-unit-preservation} : (G \ H : \text{Group}) \to (G \equiv H) \to (G \equiv' H) \)

\( \text{forget-unit-preservation} \ G \ H \ f = f \), \( e \), \( m \), \_ \)

\( \text{NB} : (G \ H : \text{Group}) \to (\equiv\text{-agreement} \ G \ H \ f) \equiv \text{forget-unit-preservation} \ G \ H \)

\( \text{NB} \ G \ H = \text{refl} _\_ \)

\( \text{forget-unit-preservation-is-equiv} : (G \ H : \text{Group}) \to \text{is-equiv} (\text{forget-unit-preservation} \ G \ H) \)

\( \text{forget-unit-preservation-is-equiv} \ G \ H = \equiv\text{-is-equiv} (\equiv\text{-agreement} \ G \ H) \)

This completes the solution of the exercise.

**Table of contents ⇑**

### The slice type

We now apply the above machinery to get a characterization of equality in the slice type.

**module slice**

\( \{U \ V : \text{Universe}\} \)

\( (R : \ V') \)

where

**open sip**

\( S : \ U' \to U \cup V' \)

\( S X = X \to R \)

\( \text{sns-data} : \text{SNS} \ S (U \cup V) \)

\( \text{sns-data} = (\iota, \rho, \theta) \)

where

\( \iota : (A B : \Sigma S) \to \{A\} \equiv \{B\} \to U \cup V' \)

\( \iota (X, g) (Y, h) (f, \_ \_ \_) = (g \equiv h \circ f) \)

\( \rho : (A : \Sigma S) \to \iota A A (\text{id} \equiv \{A\}) \)

\( \rho (X, g) = \text{refl} \ g \)

\( k : \{X : \ U'\} \to \text{canonical-map} \ \iota \ \rho \ g \ h \sim \text{id} \ (g \equiv h) \)

\( k (\text{refl} \ g) = \text{refl} \ (\text{refl} \ g) \)
Subgroups

It is common mathematical practice to regard isomorphic groups to be the same, which is a theorem in univalent mathematics, with the notion of sameness articulated by the identity type, as shown above. However, for some purposes, we may wish to consider two groups to be the same if they have the same elements. For example, in order to show that the subgroups of a group form an algebraic lattice with the finitely generated subgroups as the compact elements, it is this notion of equality that is used, with subgroup containment as the lattice order.

Asking whether two groups have the same elements in univalent mathematics doesn’t make sense unless they are subgroups of the same ambient group. In the same way that in univalent mathematics two members of the powerset are equal iff they have the same elements, two subgroups are equal if and only if they have the same elements. The existence of an isomorphism of two subgroups does not imply their equality in the type of subgroups.

The notion of subgroup can be formulated in two equivalent ways.

1. A subgroup is an element of the powerset of the underlying set of the group that is closed under the group operations. So the type of subgroups of a given group is embedded as a subtype of the powerset of the underlying set and hence inherits the characterization of equality from the powerset.

2. A subgroup of a group $G$ is a group $H$ together with a homomorphic embedding $H \to G$. With this second definition, two subgroups $H$ and $H'$ are equal iff the embeddings $H \to G$ and $H' \to G$ can be completed to a commutative triangle by a (necessarily unique) equivalence $H \to H'$. So the type of subgroups of a group $G$ with underlying set $(G : \mathcal{U})$ is embedded in the slice type $\mathcal{U} / \langle G \rangle$ and hence inherits the characterization of equality from the slice type.

Exercise. The above equivalence is characterized by induction on identifications as the function that maps the reflexive identification to the identity equivalence.
We now show that the type of subgroups is equivalent to the following type, as an application of the subtype classifier.

\[
\forall ((x : \{ G \}) \to A \to A \text{ (inv } G x))
\]

Subgroup : \( \mathcal{U} \cdot \)
Subgroup = \( \Sigma A : \mathcal{P} \{ G \}, \text{ group-closed } (_) \in A \)

\[
\langle \_ \rangle : \text{ Subgroup } \to \mathcal{P} \{ G \}
\]

\[
\langle A, u, c, \lambda \rangle A = A
\]

being-group-closed-subset-is-subsingleton : \( A : \mathcal{P} \{ G \} \to \text{ is-subsingleton } (\text{ group-closed } (_) \in A) \)
being-group-closed-subset-is-subsingleton A = \( x \cdot \text{ is-subsingleton } (\_ \in \text{ subgroup-equality } A (\text{ unit } G)) \)
\( x \cdot \text{ is-subsingleton } (\text{ Pi-is-subsingleton } dfe (\lambda x \to \text{ Pi-is-subsingleton } dfe (\lambda y \to \text{ Pi-is-subsingleton } dfe (\lambda _ \to \text{ is-subsingleton } A (x \cdot y))))) \)
\( (\text{ Pi-is-subsingleton } dfe (\lambda x \to \text{ Pi-is-subsingleton } dfe (\lambda _ \to \text{ is-subsingleton } A (\text{ inv } G x)))) \)

\[
\langle \_ \rangle \text{-is-embedding} : \text{ is-embedding } \langle \_ \rangle
\]
\( \langle \_ \rangle \text{-is-embedding} = \text{ pr}, \text{-is-embedding } \text{ being-group-closed-subset-is-subsingleton} \)

Therefore equality of subgroups is equality of their underlying subsets in the powerset:

\[
\text{ ap-} \langle \_ \rangle : (S T : \text{ Subgroup}) \to S \equiv T \to \langle S \rangle \equiv \langle T \rangle
\]
\( \text{ ap-} \langle \_ \rangle S T = \text{ ap } \langle \_ \rangle \)

\[
\text{ ap-} \langle \_ \rangle \text{-is-equiv} : (S T : \text{ Subgroup}) \to \text{ is-equiv } (\text{ ap-} \langle \_ \rangle \text{ S T})
\]
\( \text{ ap-} \langle \_ \rangle \text{-is-equiv} = \text{ embedding-gives-ap-is-equiv } \langle \_ \rangle \langle \_ \rangle \text{-is-embedding} \)

subgroups-form-a-set : \( \text{ is-set } \text{ Subgroup} \)
subgroups-form-a-set S T = \( \text{ equiv-to-subsingleton } (\text{ ap-} \langle \_ \rangle \text{ S T}, \text{ ap-} \langle \_ \rangle \text{-is-equiv } \text{ S T}) \)
(\( \text{ powersets-are-sets' } \text{ ua } \langle S \rangle \langle T \rangle \))

It follows that two subgroups are equal if and only if they have the same elements:

\[
\text{ subgroup-equality } : (S T : \text{ Subgroup}) \to (S \equiv T) \to ((x : \{ G \}) \to (x \in \langle S \rangle) \leftrightarrow (x \in \langle T \rangle))
\]
\[
\text{ subgroup-equality } S T = \gamma
\]
where
\[
f : S \equiv T \to (x : \{ G \}) \to x \in \langle S \rangle \leftrightarrow x \in \langle T \rangle
\]
\( f p x = \text{ transport } (\lambda - \to x \in \langle - \rangle) p, \text{ transport } (\lambda - \to x \in \langle - \rangle) (p ^{-1}) \)

\[
h : ((x : \{ G \}) \to x \in \langle S \rangle \leftrightarrow x \in \langle T \rangle) \to \langle S \rangle \equiv \langle T \rangle
\]
\( h \psi = \text{ subset-extensionality' } \text{ ua } \alpha \beta \)
where
\[
\alpha : \{ S \} \subseteq \{ T \}
\]
\( \alpha x = \text{ li-implication } (\psi x) \)
\[
\beta : \{ T \} \subseteq \{ S \}
\]
\( \beta x = \text{ ri-implication } (\psi x) \)

\[
g : ((x : \{ G \}) \to x \in \langle S \rangle \leftrightarrow x \in \langle T \rangle) \to S \equiv T
\]
\( g = \text{ inverse } (\text{ ap-} \langle \_ \rangle \text{ S T}) (\text{ ap-} \langle \_ \rangle \text{-is-equiv } \text{ S T}) \circ h \)

\[
y : (S \equiv T) = ((x : \{ G \}) \to x \in \langle S \rangle \leftrightarrow x \in \langle T \rangle)
\]
\( y = \text{ logically-equivalent-subsingletons-are-equivalent } _ _
\]
(\( \text{ subgroups-form-a-set } \text{ S T} \))
(\( \text{ Pi-is-subsingleton } dfe (\lambda x \to x \cdot \text{ is-subsingleton } (\text{ Pi-is-subsingleton } dfe (\lambda _ \to \text{ is-subsingleton } \langle T \rangle x)) \))
(\( \text{ Pi-is-subsingleton } dfe (\lambda _ \to \text{ is-subsingleton } \langle S \rangle x)))
(\( f, g \))

We now show that the type of subgroups is equivalent to the following type, as an application of the subtype classifier.
Subgroup' \( \Sigma H : \text{Group} \)
\( \Sigma h : (\{ H \} \to \{ G \}) \)
\( \text{is-embedding} \ h \)
\( \times \text{is-homomorphism} \ H \ G \ h \)

It will be convenient to introduce notation for the type of group structures satisfying the group axioms:

\[ T : \mathcal{U}' \to \mathcal{U}' \]
\[ T X = \Sigma ((\_ \cdot \_), e), a) : \text{group-structure} \ X, \text{group-axiom} \ X ((\_ \cdot \_), e) \]

We use an anonymous module to give common assumptions for the following few lemmas:

```plaintext
module _ \{ X : \mathcal{U}' \} (h : X \to \{ G \}) (h-is-embedding : \text{is-embedding} \ h) where

private
  h-lc : \text{left-cancellable} \ h
  h-lc = \text{embeddings-are-lc} \ h \ h-is-embedding

having-group-closed-fiber-is-subsingleton : \text{is-subsingleton} \ (\text{group-closed} \ (\text{fiber} \ h))
having-group-closed-fiber-is-subsingleton = \text{being-group-closed-subset-is-subsingleton} \ A
where
  A : \mathcal{P} \{ G \}
  A y = (\text{fiber} \ h \ y, h-is-embedding \ y)

at-most-one-homomorphic-structure : \text{is-subsingleton} \ (\Sigma \tau : T X, \text{is-homomorphism} \ (X, \tau) G h)

at-most-one-homomorphic-structure
  ((((_, unitH), maxioms), gaxiom), (pmult, punit))
  ((((_, unitH'), maxioms'), gaxiom'), (pmult', punit'))
= Y
where
  \tau' : T X
  \tau = ((_, unitH), maxioms), gaxiom
  \tau' = ((_, unitH'), maxioms'), gaxiom'

  i : \text{is-homomorphism} \ (X, \tau) G h
  i = (pmult, punit)

  i' : \text{is-homomorphism} \ (X, \tau') G h
  i' = (pmult', punit')

  p : _ \cdot_ ≡ _ \cdot'_
  p = \text{gfe} \ (\lambda x \to \text{gfe} \ (\lambda y \to h-lc \ (h \ (x \cdot y) \equiv \ (\text{ap} \ (\lambda - \to - \cdot x \ y) \ pmult \ )
  h \ x \cdot h \ y \equiv (\text{ap} \ (\lambda - \to - \cdot x \ y) \ pmult')⁻¹)
  h \ (x \cdot' y) \equiv)))

  q : unitH ≡ unitH'
  q = h-lc \ (h \ unitH \≡ \ (punit \ )
  unit G \≡ \ (punit'⁻¹)
  h \ unitH' \equiv)

  r : (_, unitH) ≡ (_, unitH')
  r = to-\cdot-≡ \ (p, q)

  δ : \tau ≡ \tau'
  δ = to-subtype-≡
      (\text{group-axiom-is-subsingleton} \ X)
      (to-subtype-≡
       (\text{monoid-axioms-subsingleton} \ X)
       r)

  \gamma : (\tau , i) ≡ (\tau' , i')
  \gamma = to-subtype-≡ \ (\lambda \tau \to \text{being-homomorphism-is-subsingleton} \ (X, \tau) G h) \ δ

\text{homomorphic-structure-gives-group-closed-fiber} : (\Sigma \tau : T X, \text{is-homomorphism} \ (X, \tau) G h)
\to \text{group-closed} \ (\text{fiber} \ h)

\text{homomorphic-structure-gives-group-closed-fiber}
  ((((_, unitH), maxioms), gaxiom), (pmult, punit))
  = (unitc, mulc, invc)
```
Conversely:

\[ \text{group-closed-fiber-gives-homomorphic-structure} : \text{group-closed} (\text{fiber } h) \rightarrow (\Sigma \tau : T X, \text{is-homomorphism} (X, \tau) G h) \]

\[ \text{group-closed-fiber-gives-homomorphic-structure} (\text{unitc} , \text{mulc} , \text{invc}) = \tau , \text{i} \]

where

\[ \varphi : (x : X) \rightarrow \text{fiber } h (h x) \]
\[ \varphi x = (x , \text{refl} (h x)) \]

\[ \text{unitH} : X \]
\[ \text{unitH} = \text{fiber-point} \text{ unitc} \]

\[ _*_* : X \rightarrow X \rightarrow X \]
\[ x * y = \text{fiber-point} (\text{mulc} (h x) (h y) (\varphi x) (\varphi y)) \]

\[ \text{invH} : X \rightarrow X \]
\[ \text{invH} x = \text{fiber-point} (\text{invc} (h x) (\varphi x)) \]

\[ \text{pmul} : (x y : X) \rightarrow (x * y) \equiv h x * h y \]
\[ \text{pmul} x y = \text{fiber-identification} (\text{mulc} (h x) (h y) (\varphi x) (\varphi y)) \]

\[ \text{punit} : h \text{unitH} \equiv \text{unit} G \]
\[ \text{punit} = \text{fiber-identification} \text{ unitc} \]

\[ \text{pinv} : (x : X) \rightarrow h (\text{invc} x) \equiv \text{inv} G (h x) \]
\[ \text{pinv} x = \text{fiber-identification} (\text{invc} (h x) (\varphi x)) \]

\[ \text{unitH-left} : (x : X) \rightarrow \text{unitH} * x \equiv x \]
\[ \text{unitH-left} x = \text{h-lc} (h (\text{unitH} * x) \equiv (\text{pmul} \text{unitH} x) \text{unitH} , h x \equiv (\text{ap} (\_, h x) \text{punit})) \]
\[ \text{unit H} * h x \equiv (\text{unit-left} G (h x)) \]
\[ h x \]

\[ \text{unitH-right} : (x : X) \rightarrow x * \text{unitH} \equiv x \]
\[ \text{unitH-right} x = \text{h-lc} (h (x * \text{unitH}) \equiv (\text{pmul} x \text{unitH}) \text{unitH} , h x \equiv (\text{ap} (h x , \_) \text{punit})) \]
\[ h x \equiv (\text{unit-right} G (h x)) \]
\[ h x \]

\[ \text{assocH} : (x y z : X) \rightarrow ((x * y) * z) \equiv (x * (y * z)) \]
\[ \text{assocH} x y z = \text{h-lc} (h ((x * y) * z) \equiv (\text{pmul} (x * y) z)) \]
\[ h (x * y) * h z \equiv (\text{ap} (\_, h z) (\text{pmul} x y)) \]
\[ (h x * h y) * h z \equiv (\text{assoc} G (h x) (h y) (h z)) \]
\[ h x * (h y * h z) \equiv (\text{ap} (h x , \_) (\text{pmul} y z))^{-1} \]
\[ h x * (h y * z) \equiv (\text{pmul} x (y * z))^{-1} \]
\[ h (x * (y * z)) \]

\[ \text{group-axiomH} : (x : X) \rightarrow \Sigma x' : X , (x * x' \equiv \text{unitH}) \times (x' * x \equiv \text{unitH}) \]
\[ \text{group-axiomH} x = \text{invH} x , \]
\[ \text{h-lc} (h (x * \text{invH} x) \equiv (\text{pmul} x (\text{invH} x)) \]
In particular, a subgroup induces a genuine group: 

\[
\begin{align*}
\text{induced-group} & : \text{Subgroup} \to \text{Group} \\
\text{induced-group} \ S & = \text{pr}_1 (', \text{characterization-of-the-type-of-subgroups} \ S) \\
\end{align*}
\]
The crucial tool to characterize equality in the alternative type of subgroups is the following embedding into the slice type.

```
forgetful-map : Subgroup' → U / { G }
forgetful-map ((X , _), h , _) = (X , h)
```

To show that this map is an embedding, we express it as a composition of maps that are more easily seen to be embeddings.

```
forgetful-map-is-embedding : is-embedding forgetful-map
forgetful-map-is-embedding = \gamma
where
  Subtype' : U → U ·
  Subtype' X = Σ (X , h) : U / { G } , is-embedding h
  f₀ : Subgroup' → Subtype { G }
  f₀ ((X , _), h , e , _) = (X , h , e)
  f₁ : Subtype { G } → Subtype' { G }
  f₁ (X , h , e) = ((X , h), e)
  f₂ : Subtype' { G } → U / { G }
  f₂ ((X , h), e) = (X , h)
  by-construction : forgetful-map ≡ f₂ ∘ f₁ ∘ f₀
  by-construction = refl _
  f₀-lc : left-cancellable f₀
  f₀-lc {(X , τ), h , e , i} ((X , τ'), h , e , i') (refl (X , h , e)) = δ
  where
    p : (τ , i) ≡ (τ' , i')
    p = at-most-one-homomorphic-structure h e (τ , i) (τ' , i')
    φ : (Σ τ : T X , is-homomorphism (X , τ) G h) → Subgroup'
    φ (τ , i) = ((X , τ), h , e , i)
    δ : ((X , τ), h , e , i) ≡ ((X , τ'), h , e , i')
    δ = ap φ p
  f₀-is-embedding : is-embedding f₀
  f₀-is-embedding = lc-maps-into-sets-are-embeddings f₀ f₀-lc (subtypes-form-set ua ( G ))
  f₁-is-equiv : is-equiv f₁
  f₁-is-equiv = invertibles-are-equis f₁ ((λ ((X , h), e) → (X , h , e)), refl , refl)
  f₁-is-embedding : is-embedding f₁
  f₁-is-embedding = equv-are-embeddings f₁ f₁-is-equiv
  f₂-is-embedding : is-embedding f₂
  f₂-is-embedding = pr₁-is-embedding (λ (X , h) → being-embedding-is-subsingleton gfe h)
  \gamma : is-embedding forgetful-map
  \gamma = ∘-embedding f₂-is-embedding (∘-embedding f₁-is-embedding f₀-is-embedding)
```

With this and the characterization of equality in the slice type, we get the promised characterization of equality of the alternative type of subgroups.

```
≡≈ : Subgroup' → Subgroup' → U ·
(H , h , _) ≡≈ (H' , h' , _) = Σ f : ((H → (H')) , is-equiv f × (h ≡ h' ∘ f))
subgroup'-equality : (S T : Subgroup') → (S ≡ T) ⇒ (S ≡ T)
subgroup'-equality S T = (S ≡ T)
  ≡≈ ( i )
  (forgetful-map S ≡ forgetful-map T) ≡≈ ( ii )
  (S ≡ T)
```

where
```
open slice { G } hiding (S)
i = ≡-sym (embedding-criterion-converse forgetful-map forgetful-map-is-embedding S T)
ii = characterization-of-/≡≈ (ua U) (forgetful-map S) (forgetful-map T)
```
The equivalence $f$ in the definition of the relation $\equiv$ is unique when it exists, because $h' \circ f \equiv h$ is an embedding and hence is left-cancellable. Moreover, the type $S \equiv T$ has at most one element:

$$
\text{subgroups'}\text{-form-a-set} : \text{is-set} \text{ Subgroup'}
\text{subgroups'}\text{-form-a-set} = \text{equiv-to-set}
\text{(=}\text{-sym characterization-of-the-type-of-subgroups)}
\text{subgroups'-form-a-set}
\equiv\text{-is-subsingleton-valued} : (S \rightarrow T : \text{Subgroup'}) \rightarrow \text{is-subsingleton} (S \equiv T)
\equiv\text{-is-subsingleton-valued} S \rightarrow T = \gamma
\text{where}
\ i : \text{is-subsingleton} (S \equiv T)
\ i = \text{subgroups'}\text{-form-a-set} S \rightarrow T
\ y : \text{is-subsingleton} (S \equiv T)
\ y = \text{equiv-to-subsingleton} (\equiv\text{-sym} (\text{subgroup'}\text{-equality} S \rightarrow T)) \ i
$$

Here is an alternative proof that avoids the equivalence $\text{Subgroup} \equiv \text{Subgroup'}$ used above to show that the alternative type of subgroups is a set:

$$
\equiv\text{-is-subsingleton-valued}' : (S \rightarrow S' : \text{Subgroup'}) \rightarrow \text{is-subsingleton} (S \equiv S')
\equiv\text{-is-subsingleton-valued}' (H, h, e, i) (H', h', e', i') = \gamma
\text{where}
\ S = (H, h, e, i)
\ S' = (H', h', e', i')
\ A = \sum f : (\{ H \} \rightarrow \{ H' \}), h' \circ f \equiv h
\ B = \sum (f, p) : A, \text{is-equiv} f
\ A\text{-is-subsingleton} : \text{is-subsingleton} A
\ A\text{-is-subsingleton} = \text{postcomp-is-embedding} gfe hfe h' \circ e' \rightarrow \text{being-equiv-is-subsingleton} gfe gfe f
\ B\text{-is-subsingleton} : \text{is-subsingleton} B
\ B\text{-is-subsingleton} = \sum\text{-is-subsingleton}
\ A\text{-is-subsingleton}
\ (\lambda (f, p) \rightarrow \text{being-equiv-is-subsingleton} gfe gfe f)
\ \delta : (S \equiv S') = B
\ \delta = \text{invertibility-gives-=} \alpha (\beta, \eta, \varepsilon)
\text{where}
\ \alpha = \lambda (f, i, p) \rightarrow ((f, (p^{-1})), i)
\ \beta = \lambda ((f, p), i) \rightarrow (f, i, (p^{-1}))
\ \eta = \lambda (f, i, p) \rightarrow \text{ap} (\lambda \rightarrow (f, i, -)) (\sim{-}\text{-involutive} p)
\ \varepsilon = \lambda ((f, p), i) \rightarrow \text{ap} (\lambda \rightarrow ((f', -), i)) (\sim{-}\text{-involutive} p)
\ y : \text{is-subsingleton} (S \equiv S')
\ y = \text{equiv-to-subsingleton} \delta B\text{-is-subsingleton}
$$

Table of contents ⇩

**Rings**

A mathematician asked us what a formalization of Noetherian local rings would look like in univalent type theory, in particular with respect to automatic preservation of theorems about rings by ring isomorphisms. In this section we consider rings, and we consider Noetherian local rings after we discuss unspecified existence.

We consider rings without unit, called *rings*, and with unit, called *rings*.

There are several options to apply the above techniques to accomplish this. One way would be to add structure to Abelian groups. However, in order to avoid having to discuss the preservation of the neutral element of addition by homomorphisms separately as above in the case of groups, we proceed from scratch, where the neutral element is not part of the structure but instead its existence is part of the axioms. Cf. the discussions of equality of monoids and groups.

**Exercise.** Proceed using the alternative approach, which should be equally easy and short (and perhaps even shorter).

We consider *rings* in a universe $\mathcal{U}$, and we assume univalence in their development. We hide the notation (__) from the module *sip* because we are going to use it for the underlying *Rng* of a *Ring*:
We derive function extensionality from univalence:

```lean
fe : global-dfunext
fe = univalence-gives-global-dfunext ua
```

We take rng structure to be the product of two magma structures:

```lean
rng-structure : U → U
rng-structure X = (X → X → X) × (X → X → X)
```

The axioms are the usual ones, with the additional requirement that the underlying type is a set, as opposed to an arbitrary ∞-groupoid:

```lean
rng-axioms : (R : U) → rng-structure R → U
rng-axioms R (+_+ , _' _') = I × II × III × IV × V × VI × VII
where
I = is-set R
II = (x y z : R) → (x + y) + z ≡ x + (y + z)
III = (x y : R) → x + y ≡ y + x
IV = Σ 0 : R , ((x : R) → x + 0 ≡ x) × ((x : R) → Σ x' : R , x + x' ≡ 0)
V = (x y z : R) → (x · y) · z ≡ x · (y · z)
VI = (x y z : R) → x · (y + z) ≡ (x · y) + (x · z)
VII = (x y z : R) → (y + z) · x ≡ (y · x) + (z · x)
```

The type of rngs in the universe U, which lives in the universe after ℰ:

```lean
Rng : U → U
Rng = Σ R : U , Σ s : rng-structure R , rng-axioms R s
```

In order to be able to apply univalence to show that the identity type ℰ ≅ ℰ' of two rngs is in canonical bijection with the type ℰ ≅ ℰ' of ring isomorphisms, we need to show that the axioms constitute property rather than data, that is, they form a subsingleton, or a type with at most one element. The proof is a mix of algebra (to show that an additive semigroup has at most one zero element, and at most one additive inverse for each element) and general facts about subsingletons (e.g. they are closed under products) and is entirely routine.

```lean
rng-axioms-is-subsingleton : (R : U) (s : rng-structure R) → is-subsingleton (rng-axioms R s)
rng-axioms-is-subsingleton R (+_+ , _' _') (i , ii , iii , iv-vii) = δ
where
A = λ (O : R) → ((x : R) → x + 0 ≡ x)
    × ((x : R) → Σ x' : R , x + x' ≡ 0)
IV = Σ A
a : (0 0' : R) → ((x : R) → x + 0 ≡ x) → ((x : R) → x + 0' ≡ x) → 0 ≡ 0'
a 0 0' f f' = 0 ≡ (f' 0)⁻¹
    (0 + 0') ≡ (iii 0 0')
    (0' + 0) ≡ (f 0'
    0' 1
b : (0 : R) → is-subsingleton ((x : R) → x + 0 ≡ x)
b 0 = Π-is-subsingleton fe (λ x → i (x + 0) x)
c : (0 : R)
   → ((x : R) → x + 0 ≡ x)
   → (x : R) → is-subsingleton (Σ x' : R , x + x' ≡ 0)
c 0 f x (x' , p') (x'' , p'') = to-subtype-≡ (λ x' → i (x + x') 0) r
where
r : x' ≡ x''
r = x' 1
```
Then the type of rng identities is in bijection with the type of ring isomorphisms by the above general machinery:

We define a rng isomorphism to be a bijection that preserves addition and multiplication, and collect all isomorphisms of two rngs $\mathcal{R}$ and $\mathcal{R}'$ in a type $\equiv[\text{Rng}] \mathcal{R}'$:

$$
\equiv[\text{Rng}]_\equiv : \text{Rng} \rightarrow \text{Rng} \rightarrow \mathcal{U}'
$$

$$(\mathcal{R}, (_+ \_ \_), \_\_) \equiv[\text{Rng}] (\mathcal{R}', (_+ \!' \_' \_), \_) =
\begin{align*}
\Sigma f : (\mathcal{R} \rightarrow \mathcal{R}') \\
, \text{is-equiv} f \\
\times ((\lambda x y \rightarrow f (x + y)) \equiv (\lambda x y \rightarrow f x + f y)) \\
\times ((\lambda x y \rightarrow f (x \cdot y)) \equiv (\lambda x y \rightarrow f x \cdot f y))
\end{align*}
$$

We define a rng isomorphism to be a bijection that preserves addition and multiplication, and collect all isomorphisms of two rngs $\mathcal{R}$ and $\mathcal{R}'$ in a type $\equiv[\text{Rng}] \mathcal{R}'$:

Then the type of rng identities is in bijection with the type of ring isomorphisms by the above general machinery:
The underlying type of a rng:

\[(\_ : (R : \text{Rng}) \to \mathcal{U} \cdot (R, \_)) = \mathcal{R}\]

We now add units to rngs to get rings.

}\text{ring-structure} : \mathcal{U}^\cdot \to \mathcal{U}^\cdot \\
\text{ring-structure} X = X \times \text{rng-structure} X

\text{ring-axioms} : (R : \mathcal{U}^\cdot) \to \text{ring-structure} R \to \mathcal{U}^\cdot \\
\text{ring-axioms} R (1, \_+_, \_\cdot_) = \text{rng-axioms} R (\_+_, \_\cdot_) \times \text{VIII}

where

\text{VIII} = (x : R) \to (x \cdot 1 \equiv x) \times (1 \cdot x \equiv x)

\text{ring-axioms-is-subsingleton} : (R : \mathcal{U}^\cdot) (s : \text{ring-structure} R) \\
\to \text{is-subsingleton} (\text{ring-axioms} R s)

\text{ring-axioms-is-subsingleton} R (1, \_+_, \_\cdot_) ((1, \text{ii-vii}), \text{viii}) = \gamma ((1, \text{ii-vii}), \text{viii})

where

\gamma : \text{is-subsingleton} (\text{ring-axioms} R (1, \_+_, \_\cdot_))
\gamma = \times\text{-is-subsingleton}

\text{ring-axioms-is-subsingleton} R (\_+_, \_\cdot_)
\text{Ring-is-subsingleton} \text{fe} (\lambda x \to \times\text{-is-subsingleton} (i (x \cdot 1) x) (i (1 \cdot x) x))

The type of rings with unit:

\text{Ring} : \mathcal{U}^\cdot \\
\text{Ring} = \Sigma R : \mathcal{U}^\cdot, \Sigma s : \text{ring-structure} R, \text{ring-axioms} R s

\_\equiv[\text{Ring}]_ : \text{Ring} \to \text{Ring} \to \mathcal{U}^\cdot \\
(R, (1, \_+_, \_\cdot_), \_) \equiv[\text{Ring}] (R', (1', \_+', \_\cdot'), \_) = \\
\Sigma f : (R \to R') \\
, \text{is-equiv} f \\
\times (f 1 \equiv 1') \\
\times ((\lambda x y \to f (x + y)) \equiv (\lambda x y \to f x +' f y)) \\
\times ((\lambda x y \to f (x \cdot y)) \equiv (\lambda x y \to f x \cdot' f y))

\text{characterization-of-ring-} \equiv : (\mathcal{R}, R', \text{Ring}) \to (\mathcal{R} \equiv R') = (\mathcal{R} \equiv[\text{Ring}] R')

\text{characterization-of-ring-} \equiv = \text{characterization-of-} \equiv (\text{ua} \mathcal{U})

\text{add-axioms} \\
\text{ring-axioms} \\
\text{ring-axioms-is-subsingleton} \\
\text{join} \\
\text{pointed-type.sns-data} \\
\text{join} \\
\text{∞-magma.sns-data} \\
\text{∞-magma.sns-data})

Table of contents ↩

Metric spaces, graphs and ordered structures

We now apply the above machinery to get a characterization of equality of metric spaces, graphs and ordered structures/

module generalized-metric-space \\
{\mathcal{U} \mathcal{V} : \text{Universe}}
We get a topological spaces.

Table of contents

Exercise. The above equivalence is characterized by induction on identifications as the function that maps the reflexive identification to the identity equivalence.

We have the following particular cases of interest:

1. **Metric spaces.** If \( \mathbb{R} \) is a type of real numbers, then the axioms can be taken to be those for metric spaces, in which case \( M \) amounts to the type of metric spaces. Then the above characterizes metric space identification as isometry.

2. **Graphs.** If \( \mathbb{R} \) is the type of truth values, and the `axioms` function is constant with value `true`, then \( M \) amounts to the type of directed graphs, and the above characterizes graph identification as graph isomorphism. We get undirected graphs by requiring the relation to be symmetric in the axioms.

3. **Partially ordered sets.** Again with \( \mathbb{R} \) taken to be the type of truth values and suitable axioms, we get posets and other ordered structures, and the above says that their identifications amount to order isomorphisms.

Table of contents ☁

### Topological spaces

We get a *type* of topological spaces when \( \mathbb{R} \) is the type of truth values and the axioms are appropriately chosen.
When $R$ is the type of truth values, the type $(X \to R)$ is the powerset of $X$, and membership amounts to function application:

\[
\begin{align*}
P & : \mathcal{W} \to \mathcal{Y} \sqcup \mathcal{W}' \\
P X &= X \to R \\
\_ \in_ : & \{X : \mathcal{W}'\} \to X \to P X \to R \\
x \in A &= A \times A
\end{align*}
\]

\[
\begin{align*}
\text{inverse-image} &: \{X Y : \mathcal{U}'\} \to (X \to Y) \to P Y \to P X \\
\text{inverse-image} &= \lambda \, x \to f \times x \in B
\end{align*}
\]

\[
\begin{align*}
P P & : \mathcal{U}' \to \mathcal{U} \sqcup \mathcal{V}' \\
P P X &= P (P X)
\end{align*}
\]

\[
\begin{align*}
\text{Space} &: \mathcal{U}' \sqcup \mathcal{V}' \\
\text{Space} &= \Sigma X : \mathcal{U}', \Sigma O : PP X, \text{axioms } X O
\end{align*}
\]

If $(X, OX, a)$ and $(Y, OY, b)$ are spaces, a **homeomorphism** can be described as a bijection $f : X \to Y$ such that the open sets of $Y$ are precisely those whose inverse images are open in $X$, which can be written as

\[
(\lambda (V : P Y) \to \text{inverse-image} f V \in OX) \equiv OY
\]

Then $\iota$ defined below expresses the fact that a given bijection is a homeomorphism:

\[
\begin{align*}
sns-data &: SNS PP (U \sqcup V) \\
sns-data &= (\iota, \rho, \theta) \\
\text{where} &: \\
\iota &: (A B : \Sigma PP) \to \{ A \} \approx \{ B \} \to U \sqcup V' \\
\iota (X, OX) (Y, OY) (f, _) &= (\lambda (V : P Y) \to \text{inverse-image} f V \in OX) \equiv OY
\end{align*}
\]

What $\rho$ says is that identity function is a homeomorphism, trivially:

\[
\begin{align*}
\rho &: (A : \Sigma PP) \to \iota A A (id- \approx A) \\
\rho (X, O) &= refl O
\end{align*}
\]

Then $\theta$ amounts to the fact that two topologies on the same set must be the same if they make the identity function into a homeomorphism.

\[
\begin{align*}
h &: \{X : \mathcal{U}'\} \{O O' : PP X\} \to \text{canonical-map} \ \iota \ O O' \sim id (O \equiv O') \\
h (refl O) &= refl (refl O) \\
\theta &: \{X : \mathcal{U}'\} \{O O' : PP X\} \to \text{is-equiv} \ (\text{canonical-map} \ \iota \ O O') \\
\theta (X) O O' &= \text{equivs-closed-under-} \sim (\text{id-is-equiv} (O \equiv O')) h
\end{align*}
\]

We introduce notation for the type of homeomorphisms:

\[
\begin{align*}
\_ \equiv_ &: \text{Space } \to \text{Space } \to U \sqcup V' \\
(X, OX, _) \equiv (Y, OY, _) &= \\
\Sigma f : (X \to Y), \text{is-equiv} f \\
\times ((\lambda V \to \text{inverse-image} f V \in OX) \equiv OY)
\end{align*}
\]

\[
\begin{align*}
\text{characterization-of-Space-} \equiv &: \text{is-univalent } U \\
\to (A B : \text{Space}) \to (A \equiv B) \equiv (A \equiv B)
\end{align*}
\]

\[
\begin{align*}
\text{characterization-of-Space-} \equiv \ua &= \text{characterization-of-} \equiv \to-axioms \ua \\
\text{sns-data} \ \text{axioms} \ \text{axioms}\ss
\end{align*}
\]

**Exercise.** The above equivalence is characterized by induction on identifications as the function that maps the reflexive identification to the identity equivalence.
But of course there are other choices for \( R \) that also make sense. For example, we can take \( R \) to be a type of real numbers, with the axioms for \( X \) and \( F : (X \to R) \to R \) saying that \( F \) is a linear functional. Then the above gives a characterization of the identity type of types equipped with linear functionals, in which case we may prefer to rephrase the above as

\[
\equiv' : \text{Space} \to \text{Space} \to \mathcal{U} \cup \mathcal{V}
\]

By a selection space we mean a type \( \text{SelectionSpace} \) equipped with a selection function, where a selection function is a map \( (X \to R) \to X \), where the type \( R \) is a parameter.

\[
\Sigma f : (X \to Y), \text{is-equiv} f \times ((\lambda (v : Y \to R) \to F(v \circ f)) \equiv G)
\]

Characterization of \( \equiv' \) by \( \text{is-univalent} \) \( \mathcal{U} \)

\[
(A B : \text{Space}) \to (A \equiv B) \equiv (A \equiv' B)
\]

Characterization of \( \equiv' \) = Characterization of \( \equiv \)

Linear functions on certain spaces correspond to special kinds of measures by the Riesz representation theorem, and hence in this case \( \text{Space} \) becomes a type of such kind of measure spaces by an appropriate choice of axioms.

Table of contents

Selection spaces

By a selection space we mean a type \( X \) equipped with a selection function, where a selection function is a map \( (X \to R) \to X \), where the type \( R \) is a parameter.

module selection-space

\[
(\mathcal{U} \mathcal{V} : \text{Universe})
\]

\[
(\mathcal{R} : \mathcal{V} \cap)
\]

\[
(\text{axioms} : (X : \mathcal{U} \cap) \to (\mathcal{R} (X \to R) \to X) \equiv \mathcal{U} \cup \mathcal{V} \cap)
\]

\[
(\text{axiomss} : (X : \mathcal{U} \cap) (\mathcal{V} : (X \to R) \to \text{is-subsingleton} (\text{axioms} X \mathcal{V}))
\]

where

open sip

open sip-with-axioms

\[
S : \mathcal{U} \cap \to \mathcal{U} \cup \mathcal{V} \cap
\]

\[
S X = (X \to R) \to X
\]

SelectionSpace : \( \mathcal{U} \cap \cup \mathcal{V} \cap \)

SelectionSpace = \( \Sigma X : \mathcal{U} \cap, X \mathcal{V} : S X \), is-subsingleton \( X \mathcal{V} \)

sns-data : SNS S (\( \mathcal{U} \cup \mathcal{V} \))

sns-data = \( \langle \lambda, \rho, \theta \rangle \)

where

\[
\lambda : (A B : \Sigma S) \to \{ A \} \equiv \{ B \} \equiv \mathcal{U} \cup \mathcal{V} \cap
\]

\[
\lambda (X, \mathcal{V}) (Y, \mathcal{V}) (\mathcal{R} (f \mathcal{V})) = (\lambda (q : Y \to R) \to f (q \circ f)) \equiv \mathcal{R}
\]

\[
\rho : (A : \Sigma S) \to \lambda A A (\text{id} \equiv \{ A \})
\]

\[
\rho (X, \mathcal{V}) = \text{refl} \mathcal{V}
\]

\[
\theta : [X : \mathcal{U} \cap] (\mathcal{V} \mathcal{R} : S X) \to \text{is-equiv} \ (\text{canonical-map} \ \lambda \rho \mathcal{V} \mathcal{R})
\]

\[
\theta [X] \mathcal{V} \mathcal{R} = \gamma
\]

where

\[
\gamma : \text{is-equiv} \ (\text{canonical-map} \ \lambda \rho \mathcal{V} \mathcal{R})
\]

\[
\gamma = \text{equivs-closed-under-} \sim (\text{id} \equiv \text{is-equiv} (\mathcal{V} \equiv \mathcal{R}) \mathcal{V})
\]

where

\[
\text{characterization-of-Space-} \equiv' : \mathcal{U} \cup \mathcal{V} \cap
\]

\[
\Sigma f : (X \to Y), \text{is-equiv} f \times ((\lambda (q : Y \to R) \to f (q \circ f)) \equiv \mathcal{R})
\]

\[
\equiv' : \text{SelectionSpace} \to \text{SelectionSpace} \to \mathcal{U} \cup \mathcal{V} \cap
\]

\[
(X, \mathcal{V}, _) \equiv (Y, \mathcal{V}, _) =
\]

\[
\Sigma f : (X \to Y), \text{is-equiv} f \times ((\lambda (q : Y \to R) \to f (q \circ f)) \equiv \mathcal{R})
\]
characterization-of-selection-space≡ : is-univalent ℱ
→ (A B : SelectionSpace) → (A ≡ B) ≃ (A ≡ B)

characterization-of-selection-space≡ ua = characterization-of≡-with-axioms ua
sns-data
axioms axiomss

Exercise. The above equivalence is characterized by induction on identifications as the function that maps the reflexive identification to the identity equivalence.

Table of contents ⇑

A contrived example

Here is an example where we need to refer to the inverse of the equivalence under consideration.

We take the opportunity to illustrate how the above boiler-plate code can be avoided by defining sns-data on the fly, at the expense of readability:

module contrived-example (ᵤ : Universe) where
  open sip
contrived-≡ : is-univalent ℱ →
  (X Y : ℱ) (φ : (X → X) → X) (γ : (Y → Y) → Y)
→
  ((X , φ) ≡ (Y , γ)) ≃ (Σ f : (X → Y)
 , Σ i : is-equiv f
 , (λ (g : Y → Y) → f (φ (inverse f i o g o f))) ≡ γ)
contrived-≡ ua X Y φ γ =
  characterization-of≡ ua
  ((λ (X , φ) (Y , γ) (f , i) → (λ (g : Y → Y) → f (φ (inverse f i o g o f))) ≡ γ),
  (λ (X , φ) → refl φ),
  (λ φ γ → equivs-closed-under-≡ (id-is-equiv (φ ≡ γ)) (λ {(refl φ) → refl (refl φ)})))
(X , φ) (Y , γ)

Many of the above examples can be written in such a concise form.

Table of contents ⇑

Functor algebras

We now characterize equality of functor algebras. In the following, we don’t need to know that the functor preserves composition or to give coherence data for the identification ℱ-id.

module generalized-functor-algebra
  {₁ ₂ : Universe}
  (F : ℱ → ₂)
  (ℱ : {X Y : ℱ} → (X → Y) → F X → F Y)
  (ℱ-id : {X : ℱ} → ℱ (id X) ≡ id (F X))
where
  open sip
S : ℱ → ℱ ⊔ ₂
S X = F X → X
sns-data : SNS S (มวล ⊔ ₂)
sns-data = (ι , ρ , θ)
where
  ι : (A B : Σ S) → { A } ≃ { B } → ℱ ⊔ ₂
  ι (X , α) (Y , β) (f , _) = f ∘ α ≡ β ∘ ℱ f
θ : {X : U} (α β : S X) → is-equiv (canonical-map ⊤ ρ α β)
θ {X} α β = y

where

i : is-equiv c
i = transport-is-equiv (α ≃_) (ρ (X, β))

h : canonical-map ⊤ ρ α β ~ c
h (refl _) = ρ (X, α) ≅ (refl-left⁻¹)

y : is-equiv (canonical-map ⊤ ρ α β)
y = equivs-closed-under-~ i h

characterization-of-functor-algebra-≃ : is-univalent U
→ (X Y : U) (α : F X → X) (β : F Y → Y)
→ (((X, α) ≃ (Y, β)) ≅ (Σ f : (X → Y), is-equiv f x (f ∘ α ≃ β ∘ F f)))

characterization-of-functor-algebra-≃ ua X Y α β =
characterization-of-≃ ua sns-data (X, α) (Y, β)

Exercise. The above equivalence is characterized by induction on identifications as the function that maps the reflexive identification to the identity equivalence.

Table of contents ⇑

Type-valued preorders

We now characterize equality of generalized preordered sets. This example is harder than the previous ones.

A type-valued preorder on a type X is a type-valued relation which is reflexive and transitive. A type-valued, as opposed to a subsingleton-valued preorder, could also be called an ∞-preorder.

type-valued-preorder-S : U → U ⊔ (V ⊔)
type-valued-preorder-S {U} {V} X = Σ _≤_ : (X → X → V)
, X (X x : X) → x ≤ x
× (X y z : X) → x ≤ y → y ≤ z → x ≤ z

A category, also known as a 1-category, is a type-valued preorder subject to suitable axioms, where the relation _≤_ is traditionally written hom, and where identities are given by the reflexivity law, and composition is given by the transitivity law.

We choose to use categorical notation and terminology for type-valued preorders.

module type-valued-preorder
(U V : Universe)
(ua : Univalence)
where

open sip

fe : global-dfunext
fe = univalence-gives-global-dfunext ua

hfe : global-hfunext
hfe = univalence-gives-global-hfunext ua

S : U → U ⊔ (V ⊔)
S = type-valued-preorder-S {U} {V}
Type-valued-preorder : (𝓤 ⊔ 𝓥⁺)⁺
Type-valued-preorder = Σ S

But we will use the shorter notation Σ S in this submodule.

The type of objects of a type-valued preorder:

Ob : Σ S → 𝓤⁺
Ob (X, homX, idX, compX) = X

Its hom-types (or preorder):

hom : (𝓧 : Σ S) → Ob 𝓧 → Ob 𝓧 → 𝓥⁺
hom (X, homX, idX, compX) = homX

Its identities (or reflexivities):

id : (𝓧 : Σ S) (x : Ob 𝓧) → hom 𝓧 x x
id (X, homX, idX, compX) = idX

Its composition law (or transitivity):

comp : (𝓧 : Σ S) (x y z : Ob 𝓧) → hom 𝓧 x y → hom 𝓧 y z → hom 𝓧 x z
comp (X, homX, idX, compX) = compX

Notice that we have the so-called diagramatic order for composition.

The functoriality of a pair F, 𝓁 (where in category theory the latter is also written F, by an abuse of notation) says that 𝓁 preserves identities and composition:

functorial : (𝓧 A : Σ S)
→ (F : Ob 𝓧 → Ob A)
→ ((x y : Ob 𝓧) → hom 𝓧 x y → hom A (F x) (F y))
→ 𝓤 ⊔ 𝓥⁺

functorial 𝓧 A F 𝓁' = pidentity × pcomposition
where

In order to express the preservation of identities and composition in traditional form, we first define, locally, symbols for composition in applicative order, making the objects implicit:

_ o_ : {x y z : Ob 𝓧} → hom 𝓧 x y z → hom 𝓧 x y → hom 𝓧 x z
g o f = comp 𝓧 _ _ _ f g

_ □_ : {a b c : Ob A} → hom A b c → hom A a b → hom A a c
g □ f = comp A _ _ _ f g

And we also make implicit the object parameters of the action of the functor on arrows:

𝓕 : {x y : Ob 𝓧} → hom 𝓧 x y → hom A (F x) (F y)
𝓕 f = 𝓁' _ _ _ f

Preservation of identities:

pidentity = (λ x → 𝓁 (id 𝓧 x)) ≡ (λ x → id A (F x))

Preservation of composition:

pcomposition = (λ x y z (f : hom 𝓧 x y) (g : hom 𝓧 y z) → 𝓁 (g o f))
≡ (λ x y z (f : hom 𝓧 x y) (g : hom 𝓧 y z) → 𝓁 g □ 𝓁 f)

This time we will need two steps to characterize equality of type-valued preorders. The first one is as above, by considering a standard notion of structure:

sns-data : SNS S (𝓤 ⊔ (𝓥⁺))
sns-data = (ι, ρ, θ)
where
ι : (𝓧 A : Σ S) → (𝓧') = (A) → 𝓤 ⊔ (𝓥⁺)
The above constructions are short thanks to computations-under-the-hood performed by Agda, and may require some effort to unravel.

The above automatically gives a characterization of equality of preorders. But this characterization uses another equality, of hom types. The second step translates this equality into an equivalence:

\[
\text{lemma } \lambda A \alpha : \Sigma \delta \eta \gamma.
\text{Functorial } \lambda A \alpha F (\lambda x y \rightarrow \text{transport} (\lambda \rightarrow x y) p)
\]

\[
\theta : \{X : \mathcal{U} \} \rightarrow \text{is-equiv} (\text{canonical-map} \lambda p \rightarrow s)
\]

\[
\theta \{x\} (\text{hom} X, \text{id} X, \text{comp} X) (\text{hom} A, \text{id} A, \text{comp} A) = g
\]

where

\[
\phi = \text{canonical-map} \lambda p (\text{hom} X, \text{id} X, \text{comp} X) (\text{hom} A, \text{id} A, \text{comp} A)
\]

\[
y : \text{codomain} \phi \rightarrow \text{domain} \phi
\]

\[
y (\text{refl } _, \text{refl } _, \text{refl } _) = \text{refl } _{}
\]

\[
\eta : y \circ \phi \sim \text{id}
\]

\[
\eta (\text{refl } _) = \text{refl } _{}
\]

\[
\epsilon : \phi \circ y \sim \text{id}
\]

\[
\epsilon (\text{refl } _, \text{refl } _, \text{refl } _) = \text{refl } _{}
\]

\[
g : \text{is-equiv} \phi
\]

\[
g = \text{invertibles-are-equis} \phi (y, \eta, \epsilon)
\]

Here Agda silently performs a laborious computation to accept the definition of item \(v\):

\[
v : (p : \text{hom} \lambda A \alpha \equiv \lambda x y \rightarrow \text{hom} A (F x) (F y))
\]

\[
\rightarrow \text{functorial } \lambda A \alpha F (\lambda x y \rightarrow \text{transport} (\lambda \rightarrow x y) p)
\]

\[
\Rightarrow \text{functorial } \lambda A \alpha F (\text{pr}_1 (r e ^ \gamma p))
\]

\[
v (\text{refl } _) = \text{Id-Eq } _- (\text{refl } _{})
\]

\[
y =
\]

\[
(\Sigma p : \text{hom} \lambda A \alpha \equiv \lambda x y \rightarrow \text{hom} A (F x) (F y))
\]

\[
, \text{functorial } \lambda A \alpha F (\lambda x y \rightarrow \text{transport} (\lambda \rightarrow x y) p) \equiv (vi)
\]

\[
(\Sigma p : \text{hom} \lambda A \alpha \equiv \lambda x y \rightarrow \text{hom} A (F x) (F y))
\]

\[
, \text{functorial } \lambda A \alpha F (\text{pr}_1 (r e ^ \gamma p)) \equiv (vii)
\]
Recall that module that again we need to combine two steps. The second step is the same, but the first step is modified to add axioms.

Now we consider type-valued preorders subject to arbitrary axioms. The only reason we need to consider this explicitly is identification to the identity functor.

Exercise. The above equivalence is characterized by induction on identifications as the function that maps the reflexive identification to the identity functor.

Combining the two steps, we get the following characterization of equality of type-valued preorders in terms of equivalences:

characterization-of-type-valued-preorder-≡:

\[
\begin{align*}
(\mathcal{X} \equiv \mathcal{A}) & \quad \Rightarrow (\mathcal{X} \equiv \mathcal{A}) \\
= & \quad (\Sigma \mathcal{F} : (\text{Ob } \mathcal{X} \to \text{Ob } \mathcal{A})) \\
& \quad \quad \quad \text{is-equiv } \mathcal{F} \\
& \quad \quad \quad \times (\Sigma p : \text{hom } \mathcal{X} \equiv (\lambda x y \to \text{hom } \mathcal{A} (F x) (F y))) \\
& \quad \quad \quad \quad \quad \text{funcrional } \mathcal{X} \mathcal{A} F (\lambda x y \to \text{transport } (\lambda - x) p)) \Rightarrow (i) \\
(\Sigma \mathcal{F} : (\text{Ob } \mathcal{X} \to \text{Ob } \mathcal{A})) & \quad \quad \quad \text{is-equiv } \mathcal{F} \\
& \quad \quad \quad \times (\Sigma \mathcal{F} : ((x y : \text{Ob } \mathcal{X}) \to \text{hom } \mathcal{X} x y \to \text{hom } \mathcal{A} (F x) (F y))) \\
& \quad \quad \quad \quad \quad \text{is-equiv } (\mathcal{F} x y) \\
& \quad \quad \quad \quad \quad \times \text{funcrional } \mathcal{X} \mathcal{A} F \mathcal{F}) \quad \Rightarrow (ii)
\end{align*}
\]

where
\[
\begin{align*}
i & = \text{characterization-of-≡ (ua } \mathcal{U} \text{ sns-data } \mathcal{X} \mathcal{A} \\
ii & = \Sigma \text{-cong } (\lambda F \to \Sigma \text{-cong } (\lambda - \text{lemma } \mathcal{X} \mathcal{A} F))
\end{align*}
\]

Exercise. The above equivalence is characterized by induction on identifications as the function that maps the reflexive identification to the identity functor.

Now we consider type-valued preorders subject to arbitrary axioms. The only reason we need to consider this explicitly is that again we need to combine two steps. The second step is the same, but the first step is modified to add axioms.

module type-valued-preorder-with-axioms

(\mathcal{U} \forall \mathcal{W} : \text{Universe})
(\ua : \text{Univalence})
\text{axioms } (X : \mathcal{U} \uparrow) \to \text{type-valued-preorder-\mathcal{S} } \{\mathcal{U}\} \{\mathcal{V}\} X \to \mathcal{W} \uparrow
\text{axioms} \text{ss } (X : \mathcal{U} \uparrow) (s : \text{type-valued-preorder-\mathcal{S} } X) \to \text{is-subsingleton } (\text{axioms } X s))

where

open sip
open sip-with-axioms
open type-valued-preorder \mathcal{U} \forall \ua

\begin{align*}
\mathcal{S}' & : \mathcal{U} \uparrow \to \mathcal{U} \uparrow (\forall \mathcal{V} \uparrow) \uparrow \mathcal{W} \\
\mathcal{S}' : & = \Sigma S : S \times, \text{axioms } X s
\end{align*}

\text{sns-data}' : \text{SNS } \mathcal{S}' (\mathcal{U} \cup (\forall \mathcal{V} \uparrow))
\text{sns-data}' = \text{add-axioms } \text{axioms } \text{axioms ss } \text{sns-data}

Recall that [...] is the map that forgets the axioms.
We now characterize equality of categories. By choosing suitable axioms for type-valued preorders, we get categories:

**module category**

\[(\mathcal{U}, \mathcal{V}) : \text{Universe}\]

\[(\text{ua} : \text{Univalence})\]

\[\text{where}\]

\[\text{open type-valued-preorder-with-axioms } \mathcal{U} \uplus (\mathcal{U} \uplus \mathcal{V}) \text{ ua}\]

\[\text{fe} : \text{global-dfunext}\]

\[\text{fe} = \text{univalence-gives-global-dfunext } \text{ua}\]

\[\text{S} : \mathcal{U} \rightarrow \mathcal{U} \uplus (\mathcal{V} \uplus)\]

\[\text{S} = \text{type-valued-preorder-S } \{\mathcal{U}\} \{\mathcal{V}\}\]

The axioms say that

1. the homs form sets, rather than arbitrary types,
2. the identity is a left and right neutral element of composition,
3. composition is associative.

**category-axioms** : \((X : \mathcal{U} \uplus) \rightarrow S X \rightarrow \mathcal{U} \uplus \mathcal{V}\)

**category-axioms** \(X \ (\text{homX} , \text{idX} , \text{compX}) = \text{hom-sets} \times \text{identityl} \times \text{identityr} \times \text{associativity}\)

\[\text{where}\]

\[\_o_ : \{x y z : X\} \rightarrow \text{homX} y z \rightarrow \text{homX} x y \rightarrow \text{homX} x z\]

\[\text{g o f} = \text{compX} _o f g\]

\[\text{hom-sets} = \forall x y \rightarrow \text{is-set} (\text{homX} x y)\]

\[\text{identityl} = \forall x y (f : \text{homX} x y) \rightarrow f o (\text{idX} x) \equiv f\]

\[\text{identityr} = \forall x y (f : \text{homX} x y) \rightarrow (\text{idX} y) o f \equiv f\]

\[\text{associativity} = \forall x y z t (f : \text{homX} x y) (g : \text{homX} y z) (h : \text{homX} z t)\]

\[\rightarrow (h o g) o f \equiv h o (g o f)\]

The first axiom is subsingleton valued because the property of being a set is a subsingleton type. The second and the third axioms are subsingleton valued in the presence of the first axiom, because equations between elements of sets are subsingletons, by definition of set. And because subsingletons are closed under products, the category axioms form a subsingleton type:
We reuse of above names in a slightly different way, taking into account that now we have axioms, which we simply ignore:

\[
\text{is-functorial} \quad \text{comp} \quad \mathbb{I}_\mathbb{D}
\]

\[
\text{Ob} \quad \text{Comp} \quad \text{Cat}
\]

\[
\text{category-axioms-subsingleton} \quad \text{category-axioms-subsingleton}
\]

\[
\text{where}
\]

\[
i : \forall x y \to \text{is-set} \ (\text{hom}_X \ x \ y)
\]

\[
i = \text{pr}_1 \ ca
\]

\[
y : \text{is-subsingleton} \ (\text{category-axioms} \ X \ (\text{hom}_X \ , \ \text{id}_X \ , \ \text{comp}_X))
\]

\[
y = \text{x-is-subsingleton} \ ss \ (\text{x-is-subsingleton} \ ls \ (\text{x-is-subsingleton} \ rs \ as))
\]

\[
\text{where}
\]

\[
ss = \text{P-is-subsingleton} \ fe
\]

\[
ls = \text{P-is-subsingleton} \ fe
\]

\[
rs = \text{P-is-subsingleton} \ fe
\]

\[
\text{as} = \text{P-is-subsingleton} \ fe
\]

\[
\text{where}
\]

\[
\text{where}
\]

We are now ready to define the type of categories, as a subtype of that of type-valued preorders:

\[
\text{Cat} = (\text{X} : X : \text{Cat}) \to \text{Ob} \ \text{X} \to \text{Ob} \ \text{X} \to \text{X}
\]

\[
\text{Cat} = \Sigma \ X : \text{Cat} \ , \ \Sigma \ s : S \ X , \ \text{category-axioms} \ X \ s
\]

We reuse of above names in a slightly different way, taking into account that now we have axioms, which we simply ignore:

\[
\text{Ob} : \text{Cat} \to \text{X}
\]

\[
\text{Ob} (X, (\text{hom}_X \ , \ \text{id}_X \ , \ \text{comp}_X), \_) = X
\]

\[
\text{hom} : (\text{X} : \text{Cat}) \rightarrow \text{Ob} \ \text{X} \to \text{Ob} \ \text{X} \to \\text{X}
\]

\[
\text{hom} (X, (\text{hom}_X \ , \ \text{id}_X \ , \ \text{comp}_X), \_) = \text{hom}_X
\]

\[
\text{id} : (\text{X} : \text{Cat}) \ (x : \text{Ob} \ \text{X}) \to \text{hom} \ \text{X} \times x
\]

\[
\text{id} (X, (\text{hom}_X \ , \ \text{id}_X \ , \ \text{comp}_X), \_) = \text{id}_X
\]

\[
\text{comp} : (\text{X} : \text{Cat}) \ (x \, y \, z : \text{Ob} \ \text{X}) \ (f : \text{hom} \ \text{X} \times y) \ (g : \text{hom} \ \text{X} \times z) \to \text{hom} \ \text{X} \times x
\]

\[
\text{comp} (X, (\text{hom}_X \ , \ \text{id}_X \ , \ \text{comp}_X), \_) = \text{comp}_X
\]

\[
\text{is-functorial} : (\text{X} \ A : \text{Cat})
\]

\[
\rightarrow (\text{F} : \text{Ob} \ \text{X} \to \text{Ob} \ \text{A})
\]

\[
\rightarrow ((x \, y : \text{Ob} \ \text{X}) \to \text{hom} \ \text{X} \times y \to \text{hom} \ \text{A} \ (\text{F} \ x) \ (\text{F} \ y))
\]

\[
\rightarrow \text{U} \ \text{U} \ \text{V}
\]

\[
\text{is-functorial} \ \text{X} \ \text{A} \ \text{F} \ \text{F}' = \text{pidentity} \times \ \text{pcomposition}
\]

\[
\text{where}
\]

\[
\text{pidentity} \quad = \ (\lambda x \to \text{F} \ (\text{id} \ \text{X} \ x)) \equiv (\lambda x \to \text{id} \ \text{A} \ (\text{F} \ x))
\]

\[
\text{id}_\text{X} = \text{id}_\text{X} \ (\lambda x \to \text{F} \ (\text{id} \ \text{X} \ x)) \equiv (\lambda x \to \text{id} \ \text{A} \ (\text{F} \ x))
\]

\[
\text{comp}_\text{X} = \text{comp}_\text{X} \ (\lambda x \to \text{F} \ (\text{id} \ \text{X} \ x)) \equiv (\lambda x \to \text{id} \ \text{A} \ (\text{F} \ x))
\]

\[
\text{pidentity} \quad = \ (\lambda x \to \text{F} \ (\text{id} \ \text{X} \ x)) \equiv (\lambda x \to \text{id} \ \text{A} \ (\text{F} \ x))
\]

\[
\text{id}_\text{X} = \text{id}_\text{X} \ (\lambda x \to \text{F} \ (\text{id} \ \text{X} \ x)) \equiv (\lambda x \to \text{id} \ \text{A} \ (\text{F} \ x))
\]

\[
\text{comp}_\text{X} = \text{comp}_\text{X} \ (\lambda x \to \text{F} \ (\text{id} \ \text{X} \ x)) \equiv (\lambda x \to \text{id} \ \text{A} \ (\text{F} \ x))
\]

\[
\text{pidentity} \quad = \ (\lambda x \to \text{F} \ (\text{id} \ \text{X} \ x)) \equiv (\lambda x \to \text{id} \ \text{A} \ (\text{F} \ x))
\]
This says that if we have a function from \( \text{is-inhabited} \) to a subsingleton, relying on function extensionality.

The following is Voevodsky’s approach to saying that a type is inhabited in such a way that the statement of inhabitation is a type. The purpose of this section is to make this precise.

**Subsingleton truncation**

The subsingleton truncation of a type \( X \), if it exists, is the universal solution of the problem of mapping \( X \) into a subsingleton type. The purpose of this section is to make this precise.

**Voevodsky’s approach to 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, relying on function extensionality.

\[
\text{is-inhabited} : \mathcal{U} \rightarrow \mathcal{U} \quad \text{is-inhabited} \{U\} X = (P : \mathcal{U}) \rightarrow \text{is-subsingleton} \quad P \rightarrow (X \rightarrow P) \rightarrow P
\]

This says that if we have a function from \( X \) to a subsingleton \( P \), then \( P \) must have a point. So this fails when \( X = \emptyset \). Considering \( P = \emptyset \), we conclude that \( \leadsto X \) if \( X \) is inhabited, which says that \( X \) is non-empty.
For simplicity in the formulation of the theorems, we assume *global* function extensionality. A type can be pointed in many ways, but inhabited in at most one way:

\[
\text{inhabitation-is-subsingleton} : \text{global-dfunext} \to (X : \mathcal{U}) \to \text{is-subsingleton} (\text{is-inhabited} X)
\]

\[
\text{inhabitation-is-subsingleton} \; \text{fe} \; X = \Pi-\text{is-subsingleton} \; \text{fe} \quad \quad \quad \quad \quad (\lambda \; P \to \Pi-\text{is-subsingleton} \; \text{fe}) \quad \quad \quad \quad \quad (\lambda \; (s : \Pi-\text{is-subsingleton} \; P) \to \Pi-\text{is-subsingleton} \; \text{fe})
\]

The following is the introduction rule for inhabitation, which says that pointed types are inhabited:

\[
\text{inhabited-intro} : \{X : \mathcal{U}\} \to X \to \text{is-inhabited} \; X
\]

\[
\text{inhabited-intro} \; x = \lambda \; P \; s \; f \to f \; x
\]

And its recursion principle:

\[
\text{inhabited-recursion} : \{X \; P : \mathcal{U}\} \to \text{is-subsingleton} \; P \to (X \to P) \to \text{is-inhabited} \; X \to P
\]

\[
\text{inhabited-recursion} \; s \; f \; \varphi = \varphi \; (\text{codomain} \; f) \; s \; f
\]

And its computation rule:

\[
\text{inhabited-recursion-computation} : \{X \; P : \mathcal{U}\}
\]

\[
\quad \quad \quad \quad \quad (i : \text{is-subsingleton} \; P)
\quad \quad \quad \quad \quad (f : X \to P)
\quad \quad \quad \quad \quad (x : X)
\]

\[
\to \text{inhabited-recursion} \; i \; f \; (\text{inhabited-intro} \; x) \equiv f \; x
\]

\[
\text{inhabited-recursion-computation} \; i \; f \; x = \text{refl} \; (f \; x)
\]

So the point \(x\) inside \text{inhabited} \; x\ is available for use by any function \(f\) into a subsingleton, via \text{inhabited-recursion}.

We can derive induction from recursion in this case, but its “computation rule” holds up to an identification, rather than judgmentally:

\[
\text{inhabited-induction} : \text{global-dfunext}
\]

\[
\to \{X : \mathcal{U}\} \{P : \text{is-inhabited} \; X \to \mathcal{U}\}
\]

\[
\quad \quad \quad \quad \quad (i : (s : \text{is-inhabited} \; X) \to \text{is-subsingleton} \; (P \; s))
\quad \quad \quad \quad \quad (f : (x : X) \to P \; (\text{inhabited-intro} \; x))
\]

\[
\to (s : \text{is-inhabited} \; X) \to P \; s
\]

\[
\text{inhabited-induction} \; \text{fe} \; \{X\} \; \{P\} \; i \; f \; s = \varphi' \; s
\]

where

\[
\varphi : X \to P \; s
\]

\[
\varphi \; x = \text{transport} \; P \; (\text{inhabitation-is-subsingleton} \; \text{fe} \; X \; (\text{inhabited-intro} \; x) \; s) \; (f \; x)
\]

\[
\varphi' : \text{is-inhabited} \; X \to P \; s
\]

\[
\varphi' = \text{inhabited-recursion} \; (i \; s) \; \varphi
\]

\[
\text{inhabited-computation} : (\text{fe} : \text{global-dfunext}) \{X : \mathcal{U}\} \{P : \text{is-inhabited} \; X \to \mathcal{U}\}
\]

\[
\quad \quad \quad \quad \quad (i : (s : \text{is-inhabited} \; X) \to \text{is-subsingleton} \; (P \; s))
\quad \quad \quad \quad \quad (f : (x : X) \to P \; (\text{inhabited-intro} \; x))
\quad \quad \quad \quad \quad (x : X)
\]

\[
\to \text{inhabited-induction} \; \text{fe} \; i \; f \; (\text{inhabited-intro} \; x) \equiv f \; x
\]

\[
\text{inhabited-computation} \; \text{fe} \; i \; f \; x = i \; (\text{inhabited-intro} \; x)
\]

\[
(\text{inhabited-induction} \; \text{fe} \; i \; f \; (\text{inhabited-intro} \; x))
\]

\[
(f \; x)
\]

The definition of inhabitation looks superficially like double negation. However, although we don’t necessarily have that \(\neg \neg P \to P\), we do have that \text{is-inhabited} \; P \to P\ if \(P\) is a subsingleton.

\[
\text{inhabited-subsingletons-are-pointed} : (P : \mathcal{U}) \to \text{is-subsingleton} \; P \to \text{is-inhabited} \; P \to P
\]

\[
\text{inhabited-subsingletons-are-pointed} \; P \; s = \text{inhabited-recursion} \; s \; (\text{id} \; P)
\]
Exercise. Show that \(\text{is-inhabited } X \iff \neg \neg X\) if and only if excluded middle holds.

\[
\text{inhabited-functorial} : \text{global-dfunext} \to (X : U') (Y : U') \to (X \to Y) \to \text{is-inhabited } X \to \text{is-inhabited } Y
\]

\[
\text{inhabited-functorial } fe \ X \ Y \ f = \text{inhabited-recursion} (\text{inhabitation-is-subsingleton} \ fe \ Y) (\text{inhabited-intro} \circ f)
\]

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:

\[
\text{image'} : \{X : U\} \{Y : V\} \to (X \to Y) \to (U \sqcup V)^{'}
\]

\[
\text{image'} f = \Sigma y : \text{codomain } f', \text{is-inhabited} (\Sigma x : \text{domain } f', f x \equiv y)
\]

An attempt to define the image of \(f\) without the inhabitation predicate would be to take it to be

\[
\Sigma y : \text{codomain } f, \Sigma x : \text{domain } f, f x \equiv y.
\]

But we already know that this is equivalent to \(X\). This is similar to what happens in set theory: the graph of any function is in bijection with its domain.

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

\[
\text{restriction'} : \{X : U\} \{Y : V\} (f : X \to Y) \to \text{image'} f \to Y
\]

\[
\text{restriction'} f (y, _) = y
\]

\[
\text{corestriction'} : \{X : U\} \{Y : V\} (f : X \to Y) \to X \to \text{image'} f
\]

\[
\text{corestriction'} f x = f x, \text{inhabited-intro} (x, \text{refl} (f x))
\]

And we can define the notion of surjection as follows:

\[
\text{is-surjection'} : \{X : U\} \{Y : V\} \to (X \to Y) \to (U \sqcup V)^{'}
\]

\[
\text{is-surjection'} f = (y : \text{codomain } f) \to \text{is-inhabited} (\Sigma x : \text{domain } f, f x \equiv y)
\]

Exercise. The type \((y : \text{codomain } f) \to \Sigma x : \text{domain } f, f x \equiv y\) is equivalent to the type \(\text{has-section } f\), 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 \(X \to \text{is-inhabited } X\) is a surjection, or that \(X \to Y\) gives \(\text{is-inhabited } X \to \text{is-inhabited } Y\) for \(X\) and \(Y\) in arbitrary universes.

There are two proposed ways to solve this kind of problem:

1. 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.

2. 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 \( X \parallel \) for the inhabitation of \( X \), called the propositional, or subsingleton, or truth-value, truncation of \( X \):

\[
\text{record subsingleton-truncations-exist : \( \mathbb{U} \) where}
\]

\[
\begin{align*}
\text{field} & \quad : \{U : \text{Universe}\} \to U \to U' \\
\text{is-subsingleton} & \quad : \{U : \text{Universe}\} \{X : U'\} \to is-subsingleton \ X \parallel X \\
\text{-recursion} & \quad : \{U \parallel : \text{Universe}\} \{X : U'\} \{P : \parallel \parallel \} \\
& \quad \to is-subsingleton \ P \to (X \to P) \to X \parallel P
\end{align*}
\]

\[
\text{infix} \ 0 \parallel \parallel
\]

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

We now assume that subsingleton truncations exist in the next few constructions, and we open the assumption to make the above fields visible.

\[
\text{module basic-truncation-development}
\]

\[
\begin{align*}
\text{(pt : subsingleton-truncations-exist)} \\
\text{(hfe : global-hfunext)} \\
\text{where}
\end{align*}
\]

\[
\text{open subsingleton-truncations-exist pt public}
\]

\[
\text{hunapply : \{X : U'\} \{A : X \to \parallel \parallel\} \{f g : \prod A\} \to f \sim g \to f \equiv g}
\]

\[
\text{hunapply = hfunext-gives-dfunext hfe}
\]

\[
\begin{align*}
\text{||-recursion-computation} & \quad : \{X : U'\} \{P : \parallel \parallel\} \\
& \quad \to (i : is-subsingleton \ P) \\
& \quad \to (f : X \to P) \\
& \quad \to (x : X) \\
& \quad \to \text{||-recursion} \ i \ f \mid x \mid \equiv f \ x
\end{align*}
\]

\[
\text{||-recursion-computation} \ i \ f \ x = i \ (\text{||-recursion} \ i \ f \mid x \mid) \ (f \ x)
\]

\[
\begin{align*}
\text{||-induction} & \quad : \{X : U'\} \{P : X \parallel \parallel \} \\
& \quad \to (s : X \parallel \parallel \parallel) \\
& \quad \to (x : X) \\
& \quad \to P \ s
\end{align*}
\]

\[
\text{||-induction \ {U} \ {V} \ {X} \ {P} \ i \ f \ s = \varphi' \ s}
\]

\[
\text{where}
\]

\[
\begin{align*}
\varphi & : X \to P \ s \\
\varphi \ x & = \text{transport} \ P \ (\text{||-is-subsingleton} \ x \mid s \mid) \ (f \ x) \\
\varphi' & : X \parallel \parallel \parallel \to P \ s \\
\varphi' & = \text{||-recursion} \ (i \ s) \ \varphi
\end{align*}
\]

\[
\begin{align*}
\text{||-computation} & \quad : \{X : U'\} \{P : X \parallel \parallel \} \\
& \quad \to (i : (s : X \parallel \parallel) \to is-subsingleton \ (P \ s)) \\
& \quad \to (f : (X : X) \to P \mid x \mid) \\
& \quad \to (x : X) \\
& \quad \to \text{||-induction} \ i \ f \mid x \mid \equiv f \ x
\end{align*}
\]

\[
\text{||-computation} \ i \ f \ x = i \mid x \mid \ (\text{||-induction} \ i \ f \mid x \mid) \ (f \ x)
\]

\[
\text{||-functor} : \{X : U'\} \{Y : \parallel \parallel\} \to (X \to Y) \to X \parallel \parallel Y \\
\text{||-functor} \ f = \text{||-recursion} \ ||-is-subsingleton \ (\lambda x \to f \ x \mid)
\]

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

\[
\begin{align*}
\text{||-agrees-with-inhabitation} & : (X : U') \to X \parallel \iff \text{is-inhabited} \ X \\
\text{||-agrees-with-inhabitation} & \ X = a , \ b
\end{align*}
\]

\[\text{where}
\]

\[
\begin{align*}
a & : X \parallel \to \text{is-inhabited} \ X
\end{align*}
\]
a = \text{-recursion} (\text{inhabitation-is-subsingleton} \ \text{hunapply} \ X) \ \text{inhabited-intro}

b : \text{is-inhabited} \ X \to \parallel X \parallel
b = \text{inhabited-recursion} \parallel-is-subsingleton \ |_|

Hence they differ only in size, and when size matters don’t get on the way, we can use \text{is-inhabited} instead of \parallel_\parallel if we wish.

Table of contents ✆

Disjunction and existence

Disjunction and existence are defined as the truncation of \( + \) and \( \Sigma \):

\[
\_\_ : \mathcal{U} \times \mathcal{V} \to \mathcal{U} \sqcup \mathcal{V}
\]

\[
\_\_ A \_\_ B = \parallel A + B \parallel
\]

\[
\text{infixl 20 \_\_}
\]

\[
\exists : \{X : \mathcal{U}\} \to (X \to \mathcal{V}) \to \mathcal{U} \sqcup \mathcal{V}
\]

\[
\exists A \_\_ = \parallel \Sigma A \parallel
\]

\[
\neg \exists : \{\mathcal{U} \sqcup \mathcal{V} : \text{Universe}\} (X : \mathcal{U}) (Y : X \to \mathcal{V}) \to \mathcal{U} \sqcup \mathcal{V}
\]

\[
\neg \exists X Y = \exists Y
\]

\[
\text{syntax} \ \neg \exists A (\lambda x \to b) = \exists x : A , b
\]

\[
\text{infixr -1 \neg -}\exists
\]

\[
\text{\_\_is-subsingleton} : \{A : \mathcal{U}\} \{B : \mathcal{V}\} \to \text{is-subsingleton} (A \_\_ B)
\]

\[
\text{\_\_is-subsingleton} = \|\_\parallel-is-subsingleton
\]

\[
\exists\text{-is-subsingleton} : \{X : \mathcal{U}\} \{A : X \to \mathcal{V}\} \to \text{is-subsingleton} (\exists A)
\]

\[
\exists\text{-is-subsingleton} = \|\_\parallel-is-subsingleton
\]

The author’s slides on \text{univalent logic} discuss further details about these notions of disjunction and existence.

Table of contents ✆

Images and surjections

The image of a function \( f : X \to Y \) is the type of \( y : Y \) for which there exists \( x : X \) with \( f x \equiv y \).

\[
\text{image} : \{X : \mathcal{U}\} \{Y : \mathcal{V}\} \to (X \to Y) \to \mathcal{U} \sqcup \mathcal{V}
\]

\[
\text{image} f = \Sigma y : \text{codomain} f , \exists x : \text{domain} f , f x \equiv y
\]

\[
\text{restriction} : \{X : \mathcal{U}\} \{Y : \mathcal{V}\} (f : X \to Y)
\]

\[
\to \text{image} \ f \to Y
\]

\[
\text{restriction} \ f \ (y , \_\_) = y
\]

\[
\text{corestriction} : \{X : \mathcal{U}\} \{Y : \mathcal{V}\} (f : X \to Y)
\]

\[
\to X \to \text{image} \ f
\]

\[
\text{corestriction} \ f \ x = f \ x , | (x , \text{refl} \ f x) |
\]

\[
\text{is-surjection} : \{X : \mathcal{U}\} \{Y : \mathcal{V}\} \to (X \to Y) \to \mathcal{U} \sqcup \mathcal{V}
\]

\[
\text{is-surjection} \ f = (y : \text{codomain} f) \to \exists x : \text{domain} f , f x \equiv y
\]

\[
\text{being-surjection-is-subsingleton} : \{X : \mathcal{U}\} \{Y : \mathcal{V}\} (f : X \to Y)
\]

\[
\to \text{is-subsingleton} (\text{is-surjection} \ f)
\]

\[
\text{being-surjection-is-subsingleton} \ f = \Pi\text{-is-subsingleton} \ \text{hunapply}
\]
corestriction-surjection : \( \{ X : U \} \) \( \{ Y : V \} \) \((f : X \to Y) \to \text{is-surjection} \) (corestriction \( f \))

\[
\text{corestriction-surjection} \ f \ (y \ , \ s) = \text{\|\|-functor} \ g \ s
\]

where
\[
g : (\Sigma \; x : \text{domain} \; f \ , \ f \; x \equiv \; y) \to \Sigma \; x : \text{domain} \; f \ , \ \text{corestriction} \; f \; x \equiv \; (y \ , \ s)
g \; (x \ , \ p) = x , \; \text{to-subtype-=} \ (\lambda \; _\_ \to \exists \text{-is-subsingleton}) \; p
\]

surjection-induction : \( \{ X : U \} \) \( \{ Y : V \} \) \((f : X \to Y) \to \text{is-surjection} \) \( f \)

\[
\text{surjection-induction} \ f \ i \; P \ j \; \alpha \; y = \text{\|\|-recursion} \ (j \; y) \; \varphi \ (i \; y)
\]

where
\[
\varphi : \text{fiber} \; f \; y \to P \; y
\]
\[
\varphi \; (x \ , \ r) = \text{transport} \; P \; r \; (\alpha \; x)
\]

Exercise. A map is an equivalence if and only if it is both an embedding and a surjection. (To be solved shortly.)

This time we can prove that the map \( x \mapsto \mid x \mid \) of \( X \) into \( \| X \| \) is a surjection without the universe levels getting in our way:

\[
\text{\|\|-is-surjection} : (X : U) \to \text{is-surjection} \; (\lambda \; (x : X) \to \mid x \mid)
\]

\[
\text{\|\|-is-surjection} \; X \; s = y
\]

where
\[
f : X \to \exists \; x : X \ , \ \mid x \mid \equiv s
f \; x = \mid (x \ , \ \text{\|\|-is-subsingleton} \; \mid x \mid) \mid
\]
\[
y : \exists \; x : X \ , \ \mid x \mid \equiv s
\]
\[
y = \text{\|\|-recursion} \; \text{\|\|-is-subsingleton} \; f \; s
\]

Saying that this surjection \( X \to \| X \| \) has a section for all \( X \) (we can pick a point of every inhabited type) amounts to global choice, which contradicts univalence, and also gives classical logic.

The subsingleton truncation of a type is also known as its support, and a type \( X \) is said to have split support if there is a choice function \( \| X \| \to X \), which is automatically a section of the surjection \( X \to \| X \| \).

Exercise. Show that a type has split support if and only it is logically equivalent to a subsingleton. In particular, the type of invertibility data has split support, as it is logically equivalent to the equivalence property.

Exercise (hard). If \( X \) and \( Y \) are types obtained by summing \( x \)- and \( y \)-many copies of the type \( 1 \), respectively, as in \( 1 + 1 + \ldots + 1 \), where \( x \) and \( y \) are natural numbers, then \( \| X \equiv Y \| \equiv (x \equiv y) \) and the type \( X \equiv X \) has \( x! \) elements.

Table of contents ⇩

A characterization of equivalences

Singletons are inhabited, of course:

\[
\text{singleton-are-inhabited} : (X : U) \to \text{is-singleton} \; X
\]

\[
\text{singleton-are-inhabited} \; X \; s = \mid \text{center} \; X \; s \mid
\]

And inhabited subsingletons are singletons:

\[
\text{inhabited-subsingletons-are-singletons} : (X : U) \to \| X \| \to \text{is-subsingleton} \; X
\]
inhabited-subsingletons-are-singletons \( X \vdash i = c, \varphi \)

where

\[
c : X
\]

\[
c = \text{\textbackslash l\textbackslash -recursion } i (id \ X) t
\]

\[
\varphi : (x : X) \rightarrow c \equiv x
\]

\[
\varphi = i \ c
\]

Hence a type is a singleton if and only if it is inhabited and a subsingleton:

\[
\text{singleton-iff-inhabited-subsingleton} : (X : \mathcal{U}^\triangledown) \rightarrow \text{is-singleton } X \iff (\parallel X \parallel \times \text{is-subsingleton } X)
\]

By considering the unique map \( X \rightarrow 1 \), this can be regarded as a particular case of the fact that a map is an equivalence if and only if it is both an embedding and a surjection:

\[
\text{equiv-iff-embedding-and-surjection} : \{X : \mathcal{U}^\triangledown\} \{Y : \mathcal{V}^\triangledown\} (f : X \rightarrow Y) \rightarrow \text{is-equiv } f \iff (\text{is-embedding } f \times \text{is-surjection } f)
\]

\[
\text{equiv-iff-embedding-and-surjection } f = (a, b)
\]

where

\[
a : \text{is-equiv } f \rightarrow \text{is-embedding } f \times \text{is-surjection } f
\]

\[
a e = (\lambda y \rightarrow \text{singletons-are-subsingletons } (\text{fiber } f y) (e y)) ,
\]

\[
(\lambda y \rightarrow \text{singletons-are-inhabited } (\text{fiber } f y) (e y))
\]

\[
b : \text{is-embedding } f \times \text{is-surjection } f \rightarrow \text{is-equiv } f
\]

\[
b (e, s) y = \text{inhabited-subsingletons-are-singletons } (\text{fiber } f y) (s y) (e y)
\]

\[
\text{equiv-≡-embedding-and-surjection} : \{X : \mathcal{U}^\triangledown\} \{Y : \mathcal{V}^\triangledown\} (f : X \rightarrow Y) \rightarrow \text{propext } (\mathcal{U} \sqcup \mathcal{V}) \rightarrow \text{is-equiv } f \equiv (\text{is-embedding } f \times \text{is-surjection } f)
\]

\[
\text{equiv-≡-embedding-and-surjection } f \ pe =
\pe (\text{being-equiv-is-subsingleton } \text{hunapply } \text{hunapply } f)
\]

\[
(\text{is-subsingleton })
\]

\[
(\text{being-embedding-is-subsingleton } \text{hunapply } f)
\]

\[
(\text{being-surjection-is-subsingleton } f))
\]

\[
(\text{lr-implication } (\text{equiv-iff-embedding-and-surjection } f))
\]

\[
(\text{rl-implication } (\text{equiv-iff-embedding-and-surjection } f))
\]

Table of contents

Exiting subsingleton truncations

We will see that global choice

\[
(X : \mathcal{U}^\triangledown) \rightarrow \parallel X \parallel \rightarrow X
\]

is inconsistent with univalence, and also implies excluded middle. However, for some types \( X \), we can prove that \( \parallel X \parallel \rightarrow X \).

We characterize such types as those that have \text{wconstant} endomaps.

Because, as we have seen, we have a logical equivalence

\[
\parallel X \parallel \leftrightarrow \text{is-inhabited } X,
\]

it suffices to discuss
which can be done in our spartan MLTT without any axioms for univalent mathematics (and hence also with axioms for univalent mathematics, including non-constructive ones such as excluded middle and choice).

For any type \( X \), we have \( \text{is-inhabited} \ X \rightarrow X \) iff \( X \) has a designated \( \text{wconstant endomap} \). To prove this we first show that the type of fixed points of a \( \text{wconstant} \) endomap is a subsingleton.

We first define the type of fixed points:

\[
\text{fix} : \{X : U \} \rightarrow (X \rightarrow X) \rightarrow U
\]

\[
\text{fix} f = \Sigma x : \text{domain} f \, , \, f \, x \equiv x
\]

Of course any fixed point of \( f \) gives an element of \( X \):

\[
\text{from-fix} : \{X : U \} (f : X \rightarrow X) \rightarrow \text{fix} f \rightarrow X
\]

\[
\text{from-fix} f = \text{pr}_1
\]

Conversely, if \( f \) is \( \text{wconstant} \) then for any \( x : X \) we have that \( f \, x \) is a fixed point of \( f \), and hence from any element of \( X \) we get a fixed point of \( f \):

\[
\text{to-fix} : \{X : U \} (f : X \rightarrow X) \rightarrow \text{wconstant} f
\]

\[
\text{to-fix} f \, \kappa \, x = f \, x \, , \, \kappa (f \, x) \, x
\]

The following is trivial if the type \( X \) is a set. What may be surprising is that it holds for arbitrary types, because in this case the identity type \( f \, x \equiv x \) is in general not a subsingleton.

\[
\text{fix-is-subsingleton} : \{X : U \} (f : X \rightarrow X) \rightarrow \text{wconstant} f \rightarrow \text{is-subsingleton} (\text{fix} f)
\]

\[
\text{fix-is-subsingleton} \{U\} \{X\} f \, \kappa = \gamma
\]

\[
\text{where} \quad a : (y \, x : X) \rightarrow (f \, x \equiv x) \equiv (f \, y \equiv x)
\]

\[
a \, y \, x = \text{transport} (_\equiv x) (\kappa \, x \, y) , \text{transport-is-equiv} (_\equiv x) (\kappa \, x \, y)
\]

\[
b : (y : X) \rightarrow \text{fix} f \equiv \text{singleton-type'} (f \, y)
\]

\[
b \, y = \Sigma\text{-cong} (a \, y)
\]

\[
c : X \rightarrow \text{is-singleton} (\text{fix} f)
\]

\[
c \, y = \text{equiv-to-singleton} (b \, y) (\text{singleton-types'-are-singletons} X \, (f \, y))
\]

\[
d : \text{fix} f \rightarrow \text{is-singleton} (\text{fix} f)
\]

\[
d = c \circ \text{from-fix} f
\]

\[
y : \text{is-subsingleton} (\text{fix} f)
\]

\[
y = \text{subsingleton-criterion} \, d
\]

**Exercise.** Formulate and prove the fact that the type \( \text{fix} \, f \) has the universal property of the subsingleton truncation of \( X \) if \( f \) is \( \text{wconstant} \). Moreover, argue that the computation rule for recursion holds definitionally in this case. This is an example of a situation where the truncation of a type just is available in MLTT without axioms or extensions.

We use \( \text{fix-is-subsingleton} \) to show that the type \( \text{is-inhabited} \ X \rightarrow X \) is logically equivalent to the type \( \text{wconstant-endomap} \, X \), where one direction uses function extensionality. We refer to a function \( \text{is-inhabited} \ X \rightarrow X \) as a choice function for \( X \). So the claim is that a type has a choice function if and only if it has a designated \( \text{wconstant} \) endomap.

\[
\text{choice-function} : U \rightarrow U ^+ \ ^+
\]

\[
\text{choice-function} \, X = \text{is-inhabited} \ X \rightarrow X
\]

With a constant endomap of \( X \), we can exit the truncation \( \text{is-inhabited} \ X \) in pure MLTT:

\[
\text{wconstant-endomap-gives-choice-function} : \{X : U \} \rightarrow \text{wconstant-endomap} \, X \rightarrow \text{choice-function} \, X
\]

\[
\text{wconstant-endomap-gives-choice-function} \{U\} \{X\} (f \, , \, \kappa) = \text{from-fix} f \circ \gamma
\]
The crucial property of the function of roots:

Given a root, we find a minimal root (below it, of course) by bounded search, and this gives a constant endomap of the type

\[
\text{module } \\
\text{with the information that there is some root, we can find an explicit root.}
\]

As an application, we show that if the type of roots of a function of the form

\[
\text{choice-function-gives-wconstant-endomap}
\]

For the converse we use function extensionality (to know that the type is inhabited \( X \) is a subsingleton in the construction of the \( w\text{constant} \) endomap):

\[
\text{module find-hidden-root where}
\]

Given a root, we find a minimal root (below it, of course) by bounded search, and this gives a constant endomap of the type of roots:

\[
\mu p : (f : \mathbb{N} \to \mathbb{N}) \to \text{root } f \to \text{root } f
\]

\[
\mu p f r = \text{minimal-root-is-root } f (\text{root-gives-minimal-root } f r)
\]

\[
\mu p - \text{root} : (f : \mathbb{N} \to \mathbb{N}) \to \text{root } f \to \mathbb{N}
\]

\[
\mu p - \text{root} f r = \text{pr}_1 (\mu p f r)
\]

\[
\mu p - \text{root-is-root} : (f : \mathbb{N} \to \mathbb{N}) (r : \text{root } f) \to f (\mu p - \text{root } f r) \equiv \varnothing
\]

\[
\mu p - \text{root-is-root} f r = \text{pr}_2 (\mu p f r)
\]

\[
\mu p - \text{root-minimal} : (f : \mathbb{N} \to \mathbb{N}) (m : \mathbb{N}) (p : f m \equiv \varnothing) \to (n : \mathbb{N}) \to f n \equiv \varnothing \to \mu p - \text{root } f (m , p) \leq n
\]

\[
\mu p - \text{root-minimal} f m p n q = \text{not-<}-\text{gives}-\geq (\mu p - \text{root } f (m , p)) n y
\]

\[
\phi : \neg (f n \equiv \varnothing) \to \neg (n < \mu p - \text{root } f (m , p))
\]

\[
\phi = \text{contrapositive} (\text{pr}_2(\text{pr}_2 (\text{root-gives-minimal-root} f (m , p))) n)
\]

\[
y : \neg (n < \mu p - \text{root } f (m , p))
\]

\[
y = \phi (\text{dn1} (f n \equiv \varnothing) q)
\]

The crucial property of the function \( \mu p f \) is that it is \( w\text{constant} \):

\[
\mu p - \text{wconstant} : (f : \mathbb{N} \to \mathbb{N}) \to w\text{constant } (\mu p f)
\]

\[
\mu p - \text{wconstant} f (n , p) (n' , p') = r
\]

\[
\mu p - \text{wconstant} f (n , p) (n' , p') = r
\]

\[
\text{where}
\]

\[
m m' : \mathbb{N}
\]

\[
m = \mu p - \text{root } f (n , p)
\]

\[
m' = \mu p - \text{root } f (n' , p')
\]

\[
l : m \leq m'
\]

\[
l = \mu p - \text{root-minimal} f n p m' (\mu p - \text{root-is-root } f (n' , p'))
\]

\[
l' : m' \leq m'
\]

\[
l' = \mu p - \text{root-minimal} f n' p' m (\mu p - \text{root-is-root } f (n , p))
\]

\[
q : m \equiv m'
\]

\[
q = \text{s-anti } l l'
\]

\[
r : \mu p f (n , p) \equiv \mu p f (n' , p')
\]

\[
r = \text{to-subtype-} \equiv (\lambda _ \to \mathbb{N}-\text{set} (f _ \to \varnothing)) q
\]
Using the $w$constancy of $\mu \rho f$, if a root of $f$ exists, then we can find one (which in fact will be the minimal one):

$$\text{find-existing-root} : (f : \mathbb{N} \to \mathbb{N}) \to \text{is-inhabited} (\text{root } f) \to \text{root } f$$

$$\text{find-existing-root } f = h \circ g$$

where

$$y : \text{root } f \to \text{fix} (\mu \rho f)$$
$$y = \text{to-fix} (\mu \rho f) (\mu \rho \text{-constant } f)$$

$$g : \text{is-inhabited} (\text{root } f) \to \text{fix} (\mu \rho f)$$
$$g = \text{inhabited-recursion} (\text{fix-is-subsingleton} (\mu \rho f) (\mu \rho \text{-constant } f)) y$$

$$h : \text{fix} (\mu \rho f) \to \text{root } f$$
$$h = \text{from-fix} (\mu \rho f)$$

In the following example, we first hide a root with $\text{inhabited-intro}$ and then find the minimal root with search bounded by this hidden root:

```plaintext
module find-existing-root-example where

f : \mathbb{N} \to \mathbb{N}
f 0 = 1
f 1 = 1
f 2 = 0
f 3 = 1
f 4 = 0
f 5 = 1
f 6 = 1
f 7 = 0
f (\text{succ} (\text{succ} (\text{succ} (\text{succ} (\text{succ} (\text{succ} (\text{succ} x))))))) = x
```

We hide the root 8 of $f$:

```plaintext
root-existence : \text{is-inhabited} (\text{root } f)
root-existence = \text{inhabited-intro} (8, \text{refl } 0)
```

```plaintext
r : \text{root } f
r = \text{find-existing-root } f \text{ root-existence}
```

```plaintext
x : \mathbb{N}
x = \text{pr}_1 r
```

```plaintext
x-is-root : f x \equiv 0
x-is-root = \text{pr}_2 r
```

We have that $x$ evaluates to 2, which is clearly the minimal root of $f$:

```plaintext
p : x \equiv 2
p = \text{refl } 2
```

Thus, even though the type $\text{is-inhabited } A$ is a subsingleton for any type $A$, the function $\text{inhabited-intro} : A \to \text{is-inhabited } A$ doesn’t erase information. We used the information contained in $\text{root-existence}$ as a bound for searching for the minimal root.

Notice that this construction is in pure (spartan) MLTT without assumptions. Now we repeat part of the above using the existence of small truncations and functional extensionality as assumptions, replacing $\text{is-inhabited}$ by $\parallel_\_\parallel$:

```plaintext
module exit-\parallel\parallel (pt : \text{subsingleton-truncations-exist}) (hfe : \text{global-hfunext})
where

open \text{basic-truncation-development} pt hfe
open \text{find-hidden-root}

\text{find-\parallel\parallel-existing-root} : (f : \mathbb{N} \to \mathbb{N})
\to (\exists n : \mathbb{N}, f n \equiv 0)
\to \Sigma n : \mathbb{N}, f n \equiv 0
```

```plaintext
\text{find-\parallel\parallel-existing-root} f = k
```
where
\( \gamma : \text{root } f \to \text{fix} (\mu f) \)
\( \gamma = \text{to-fix} (\mu f) (\mu \text{-wconstant } f) \)

\( g : \parallel \text{root } f \parallel \to \text{fix} (\mu f) \)
\( g = \parallel \parallel \text{-recursion} (\text{fix-is-subsingleton} (\mu f) (\mu \text{-wconstant } f)) \parallel \gamma \)

\( h : \text{fix} (\mu f) \to \text{root } f \)
\( h = \text{from-fix} (\mu f) \)

\( k : \parallel \text{root } f \parallel \to \text{root } f \)
\( k = h \circ g \)

module find-\parallel \parallel -existing-root-example where

\( f : \mathbb{N} \to \mathbb{N} \)
\( f \ 0 = 1 \)
\( f \ 1 = 1 \)
\( f \ 2 = 0 \)
\( f \ 3 = 1 \)
\( f \ 4 = 0 \)
\( f \ 5 = 1 \)
\( f \ 6 = 1 \)
\( f \ 7 = 0 \)
\( f (\text{succ} (\text{succ} (\text{succ} (\text{succ} (\text{succ} (\text{succ} x))))) = x \)

\( \text{root-\parallel \parallel -existence} : \parallel \text{root } f \parallel \)
\( \text{root-\parallel \parallel -existence} = \parallel 8 , \text{refl } 0 \parallel \)

\( r : \text{root } f \)
\( r = \text{find-\parallel \parallel -existing-root } f \text{ root-\parallel \parallel -existence} \)

\( x : \mathbb{N} \)
\( x = \text{pr}_1 r \)

\( x\text{-is-root} : f x \equiv 0 \)
\( x\text{-is-root} = \text{pr}_2 r \)

This time, because the existence of propositional truncations is an assumption for this submodule, we don’t have that \( x \) evaluates to 2, because the computation rule for truncation doesn’t hold definitionally. But we do have that \( x \) is 2, applying the computation rule manually.

\( \text{NB} : \text{find-\parallel \parallel -existing-root } f \)
\( \equiv \text{from-fix} (\mu f) \circ \parallel \parallel \text{-recursion} \)
\( (\text{fix-is-subsingleton} (\mu f) (\mu \text{-wconstant } f)) \)
\( (\text{to-fix} (\mu f) (\mu \text{-wconstant } f)) \)

\( \text{NB} = \text{refl } _{\parallel \parallel -existence} \)

\( p : x \equiv 2 \)
\( p = \text{ap} (\text{pr}_1 \circ \text{from-fix } (\mu f)) \)
\( (\parallel \parallel \text{-recursion-computation} \)
\( (\text{fix-is-subsingleton} (\mu f) (\mu \text{-wconstant } f)) \)
\( (\text{to-fix} (\mu f) (\mu \text{-wconstant } f)) \)
\( (8 , \text{refl } _{\parallel \parallel -existence}) \)

In cubical Agda, with the truncation defined as a higher inductive type, \( x \) would compute to 2 automatically, like in our previous example using Voevodsky’s truncation is-inhabited. This concludes the example. We also have:

\( \text{wconstant-endomap-gives-\parallel \parallel -choice-function} : \{ X : U \} \to \text{wconstant-endomap } X \)
\( \to (\parallel X \parallel \to X) \)

\( \text{wconstant-endomap-gives-\parallel \parallel -choice-function} \{ U \} \{ X \} (f, \kappa) = \text{from-fix } f \circ \gamma \)

where
\( \gamma : \parallel X \parallel \to \text{fix } f \)
\( \gamma = \parallel \parallel \text{-recursion} (\text{fix-is-subsingleton } f \kappa) (\text{to-fix } f \kappa) \)

\( \parallel \parallel \text{-choice-function-gives-wconstant-endomap} : \{ X : U \} \)
\( \to (\parallel X \parallel \to X) \)
There is another situation in which we can eliminate truncations that is often useful in practice. The universal property of subsingleton truncation says that we can get a function \( \parallel X \parallel \to Y \) provided the type \( Y \) is a subsingleton and we have a given function \( X \to Y \). Because \( Y \) is a subsingleton, the given function is automatically \( \text{wconstant} \). Hence the following generalizes this to the situation in which \( Y \) is a set:

\[
\begin{align*}
\text{∥ ∥-recursion-set} & : (X : \mathcal{U}) (Y : \mathcal{V}^{'}) \\
& \to \text{is-set} Y \\
& \to (f : X \to Y) \\
& \to \text{wconstant} f \\
& \to \parallel X \parallel \to Y
\end{align*}
\]

\[
\begin{align*}
\text{∥ ∥-recursion-set} \ {\{\mathcal{U}\} \ {\mathcal{V}}} & X Y s f \ k = f' \\
\text{where}
\psi : (y \ y' : Y) \to (\Sigma x : X , f x \equiv y) \to (\Sigma x' : X , f x' \equiv y') \to y \equiv y' \\
\psi y y' (x , r) (x' , r') = y \equiv (r^{-1}) \\
f x \equiv (k \times x') \\
f x' \equiv (r') \\
y' \equiv y' \\
\phi : (y \ y' : Y) \to (\exists x : X , f x \equiv y) \to (\exists x' : X , f x' \equiv y') \to y \equiv y' \\
\phi y y' u u' = \text{∥ ∥-recursion} (s y y') (\lambda - \to \text{∥ ∥-recursion} (s y y') (\psi y y' -) u') u \\
P : \mathcal{U} \cup \mathcal{V}^{'}, \\
P = \text{image} f \\
\mathcal{I} : \text{is-subsingleton} P \\
\mathcal{I} (y , u) (y' , u') = \text{to-subtype-≡} (\lambda _ - \to \exists\text{-is-subsingleton}) (\phi y y' u u') \\
g : \parallel X \parallel \to P \\
g = \text{∥ ∥-recursion} \mathcal{I} (\text{corestriction} f) \\
h : P \to Y \\
h = \text{restriction} f \\
f' : \parallel X \parallel \to Y \\
f' = h \circ g
\end{align*}
\]

If we try to do this with Voevodsky’s truncation \( \text{is-inhabited} \), we stumble into an insurmountable problem of size.

### Equality of Noetherian local rings

This section has that on \text{rings} as a pre-requisite. We now apply the notion of subsingleton truncation to give the promised examples of Noetherian rings and commutative Noetherian local rings. Subsingleton truncation is needed to have the existential quantifier \( \exists \) available, in order to be able to define the notion of Noetherian ring.

\[
\begin{align*}
\text{module} \text{ noetherian-local-ring} \\
& (\text{pt} : \text{subsingleton-truncations-exist}) \\
& (\mathcal{U} : \text{Universe}) \\
& (\text{ua} : \text{Univalence})
\end{align*}
\]

\[
\begin{align*}
\text{where}
\text{open ring} \ {\mathcal{U}} \ \text{ua} \\
\text{open basic-truncation-development} \ \text{pt} \ \text{hfe} \\
\text{open N-order}
\end{align*}
\]

We first consider left Noetherian rings and then Noetherian local rings.
Hence properties of left Noetherian rngs are invariant under isomorphism. More generally, we can transport along type-

For this we need the notion of left ideal of a rng $\mathcal{R}$, which is an element of the powerset $\mathcal{P}(\mathcal{R})$ of the underlying set of $\mathcal{R}$:

$$
is-left-ideal : (\mathcal{R} : \text{Rng}) \to \mathcal{P}(\mathcal{R})
$$

$$
is-left-ideal (R, (_+_, _-_), (i, ii, iii, (0, _), _)) I =
\begin{align*}
& (0 \in I) \\
& \times ((x \cdot y : R) \to x \in I \to y \in I \to (x + y) \in I) \\
& \times ((x \cdot y : R) \to y \in I \to (x \cdot y) \in I)
\end{align*}
$$

$$
is-left-noetherian : (\mathcal{R} : \text{Rng}) \to \mathcal{U} \quad
is-left-noetherian \mathcal{R} = (I : \mathcal{N} \to \mathcal{P}(\mathcal{R})) \to
\begin{align*}
& (((n : \mathcal{N}) \to is-left-ideal \mathcal{R} (I n)) \\
& \quad \to ((n : \mathcal{N}) \to I n \in I (\text{succ } n)) \\
& \quad \to \exists m : \mathcal{N}, ((n : \mathcal{N}) \to m \leq n \to I m \equiv I n)
\end{align*}
$$

$$
\mathcal{LNRng} : \mathcal{U} \quad
\mathcal{LNRng} = \Sigma \mathcal{R} : \text{Rng}, \text{is-left-noetherian } \mathcal{R}
$$

In order to be able to characterize equality of left Noetherian rngs, we again need to show that $\text{is-left-noetherian}$ is a property rather than data:

$$
\text{being-ln-is-subsingleton} : (\mathcal{R} : \text{Rng}) \to \text{is-subsingleton} (\text{is-left-noetherian } \mathcal{R})
$$

$$
\text{being-ln-is-subsingleton } \mathcal{R} = \Pi \text{-is-subsingleton } fe \\
\begin{align*}
& (\lambda I \to \Pi \text{-is-subsingleton } fe) \\
& (\lambda _ \to \Pi \text{-is-subsingleton } fe) \\
& (\lambda _ \to \exists \text{-is-subsingleton}))
\end{align*}
$$

$$
\text{forget-LN} : \mathcal{LNRng} \to \text{Rng}
$$

$$
\text{forget-LN } (\mathcal{R}, _) = \mathcal{R}
$$

$$
\text{forget-LN-is-embedding} : \text{is-embedding forget-LN}
$$

$$
\text{forget-LN-is-embedding} = \text{pr}_1\text{-is-embedding being-ln-is-subsingleton}
$$

Isomorphism of left Noetherian rngs:

$$
\equiv[\mathcal{LNRng}]_\sim : \mathcal{LNRng} \to \mathcal{LNRng} \to \mathcal{U}
$$

$$
((\mathcal{R}, (_+_, _-_), _), _) \equiv[\mathcal{LNRng}] ((\mathcal{R}', (_+\prime_, _-\prime_), _), _) =
\begin{align*}
& \Sigma f : (R \to R') \\
& \quad \text{is-equiv } f \\
& \quad \times ((\lambda x y \leftarrow f (x + y)) \equiv (\lambda x y \to f x + f y)) \\
& \quad \times ((\lambda x y \leftarrow f (x \cdot y)) \equiv (\lambda x y \to f x + f y))
\end{align*}
$$

$$
\text{NB} : (\mathcal{R}, \mathcal{R}') : \text{LNRng} \to (\mathcal{R} \equiv[\mathcal{LNRng}] \mathcal{R}') \equiv (\text{forget-LN } \mathcal{R} \equiv[\text{Rng}] \text{forget-LN } \mathcal{R}')
$$

$$
\text{NB } \mathcal{R} \mathcal{R}' = \text{refl } _\sim
$$

Again the identity type of Noetherian rngs is in bijection with the type of Noetherian rng isomorphisms:

$$
\text{characterization-of-LNRng=} : (\mathcal{R}, \mathcal{R}') : \text{LNRng} \to (\mathcal{R} \equiv \mathcal{R}') \Rightarrow (\mathcal{R} \equiv[\mathcal{LNRng}] \mathcal{R}')
$$

$$
\text{characterization-of-LNRng=} \mathcal{R} \mathcal{R}' = (\mathcal{R} \equiv \mathcal{R}') \equiv\{ i \} \\
\quad \{ \text{forget-LN } \mathcal{R} \equiv \text{forget-LN } \mathcal{R}' \} \equiv\{ ii \} \\
\quad (\mathcal{R} \equiv[\mathcal{LNRng}] \mathcal{R}')
$$

where

$$
i = \equiv\text{-sym } \text{(embedding-criterion-converse forget-LN)
\text{forget-LN-is-embedding } \mathcal{R} \mathcal{R}')
$$

$$
ii = \text{characterization-of-rng=} \equiv (\text{forget-LN } \mathcal{R}) (\text{forget-LN } \mathcal{R}')
$$

Hence properties of left Noetherian rngs are invariant under isomorphism. More generally, we can transport along type-

valued functions of left Noetherian rngs, with values in an arbitrary universe $\mathcal{V}$, rather than just truth-valued ones:

$$
isomorphic-LNRng-transport : (A : \text{LNRng} \to \mathcal{V}) (\mathcal{R}, \mathcal{R}') : \text{LNRng} \\
\to A \equiv[\text{LNRng}] \mathcal{R}' \to A \mathcal{R} \to A \mathcal{R}'
$$

$$
isomorphic-LNRng-transport A \mathcal{R} \mathcal{R}' i a = a'
$$

where
In particular, any theorem about a left Noetherian rng automatically applies to any left Noetherian rng isomorphic to it.

One can similarly define right Noetherian rng and Noetherian rng, and then obtain the expected characterizations of their identity types (exercise).

We now consider Noetherian local rings.

```plaintext
is-commutative : Rng → U
is-commutative (R , (_,+_ , _-)) = (x y : R) → x · y ≡ y · x

being-commutative-is-subsingleton : (R : Rng) → is-subsingleton (is-commutative R)
being-commutative-is-subsingleton (R , (_,+_ , _-)) = (Π x → Π is-subsingleton (being-commutative-is-subsingleton R))
```

In the presence of commutativity, there is no difference between left ideal, right ideal and two-sided ideal, and so one speaks of simply ideals. A commutative rng is said to be local if it has a unique maximal proper ideal.

```plaintext
is-Noetherian-Local : Ring → U
is-Noetherian-Local (R , (1 , _+_ , _-)) = is-commutative R × is-noetherian R

where
R : Rng
R = (R , (_,+_ , _-))

is-ideal = is-left-ideal R
is-noetherian = is-left-noetherian R

is-proper-ideal : P R → U
is-proper-ideal I = is-ideal I × (∃ x : (R ) , x ∉ I)

is-local = ∃! I : P R , is-proper-ideal I × (∃ J : P R → is-proper-ideal J → J ⊆ I)
```

```plaintext
being-NL-is-subsingleton : (R : Ring) → is-subsingleton (is-Noetherian-Local R)
being-NL-is-subsingleton (R , (1 , _+_ , _-)) = (×-is-subsingleton (being-commutative-is-subsingleton R))
```

where
R : Rng
R = (R , (_,+_ , _-))

```plaintext
NL-Ring : U
NL-Ring = Σ R : Ring , is-Noetherian-Local R

≡[NL] : NL-Ring → NL-Ring → U
≡[NL] (R , (1 , _+_ , _-)) = (Π f : (R → R') , is-equiv f)
```
forget-NL : NL-Ring → Ring
forget-NL (⟂, _) = ⟂

forget-NL-is-embedding : is-embedding forget-NL
forget-NL-is-embedding = pr₁-is-embedding being-NL-is-subsingleton

NB' : (𝓡 𝓡' : NL-Ring) → (𝓡 ≅[NL] 𝓡') ≡ (forget-NL 𝓡 ≡[Ring] forget-NL 𝓡')
NB' 𝓡 𝓡' = refl _

characterization-of-NL-ring≡ : (𝓡 𝓡' : NL-Ring) → (𝓡 ≃ (forget-NL 𝓡 ≡[Ring] forget-NL 𝓡'))
characterization-of-NL-ring≡ 𝓡 𝓡' = (forget-NL 𝓡 ≃ forget-NL 𝓡') ≡ i
(characterization-of-NL-ring≡ 𝓡 𝓡') ≡ ii

where
i = ≃-sym (embedding-criterion-converse forget-NL
forget-NL-is-embedding 𝓡 𝓡')
ii = characterization-of-ring≡ (forget-NL 𝓡) (forget-NL 𝓡')

isomorphic-NL-Ring-transport : (A : NL-Ring → 𝜅⁻¹) (𝓡 𝓡' : NL-Ring)
→ 𝓡 ≃[NL] 𝓡' → A 𝓡 → A 𝓡'
isomorphic-NL-Ring-transport A 𝓡 𝓡' i a = a'
where
p : 𝓡 ≃[NL] 𝓡'
p = ≃-sym (characterization-of-NL-ring≡ 𝓡 𝓡') ^ i
a' : A 𝓡'
a' = transport A p a

We remark that alternative definitions of the above notions are adopted for the purposes of constructive algebra.

Table of contents ⦿

Choice in univalent mathematics

We discuss unique choice, univalent choice and global choice.

1. A simple form of unique choice just holds in our spartan MLTT.

2. The full form of unique choice is logically equivalent to function extensionality.

3. Univalent choice implies excluded middle and is not provable or disprovable, but is consistent with univalence.

4. Global choice contradicts univalence, because it is not possible to choose an element of every inhabited type in a way that is invariant under automorphisms.

Table of contents ⦿

The principle of unique choice

The principle of simple unique choice says that

if for every x : X there is a unique a : A x with R x a,

then

there is a specified function f : (x : X) → A x such that R x (f x) for all x : X.

This just holds and is trivial, given by projection:

simple-unique-choice : (X : 𝓤⁻¹) (A : X → 𝜅⁻¹) (R : (x : X) → A x → 𝜅⁻¹)
→ ((x : X) → ∃! a : A x, R x a)
Here is an alternative proof that derives \( \text{unique-choice} \) instead:

\[
\text{simple-unique-choice} : X A R s = f, \ \varphi
\]

where

\[
f : (x : X) \to A x
\]

\[
f x = \text{pr}_1 (\text{center} (\Sigma a : A x, R x a) (s x))
\]

\[
\varphi : (x : X) \to R x (f x)
\]

\[
\varphi x = \text{pr}_1 (\text{center} (\Sigma a : A x, R x a) (s x))
\]

Below we also consider a \textit{variation} of simple unique choice that works with \( \exists \) (truncated \( \Sigma \)) rather than \( \exists ! \).

A full form of unique choice is Voevodsky’s formulation \textit{vvfunext} of function extensionality, which says that products of singletons are singletons. We show that this is equivalent to our official formulation of unique choice:

\[
\text{Unique-Choice} : (U \uplus W : \text{Universe}) \to (U \uplus U \uplus W)^+ \quad \text{Unique-Choice} U \uplus W = (X : U^-) (A : X \to V^-) (R : (x : X) \to A x \to W^-)
\]

\[
\to ((x : X) \to \exists! a : A x, R x a)
\]

\[
\to \exists f : \Pi A x, ((x : X) \to R x (f x))
\]

This can be read as saying that every single-valued relation is the graph of a unique function.

\[
\text{vvfunext-gives-unique-choice} : \text{vvfunext} U (U \uplus W) \to \text{Unique-Choice} U \uplus W
\]

\[
\text{vvfunext-gives-unique-choice} v v X A R s = c
\]

where

\[
a : ((x : X) \to \Sigma a : A x, R x a)
\]

\[
= (\Sigma f : ((x : X) \to A x), ((x : X) \to R x (f x)))
\]

\[
a = \Pi \Sigma \text{-distr} =
\]

\[
b : \text{is-singleton} ((x : X) \to \Sigma a : A x, R x a)
\]

\[
b = v v s
\]

\[
c : \text{is-singleton} (\Sigma f : ((x : X) \to A x), ((x : X) \to R x (f x)))
\]

\[
c = \text{equiv-to-singleton'} a b
\]

\[
\text{unique-choice-gives-vvfunext} : \text{Unique-Choice} U \uplus W \to \text{vvfunext} U \uplus W
\]

\[
\text{unique-choice-gives-vvfunext} \{U\} \{W\} uc \{X\} \{A\} \varphi = \gamma
\]

where

\[
R : (x : X) \to A x \to \text{V'}
\]

\[
R x a = A x
\]

\[
s' : (x : X) \to \text{is-singleton} (A x \times A x)
\]

\[
s' x = x \text{-is-singleton} (\varphi x) (\varphi x)
\]

\[
s : (x : X) \to \exists! y : A x, R x y
\]

\[
s = s'
\]

\[
e : \exists! f : \Pi A x, ((x : X) \to R x (f x))
\]

\[
e = uc X A R s
\]

\[
e' : \text{is-singleton} (\Pi A x \times A)
\]

\[
e' = e
\]

\[
\rho : \Pi A x \leftarrow \Pi A x \times A
\]

\[
\rho = \text{pr}_{1}, (\lambda y \to y, y), \text{refl}
\]

\[
\gamma : \text{is-singleton} (\Pi A)
\]

\[
\gamma' = \text{retract-of-singleton} \rho e'
\]

Here is an alternative proof that derives \textit{hfunext} instead:

\[
\text{unique-choice-gives-hfunext} : \text{Unique-Choice} U \uplus W \to \text{hfunext} U \uplus W
\]

\[
\text{unique-choice-gives-hfunext} \{U\} \{W\} uc = \text{-hfunext} \gamma
\]

where

\[
y : (X : U^-) (A : X \to W^-) (f : \Pi A) \to \exists! g : \Pi A, f \sim g
\]

\[
y X A f = uc X A R e
\]

where

\[
R : (x : X) \to A x \to \text{V'}
\]
The above is not quite the converse of the previous, as there is a universe mismatch, but we do get a logical equivalence by taking \( \mathcal{V} \) to be \( \mathcal{V}' \):

\[
\text{unique-choice} \equiv \mathcal{V} \land \mathcal{V}' \equiv \mathcal{V} \land \mathcal{V}'
\]

\[
\text{unique-choice} \equiv \mathcal{V} \land \mathcal{V}'
\]

\[
\text{vvfunext} \Rightarrow \text{unique-choice-gives-vvfunext} \approx \text{vvfunext-gives-unique-choice}
\]

We now give a different derivation of unique choice from function extensionality, in order to illustrate transport along the inverse of \( \text{happly} \). For simplicity, we assume global function extensionality in the next few constructions.

\[
\text{module } (\text{hfe} : \text{global-hfunext}) \text{ where}
\]

\[
\begin{align*}
\text{private} & \quad \text{hunapply} : \{X : \mathcal{U}' \} \{A : X \rightarrow \mathcal{V}' \} \{f, g : \Pi A \rightarrow f \sim g \rightarrow f \equiv g
g \equiv \text{inverse} (\text{happly} _{-}) (\text{hfe} _{-}) & \\
\text{transport-hunapply} : \{X : \mathcal{U}' \} (A : X \rightarrow \mathcal{V}') \{R : (x : X) \rightarrow A x \rightarrow \mathcal{V}' \}
\end{align*}
\]

\[
\begin{align*}
& \quad (f, g : \Pi A) \\
& \quad \{\phi : (x : X) \rightarrow R x (f x)\} \\
& \quad (h : f \sim g) \\
& \quad (x : X) \\
& \quad \rightarrow \text{transport} (\lambda x \rightarrow (x : X) \rightarrow R x (- x)) (\text{hunapply} h) \phi x \\
& \quad \Rightarrow \text{transport} (R x) (h x) (\phi x)
\end{align*}
\]

\[
\begin{align*}
\text{transport-hunapply} & \quad A R f g \phi h x = \\
& \quad \text{transport} (\lambda x \rightarrow \forall x \rightarrow R x (- x)) (\text{hunapply} h) \phi x \equiv (\text{i } ) \\
& \quad \text{transport} (R x) (\text{happly} f g (\text{hunapply} h) x) (\phi x) \equiv (\text{ii } ) \\
& \quad \text{transport} (R x) (h x) (\phi x)
\end{align*}
\]

\[
\begin{align*}
\text{where}
& \quad a : \{f, g : \Pi A\} \{\phi : \forall x \rightarrow R x (f x)\} (p : f \equiv g) (x : \text{domain} A) \\
& \quad \rightarrow \text{transport} (\lambda x \rightarrow \forall x \rightarrow R x (- x)) p \phi x \\
& \quad \Rightarrow \text{transport} (R x) (\text{happly} f g p x) (\phi x)
\end{align*}
\]

\[
\begin{align*}
& \quad a (\text{refl } x) x = \text{refl } x \\
& \quad b : \text{happly} f g (\text{hunapply} h) \equiv h \\
& \quad b = \text{inverses-are-sections} (\text{happly} f g) (\text{hfe} f g) h \\
& \quad i = a (\text{hunapply} h) x \\
& \quad ii = \text{ap} (\lambda x \rightarrow \text{transport} (R x) (- x) (\phi x)) b
\end{align*}
\]

\[
\begin{align*}
\text{unique-choice} & \quad (X : \mathcal{U}') (A : X \rightarrow \mathcal{V}') (R : (x : X) \rightarrow A x \rightarrow \mathcal{V}') \\
& \quad \rightarrow ((x : X) \rightarrow \exists! a : A x , R x a) \\
& \quad \rightarrow \exists! f : ((x : X) \rightarrow A x) , ((x : X) \rightarrow R x (f x))
\end{align*}
\]

\[
\begin{align*}
\text{unique-choice} & \quad X A R s = C , \Theta
\end{align*}
\]

\[
\begin{align*}
\text{where}
& \quad f_0 : (x : X) \rightarrow A x \\
& \quad f_0 x = \text{pr}_1 (\text{center} (\Sigma a : A x , R x a) (s x)) \\
& \quad \phi_0 : (x : X) \rightarrow R x (f_0 x) \\
& \quad \phi_0 x = \text{pr}_2 (\text{center} (\Sigma a : A x , R x a) (s x)) \\
& \quad \Sigma C : (\Sigma f : ((x : X) \rightarrow A x) , ((x : X) \rightarrow R x (f x))) \\
& \quad C = f_0 , \phi_0 \\
& \quad c : (x : X) \rightarrow (\tau : \Sigma a : A x , R x a) \rightarrow f_0 x , \phi_0 x \equiv \tau \\
& \quad c x = \text{centrality} (\Sigma a : A x , R x a) (s x) \\
& \quad c_1 : (x : X) (a : A x) (r : R x a) \rightarrow f_0 x \equiv a \\
& \quad c_1 x a r = \text{ap} \text{pr}_1 (c x (a , r))
\end{align*}
\]
The axiom of choice in univalent mathematics says that

Where

The univalent axiom of choice

Table of contents

In the next few subsections we continue working within the submodule choice, in order to have the existence of propositional truncations available, so that we can use the existential quantifier \( \exists \).

The univalent axiom of choice

The axiom of choice in univalent mathematics says that

if for every \( x : X \) there exists \( a : A x \) with \( R x a \),

where \( R \) is some given relation,
These two forms of choice are logically equivalent (and hence equivalent, as both are subsingletons assuming function
notion).

because choice implies excluded middle, and excluded middle implies that non-emptyness and inhabitation are the same
non-empty, where in a constructive setting we strengthen

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 strengthen

This is not provable or disprovable in spartan univalent type theory, but, with this proviso, it does hold in Voevodsky’s simplicial model of our univalent type theory, and hence is consistent. 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.

provided

1. X is a set,
2. A is a family of sets,
3. 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)) → Θ f : (x : X) → Θ a : A x , R x a

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 gives IChoice

Choice gives IChoice

Table of contents

A second formulation of univalent choice

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 strengthen non-empty to inhabited (but this strengthening 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 gives IChoice

IChoice gives IChoice

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 = ⊥ x x

h : (x : X) → Y x → Σ y : Y x , R x y
h x y = (y , refl x)

g : (x : X) → Θ y : Y x , R x y
g x = |||--functor (h x) (φ x)

c : Θ f : Π Y , ((x : X) → R x (f x))
c = ac X Y i j R k g

y : || Π Y ||
A third formulation of univalent choice

We will see that exiting truncations in the sense of

\[(A : X \to \mathcal{U}') (x : X) \to \top A x \parallel \to A x\]

amounts to global choice and is inconsistent with univalence. However, exiting truncations in an unspecified way, in the sense of

\[(A : X \to \mathcal{U}') \to \top (x : X) \to \top A x \parallel \to A x\]

is consistent and equivalent to the above two versions of univalent choice, as we show now.

\[
\begin{align*}
\text{TAC} : & \quad (X : \mathcal{U}') (A : X \to \mathcal{U}') \to \text{is-set } X \to ((x : X) \to \text{is-set } (A x)) \to \mathcal{U} \cup \mathcal{V}' \\
\text{TAC} X A \quad & \quad \parallel ((x : X) \to \top A x \parallel \to A x) \parallel
\end{align*}
\]

\[
\begin{align*}
\text{TChoice} : & \quad \forall X : \mathcal{U} \to \mathcal{U}' \quad \\
\text{TChoice } & \quad \mathcal{U} = (X : \mathcal{U}') (A : X \to \mathcal{U}') (i : \text{is-set } X) (j : (x : X) \to \text{is-set } (A x)) \\
& \quad \to \text{TAC } X A \quad i \quad j
\end{align*}
\]

Notice that we use the hypothesis twice to get the conclusion in the following:
We apply the third formulation of univalent choice to show that choice implies excluded middle. We begin with the following lemma.

\[
\text{decidable-equality-criterion} : \{X : \mathcal{U} \} \to (\alpha : 2 \to X) \to \text{decidable}(\exists n : 2, \alpha n \equiv x) \to \text{decidable}(\Sigma n : 2, \alpha n \equiv x)
\]

\[
\text{decidable-equality-criterion} \; \alpha \; c = y \; d
\]

where

\[
r : 2 \to \text{image} \; \alpha
\]

\[
r = \text{corestriction} \; \alpha
\]

\[
\sigma : (y : \text{image} \; \alpha) \to \Sigma n : 2, \; r \; n \equiv y
\]

\[
\sigma (x, \; t) = f \; u
\]

where

\[
u : \Sigma n : 2, \; \alpha n \equiv x
\]

\[
u = c \; x \; t
\]

\[
f : (\Sigma n : 2, \; \alpha n \equiv x) \to \Sigma n : 2, \; r \; n \equiv (x, \; t)
\]

\[
f (n, \; p) = n, \; \text{to-subtype-≡} (\lambda _- \to \exists-is-subsingleton) \; p
\]

\[
s : \text{image} \; \alpha \to 2
\]

\[
s \; y = pr_1 (\sigma \; y)
\]

\[
\eta : (y : \text{image} \; \alpha) \to r (s \; y) \equiv y
\]

\[
\eta \; y = pr_2 (\sigma \; y)
\]

\[
l : \text{left-cancellable} \; s
\]

\[
l = \text{sections-are-lc} \; s \; (r, \; \eta)
\]

\[
ar : \{m \; n : 2\} \to \alpha m \equiv \alpha n \to r \; m \equiv r \; n
\]

\[
ar \; p = \text{to-subtype-≡} (\lambda _- \to \exists-is-subsingleton) \; p
\]

\[
ra : \{m \; n : 2\} \to r \; m \equiv r \; n \to \alpha m \equiv \alpha n
\]

\[
ra = \text{ap} \; \text{pr}_1
\]

\[
as : \{m \; n : 2\} \to \alpha m \equiv \alpha n \to s (r \; m) \equiv s (r \; n)
\]

\[
as \; p = \text{ap} \; s \; (ar \; p)
\]

\[
ra : \{m \; n : 2\} \to s (r \; m) \equiv s (r \; n) \to \alpha m \equiv \alpha n
\]

\[
ra \; p = ra \; (lp)
\]

\[
y : \text{decidable} \; (s (r_0) \equiv s (r_1)) \to \text{decidable}(\alpha_0 \equiv \alpha_1)
\]

\[
y (\text{inl} \; p) = \text{inl} \; (sa \; p)
\]

\[
y (\text{inr} \; u) = \text{inr} \; (\text{contrapositive} \; as \; u)
\]

\[
d : \text{decidable} \; (s (r_0) \equiv s (r_1))
\]

\[
d = \text{2-has-decidable-equality} \; (s (r_0)) \; (s (r_1))
\]

The first consequence of this lemma is that choice implies that every set has decidable equality.
choice-gives-decidable-equality : \( TChoice \mathcal{U} \rightarrow (X : \mathcal{U}^\prime) \rightarrow is-set X \rightarrow has-decidable-equality X \)

\[
choice-gives-decidable-equality \{\mathcal{U}\} \text{ tac } X \text{ i } x_0 \text{ x}_1 = \gamma
\]

where

\[
\alpha : 2 \rightarrow X \\
\alpha_0 = x_0 \\
\alpha_1 = x_1
\]

\[
A : X \rightarrow \mathcal{U}^\prime \\
A x = \Sigma n : 2 \, \alpha n = x
\]

\[
1 : \text{is-subsingleton} (\text{decidable} (x_0 \equiv x_1)) \\
1 = (+ \text{-is-subsingleton}' \text{ hunapply} (1 (\alpha_0) (\alpha_1))}
\]

\[
\delta : \|((x : X) \rightarrow A x \| \rightarrow A X)\| \rightarrow \text{decidable} (x_0 \equiv x_1) \\
\delta = \|\| \text{-recursion} l (\text{decidable-equality-criterion} \alpha)
\]

\[
\iota : (x : X) \rightarrow \text{is-set} (A x) \\
\iota x = \text{subsets-of-sets-are-sets} 2 (\lambda n \rightarrow \alpha n \equiv x) \, 2 \text{-is-set} (\lambda n \rightarrow \iota (\alpha n) x)
\]

\[
h : \|((x : X) \rightarrow A x \| \rightarrow A X)\| \\
h = \text{tac} X A \iota \iota
\]

\[
y : \text{decidable} (x_0 \equiv x_1) \\
y = \delta h
\]

Applying the above to the object \( \Omega \mathcal{U} \) of truth-values in the universe \( \mathcal{U} \), we get excluded middle:

\[
\text{choice-gives-EM} : \text{propext} \mathcal{U} \rightarrow TChoice (\mathcal{U}^\prime) \rightarrow \text{EM} \mathcal{U}
\]

\[
\text{choice-gives-EM} \{\mathcal{U}\} \text{ pe} \text{ tac} = \text{em}
\]

where

\[
T : \Omega \mathcal{U} \\
T = (\text{Lift} \mathcal{U} \, 1, \text{equiv-to-subsingleton} (\text{Lift} \simeq 1) \, 1 \text{-is-subsingleton})
\]

\[
\delta : (\omega : \Omega \mathcal{U}) \rightarrow \text{decidable} (T \equiv \omega) \\
\delta = \text{choice-gives-decidable-equality} \text{ tac} (\Omega \mathcal{U}) (\Omega \text{-is-a-set} \text{ hunapply} \text{ pe} ) T
\]

\[
\text{em} : (P : \mathcal{U}^\prime) \rightarrow \text{is-subsingleton} P \rightarrow P + \neg P \\
\text{em} P \iota = \gamma (\delta (P , \iota))
\]

where

\[
y : \text{decidable} (T \equiv (P , \iota)) \rightarrow P + \neg P
\]

\[
y \, \text{inl} \, r = \text{inl} (\text{Id}-\text{fun} \, s (\text{lift} *))
\]

where

\[
s : \text{Lift} \mathcal{U} \, 1 \equiv P \\
s = \text{ap} \, \text{pr}_1 \, r
\]

\[
y \, \text{inr} \, n = \text{inr} (\text{contrapositive} \, f \, n)
\]

where

\[
f : P \rightarrow T \equiv P , \iota \\
f \, p = \Omega \text{-ext} \text{ hunapply} \text{ pe} (\lambda (\_ : \text{Lift} \mathcal{U} \, 1) \rightarrow p) (\lambda (\_ : P) \rightarrow \text{lift} *)
\]

For more information with Agda code, see this.

**Table of contents**

**Global choice**

The following says that, for any given \( X \), we can either choose a point of \( X \) or tell that \( X \) is empty:

\[
global-choice : (\mathcal{U} : \text{Universe}) \rightarrow \mathcal{U}^\prime \\
global-choice \mathcal{U} = (X : \mathcal{U}^\prime) \rightarrow X + is-empty X
\]

And the following says that we can pick a point of every inhabited type:

\[
global-∥-choice : (\mathcal{U} : \text{Universe}) \rightarrow \mathcal{U}^\prime \\
global-∥-choice \mathcal{U} = (X : \mathcal{U}^\prime) \rightarrow ∥ X ∥ \rightarrow X
\]
Two forms of globally global choice:

open exit-∥∥ pt hfe

global-choice-gives-wconstant : global-choice 𝓤 → (X : 𝓤̇) → wconstant-endomap X

global-choice-gives-wconstant g X = decidable-has-wconstant-endomap (g X)

global-choice-gives-global-∥∥-choice : global-choice 𝓤 → global-∥∥-choice 𝓤

where
c X = y (c X)

γ : X + is-empty X → ∥ X ∥ → X
γ (inl x) s = x
γ (inr n) s = !∅ X (∥-recursion ∅-is-subsingleton n s)

global-∥∥-choice-gives-all-types-are-sets : global-∥∥-choice 𝓤 → (X : 𝓤̇) → is-set X

global-∥∥-choice-gives-universe-is-set : global-∥∥-choice (𝓤⁺) → is-set (𝓤̇)

global-∥∥-choice-gives-universe-is-set {𝓤} c = global-∥∥-choice-gives-all-types-are-sets c (𝓤⁺)

global-∥∥-choice-gives-choice : global-∥∥-choice 𝓤 → TChoice 𝓤

global-∥∥-choice-gives-choice {𝓤} c X A i j = |(λ x → c (A x))|

global-∥∥-choice-gives-EM : propext 𝓤 → global-∥∥-choice (𝓤⁺) → EM 𝓤

global-∥∥-choice-gives-EM {𝓤} pe c = choice-gives-EM pe (global-∥∥-choice-gives-choice c)

global-∥∥-choice-gives-global-choice : propext 𝓤 → global-∥∥-choice 𝓤 → global-∥∥-choice (𝓤⁺) → global-choice 𝓤

where
d : decidable ∥ X ∥
d = global-∥∥-choice-gives-EM pe c ∥ X ∥-is-subsingleton

f : decidable ∥ X ∥ → X + is-empty X
f (inl i) = inl (c X i)
f (inr φ) = inr (contrapositive |_| φ)

γ : X + is-empty X
γ = f d
Global-Choice Global-∥-Choice : \{\omega
Global-Choice \quad = \forall \ U \rightarrow \text{global-choice} \ U
Global-∥-Choice = \forall \ U \rightarrow \text{global-∥-choice} \ U

Which are equivalent, where one direction uses propositional extensionality:

Global-Choice-gives-Global-∥-Choice : Global-Choice \rightarrow Global-∥-Choice
Global-Choice-gives-Global-∥-Choice c U =
global-choice-gives-global-∥-choice (c U)

Global-∥-Choice-gives-Global-Choice : \text{global-propext}
\rightarrow Global-∥-Choice \rightarrow Global-Choice

Global-∥-Choice-gives-Global-Choice pe c U =
global-∥-choice-gives-global-choice pe (c U) (c (U \ '))

And which are inconsistent with univalence:

global-∥-choice-inconsistent-with-univalence : Global-∥-Choice
\rightarrow \text{Univalence}
\rightarrow \emptyset

global-∥-choice-inconsistent-with-univalence g ua = \gamma (g \ U_i) (ua \ U_0)

where
open example-of-a-nonset

\gamma : \text{global-∥-choice} \ U_i \rightarrow \text{is-univalent} \ U_0 \rightarrow \emptyset
\gamma g ua = \text{is-not-a-set} ua (global-∥-choice-gives-universe-is-set g)

global-choice-inconsistent-with-univalence : Global-Choice
\rightarrow \text{Univalence}
\rightarrow \emptyset

global-choice-inconsistent-with-univalence g =
global-∥-choice-inconsistent-with-univalence (Global-Choice-gives-Global-∥-Choice g)

See also Theorem 3.2.2 and Corollary 3.2.7 of the HoTT book for a different argument that works with a single, arbitrary universe.

global-choice-gives-all-types-are-sets : global-choice \ U
\rightarrow (X : U) \rightarrow \text{is-set} \ X

global-choice-gives-all-types-are-sets \{ U \} c x = \text{hedberg} (\lambda x y \rightarrow c (x \equiv y))

Table of contents

Propositional resizing, truncation and the powerset

Voevodsky considered resizing rules for a type theory for univalent foundations. These rules govern the syntax of the formal system, and hence are of a meta-mathematical nature.

Here we instead formulate, in our type theory without such rules, mathematical resizing principles. These principles are provable in the system with Voevodsky’s rules.

The consistency of the resizing rules is an open problem at the time of writing, but the resizing principles are consistent relative to ZFC with Grothendieck universes, because they follow from excluded middle, which is known to be validated by the simplicial-set model.

It is also an open problem whether the resizing principles discussed below have a computational interpretation.

Table of contents

Propositional resizing
We say that a type $X$ has size $\nu$ if it is equivalent to a type in the universe $\nu$:

$\text{has-size} : \nu \to (\nu : \text{Universe}) \to \nu \uplus \nu$

$X \text{ has-size } \nu = \Sigma Y : \nu \mid X \simeq Y$

The propositional resizing principle from a universe $\nu$ to a universe $\nu'$ says that every subsingleton in $\nu$ has size $\nu'$:

$\text{propositional-resizing} : ((\nu \nu' : \text{Universe}) \to (\nu \uplus \nu'))' \to \text{is-subsingleton}
\to \text{has-size } \nu'$

We also consider global propositional resizing:

$\text{propositional-resizing} : \nu \nu' \to \text{propositional-resizing } \nu \nu'$

Resizing from a universe to a higher universe just holds, of course:

$\text{upper-resizing} : \forall \nu \nu' (X : \nu) \to X \text{ has-size } \nu \to X \text{ has-size } \nu \uplus \nu'$

$\text{upper-resizing } \nu X = (\text{Lift } \nu X, \simeq \text{-Lift } X)$

Moreover, the notion of size is upper closed:

$\text{has-size-is-upper} : (X : \nu') \to X \text{ has-size } \nu \to X \text{ has-size } (\nu \uplus \nu)$

$\text{has-size-is-upper} \nu \nu' X Y = Z \gamma$

$\delta : \text{P has-size } \nu' \gamma = \text{has-size-is-upper } \nu \nu' X \text{P i} = \text{upper-resizing } \nu P$

We say that a type is small if it has a copy in the first universe:

$\text{is-small} : \nu \to \nu \uplus \nu_0'$

$\text{is-small } X = X \text{ has-size } \nu_0$

Then propositional resizing is equivalent to saying that all propositions of every universe are small:

$\text{all-propositions-are-small} : \forall \nu \to \nu'$

$\text{all-propositions-are-small } \nu = (P : \nu) \to \text{is-prop } P \to \text{is-small } P$

$\text{all-propositions-are-small-means-PR}_0 : \text{all-propositions-are-small } \nu \equiv \text{propositional-resizing } \nu \nu_0$

$\text{all-propositions-are-small-means-PR}_0 = \text{refl }$

$\text{all-propositions-are-small-gives-PR} : \text{all-propositions-are-small } \nu \to \text{propositional-resizing } \nu \nu$

$\text{all-propositions-are-small-gives-PR } \nu \nu P i = \gamma$

$\delta : \text{P has-size } \nu_0 \gamma = \text{has-size-is-upper } \nu \nu P \delta$

A global version:

$\text{All-propositions-are-small} : \nu \nu'$

$\text{All-propositions-are-small } \nu = \text{all-propositions-are-small } \nu \nu_0$
All-propositions-are-small = \forall \ U \rightarrow \text{all-propositions-are-small} \ U

PR-gives-All-propositions-are-small : \text{Propositional-resizing} \\
\quad \rightarrow \text{All-propositions-are-small} \ U

PR-gives-All-propositions-are-small \ U = \text{PR} \\

All-propositions-are-small-gives-PR : \text{All-propositions-are-small} \\
\quad \rightarrow \text{Propositional-resizing}

All-propositions-are-small-gives-PR \ a \ \{\ U \} \ \{\ V \} = \text{all-propositions-are-small-gives-PR} \ (a \ \{\ U \})

We use the following to work with propositional resizing more abstractly:

\text{resize} : \text{propositional-resizing} \ U \ V \\
\quad \rightarrow \ (P : \ U \ ' ) (i : \text{is-subsingleton} \ P) \rightarrow \ V \ '

\text{resize} \ \rho \ P \ i = \text{pr}_1 (\rho \ P \ i)

\text{resize-is-subsingleton} : (\rho : \text{propositional-resizing} \ U \ V) \\
\quad \rightarrow (P : \ U \ ' ) (i : \text{is-subsingleton} \ P) \\
\quad \rightarrow \text{is-subsingleton} \ (\text{resize} \ \rho \ P \ i)

\text{resize-is-subsingleton} \ \rho \ P \ i = \text{equiv-to-subsingleton} \ (\cong \text{-sym} (\text{pr}_2 (\rho \ P \ i))) \ i

\text{to-resize} : (\rho : \text{propositional-resizing} \ U \ V) \\
\quad \rightarrow (P : \ U \ ' ) (i : \text{is-subsingleton} \ P) \\
\quad \rightarrow P \rightarrow \text{resize} \ \rho \ P \ i

\text{to-resize} \ \rho \ P \ i = \text{"} \text{pr}_3 (\rho \ P \ i) \text{"}

\text{from-resize} : (\rho : \text{propositional-resizing} \ U \ V) \\
\quad \rightarrow (P : \ U \ ' ) (i : \text{is-subsingleton} \ P) \\
\quad \rightarrow \text{resize} \ \rho \ P \ i \rightarrow P

\text{from-resize} \ \rho \ P \ i = \text{"} \text{=\text{-sym}(pr}_2 (\rho \ P \ i)) \text{"}

Table of contents ↑

**Excluded middle gives propositional resizing**

Propositional resizing is consistent because it is implied by excluded middle, which is consistent (with or without univalence):

\text{EM-gives-all-propositions-are-small} : \text{EM} \ U \rightarrow \text{all-propositions-are-small} \ U

where

- \(Q : P + \neg P \rightarrow \mathcal{U}_0\ '\)
- \(Q \ (\text{inl} \ _) = \bot\)
- \(Q \ (\text{inr} \ _) = \emptyset\)

- \(j : (d : P + \neg P) \rightarrow \text{is-subsingleton} \ (Q \ d)\)
- \(j \ (\text{inl} \ p) = \bot\text{-is-subsingleton}\)
- \(j \ (\text{inr} \ n) = \emptyset\text{-is-subsingleton}\)

- \(f : (d : P + \neg P) \rightarrow P \rightarrow Q \ d\)
- \(f \ (\text{inl} \ _) = \star\)
- \(f \ (\text{inr} \ n) \ p = !\emptyset \ (n \ p)\)

- \(g : (d : P + \neg P) \rightarrow Q \ d \rightarrow P\)
- \(g \ (\text{inl} \ p) \ _ = \ p\)
- \(g \ (\text{inr} \ _) \ q = !\emptyset \ P \ q\)

- \(e : P \cong Q \ (\text{em} \ P \ i)\)
- \(e = \text{logically-equivalent-subsingletons-are-equivalent}\)
- \(P \ (Q \ (\text{em} \ P \ i)) \ i \ (j \ (\text{em} \ P \ i)) \ (f \ (\text{em} \ P \ i), \ g \ (\text{em} \ P \ i))\)
γ : is-small P
γ = Q (em P i), e

Hence excluded middle implies propositional resizing:

EM-gives-PR : EM U → propositional-resizing U V
EM-gives-PR {U} {V} em = all-propositions-are-small-gives-PR
(EM-gives-all-propositions-are-small em)

Table of contents

The propositional resizing axiom is a subsingleton

To show that the propositional resizing principle is a subsingleton, we use univalence here.

has-size-is-subsingleton : Univalence
→ (X : U`) (V : Universe)
→ is-subsingleton (X has-size V)

has-size-is-subsingleton {U} ua X V = univalence→ (ua V) (ua (U U V)) X

PR-is-subsingleton : Univalence → is-subsingleton (propositional-resizing U V)
PR-is-subsingleton {U} {V} ua =
Π-is-subsingleton (univalence-gives-global-dfunext ua)
(λ P → Π-is-subsingleton (univalence-gives-global-dfunext ua)
(λ i → has-size-is-subsingleton ua P V))

Exercise. It is possible to show that the propositional resizing principle is a subsingleton using propositional and functional extensionality instead of univalence.

Table of contents

Propositional impredicativity

We consider binary and unary notions of propositional impredicativity.

The binary notion of impredicativity, for two universes U and V says that the type of propositions in the universe U has a copy in the universe V.

Impredicativity : (U V : Universe) → (U U V)`
Impredicativity U V = Ω U has-size V

The unary notion of impredicativity, for one U, says that the type Ω U of propositions, which lives in the universe U`, has a copy in U itself:

is-impredicative : (U : Universe) → U `
is-impredicative U = Impredicativity U U

We can rephrase this in terms of a notion of relative smallness. We say that a type X in a successor universe U` is relatively small if it has a copy in the universe U:

is-relatively-small : U` → U `
is-relatively-small {U} X = X has-size U

impredicativity-is-Ω-smallness : Π {U} → is-impredicative U ≡ is-relatively-small (Ω U)
impredicativity-is-Ω-smallness {U} = refl

Propositional resizing gives impredicativity, in the presence of propositional and functional extensionality:

PR-gives-Impredicativity` : global-propext
→ global-dfunext
→ propositional-resizing V U
→ propositional-resizing U V
Propositional resizing doesn't imply that the first universe $\mathcal{U}_0$ is impredicative, but it does imply that all other, successor, universes $\mathcal{U}^+$ are.

What we get with propositional resizing is that all types of propositions of any universe $\mathcal{U}$ are equivalent to $\Omega \mathcal{U}_0$, which lives in the second universe $\mathcal{U}_1$:

We can rephrase this as follows:

all-propositions-are-small-gives-impredicativity$_1$ : global-propext
  → global-dfunext
  → propositional-resizing $\mathcal{U}_0$ $\mathcal{U}_1$
  → is-impredicative $\mathcal{U}_1$

PR-gives-impredicativity$_1$ : global-propext
  → global-dfunext
  → propositional-resizing $\mathcal{U}_0$ $\mathcal{U}_1$
  → Impredicativity $\mathcal{U}_1$ $\mathcal{U}_0$

We can rephrase this as follows:
all-propositions-are-small-gives-impredicativity₁ = PR-gives-impredicativity₁

Exercise. Excluded middle gives the impredicativity of the first universe, and of all other universes.

We also have that moving Ω around universes moves subsingletons around universes:

\[ \text{Impredicativity-gives-PR} : \text{propext } \mathcal{U} \]
\[ \rightarrow \text{dfunext } \mathcal{U} \mathcal{U} \]
\[ \rightarrow \text{Impredicativity } \mathcal{U} \mathcal{V} \]
\[ \rightarrow \text{propositional-resizing } \mathcal{U} \mathcal{V} \]

\[ \text{Impredicativity-gives-PR} \{\mathcal{U}\} \{\mathcal{V}\} \text{ pe fe } (O, e) P i = Q, e \]

where
\[ \mathbb{I}' : \mathcal{U} \]
\[ \mathbb{I}' = \text{Lift } \mathcal{U} \mathbb{I} \]
\[ k : \text{is-subsingleton } \mathbb{I}' \]
\[ k (\text{lift } *) (\text{lift } *) = \text{refl } (\text{lift } *) \]
\[ \text{down} : \Omega \mathcal{U} \rightarrow O \]
\[ \text{down} = \text{e } \]

\[ \text{O-is-set} : \text{is-set } O \]
\[ \text{O-is-set} = \text{equiv-to-set } (\Rightarrow \text{-sym e}) (\text{O-is-a-set } \text{fe pe}) \]

\[ Q : \mathcal{V}' \]
\[ Q = \text{down } (\mathbb{I}', k) \equiv \text{down } (P, i) \]

\[ j : \text{is-subsingleton } Q \]
\[ j = \text{O-is-set } (\text{down } (\text{Lift } \mathcal{U} \mathbb{I}, k)) (\text{down } (P, i)) \]

\[ \varphi : Q \rightarrow P \]
\[ \varphi q = \text{Id-fun } \]
\[ (\text{ap } \text{holds } (\text{equivs-are-lc } \text{down } (\Rightarrow \text{-equiv e}) q)) (\text{lift } *) \]

\[ \gamma : P \rightarrow Q \]
\[ \gamma p = \text{ap } \text{down } (\text{to-subtype-≡ } \]
\[ (\lambda _ \rightarrow \text{being-subsingleton-is-subsingleton } \text{fe}) \]
\[ (\text{pe } k i (\lambda _ \rightarrow p) (\lambda _ \rightarrow \text{lift } *)) ) \]

\[ \varepsilon : P \equiv Q \]
\[ \varepsilon = \text{logically-equivalent-subsingletons-are-equivalent } P Q i j (\gamma, \varphi) \]

Exercise. \text{propext} and \text{funext} and excluded middle together imply \text{that } Ω \mathcal{U} \text{ has size } \mathcal{U}_0.

Table of contents ⇩

Propositional resizing gives subsingleton truncation

Using Voevodsky’s construction and propositional resizing, we get that function extensionality implies that subsingleton truncations exist:

\[ \text{PR-gives-existence-of-truncations} : \text{global-dfunext} \]
\[ \rightarrow \text{Propositional-resizing} \]
\[ \rightarrow \text{subsingleton-truncations-exist} \]

\[ \text{PR-gives-existence-of-truncations} \text{ fe } R = \]
\[ \text{record} \]
\[ \{ \]
The powerset in the presence of propositional resizing

As a second, important, use of resizing, we revisit the powerset. First, given a set of subsets, that is, an element of the double powerset, we would like to consider its union. We investigate its existence in a submodule with assumptions.

module powerset-union-existence
  (pt : subsingleton-truncations-exist)
  (hfe : global-hfunext)
where

open basic-truncation-development pt hfe

Unions of families of subsets exist under the above assumption of the existence of truncations, provided the index set \( I \) is in a universe equal or below the universe of the type \( X \) we take the powerset of:

\[
\text{family-union} : \{ X : \mathcal{U} \uplus \mathcal{V} \} \{ I : \mathcal{V} \} \to (I \to \mathcal{P} X) \to \mathcal{P} X
\]

Notice the increase of universe levels in the iteration of powersets:

\[
\mathcal{P}\mathcal{P} : \mathcal{U} \to \mathcal{U} ^{\uplus}
\]
\[
\mathcal{P}\mathcal{P} X = \mathcal{P} (\mathcal{P} X)
\]

The following doesn’t assert that unions of collections of subsets are available. It says what it means for them to be available:

\[
\text{existence-of-unions} : (\mathcal{U} : \text{Universe}) \to \mathcal{U} ^{\uplus}
\]

One may wonder whether there are different universe assignments for the above definition, for instance whether we can instead assume \( \mathcal{A} : (X \to \Omega \mathcal{V}) \to \Omega \mathcal{W} \) for suitable universes \( \mathcal{V} \) and \( \mathcal{W} \), possibly depending on \( \mathcal{U} \). That this is not the case can be checked by writing the above definition in the following alternative form:

\[
\text{existence-of-unions,} : (\mathcal{U} : \text{Universe}) \to _{\uplus}
\]

\[
\text{existence-of-unions,} \mathcal{U} =
\]

\[
(X : \mathcal{U} ^{\uplus})
\]
\[
(A : (X \to \Omega _{\mathcal{U}}) \to \Omega _{\mathcal{U}})
\]
\[
\to \Sigma B : (X \to \Omega _{\mathcal{U}}), ((x : X) \to (x \in B) \leftrightarrow (\exists A : (X \to \Omega _{\mathcal{U}}), (A \in A) \times (x \in A)))
\]
The fact that Agda accepts the above definition without errors means that there is a unique way to fill each \( _ \), which has to be the following.

\[
\text{existence-of-unions}_2 : (U : \text{Universe}) \rightarrow U \rightsquigarrow \\
\text{existence-of-unions}_2 U = \\
(X : U \rightsquigarrow ) \\
(A : (X \rightarrow U \rightsquigarrow ) \rightarrow \Omega (U \rightsquigarrow )) \\
\rightarrow \Sigma B : (X \rightarrow U \rightsquigarrow ) , ((x : X) \rightarrow (x \in B) \leftrightarrow (\exists A : (X \rightarrow U \rightsquigarrow ) , (A \in A) \times (x \in A)))
\]

\[
\text{existence-of-unions-agreement} : \text{existence-of-unions} U \equiv \text{existence-of-unions}_2 U \\
\text{existence-of-unions-agreement} = \text{refl}_U
\]

To get the universe assignments explicitly, using Agda’s interactive mode, we can write holes \( ? \) instead of \( _ \) and then ask Agda to fill them uniquely by solving constraints, which is what we did to construct \( \text{existence-of-unions}_2 \).

**Exercise.** Show that the existence of unions is a subsingleton type.

Without propositional resizing principles, it is not possible to establish the existence.

\[
\text{existence-of-unions-gives-PR} : \text{existence-of-unions} U \\
\rightarrow \text{propositional-resizing} (U \rightsquigarrow ) U
\]

\[
\text{existence-of-unions-gives-PR} \{U\} \alpha = \gamma \\
\text{where} \\
\gamma : (P : U \rightsquigarrow ) \rightarrow (i : \text{is-subsingleton} P) \rightarrow P \text{ has-size } U \\
\gamma P i = Q , e \\
\text{where} \\
\iota_U : U \rightsquigarrow \\
\iota_U = \text{Lift} U \iota \\
\ast_u : \iota_U \\
\ast_u = \text{lift} \ast \\
\iota_U-is-subsingleton : \text{is-subsingleton} \iota_U \\
\iota_U-is-subsingleton = \text{equiv-to-subsingleton} (\text{Lift} \riveq \iota) \iota-is-subsingleton \\
\A : \text{PP} \iota_U \\
\A = \lambda (A : \text{P} \iota_U) \rightarrow P , i \\
\B : \text{P} \iota_U \\
\B = \text{pr}_1 (\alpha \iota U A) \\
\phi : (x : \iota_U) \rightarrow (x \in B) \leftrightarrow (\exists A : \text{P} \iota_U , (A \in A) \times (x \in A)) \\
\phi = \text{pr}_2 (\alpha \iota U A) \\
\Q : U \rightsquigarrow \\
\Q = \ast_u \in B \\
\j : \text{is-subsingleton} Q \\
\j = \in\text{-is-subsingleton} B \ast_u \\
\f : P \rightarrow Q \\
\f p = b \\
\text{where} \\
\a : \Sigma A : \text{P} \iota_U , (A \in A) \times (\ast_u \in A) \\
\a = (\lambda (x : \iota_U) \rightarrow \iota_U , \iota_U-is-subsingleton) , p , \ast_u \\
\b : \ast_u \in B \\
\b = \text{rl-implication} (\phi \ast_u) \mid a \mid \\
\g : Q \rightarrow P \\
\g q = \parallel \parallel \text{-recursion} i b a \\
\text{where} \\
\a : \exists A : \text{P} \iota_U , (A \in A) \times (\ast_u \in A) \\
\a = \text{lr-implication} (\phi \ast_u) q \\
\b : (\Sigma A : \text{P} \iota_U , (A \in A) \times (\ast_u \in A)) \rightarrow P \\
\b (A , m , \_ ) = m
The construction of intersections is as that of unions using propositional resizing:

\[
\text{PR-gives-existence-of-unions} : \text{propositional-resizing} \ (\mathcal{U}^+) \ \mathcal{U} \\
\rightarrow \text{existence-of-unions} \ \mathcal{U}
\]

The converse also holds, with an easier construction:

\[
\text{PR-gives-existence-of-unions} \ \{\mathcal{U}\} \ \rho \ \mathcal{X} \ A = \mathcal{B} , \ (\lambda \ x \rightarrow \text{lr} \ x , \ \text{rl} \ x)
\]

where

\[
\beta : X \rightarrow \mathcal{U}^+ \\
\beta x = \exists A : \mathcal{P} X , (A \in A) \times (x \in A)
\]

\[
i : (X : X) \rightarrow \text{is-subsingleton} \ (\beta x) \\
i x = \exists \text{-is-subsingleton}
\]

\[
B : \mathcal{P} X \\
B x = (\text{resize} \ \rho (\beta x) \ (i x)) , \ \text{resize-is-subsingleton} \ \rho (\beta x) \ (i x))
\]

\[
\text{lr} : (X : X) \rightarrow x \in B \rightarrow \exists A : \mathcal{P} X , (A \in A) \times (x \in A)
\]

\[
\text{lr} x = \text{from-resize} \ \rho (\beta x) \ (i x)
\]

\[
\text{rl} : (X : X) \rightarrow (\exists A : \mathcal{P} X , (A \in A) \times (x \in A)) \rightarrow x \in B \\
\text{rl} x = \text{to-resize} \ \rho (\beta x) \ (i x)
\]

We now close the above submodule and start another one with different assumptions:

module basic-powerset-development

(hfe : global-hfunext)

(\rho : \text{Propositional-resizing})

where

pt : \text{subsingleton-truncations-exist}

pt = \text{PR-gives-existence-of-truncations} \ (\text{hfunext-gives-dfunext} \ hfe) \ \rho

open basic-truncation-development pt hfe

open powerset-union-existence pt hfe

\[
\mathcal{U} : \{X : \mathcal{U}^+\} \rightarrow \mathcal{P} \mathcal{X} \rightarrow \mathcal{P} X
\]

\[
\mathcal{U} \mathcal{A} = \text{pr}_1 (\text{PR-gives-existence-of-unions} \ \rho \ _\mathcal{A})
\]

\[
\mathcal{U}\text{-property} : \{X : \mathcal{U}^+\} \ (A : \mathcal{P} X)
\rightarrow (X : X) \rightarrow (x \in \mathcal{U} \mathcal{A}) \leftrightarrow (\exists A : \mathcal{P} X , (A \in A) \times (x \in A))
\]

\[
\mathcal{U}\text{-property} \ \mathcal{A} = \text{pr}_2 (\text{PR-gives-existence-of-unions} \ \rho \ _\mathcal{A})
\]

The construction of intersections is as that of unions using propositional resizing:

\[
\text{intersections-exist} : \ \\
(X : \mathcal{U}^+) \\
(A : \mathcal{P} X)
\rightarrow \Sigma \mathcal{B} : \mathcal{P} X , ((X : X) \rightarrow (x \in B) \leftrightarrow ((A : \mathcal{P} X) \rightarrow A \in A \rightarrow x \in A))
\]

\[
\text{intersections-exist} \ \{\mathcal{U}\} X \mathcal{A} = \mathcal{B} , \ (\lambda x \rightarrow \text{lr} \ x , \ \text{rl} \ x)
\]

where

\[
\beta : X \rightarrow \mathcal{U}^+ \\
\beta x = (A : \mathcal{P} X) \rightarrow A \in A \rightarrow x \in A
\]

\[
i : (X : X) \rightarrow \text{is-subsingleton} \ (\beta x) \\
i x = \text{π-is-subsingleton} \ \text{hunapply}
\]

\[
(B : \mathcal{P} X) \\
B x = (\text{resize} \ \rho (\beta x) \ (i x)) , \ \text{resize-is-subsingleton} \ \rho (\beta x) \ (i x))
\]

\[
\text{lr} : (X : X) \rightarrow x \in B \rightarrow (A : \mathcal{P} X) \rightarrow A \in A \rightarrow x \in A \\
\text{lr} x = \text{from-resize} \ \rho (\beta x) \ (i x)
\]
Infix conventions call for intersections and arbitrary unions. For finite intersections we consider the unary case:

For example, with this we can define the type of topological spaces as follows, where

Topological spaces in the presence of propositional resizing

Table of contents

Topological spaces in the presence of propositional resizing

For example, with this we can define the type of topological spaces as follows, where \( \mathcal{O} \) consists of designated sets, conventionally called open and collectively referred to as the topology on \( X \), which are stipulated to be closed under finite intersections and arbitrary unions. For finite intersections we consider the unary case \texttt{full} and the binary case \texttt{∩}. Because the empty set is the union of the empty collection (exercise), it is automatically included among the open sets.

\[
\text{Top} : (U : \text{Universe}) \to U \to -
\]

\[
\text{Top} U = \Sigma X : U \to - \times (\Sigma \mathcal{O} : \mathcal{P} X \to \mathcal{O} \to X \to \mathcal{O}) \times (\mathcal{O} \to \mathcal{O} \to X \to \mathcal{O}) \]

infix 20 _∩_  

infix 20 _∪_

Table of contents
Notice that this jumps two universes. It is also possible, with \( \Omega \)-resizing, to construct the powerset in such a way that the powerset of any type lives in the same universe as the type (exercise), and hence so that the type of topological spaces in a base universe lives in the next universe (exercise), rather than two universes above the base universe.

**Exercise.** For a function \( X \to Y \), define its inverse image \( \mathcal{P} Y \to \mathcal{P} X \) and its direct image \( \mathcal{P} X \to \mathcal{P} Y \). Define the notion of a continuous map of topological spaces, namely a function of the underlying sets whose inverse images of open sets are open. Show that the identity function is continuous and that continuous maps are closed under composition.

---

### Quotients

We now construct quotients using a technique proposed by Voevodsky, who assumed propositional resizing for that purpose, so that the quotient of a given type by a given equivalence relation would live in the same universe as the type. But the requirement that the quotient lives in the same universe is not needed to prove the universal property of the quotient.

We construct the quotient using propositional truncations, assuming functional and propositional extensionality, *without assuming resizing*.

A binary relation \( \approx \) on a type \( X : \mathcal{U} \) with values in a universe \( \mathcal{V} \) (which can of course be \( \mathcal{U} \)) is called an *equivalence relation* if it is subsingleton-valued, reflexive, symmetric and transitive. All these notions have the same type

\[
\{ X : \mathcal{U} \} \to (X \to X \to \mathcal{V}) \to \mathcal{U} \sqcup \mathcal{V}
\]

and are defined by

\[
\text{is-subsingleton-valued} \quad \approx = \quad \forall x y \to \text{is-subsingleton} \ (x \approx y)
\]

\[
\text{reflexive} \quad \approx = \quad \forall x \to x \approx x
\]

\[
\text{symmetric} \quad \approx = \quad \forall x y \to x \approx y \rightarrow y \approx x
\]

\[
\text{transitive} \quad \approx = \quad \forall x y z \to x \approx y \rightarrow y \approx z \rightarrow x \approx z
\]

\[
\text{is-equivalence-relation} \quad \approx = \quad \text{is-subsingleton-valued} \ \approx \quad \times \text{reflexive} \ \approx \quad \times \text{symmetric} \ \approx \quad \times \text{transitive} \ \approx
\]

We now work with a submodule with parameters to quotient a given type \( X \) by a given equivalence relation \( \approx \). We assume not only the existence of propositional truncations, as discussed above, but also functional and propositional extensionality.

```plaintext
module quotient
{\mathcal{U} \mathcal{V} : \text{Universe}}
(pt : \text{subsingleton-truncations-exist})
(hfe : \text{global-hfunext})
(pe : \text{propext} \ \mathcal{V})
(X : \mathcal{U})
(\approx = : X \to X \to \mathcal{V})
(=p : \text{is-subsingleton-valued} \ \approx)
(=r : \text{reflexive} \ \approx)
(=s : \text{symmetric} \ \approx)
(=t : \text{transitive} \ \approx)
where

open \text{basic-truncation-development} \ pt \ hfe

From the given relation

\[
\approx = X \to X \to \mathcal{V}
\]
```
we define a function

\[ X \to (X \to \Omega), \]

and we take the quotient \( X/\approx \) to be the image of this function. It is for constructing the image that we need subsingleton truncations. Functional and propositional extensionality are then used to prove that the quotient is a set.

\[ \text{equiv-rel} : X \to (X \to \Omega) \]
\[ \text{equiv-rel} \ x \ y = (x = y), \approx \ x \ y \]
\[ X/\approx : \forall \cdot \exists \ \forall \cdot \forall \]
\[ X/\approx = \text{image equiv-rel} \]
\[ X/\approx -\text{is-set} : \text{is-set} X/\approx \]
\[ X/\approx -\text{is-set} = \text{subsets-of-sets-are-sets} (X \to \Omega) -\]
\[ (\text{powersets-are-sets} (\text{dfunext-gives-hfunext hunapply hunapply pe}) \]
\[ (\lambda \_ \to \exists\text{-is-subsingleton}) \]
\[ \eta : X \to X/\approx \]
\[ \eta = \text{corestriction equiv-rel} \]

We show that \( \eta \) is the universal solution to the problem of transforming equivalence \( _=\_ \) into equality \( _\equiv_\). By construction, \( \eta \) is a surjection, of course:

\[ \eta\text{-surjection} : \text{is-surjection} \eta \]
\[ \eta\text{-surjection} = \text{corestriction-surjection equiv-rel} \]

It is convenient to use the following induction principle for reasoning about the image \( X/\approx \).

\[ \eta\text{-induction} : (P : X/\approx \to \Omega) -\]
\[ \rightarrow ((x' : X/\approx) \to \exists\text{-is-subsingleton} (P x')) \]
\[ \rightarrow ((x : X) \to P (\eta x)) \]
\[ \rightarrow (x' : X/\approx) \to P x' \]
\[ \eta\text{-induction} = \text{surjection-induction} \eta \eta\text{-surjection} \]

The first part of the universal property of \( \eta \) says that equivalent points are mapped to identified points:

\[ \eta\text{-equiv-equal} : \{x \ y : X\} \to x = y \to \eta x \equiv \eta y \]
\[ \eta\text{-equiv-equal} \ {\{x\}} \ {\{y\}} \ e = \]
\[ \text{to-subtype-}\equiv \]
\[ (\lambda \_ \to \exists\text{-is-subsingleton}) \]
\[ (\text{hunapply} (\lambda \_ \to \text{being-subsingleton-is-subsingleton hunapply}) \]
\[ (\text{pe} (=p \ x \ z) (=p \ y \ z) (=t \ y \ x \ z) (=s \ x \ y \ z)) (\equiv t \ x \ y \ z) )) \]

To prove the required universal property, we also need the fact that \( \eta \) reflects equality into equivalence:

\[ \eta\text{-equal-equiv} : \{x \ y : X\} \to \eta x \equiv \eta y \to x = y \]
\[ \eta\text{-equal-equiv} \ {\{x\}} \ {\{y\}} \ p = \text{equiv-rel-reflect} (\text{ap pr1 p}) \]
\[ \text{where} \]
\[ \text{equiv-rel-reflect} : \text{equiv-rel} x \equiv \text{equiv-rel} y \to x = y \]
\[ \text{equiv-rel-reflect} q = b (=r y) \]
\[ \text{where} \]
\[ a : (y = y) \equiv (x = y) \]
\[ a = \text{ap} (\lambda \_ \to \text{pr1}(- \ y)) (q ^{-1}) \]
\[ b : y = y \to x = y \]
\[ b = \text{Id-fun} \ a \]

We are now ready to formulate and prove the required universal property of the quotient. What is noteworthy here, regarding universes, is that the universal property says that we can eliminate into any set \( A \) of any universe \( \forall \).

\[ \text{universal-property} : (A : \forall \cdot) \]
\[ \rightarrow \text{is-set} A \]
\[ \rightarrow (f : X \to A) \]
\[ \rightarrow (\{x \ x' : X\} \to x = x' \to f x \equiv f x') \]
\[ \rightarrow \exists! f' : (X/\approx \to A), f' \circ \eta \equiv f \]
universal-property \{W\} A i f \tau = e
where
G : X/\approx \rightarrow \mathcal{V} \cup \mathcal{U} \cup \mathcal{W} \cup \mathcal{V}^2
G x' = \Sigma a : A , \exists x : X , (\eta x \equiv x') \times (f x \equiv a)

\phi : (x' : X/\approx) \rightarrow \text{is-subsingleton } (G x')
\phi = \eta\text{-induction } \_ y \text{ induction-step}

where
induction-step : (y : X) \rightarrow \text{is-subsingleton } (G (\eta y))
induction-step x (a , d) (b , e) = \text{to-subtype-=} (\lambda \_ \rightarrow \exists\text{-is-subsingleton} ) p

where
h : (\Sigma x' : X , (\eta x' \equiv \eta x) \times (f x' \equiv a))
\rightarrow (\Sigma y' : X , (\eta y' \equiv \eta x) \times (f y' \equiv b))
\rightarrow a \equiv b
h (x' , r , s) (y' , t , u) = a \quad \text{η-s}^{-1}
\begin{cases}
\text{f x' \equiv (s -1 \text{η-equal-equiv } (r \cdot t^{-1}))} \\
\text{f y' \equiv (u)} \\
b 
\end{cases}

p : a \equiv b
p = \text{≡-recursion } (i a b) (\lambda \sigma \rightarrow \equiv\text{-recursion } (i a b) (h \sigma) e) d

y : (x' : X/\approx) \rightarrow \text{is-subsingleton } (\text{is-subsingleton } (G x'))
y x' = \text{being-subsingleton-is-subsingleton } \text{hunapply}

k : (x' : X/\approx) \rightarrow G x'
k = \eta\text{-induction } \_ \phi \text{ induction-step}

where
induction-step : (y : X) \rightarrow G (\eta y)
induction-step x = f x , t x , \text{refl } (\eta x) , \text{refl } (f x) |

f' : X/\approx \rightarrow A
f' x' = \text{pr}_1 (k x')

r : f' \circ \eta \equiv f
r = \text{hunapply } h

where
g : (y : Y) \rightarrow \Sigma x : X , (\eta x \equiv \eta y) \times (f x \equiv f' (\eta y))
g y = \text{pr}_2 (k (\eta y))

j : (y : Y) \rightarrow (\Sigma x : X , (\eta x \equiv \eta y) \times (f x \equiv f' (\eta y))) \rightarrow f' (\eta y) \equiv f y
j y (x , p , q) = f' (\eta y) \equiv (q -1)
\begin{cases}
f x \equiv (\text{η-equal-equiv } p) \\
f y
\end{cases}

h : (y : X) \rightarrow f' (\eta y) \equiv f y
h y = \equiv\text{-recursion } (i (f' (\eta y) ) (f y)) (j y) (g y)

c : (\sigma : \Sigma f' : (X/\approx \rightarrow A) , f' \circ \eta \equiv f) \rightarrow (f' , r) \equiv \sigma
c (f' , s) = \text{to-subtype-=} (\lambda g \rightarrow \Pi\text{-is-set} ) \text{hfe } (\lambda \_ \rightarrow 1) (g \circ \eta) f t

where
w : \forall x \rightarrow f'(\eta x) \equiv f'(\eta x)
w = \text{happily } (f' \circ \eta ) (f' \circ \eta) (r \cdot s^{-1})
t : f' \equiv f'
t = \text{hunapply } (\eta\text{-induction } \_ (\lambda x' \rightarrow i (f' x') (f'' x')) w)

e : \exists! f' : (X/\approx \rightarrow A) , f' \circ \eta \equiv f
e = (f' , r) , c

As mentioned above, if one so wishes, it is possible to resize down the quotient \(X/\approx\) to the same universe as the given type \(X\) lives by assuming propositional resizing. But we don’t see any mathematical need or benefit to do so, as the constructed quotient, regardless of the universe it inhabits, has a universal property that eliminates into any desired universe, lower, equal or higher than the quotiented type.
The following axioms are together consistent by considering Voevodsky’s simplicial-set model:

1. Function extensionality.
2. Propositional extensionality.
3. Univalence.
4. Existence of propositional truncations.
5. Excluded middle.
7. Propositional resizing and impredicativity.

We have that:

- The first four admit a constructive interpretation via cubical type theory with an implementation in cubical Agda.
- Univalence implies function extensionality and propositional extensionality.
- Simple unique choice just holds.
- Full unique choice is equivalent to function extensionality.
- Choice implies excluded middle, as usual, and both are non-constructive.
- Excluded middle implies propositional resizing and impredicativity.
- The constructive status of propositional resizing and impredicativity is open.
- Function extensionality and propositional resizing imply the existence of propositional truncations, and hence so do function extensionality and excluded middle.

The avoidance of excluded middle and choice makes the theory not only constructive but also applicable to more models. However, one is free to assume excluded middle and choice for pieces of mathematics that require them, or just if one simply prefers classical reasoning.

Univalent foundations have enough room for the constructive, non-constructive, pluralistic and neutral approaches to mathematics, and in this sense they are no different from e.g. set theoretic foundations.

An example of a theorem of univalent mathematics that requires excluded middle is Cantor-Schröder-Bernstein for homotopy types, or ∞-groupoids, which is also discussed in this blog post and is implemented in Agda.

A major omission in these notes is a discussion of higher-inductive types. On the other hand, these notes completely cover the foundational principles officially supported by UniMath, namely (1)-(7) above.

Table of contents ⇧

Appendix

Table of contents ⇧

Solutions to some exercises

```agda
module ℕ-order-exercise-solution where

≤' : ℕ → ℕ → 𝓤₀
≤' = ℕ-iteration (ℕ → 𝓤₀) (λ y → 1)
  (λ f → ℕ-recursion (𝓤₀) ⊥ (λ y P → f y))

open ℕ-order

≤-and-≤'-coincide : (x y : ℕ) → (x ≤ y) ≡ (x ≤' y)
≤-and-≤'-coincide 0 y = refl
≤-and-≤'-coincide (succ x) 0 = refl
≤-and-≤'-coincide (succ x) (succ y) = ≤-and-≤'-coincide x y
```
module N-more where

open Arithmetic renaming (\_+_ to \_+_)
open basic-arithmetic-and-order

\leq\text{-prop-valued} : (x : \mathbb{N}) \to is-subsingleton (x \leq y)
\leq\text{-prop-valued} \Theta y = \imath\text{-is-subsingleton}
\leq\text{-prop-valued} (\text{succ } x) \text{ zero} = \Theta\text{-is-subsingleton}
\leq\text{-prop-valued} (\text{succ } x) (\text{succ } y) = \leq\text{-prop-valued} x y

\leq\text{-prop-valued} : (x : \mathbb{N}) \to is-subsingleton (x \leq y)
\leq\text{-prop-valued} x y (z , p) (z' , p') = y

\text{where}
\begin{align*}
q : z \equiv z' \\
q = +\text{-lc } x z z' (x + z \equiv p) \\
y \equiv (p' \text{-}1) \\
x + z' \equiv
\end{align*}

y : z , p \equiv z' , p'
y = to-subtype\equiv (\lambda z : \mathbb{N}\text{-is-set } (x + z) y) q

\leq\text{-charac} \triangleright (x y : \mathbb{N}) \to (x \leq y) \equiv (x \equiv y)
\leq\text{-charac} pe x y = pe (\leq\text{-prop-valued} x y) (\leq\text{-prop-valued} x y)

\leq\text{-gives-} \equiv \leq \equiv \leq\text{-gives-} x y

the-subsingletons-are-the-subtypes-of-a-singleton : (X : \U \text{-}1) \to is-subsingleton X \equiv (X \equiv \imath)

the-subsingletons-are-the-subtypes-of-a-singleton X = \varphi , \psi

\text{where}
\begin{align*}
i : \text{is-subsingleton } X \to \text{is-embedding } (\imath\text{-}1 \text{ X}) \\
i s = (x , \text{refl } *) (y , \text{refl } *) = \text{ap } (\lambda x \to , \text{refl } *) (s x y)
\end{align*}

\varphi : \text{is-subsingleton } X \to X \equiv \imath
\text{where}
\begin{align*}
a : x \equiv y \to f x \equiv f y \\
a = \text{ap } f \{x\} \{y\}
\end{align*}

\psi (f , e) x y = d
\text{where}
\begin{align*}
b : \text{is-equiv } a \\
b = \text{embedding-gives-ap-is-equiv } f e x y
\end{align*}

\begin{align*}
c : f x \equiv f y \\
c = \imath\text{-is-subsingleton } (f x) (f y)
\end{align*}

\begin{align*}
d : x \equiv y \\
d = \text{inverse } a b c
\end{align*}

the-subsingletons-are-the-subtypes-of-a-singleton' : \triangleright (X : \U \text{-}1) \to \text{is-subsingleton } X \equiv (X \equiv \imath)

the-subsingletons-are-the-subtypes-of-a-singleton' \ (\text{pe } fe X = y)
\text{where}
\begin{align*}
a : \text{is-subsingleton } X \equiv (X \equiv \imath) \\
a = \text{the-subsingletons-are-the-subtypes-of-a-singleton } X
\end{align*}

\begin{align*}
b : \text{is-subsingleton } (X \equiv \imath) \\
b (f , e) (f' , e') = \text{to-subtype-=} \\
\quad \text{(being-embedding-is-subsingleton } fe) \\
\quad (fe (\lambda x : \imath\text{-is-subsingleton } (f x) (f' x)))
\end{align*}

\begin{align*}
y : \text{is-subsingleton } X \equiv (X \equiv \imath) \\
y = \text{pe } (\text{being-subsingleton-is-subsingleton } fe) b (pr_1 a) (pr_2 a)
\end{align*}

\text{G\text{-}\equiv-equation} : (ua : \text{is-univalent } (\U \cup \W))
\text{where}
\begin{align*}
(\U \text{-}1) \to (\Sigma Y : \U \cup \W \equiv , X = Y) \to \W \equiv \\
(\lambda a : \text{A } (\text{Lift } \W X , =\text{-Lift } X))
\end{align*}
module retraction-classifier
pointed-types
retractions-into
has-section-charac
≃
H↑-
H↑-
≃
G↑-
≃
where
t : (Σ Y : ℰ ∪ ℰ', X ≃ Y)
t = (Lift ℰ X , ≃-Lift X)
p : t ≡ t
p = univalence-"" {ℓ} {ℰ ∪ ℰ} ua X t t
q : p ≡ refl t
q = subsingletons-are-sets (Σ Y : ℰ ∪ ℰ', X ≃ Y)
(univalence-"" {ℓ} {ℰ ∪ ℰ} ua X) t t p (refl t)
H↑-≡-equation : (ua : is-univalent (ℰ ∪ ℰ))
→ (X : ℰ)
→ (A : (Y : ℰ ∪ ℰ') → X ≃ Y → ℰ')
→ (a : A (Lift ℰ X) (≃-Lift X))
→ H↑-≡-equation (ua X A a (Lift ℰ X) (≃-Lift X) ≡ a
has-section-charac : {X : ℰ'} {Y : ℰ'} (f : X → Y)
→ ((y : Y) → Σ x : X , f x ≡ y) ≡ has-section f
has-section-charac f = ΠΣ-distr-≈
retractions-into : ℰ' → ℰ' ≃
retractions-into {ℓ} Y = Σ X : ℰ' , Y ≦ X
pointed-types : (Ε : Universe) → ℰ' ≃
pointed-types Ε = Σ X : ℰ' , X
retraction-classifier : Univalence
→ (Y : ℰ') → retractions-into Y ≃ (Y → pointed-types Ε)
retraction-classifier
{ℓ} ua Y =
retractions-into Y
≡( i )
(Σ X : ℰ' , Σ f : (X → Y) , ((y : Y) → Σ x : X , f x ≡ y))
≡( ii )
((Ε /[ id ] Y))
≡( iii )
(Y → pointed-types Ε)
where
i = ≃-sym (Σ-cong (λ X → Σ-cong (λ f → ΠΣ-distr-≈)))
ii = Id-Eq _ _ (refl _)
iii = special-map-classifier (ua Ε)
(univalence-gives-dfunext' (ua Ε) (ua (ℰ'))) id Y

module surjection-classifier
(pt : subsingleton-truncations-exist)
(ua : Univalence)

where

hfe : global-hfunext
hfe = univalence-gives-global-hfunext ua

open basic-truncation-development pt hfe public

≡→ : ℰ' → ℰ' → ℰ ∪ ℰ'
X → Y = Σ f : (X → Y) , is-surjection f
surjections-into : ℰ' → ℰ'
surjections-into {ℓ} Y = Σ X : ℰ' , X → Y
inhabited-types : (Ε : Universe) → ℰ' ≃
inhabited-types Ε = Σ X : ℰ' , ⊥ X ⊥
surjection-classifier : Univalence
→ (Y : ℰ')
surjection-classifier Y = (Y → inhabited-types Ε)
Additional exercises

Solutions are available at the end.

Exercise. A sequence of elements of a type $X$ is just a function $\mathbb{N} \to X$. 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 : $\mathbb{N} \to (\mathbb{N} \to \mathbb{N}$)) ↠ $\Sigma$ $\alpha$ : $(\mathbb{N} \to \mathbb{N})$, $(\alpha : \mathbb{N} \to \mathbb{N}) \to \Sigma$ $n$ : $\mathbb{N}$, $\alpha \equiv e n$

cantors-diagonal : ¬($\Sigma$ $e$ : $(\mathbb{N} \to (\mathbb{N} \to \mathbb{N})$), ($(\alpha : \mathbb{N} \to \mathbb{N}) \to \Sigma$ $n$ : $\mathbb{N}$, $\alpha \equiv e n$))

Hint. It may be helpful to prove that the function $\text{succ}$ has no fixed points, first.

Exercise.

$\mathbb{2}$-has-$2$-automorphisms : dfunext $\mathcal{U}_0 \mathcal{U}_0 \to (\mathbb{2} \equiv \mathbb{2}) \equiv \mathbb{2}$

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

lifttwo : is-univalent $\mathcal{U}_0 \to$ is-univalent $\mathcal{U}_1 \to (\mathbb{2} \equiv \mathbb{2}) \equiv \text{Lift } \mathcal{U}_1 \mathbb{2}$

Exercise. Having decidable equality is a subsingleton type.

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

DNE : $\forall$ $\mathcal{U} \to \mathcal{U}$ $\cdot$' $\cdot$

DNE $\mathcal{U} = (P : \mathcal{U} \cdot) \to \text{is-subsingleton } P \to \neg
\neg P \to P$

ne : $(X : \mathcal{U} \cdot) \to \neg\neg(X + \neg X)$

DNE-gives-EM : dfunext $\mathcal{U} \mathcal{U}_0 \to \text{DNE } \mathcal{U} \to \text{EM } \mathcal{U}$

EM-gives-DNE : $\text{EM } \mathcal{U} \to \text{DNE } \mathcal{U}$

The following says that excluded middle holds if and only if every subsingleton is the negation of some type.

SN : $\forall$ $\mathcal{U} \to \mathcal{U}$ $\cdot$' $\cdot$

SN $\mathcal{U} = (P : \mathcal{U} \cdot) \to \text{is-subsingleton } P \to \Sigma$ $X : \mathcal{U}$ $\cdot$, $P \leftrightarrow \neg X$

SN-gives-DNE : SN $\mathcal{U} \to \text{DNE } \mathcal{U}$

DNE-gives-SN : DNE $\mathcal{U} \to \text{SN } \mathcal{U}$

Solutions to additional exercises

succ-no-fixed-point : (n : $\mathbb{N}$) $\to$ succ n $\not\equiv$ n

succ-no-fixed-point ${\text{0}}$ = positive-not-zero ${\text{0}}$

succ-no-fixed-point (succ n) = γ

where

IH : succ n $\not\equiv$ n

IH = succ-no-fixed-point n

γ : succ (succ n) $\not\equiv$ succ n

γ p = IH (succ-lc p)

positive-cantors-diagonal = sol

where
sol : (e : N → (N → N)) → Σ a : (N → N), ((n : N) → a ≠ e n)
sol a = (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 ≡ ∎
ocantors-diagonal = sol
where
  sol : ¬(Σ e : (N → (N → N)) , ((α : N → N) → Σ n : N , α ≠ e n))
sol (e , y) = 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 = y α

c : ∅
c = φ (pr₁ b) (pr₂ b)
oc2-has-2-automorphisms = sol
where
  sol : dfunext Λ₀ Λ₀ → (2 ≅ 2) ≅ 2
sol fe = invertibility-gives-≅ f (g , η , ε)
where
  f : (2 ≅ 2) → 2
  f (h , e) = h o
  g : 2 → (2 ≅ 2)
g₀ = id , id-is-equiv 2
  g₁ = swap₂ , swap₂-is-equiv
η : (e : 2 ≅ 2) → g (f e) ≡ e
η (h , e) = y (h₀) (h₁) (refl (h₀)) (refl (h₁))
where
  y : (m n : 2) → h₀ ≡ m → h₁ ≡ n → g (h₀) ≡ (h , e)
y₀₀ p q = !∅ (g (h₀) ≡ (h , e))
y₀₀ p q = to-subtype-≡
  (being-equiv-is-subsingleton fe fe)
  (fe (2-induction (λ n → pr₁ (g (h₀))) n ≡ h n))
  (pr₁ (g (h₀))₀ ≡ ( ap (λ → → pr₁ (g -)₀) p )
  pr₁ (g₀)₀ ≡ ( refl₀ )
 ₀ ≡ ( p⁻¹ )
  h₀ ≡ ∎)
  (pr₁ (g (h₀))₁ ≡ ( ap (λ → → pr₁ (g -)₁) p )
  pr₁ (g₁)₁ ≡ ( refl₁ )
 ₁ ≡ ( q⁻¹ )
  h₁ ≡ ∎))
y₀₁ p q = to-subtype-≡
  (being-equiv-is-subsingleton fe fe)
  (fe (2-induction (λ n → pr₁ (g (h₀))) n ≡ h n))
  (pr₁ (g (h₀))₀ ≡ ( ap (λ → → pr₁ (g -)₀) p )
  pr₁ (g₀)₀ ≡ ( refl₀ )
 ₀ ≡ ( p⁻¹ )
  h₀ ≡ ∎)
\[
\begin{align*}
\text{lifttwo} &= \text{sol} \\
\text{hde-is-subsingleton} : \text{dfunext } \mathcal{U} &\rightarrow \text{DNE } \mathcal{U} \\
\text{EM-gives-DNE} : \text{EM } \mathcal{U} &\rightarrow \text{DNE } \mathcal{U} \\
\text{SN-gives-DNE} : \text{SN } \mathcal{U} &\rightarrow \text{DNE } \mathcal{U}
\end{align*}
\]
\[ g = \text{pr}_2 (\text{pr}_2 (\text{sn} \; \text{P} \; \text{i})) \]

\[ f' : \neg \; P \rightarrow \neg (\neg \; X) \]
\[ f' = \text{contrapositive (contrapositive } f) \]

\[ h : \neg \; P \rightarrow P \]
\[ h = g \circ \text{tno X} \circ f' \]

\[ h' : \neg \; P \rightarrow P \]
\[ h' \varphi = g (\lambda (x : X) \rightarrow \varphi (\lambda (p : P) \rightarrow f \; p \; x)) \]

\[ \text{DNE-gives-SN} = \text{sol} \]

\[ \text{where} \]
\[ \text{sol : DNE } \mathcal{U} \rightarrow \text{SN } \mathcal{U} \]
\[ \text{sol} \; \text{dne} \; \text{P} \; \text{i} = \neg \; \text{P} \; , \; \text{dni} \; \text{P} \; , \; \text{dne} \; \text{P} \; \text{i} \]

**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.

\[
\text{infix } 0 \; _\sim \\
\text{infixr } 50 \; _\sim' \\
\text{infixr } 30 \; _\times \\
\text{infixr } 20 \; _+ \\
\text{infixl } 70 \; _\circ \\
\text{infix } 0 \; \text{Id} \\
\text{infix } 0 \; _\equiv \\
\text{infix } 10 \; _\Leftrightarrow \\
\text{infixl } 30 \; _\cdot \\
\text{infixr } 0 \; _\equiv (_).
\]

\[
\text{infix } 1 \; _\leftarrow \\
\text{infix } 40 \; _\leftarrow^{-1} \\
\text{infix } 10 \; _\triangleleft \\
\text{infixr } 0 \; _\triangleleft((). \\
\text{infix } 1 \; _\leftarrow\uparrow \\
\text{infix } 10 \; _\Leftarrow \\
\text{infixl } 30 \; _\cdot_\leftarrow \\
\text{infixr } 0 \; _\Leftarrow(_._
\text{infix } 1 \; _\leftarrow \\
\text{infix } 40 \; _\in \\
\text{infix } 30 \; [_,_] \\
\text{infixr } -1 \; _\Sigma \\
\text{infixr } -1 \; _\Pi \\
\text{infixr } -1 \; _\exists!
\]

Table of contents ↑